On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values

CHRISTOPH KOCH
Saarland University, Saarbrücken, Germany

This article studies the complexity of evaluating functional query languages for complex values such as monad algebra and the recursion-free fragment of XQuery. We show that monad algebra with equality restricted to atomic values is complete for the class $\mathrm{TA}[2^{O(n)}, O(n)]$ of problems solvable in linear exponential time with a linear number of alternations if the query is assumed to be part of the input. The monotone fragment of monad algebra with atomic value equality but without negation is NEXPTIME-complete. For monad algebra with deep value equality, that is, equality of complex values, we establish $\mathrm{TA}[2^{O(n)}, O(n)]$ lower and exponential-space upper bounds. We also study a fragment of XQuery, Core XQuery, that seems to incorporate all the features of a query language on complex values that are traditionally deemed essential. A close connection between monad algebra on lists and Core XQuery (with "child" as the only axis) is exhibited. The two languages are shown expressively equivalent up to representation issues. We show that Core XQuery is just as hard as monad algebra with respect to query and combined complexity, and that it is in TC_0 if the query is assumed fixed. As Core XQuery is NEXPTIME-hard, the best-known techniques for processing such problems require exponential amounts of working memory and doubly exponential time in the worst case. We present a property of queries – the lack of a certain form of composition – that virtually all real-world XQueries have and that allows for query evaluation in PSPACE and thus singly exponential time. Still, we are able to show for an important special case – Core XQuery with equality testing restricted to atomic values – that the composition-free language is just as expressive as the language with composition. Thus, under widely-held complexity-theoretic assumptions, the language with composition is an exponentially more succinct version of the composition-free language.

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

General Terms: Languages, Algorithms

Additional Key Words and Phrases: Complex values, Monad algebra, Nested-relational algebra, conservativity, XML, XQuery, Complexity, Expressiveness

Author's address: Lehrstuhl für Informationssysteme, Universität des Saarlandes, Im Stadtwald, D-66123 Saarbrücken, Germany, `koch@infosys.uni-sb.de`

The results of this article are based on [Koch 2005a], "On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values", *Proc. 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Baltimore, Maryland, 2005, and on [Koch 2005b], "On the Role of Composition in XQuery", which was presented at the *8th International Workshop on the Web and Databases (WebDB)*, Baltimore, Maryland, 2005.

1. INTRODUCTION

Complex values form part of various data models for advanced database applications, such as object-oriented, object-relational, and semistructured data models. A large amount of theoretical work on query languages for complex values has been carried out (e.g. [Jaeschke and Schek 1982; Kuper and Vardi 1993b; Abiteboul and Beeri 1995; Hull and Su 1989; Grumbach and Vianu 1995; Tannen et al. 1992; Grumbach and Milo 1996; Hull and Su 1993; Buneman et al. 1995; Abiteboul and Hillebrand 1995; Paredaens and Van Gucht 1988; Wong 1996; Dantsin and Voronkov 1997; Libkin and Wong 1997; Vorobyov and Voronkov 1998; Dantsin and Voronkov 2000]), and this has laid the foundations for object-oriented query languages as well as SQL 1999 or XQuery.

Earlier complexity studies on query languages for complex values have almost entirely focused on logic- [Kuper and Vardi 1993a] and particularly logic programming-based query languages [Vorobyov and Voronkov 1998; Dantsin and Voronkov 2000; Dantsin et al. 2001], and fixpoint languages (e.g. [Grumbach and Vianu 1995]). However, the query languages considered by many researchers to be most natural for complex values (such as *complex value algebra without powerset* [Abiteboul and Beeri 1995; Abiteboul et al. 1995], its syntactic variant *monad algebra* [Tannen et al. 1992; Buneman et al. 1995], and XQuery) are functional.

Monad algebra. Monad algebra is a clean, compositional, variable-free functional query language that derives its power to manipulate complex values from its support for defining higher-order operations. It was shown expressively equivalent to a number of other important complex-value query languages such as *nested relational algebra* [Jaeschke and Schek 1982] and complex value algebra without powerset in earlier research [Tannen et al. 1992]. (Complex value algebra *with powerset* [Kuper and Vardi 1993b; Abiteboul and Beeri 1995; Grumbach and Milo 1996] can take hyperexponential runtime. Queries that really need the powerset operator are usually too costly to evaluate.)

Since some of these languages were developed driven by practical requirements rather than from first principles as is the case for monad algebra, and nevertheless all the languages ended up with the same expressive power, it appears that the expressiveness of these languages on complex values is "the right one" to many researchers and plays a role analogous to that of the power of first-order logic (or relational algebra) on the relational model.

One known result [Suciu and Tannen 1997] is that monad algebra is in TC_0 with respect to *data complexity* (i.e., if the query is assumed fixed [Vardi 1982]). However, the complexity of monad algebra if the query is assumed variable (*query/combined complexity* [Vardi 1982]) is open. In this article, we study the complexity of monad algebra under the latter assumption.

XQuery. XQuery is destined to become the dominant data-transformation query language for XML data and to take a role analogous to the one occupied by SQL for relational databases.

It is folklore that full XQuery is Turing-complete, but it is also obvious that queries without recursion are guaranteed to terminate already in straightforward functional implementations of the XQuery language. Recursion in XQuery is rarely used in practice (see also [XQueryUseCases 2005]); recursive XML transformations are usually implemented in XSLT.

In essence, XQuery is a quite natural typed functional programming language for XML; still it is sometimes criticized by the research community as huge and clumsy. In this article we study a substantial recursion-free fragment of XQuery, which we call Core XQuery. It seems that Core XQuery contains all and only the features one would expect from a functional query language for unranked trees in the spirit of complex-value algebra without powerset.[1]

Little foundational research on XQuery has been done to date. There are only some cautious first attempts at finding clean formalizations of and algebras for the language [Hidders et al. 2004; Fernandez et al. 2000; World Wide Web Consortium 2005]. Most other recent work has focused on engineering good query processors for XQuery [Ludäscher et al. 2002; Marian and Siméon 2003; Florescu et al. 2003; Fernandez and Siméon 2004; Koch et al. 2004].

In this article, we attempt a first closer look at the complexity of XQuery, or more precisely, of the Core XQuery fragment. We attempt to do this in a principled manner, establishing connections to earlier, well-studied formalisms for functional queries on complex-value databases [Paredaens and Van Gucht 1988; Tannen et al. 1992; Buneman et al. 1995; Wong 1996; Grumbach et al. 1996; Libkin and Wong 1997]. Indeed our results on the complexity of monad algebra quite directly yield a characterization of the complexity of Core XQuery.

Contributions. The technical contributions of this article are as follows.

—We introduce the Core XQuery language, a simple yet powerful nonrecursive fragment of XQuery.

—Monad algebra on lists is incomparable with Core XQuery in the strict sense due to differences in the data models employed (trees versus complex values). In particular, Core XQuery cannot simulate tuples.[2] Nevertheless, we exhibit a close connection between XQuery and monad algebra on lists and show that the Core XQuery queries that use only the child axis for navigation in data trees capture monad algebra on lists *up to representation issues* (using mappings that factor these differences in the data models out).

The established mappings are efficiently computable. This allows us to prove complexity results interchangeably for monad algebra and Core XQuery, but it also gives a very concise formal semantics to Core XQuery – through monad algebra.

—We show that monad algebra (on sets, lists and bags) and Core XQuery, both with equality on atomic values but without negation, are NEXPTIME-complete with respect to query/combined complexity.

This gives a negative answer (under the complexity-theoretic assumption that PSPACE $\neq$ NEXPTIME) to the longstanding open question [Van den Bussche 2005] whether there is a polynomial-time mapping from nested-relational algebra on flat relations to classical (flat) relational algebra (which is just PSPACE-

[1]Core XQuery is not to be confused with the XQuery Core [World Wide Web Consortium 2005], which is a much larger fragment of XQuery for which this expressive correspondence with monad algebra, nested relational algebra, and similar languages does not hold.

[2]Full XQuery can, however – using either position arithmetics or attributes.

complete [Stockmeyer 1974], see also [Abiteboul et al. 1995] for a more recent exposition).

—We show that monad algebra and Core XQuery in the presence of negation and equality on atomic values are complete for TA[$2^{O(n)}, O(n)$] with respect to query/combined complexity.

—For the case of our query languages with deep equality, we obtain an EXPSPACE upper bound. A TA[$2^{O(n)}, O(n)$] lower bound follows from the fact that negation (and therefore universal quantification "every" in XQuery) is easily definable using deep equality.
Note that the EXPSPACE upper bound is a rather robust one and should extend to all or at least a much larger part of nonrecursive XQuery.

—We show that Core XQuery is in TC_0 with respect to data complexity.

Core XQuery is NEXPTIME-hard with respect to query complexity, and as a consequence, it is commonly believed that any query evaluation algorithm for nonrecursive XQuery must consume doubly exponential time and exponential space for query evaluation in the worst case (cf. e.g. [Johnson 1990]). This is by an exponential factor worse than the complexity of relational algebra or calculus [Stockmeyer 1974].

We present a syntactic property – the lack of a certain form of composition – that virtually all real-world XQueries have and which renders *composition-free* Core XQuery just as hard as relational algebra.

By composition, informally, we refer to the assignment of all or part of a constructed data value to a variable or the use of a constructed value in a where-condition. A data value is called *constructed* iff it is defined by an XQuery expression other than an XPath statement. For example, the query

```
<books_2004>
{ for $x in /bib/book where $x/year=2004 return
     <book>
        {$x/title}
        <authors>
           { for $y in $x/author return
               <author> {$y/lastname} </author>
           }
        </authors>
     </book>
}
</books_2004>
```

is a composition-free query (so nesting queries, by using FLWR-statements in the return clauses of other FLWR statements, is not a problem) while

```
<books>
{ let $x := <a>{ for $w in /bib/book return <b> {$w} </b> }</a>
  for $y in $x/b return $y/*
}
</books>
```

Monad algebra and Core XQuery with composition:

	with negation	without negation
deep equality	in EXPSPACE; TA[$2^{O(n)}, O(n)$]-hard	
equality on atomic values	TA[$2^{O(n)}, O(n)$]-complete	NEXPTIME-complete

Core XQuery without composition:

	with negation	without negation
deep equality	PSPACE-complete	NP-complete
equality on atomic values	PSPACE-complete	NP-complete

Table I. Query/combined complexity of monad algebra and Core XQuery.

is not composition-free because it uses a let-expression that assigns a constructed tree to variable $\$x$. The equivalent query

```
<books>
{ for $y in (for $w in /bib/book return <b> {$w} </b>) return $y/* }
</books>
```

is still not composition-free because the "in"-expression of the outer for-loop contains a for-loop. However, there is an equivalent composition-free query,

```
<books>
   { for $w in /bib/book return $w }
</books>
```

Our contributions regarding composition-free Core XQuery are as follows.

—It is shown that composition-free Core XQuery can be evaluated in polynomial space and thus also in singly exponential time. In fact, composition-free nonrecursive XQuery is PSPACE-complete with respect to query/combined complexity.

—We show that composition-free Core XQuery without negation is NP-complete.

—Still, we are able to show for an important special case – equality is restricted to atomic values – that composition-free Core XQuery is just as expressive as Core XQuery with composition, i.e., *composition-free Core XQuery with atomic equality is closed under composition*. Thus, under the usual complexity-theoretic assumptions, the language with composition is exponentially more succinct than the composition-free language.

Table I summarizes our complexity results for query and combined complexity. Since the variables in composition-free XQuery range only over subtrees of the input tree, supporting deep value equality has no influence on the complexity of queries, differently from the case of Core XQuery with composition.

To the best of the author's knowledge, this is the first work characterizing the complexity of XQuery. The mappings to and from monad algebra also give an argument that Core XQuery is a well-designed language that offers the "right" degree of expressive power.

Nonrecursive composition-free XQuery is an important class of queries, and indeed, most practical XQueries belong to this class. (For instance, only a handful of the XML Query Use Case queries [XQueryUseCases 2005] employ composition.)

Composition-free (Core) XQuery is also popular among implementors of limited prototype XQuery engines, e.g. [Koch et al. 2004]. Our preliminary expressiveness results show that restricting oneself to implementing composition-free Core XQuery does not cause a loss of generality, at least if equality checking is limited to atomic value equality. The expressiveness result also gives a partial explanation for why practical XQueries tend to be composition-free, as observed above.

Note that other functional languages such as monad algebra do not seem to have natural "composition-free" fragments that remain expressive.

A major motivation of this work is to define simple but relevant fragments of XQuery suitable for research prototype implementations and theoretical study (see also [Hidders et al. 2004] for another attempt towards the latter goal). Indeed, composition-free Core XQuery may allow for special, efficient implementation techniques because all XQuery variables only range over nodes in the input tree (never over nodes from intermediate query results).

Related work. It seems that the most relevant work regarding the problems studied in this article – apart from the characterization of the data complexity of monad algebra in [Suciu and Tannen 1997] – is on the complexity of nonrecursive logic programming.

For nonrecursive logic programming, a full complexity characterization [Dantsin et al. 2001] has been obtained for the most common forms of complex values (that is, values built from sets, lists, bags, tuples, and atomic values) and various classes of logic programs (with and without negation, range-restriction, and types). It turns out that the complexity of nonrecursive logic programming is robustly (for various kinds of complex values, and with or without range-restriction) NEXPTIME-complete. In the presence of negation (and necessarily range-restriction), nonrecursive logic programming is known to be in the class $\text{TA}[2^{O(n)}, O(n)]$ [Vorobyov and Voronkov 1998] and hard for the class $\text{TA}[2^{O(n/\log n)}, O(n/\log n)]$ [Voronkov 2004; Dantsin et al. 2001].

A main difference between functional languages such as monad algebra and XQuery and logic programming as studied in [Dantsin and Voronkov 1997; Vorobyov and Voronkov 1998] is the form of nonmonotonicity employed. In functional languages that have the power to check the equality of complex values, negation is usually a redundant operation. Equality does introduce nonmonotonicity into the functional languages, while the seemingly same deep equality in logic programming languages does not. Nonmonotonicity in the functional languages is different from and seemingly more powerful than that obtained through negation in nonrecursive normal logic programming.

For example, in monad algebra, we can compute two different complex values of doubly exponential size each and then check their equality. A priori, one would assume that such a check requires a "proof" involving a doubly exponentially sized proof tree, or in other words, one would assume that comparing two values requires reading all of their data. However, this is not the case. The upper bounds on the complexity of nonrecursive logic programming rely on the fact that unifiers in SLD resolution proofs of nonrecursive logic programs cannot grow beyond singly exponential size.

The work [Grumbach and Vianu 1995; Kuper and Vardi 1993a] is on more expressive query languages. [Kuper and Vardi 1993a] proves *LDM logic* without powerset complete for the class TA$[2^{O(n)}, O(n)]$. Differently from monad algebra, LDM logic is a logical language with quantification, operates on cyclic data, and cannot express deep equality.

Structure. The structure of this article is as follows. Section 2 discusses the notions from complexity theory used in the technical sections of the article and introduces complex values and monad algebra (on sets, lists, and bags). Section 3 defines the Core XQuery fragment and provides efficiently computable mappings between monad algebra on lists and Core XQuery. Section 4 gives the EXPSPACE bound on the query complexity of monad algebra and Core XQuery in the presence of deep equality. Section 5 presents the complexity results for the languages with atomic equality, with and without negation. It starts with the upper bounds in Section 5.1 and follows up with the corresponding lower bounds in Section 5.2. Section 6 presents our results on the data complexity of XQuery (and summarizes analogous results for monad algebra). In Section 7 we introduce composition-free Core XQuery. In Section 7.1, we prove the PSPACE- and NP-completeness results for the complexity of composition-free Core XQuery. Finally, in Section 7.2, we prove the expressiveness result that composition-free Core XQuery with atomic equality captures full Core XQuery with atomic equality.

2. PRELIMINARIES

2.1 Complexity-Theoretic Background

By AC$_0$ we refer to the class of languages recognizable by LOGSPACE-uniform families of circuits of polynomial size and constant depth using and- and or-gates of unbounded fan-in. By TC$_0$ we refer to the same class except that in addition so-called majority-gates are permitted, which compute “true” iff more than half of their inputs are true. For details on circuit complexity and the notion of uniformity we refer to [Greenlaw et al. 1995; Johnson 1990].

We assume deterministic, nondeterministic, and alternating Turing machines known and refer to e.g. [Johnson 1990] for definitions. By DTIME$[t(n)]$ and NTIME$[t(n)]$, we denote the classes of all problems solvable in time $t(n)$ (where n is the size of the input) on deterministic and nondeterministic Turing machines, respectively. By DSPACE$[s(n)]$, we denote the classes of all problems solvable in space $s(n)$ on deterministic Turing machines. By TA$[t(n), a(n)]$, we denote the class of problems solvable in time $t(n)$ using $a(n)$ alternations on alternating Turing machines.

We will use the following abbreviations for complexity classes in this article:

$$\begin{aligned}
\text{NETIME} &= \text{NTIME}[2^{O(n)}] \\
\text{NEXPTIME} &= \text{NTIME}[2^{n^{O(1)}}] \\
\text{2ETIME} &= \text{DTIME}[2^{2^{O(n)}}] \\
\text{2EXPTIME} &= \text{DTIME}[2^{2^{n^{O(1)}}}] \\
\text{LOGSPACE} &= \text{DSPACE}[O(\log n)] \\
\text{EXPSPACE} &= \text{DSPACE}[2^{n^{O(1)}}]
\end{aligned}$$

It is known that $\mathrm{AC}_0 \subseteq \mathrm{TC}_0 \subseteq \mathrm{LOGSPACE} \subset \mathrm{NEXPTIME} \subseteq \mathrm{TA}[2^{n^{O(1)}}, 1] \subseteq \mathrm{TA}[2^{n^{O(1)}}, n^{O(1)}] \subseteq \mathrm{TA}[2^{n^{O(1)}}, 2^{n^{O(1)}}] = \mathrm{EXPSPACE} \subseteq \mathrm{2EXPTIME}$. Moreover,

$$\mathrm{NETIME} \subseteq \mathrm{TA}[2^{O(n)}, O(n)] \subseteq \mathrm{2ETIME} \subset \mathrm{2EXPTIME}$$

(cf. e.g. [Johnson 1990; Chandra et al. 1981]).

The complexity classes NETIME and 2ETIME are not robust – they are not closed under LOGSPACE-reductions, as can be verified using a simple padding argument and the Time Hierarchy theorem [Hartmanis et al. 1965]. We will consider completeness for those classes as well as of $\mathrm{TA}[2^{O(n)}, O(n)]$ under *LOGLIN-reductions*, under which they are known to be closed (cf. e.g. [Dantsin et al. 2001]). By a LOGLIN reduction, we denote a LOGSPACE reduction that produces output of linear size. $\mathrm{TA}[2^{O(n)}, O(n)]$ has important complete problems from logic, such as deciding the Theory of Real Addition [Berman 1980; Ferrante and Rackoff 1975].

There are a number of alternative ways of stating the query evaluation problem. In this article, we study the complexity of Boolean queries. For both monad algebra and XQuery, we think of a nonempty collection (set, list, or bag) as "true" and an empty one as "false".

We study three kinds of complexity of query evaluation, *data complexity* (where queries are assumed to be fixed and data variable, that is, part of the input), *query complexity* (where the query is variable and the data is assumed to be fixed), and *combined complexity* (where both data and query are considered variable) [Vardi 1982].

2.2 Complex Values and Monad Algebra on Sets

We now introduce monad algebra on sets. We consider complex values constructed from sets, tuples, and atomic values from a single-sorted domain[3]. Types are terms of the grammar

$$\tau ::= \mathrm{Dom} \mid \{\tau\} \mid \langle A_1 : \tau_1, \ldots, A_k : \tau_k \rangle$$

where $k \geq 0$ and $A_1, \ldots, A_k$ are called attribute values.

Consider the query language on complex values consisting of expressions built from the following operations (the types of the operations are provided as well):

(1) identity

$$\mathrm{id} : x \mapsto x \qquad \tau \to \tau$$

(2) composition[4]

$$f \circ g : x \mapsto g(f(x)) \qquad \frac{f : \tau \to \tau',\ g : \tau' \to \tau''}{f \circ g : \tau \to \tau''}$$

(3) constants from $\mathrm{Dom} \cup \{\emptyset, \langle\rangle\}$ ($\langle\rangle$ is the nullary tuple)

(4) singleton set construction

$$\mathrm{sng} : x \mapsto \{x\} \qquad \tau \to \{\tau\}$$

[3]All results in this article immediately generalize to many-sorted domains.
[4]Again, our convention throughout the article is that $(f \circ g)(x) = g(f(x))$, not $f(g(x))$.

(5) application of a function to every member of a set

$$\text{map}(f) : X \mapsto \{f(x) \mid x \in X\} \qquad \frac{f : \tau \to \tau'}{\text{map}(f) : \{\tau\} \to \{\tau'\}}$$

(6) unnesting sets of sets:

$$\text{flatten} : X \mapsto \bigcup X \qquad \{\{\tau\}\} \to \{\tau\}$$

(7) pairing

$$\text{pairwith}_{A_1} : \langle A_1 : X_1, A_2 : x_2, \ldots, A_n : x_n\rangle \mapsto \{\langle A_1 : x_1, A_2 : x_2, \ldots, A_n : x_n\rangle \mid x_1 \in X_1\}$$

$$\langle A_1 : \{\tau_1\}, A_2 : \tau_2, \ldots, A_n : \tau_n\rangle \to \{\langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle\}$$

(pairwith$_{A_i}$ for $i > 1$ is defined analogously.)

(8) tuple formation

$$\langle A_1 : f_1, \ldots, A_n : f_n\rangle : x \mapsto \langle A_1 : f_1(x), \ldots, A_n : f_n(x)\rangle$$

$$\frac{f_1 : \tau \to \tau_1, \ldots, f_n : \tau \to \tau_n}{\langle A_1 : f_1, \ldots, A_n : f_n\rangle : \tau \to \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle}$$

(9) projection

$$\pi_{A_i} : \langle A_1 : x_1, \ldots, A_i : x_i, \ldots, A_n : x_n\rangle \mapsto x_i$$

$$\pi_{A_i} : \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle \to \tau_i$$

The language just defined has a nice theoretical foundation from programming language theory, that of structural recursion on sets extended by a small amount of machinery for creating and destroying tuples [Tannen et al. 1992]. Formally, the language above is a Cartesian category with a *strong monad* on it (where "strong" refers to so-called *tensorial strength* introduced by the "pairwith" operation). We call this language $\mathcal{M}$ [Tannen et al. 1992].

We use flatmap(f) as a shortcut for map$(f) \circ$ flatten. Observe that projection π is applied to tuples rather than to sets of tuples as in relational algebra. For example, the relational algebra expression π_{AB} corresponds to the expression map$(\langle A : \pi_A, B : \pi_B\rangle)$ in $\mathcal{M}$.

EXAMPLE 2.1. The Cartesian product $f \times g$ can be defined as $\langle 1 : f, 2 : g\rangle \circ$ pairwith$_1 \circ$ flatmap(pairwith$_2$).

Observe the difference from the product of relational algebra. For instance, the query id$\times$id on a set of pairs S computes $\{\langle\langle x_1, x_2\rangle, \langle x_3, x_4\rangle\rangle \mid \langle x_1, x_2\rangle, \langle x_3, x_4\rangle \in S\}$ rather than $\{\langle x_1, x_2, x_3, x_4\rangle \mid \langle x_1, x_2\rangle, \langle x_3, x_4\rangle \in S\}$. □

It is customary to define Boolean queries ("predicates") as queries that produce values of type $\{\langle\rangle\}$, i.e., that either return $\{\langle\rangle\}$ ("true") or $\emptyset$ ("false") [Tannen et al. 1992]. Note that the logical conjunction $\gamma \wedge \delta$ of two predicates γ and δ can be computed as $\gamma \times \delta$.

By *positive monad algebra* $\mathcal{M}_\cup$, we denote $\mathcal{M}$ extended by the set union operation $\cup$. This language has a number of nice properties [Tannen et al. 1992; Buneman et al. 1995], but it is known to be incomplete as a practical query language because it cannot yet express a value equality predicate

$$(A_i = A_j) : \langle A_1 : \tau_1, \ldots, A_k : \tau_k\rangle \to \{\langle\rangle\}.$$

Equality of atomic values, $(A_i =_{atomic} A_j)$ with $\tau_i = \tau_j = \mathrm{Dom}$ is defined as

$$\langle A_1 : x_1, \ldots, A_k : x_k\rangle \mapsto \{\langle\rangle \mid x_i = x_j\}.$$

In the following we will also consider "deep" equality of arbitrary complex values, denoted $=_{deep}$, which is $=_{atomic}$ on atomic values and inductively holds on tuple values $\langle A_1 : x_1, \ldots, A_k : x_k\rangle$, $\langle A_1 : y_1, \ldots, A_k : y_k\rangle$ of the same type iff $x_1 =_{deep} y_1 \wedge \cdots \wedge x_k =_{deep} y_k$ and on set values X, Y of the same type iff for all $x \in X$ there is a $y \in Y$ such that $x =_{deep} y$ and vice versa. We will use the symbol $=$ whenever a statement is made about both forms of equality.

If we extend $\mathcal{M}_\cup$ by any nonempty subset of the operations deep value equality $(A =_{deep} B)$, testing set membership $(A \in B)$ or containment $(A \subseteq B)$, selection $\sigma_{A=B}$, set difference "$-$", set intersection $\cap$, or nesting[5], we always get the same expressive power.[6] We will call any one of these extended languages *full monad algebra.*

Theorem 2.2 [Tannen et al. 1992].

$$\mathcal{M}_\cup[=_{deep}] \equiv \mathcal{M}_\cup[\sigma] \equiv \mathcal{M}_\cup[-] \equiv \mathcal{M}_\cup[\cap] \equiv \mathcal{M}_\cup[\subseteq] \equiv \mathcal{M}_\cup[\in] \equiv \mathcal{M}_\cup[nest].$$

Moreover, generalizing selections to test against constants or to support "$\in$", "$\subseteq$" or allowing for Boolean combinations of conditions does not increase the expressiveness of full monad algebra [Tannen et al. 1992].

Example 2.3. Given a Boolean predicate γ, selection σ_γ can be expressed as $\mathrm{flatmap}\big(\langle 1 : \mathrm{id}, 2 : \mathrm{id} \circ \gamma\rangle \circ \mathrm{pairwith}_2 \circ \mathrm{map}(\pi_1)\big)$. Predicate $(A \subseteq B)$ can be expressed in $\mathcal{M}_\cup[=_{deep}]$ as $\langle A : \pi_A, A' : \pi_A \cap \pi_B\rangle \circ (A =_{deep} A')$ where $f \cap g := (f \times g) \circ \sigma_{1=2} \circ \mathrm{map}(\pi_1)$. A predicate $(f \subseteq g)$ can be expressed as $\langle 1 : f, 2 : g\rangle \circ (1 \subseteq 2)$. □

Example 2.4. Given a complex value of type $\langle R : \{\tau\}, S : \{\tau\}\rangle$, difference $R - S$ can be implemented in $\mathcal{M}_\cup[\sigma]$ as

$$\mathrm{pairwith}_R \circ \mathrm{map}\big(\langle R : \pi_R, S_R : \langle R : \pi_R, S : \pi_S\rangle \circ \mathrm{pairwith}_S \circ \sigma_{R=S}\rangle\big) \circ \sigma_{S=\emptyset} \circ \mathrm{map}(\pi_R).$$

The idea is to compute, for each element r of R, the set S_R of elements in S that are equal to r and then to select those elements r of R for which S_R is empty. □

Theorem 2.2 demonstrates that full monad algebra (w.l.o.g., $\mathcal{M}_\cup[=_{deep}]$) is a very robust notion. It can serve as an "expressiveness benchmark" for query languages on complex-value databases. Indeed, it has been shown that full monad algebra is a *conservative extension* of relational algebra.

By a flat relational database, we denote a relational database in the classical sense [Codd 1970; Abiteboul et al. 1995]. In our data model, a (flat) relational database

[5]The "nest" operation of complex value algebra without powerset [Abiteboul and Beeri 1995] groups tuples by some of their attributes. For example, $\mathrm{nest}_{C=(B)}(R)$ on relation $R(AB)$ computes the value $\{\langle A\colon x, C\colon \{\langle B\colon y\rangle \mid \langle A\colon x, B\colon y\rangle \in R\}\rangle \mid (\exists y)\langle A\colon x, B\colon y\rangle \in R\}$.
[6]No analogous statement can be made about flat relational algebra.

can be represented as a tuple of (flat) relations, where a (flat) relation of arity k is a set of tuples of atomic values, i.e. a value of type $\{\langle A_1 : \mathrm{Dom}, \ldots, A_k : \mathrm{Dom}\rangle\}$.

THEOREM 2.5 [PAREDAENS AND VAN GUCHT 1988]. *A mapping from a (flat) relational database to a (flat) relation is expressible in $\mathcal{M}_\cup[=_{deep}]$ if and only if it is expressible in relational algebra.*

A generalized version of Theorem 2.5 can be found in [Wong 1996].

2.3 Monad Algebra on Lists and Bags

We will also consider monad algebra on lists $\mathcal{M}_\cup^{[\,]}$ and bags $\mathcal{M}_\cup^{\{\!|\,|\!\}}$ in this article (see also [Tannen et al. 1992; Buneman et al. 1995; Libkin and Wong 1997]). We will be parsimonious with notation here, but this should not lead to confusion throughout the article. We will use the same syntax and operation names as for monad algebra on sets, but now, for instance, $\cup$ on lists denotes the concatenation of two lists and "flatten" concatenates the list-typed members of a list in order of appearance. For bags, these operations ignore order but preserve duplicates. Of course, two lists are equal iff they are of the same length and for each i, the i-th members of the two lists are equal. Two bags are equal iff each member of either bag occurs the same number of times in both bags.

For bags, we will also consider the additional operations "unique", which simply eliminates duplicates from bags, and "monus" (a powerful version of difference which allows to express arithmetics in monad algebra on bags), defined such that b monus b' is the bag consisting of the elements x of b with multiplicity $\#x(b \text{ monus } b') = \max(0, \#x(b) - \#x(b'))$; for instance,

$$\{\!|a, a, a, b, b, b, c, d|\!\} \text{ monus } \{\!|a, a, b, c, e|\!\} = \{\!|a, b, b, d|\!\}.$$

In [Libkin and Wong 1997] it was shown that adding either of these two operations strictly increases the expressive power of the language (and adding both makes the language yet stronger).

We also consider the extension of $\mathcal{M}_\cup^{[\,]}$ by a further operation "true" which evaluates to $[\langle\rangle]$ (true) on a list if it is nonempty and to $[\,]$ (false) otherwise. We use "true" to eliminate duplicate entries from list-typed truth values (e.g., from $[\langle\rangle, \ldots, \langle\rangle]$).

3. CORE XQUERY

We consider the fragment of XQuery with abstract syntax

$$\begin{array}{rcl}
\mathit{query} & ::= & () \mid \langle a\rangle \mathit{query}\langle/a\rangle \mid \mathit{query}\ \mathit{query} \mid \mathit{var} \mid \mathit{var}/\mathit{axis} :: \nu \\
& \mid & \text{for } \mathit{var} \text{ in } \mathit{query} \text{ return } \mathit{query} \\
& \mid & \text{if } \mathit{cond} \text{ then } \mathit{query} \\
\mathit{cond} & ::= & \mathit{var} = \mathit{var} \mid \mathit{query}
\end{array}$$

where a denotes the XML tags, *axis* the XPath axes [7], *var* a set of XQuery variables $\$x$, $\$x_1$, $\$x_2$, ..., $\$y$, $\$z$, ... with a distinguished *root variable* (which is the unique

[7]For simplicity, particularly of the following semantics, we only consider the child and the descendant axis, but some complexity upper bounds in this article hold for all XPath axes. Such theorems refer to "all axes".

$$
\begin{array}{rcl}
[\![()]\!]_k(\vec{e}) & := & [\,] \\
[\![\langle a\rangle\alpha\langle/a\rangle]\!]_k(\vec{e}) & := & [\langle a\rangle[\![\alpha]\!]_k(\vec{e})\langle/a\rangle] \\
[\![\alpha\ \beta]\!]_k(\vec{e}) & := & [\![\alpha]\!]_k(\vec{e}) + [\![\beta]\!]_k(\vec{e}) \\
[\![\$x_i]\!]_k(t_1,\dots,t_k) & := & [t_i] \\
[\![\$x_i/\chi :: \nu]\!]_k(t_1,\dots,t_k) & := & \text{return list } [t'_1,\dots,t'_l] \text{ of subtrees of } t_i \text{ such that} \\
 & & \{v \mid \chi^{t_i}(root^{t_i}, v) \wedge \mathrm{lab}^{t_i}_{\nu}(v)\} = \{root^{t'_1},\dots,root^{t'_l}\} \\
 & & \text{and } root^{t'_1} <^{t_i}_{doc} \cdots <^{t_i}_{doc} root^{t'_l} \\
[\![\text{for } \$x_{k+1} \text{ in } \alpha \text{ return } \beta]\!]_k(\vec{e}) & := & \text{let } [\![\alpha]\!]_k(\vec{e}) = [t_1,\dots,t_l]; \\
 & & \ \text{return } [\![\beta]\!]_{k+1}(\vec{e},t_1) + \cdots + [\![\beta]\!]_{k+1}(\vec{e},t_l) \\
[\![\text{if } \phi \text{ then } \alpha]\!]_k(\vec{e}) & := & \text{if } [\![\phi]\!]_k(\vec{e}) \text{ is nonempty then } [\![\alpha]\!]_k(\vec{e}) \text{ else } [\,] \\
[\![\$x_i = \$x_j]\!]_k(t_1,\dots,t_k) & := & \text{if } t_i = t_j \text{ then } [\langle\text{yes}/\rangle] \text{ else } [\,]
\end{array}
$$

Fig. 1. Semantics of Core XQuery.

free variable in the query), and ν a *node test* (either a tag name or "*"). We refer to this fragment as *Core XQuery*, or *XQ* for short.

For simplicity, we will work with pure node-labeled unranked ordered trees, and by atomic values, we will refer to leaves (or equivalently, their labels). This requires to assume an infinite labeling alphabet, but for our results this does not cause a problem.[8]

XQuery supports several forms of equality. We will not try to use the same syntax (=, eq, or deep_equal) as in the current standards proposal – it is not clear whether the syntax has stabilized. Throughout this article, equality is by value, that is, by value as an ordered unranked tree or equivalently by value of the corresponding XML document as a text string[9]. Other notions of equality such as equality of node identifiers will not be considered. We will write $=_{deep}$ and $=_{atomic}$ for deep and atomic equality, respectively. We will use $=$ for statements that apply to both forms of equality.

Our only other divergence from XQuery syntax is that we assume if-expressions of the form "if ϕ then α" rather than "if ϕ then α else β". Of course, our if-expressions can be considered as a shortcut for "if ϕ then α else ()" and else-branches can be simulated using negation, "if not(ϕ) then β".

We use the shortcuts $\langle a/\rangle$ for $\langle a\rangle()\langle/a\rangle$ and $\$x/a$ for $\$x/$child::$a$.

We define the semantics of an *XQ* expression α with k free variables using a function $[\![\alpha]\!]_k$ – given in Figure 1 – that takes a k-tuple $\vec{e}$ of trees as input. On input tree t, query Q evaluates to $[\![Q]\!]_1(t)$. The symbol $+$ in Figure 1 denotes list concatenation, by subtrees of t we refer to subtrees induced by a node of t and all of its descendants in t, $<^t_{doc}$ is the preorder depth-first left-to-right traversal order

[8]An equivalent alternative would be a separate infinite value domain, but this would only cause heavier notation.

[9]We leave concerns regarding normalization of whitespace aside here.

through tree t, χ^t is the binary axis relation χ on t, for instance, $\mathrm{Child}^t(u, v)$ is true iff node v is a child of node u in tree t, $root^t$ is the root node of tree t, lab^t_* is true on all nodes of t, and lab^t_a, for a a tag name, is true on those nodes of t labeled a. All *XQ* queries evaluate to lists of trees. Values can only be assigned to variables in for-expressions, which assure that variables always bind to single trees rather than lists. For the restricted syntax of Core XQuery, this semantics is (observationally) consistent with XQuery as currently undergoing standardization with the W3C [World Wide Web Consortium 2005].

In our definition of the syntax of Core XQuery, we have been economical with operators introduced. Since a condition is true iff it evaluates to a nonempty collection,

$$\begin{aligned}
\text{true} &:= \langle\text{nonempty}/\rangle \\
\phi \text{ or } \psi &:= \phi\ \psi \\
\phi \text{ and } \psi &:= \text{if } \phi \text{ then } \psi \\
\text{some } \$x \text{ in } \alpha \text{ satisfies } \phi &:= \text{for } \$x \text{ in } \alpha \text{ return } \phi \\
\$x = \langle a/\rangle &:= \text{some } \$y \text{ in } \langle a/\rangle \text{ satisfies } \$x = \$y \\
(\text{let } \$x := \langle a\rangle\alpha\langle/a\rangle)\ \beta &:= \text{for } \$x \text{ in } \langle a\rangle\alpha\langle/a\rangle \text{ return } \beta
\end{aligned}$$

Using deep equality, we can define negation,

$$\text{not } \phi := \big(\phi =_{deep} ()\big).$$

Conditions "every \$x in α satisfies ϕ" can be defined using "not" and "some".

The following result follows immediately from these definitions.

Proposition 3.1. *Let* $\mathbf{X}$ *be a set of operations and axes.*

—*Each* $XQ[$*true, and, or, some, let*$, \mathbf{X}]$ *query can be translated in LOGLIN into an equivalent* $XQ[\mathbf{X}]$ *query.*

—*Each* $XQ[=_{deep}, not, every, \mathbf{X}]$ *query can be translated in LOGLIN into an equivalent* $XQ[=_{deep}, \mathbf{X}]$ *query.*

Next, we provide mappings between Core XQuery (using only the child axis, since there is no feature corresponding to e.g. the descendant axis in monad algebra) and monad algebra on lists. These show the *equivalence of these languages up to representation issues*, but our main aim is to provide reductions for the study of the complexity of XQuery.

Translation from Core XQuery to $\mathcal{M}^{[]}_{\cup}$. We recursively map the data tree t to a complex value $C(t)$ as follows: Each tree node with label a and children subtrees $t_1, \ldots, t_n$ $(n \geq 0)$ is mapped to a tuple $\langle \text{label} : a, \text{children} : [C(t_1), \ldots, C(t_n)]\rangle$. Moreover, the function C' maps a list of trees $[t_1, \ldots, t_n]$ to the list-typed complex value $[C(t_1), \ldots, C(t_n)]$.

Modulo representation issues captured in the tree translation function C, there is an equivalent monad algebra query for each $XQ[=, \text{child}, \text{not}]$ query, for $=$ either $=_{deep}$ or $=_{atomic}$.

Lemma 3.2. *There is a mapping* $MA : XQ[=, \textit{child, not}] \rightarrow \mathcal{M}^{[]}_{\cup}[=, \textit{not}]$ *such that for each* $XQ[=, \textit{child, not}]$ *query* Q,

$$\begin{aligned}
MA &: XQ[=,\text{child},\text{not}] \rightarrow [\langle N:\text{varname}, V:\tau\rangle] \rightarrow [\tau'] \\
\\
MA(\alpha\ \beta) &:= MA(\alpha) \cup MA(\beta) \\
MA(\,()\,) &:= [\,] \\
MA(\langle a\rangle\alpha\langle/a\rangle) &:= \langle \text{label}:a, \text{children}: MA(\alpha)\rangle \circ \text{sng} \\
MA(\$x) &:= \sigma_{\text{N=\$x}} \circ \text{map}(\pi_V) \\
MA(\$x/*) &:= \sigma_{\text{N=\$x}} \circ \text{flatmap}(\pi_V \circ \pi_{\text{children}}) \\
MA(\$x/a) &:= \sigma_{\text{N=\$x}} \circ \text{flatmap}(\pi_V \circ \pi_{\text{children}} \circ \sigma_{\text{label}=a}) \\
MA(\text{for } \$x \text{ in } \alpha \text{ return } \beta) &:= \langle 1:\text{id}, 2: MA(\alpha)\rangle \circ \text{pairwith}_2 \circ \\
&\quad\ \ \text{flatmap}\big((\pi_1 \cup (\langle N:\$x, V:\pi_2\rangle \circ \text{sng})) \circ MA(\beta)\big) \\
MA(\text{if } \alpha \text{ then } \beta) &:= \langle 1:\text{id}, 2: MA(\alpha) \circ \text{true}\rangle \circ \text{pairwith}_2 \circ \\
&\quad\ \ \text{flatmap}(\pi_1 \circ MA(\beta)) \\
\\
MA(\text{not } \alpha) &:= MA(\alpha) \circ \text{map}(\langle\rangle) \circ \text{not} \\
MA(\$x = \$y) &:= \langle 1:\sigma_{N=\$x}, 2:\sigma_{N=\$y}\rangle \circ \text{pairwith}_1 \circ \\
&\quad\ \ \text{flatmap}(\text{pairwith}_2) \circ \sigma_{1.V=2.V}
\end{aligned}$$

Fig. 2. Mapping from $XQ[=,\text{child},\text{not}]$ to $\mathcal{M}^{[]}_{\cup}[=,\text{not}]$.

(*1*) *for any XML tree t,* $C'(\llbracket Q\rrbracket_1(t)) = MA(Q)\big([\langle N:\$ROOT, V:C(t)\rangle]\big)$*,*

(*2*) $MA(Q)$ *can be computed in space* $O(\log|Q|)$*, and*

(*3*) $|MA(Q)| = O(|Q|)$.

Proof. Consider the function MA of Figure 2. It is easy to verify by induction that MA satisfies the invariant that for all XQ expressions α and k-ary environments $\vec{e} = (t_1,\dots,t_k)$ such that the free variables of α are included in $\{x_1,\dots,x_k\}$,

$$C'(\llbracket\alpha\rrbracket_k(\vec{e})) = MA(\alpha)([\langle N:x_1, V:C(t_1)\rangle,\dots,\langle N:x_k, V:C(t_k)\rangle]).$$

We check this for the two most interesting cases, expressions $\$x$ (or $\$x/a$, or $\$x/*$) which access the tree associated with variable $\$x$ in the environment and for-expressions which extend the environment by a new variable.

Let $E = [\langle N:x_1, V:C(t_1)\rangle,\dots,\langle N:x_k, V:C(t_k)\rangle]$.

—We compute $MA(\$x_i)(E) = (\sigma_{N=\$x_i} \circ \text{map}(\pi_V))(E)$. We have $\sigma_{N=\$x_i}(E) = [\langle N:\$x_i, V:C(t_i)\rangle]$ and $\text{map}(\pi_V)([\langle N:\$x_i, V:C(t_i)\rangle]) = [C(t_i)] = C'([t_i])$. Since $\llbracket\$x_i\rrbracket_k(t_1,\dots,t_k) = [t_i]$, the induction hypothesis holds for expressions $\$x_i$.

—We compute $MA(\text{for } \$x_{k+1} \text{ in } \alpha \text{ return } \beta)(E)$. Let $\llbracket\alpha\rrbracket_k(t_1,\dots,t_k) = [t'_1,\dots,t'_l]$. By the induction hypothesis, $MA(\alpha)(E) = [C(t'_1),\dots,C(t'_l)]$. But then

$$(\langle 1:\text{id}, 2:MA(\alpha)\rangle \circ \text{pairwith}_2)(E) = [\langle 1:E, 2:C(t'_1)\rangle,\dots,\langle 1:E, 2:C(t'_l)\rangle].$$

If we apply $\text{flatmap}((\pi_1 \cup (\langle N:\$x_{k+1}, V:\pi_2\rangle \circ \text{sng})) \circ MA(\beta))$ to this we get

$$MA(\beta)(E \cup [\langle N:\$x_{k+1}, V:C(t'_1)\rangle]) \cup \dots \cup MA(\beta)(E \cup [\langle N:\$x_{k+1}, V:C(t'_l)\rangle]).$$

Applying the induction hypothesis l times yields

$$C'([\![\beta]\!]_{k+1}(t_1,\ldots,t_k,t'_1)) \cup \cdots \cup C'([\![\beta]\!]_{k+1}(t_1,\ldots,t_k,t'_l))$$

which in turn is equal to $C'([\![\text{for } \$x_{k+1} \text{ in } \alpha \text{ return } \beta]\!]_k(t_1,\ldots,t_k))$.

By definition, a query has one free variable ("\$ROOT"). Claim (1) of our theorem, which is the restriction of the induction hypothesis to $k = 1$, follows immediately. Regarding claims (2) and (3), it is easy to check by inspection of Figure 2 that MA can be computed in LOGSPACE and that the result is of linear size. □

For atomic equality, $\sigma_{1.V=2.V}$ in the definition of MA is to be implemented as $\sigma_{1.V.\text{label}=_{atomic}2.V.\text{label}}$. Note that on a $XQ[=,\text{child}]$ query Q, $MA(Q)$ is a $\mathcal{M}^{[]}_{\cup}[=]$ query.

Translation from $\mathcal{M}^{[]}_{\cup}$ to Core XQuery. Let T be the following canonical translation from complex values to trees:

$$\begin{aligned} T(\langle A_1 : v_1, A_2 : v_2\rangle) &= \langle tup\rangle\langle a_1\rangle T(v_1)\langle/a_1\rangle\langle a_2\rangle T(v_2)\langle/a_2\rangle\langle/tup\rangle \\ T([v_1,\ldots,v_n]) &= \langle list\rangle T(v_1)\ldots T(v_n)\langle/list\rangle \end{aligned}$$

Note that T is not the inverse of the mapping C that we introduced above to map from XML trees to complex values constructed from tuples and lists.

Let $\mathcal{M}^{[],\langle\cdot,\cdot\rangle}_{\cup}$ denote the monad algebra queries on lists and pairs (rather than on tuples of arbitrary arity). For both $=_{deep}$ and $=_{atomic}$, we have

Lemma 3.3. *There is a mapping $XQ : \mathcal{M}^{[]}_{\cup}[=] \to XQ[=, child]$ such that for each $\mathcal{M}^{[]}_{\cup}[=]$ query Q,*

(*1*) *for any complex value v, $T(Q(v)) = [\![XQ(Q)(\$ROOT)]\!]_1\big(T(v)\big)$,*

(*2*) *$XQ(Q)$ can be computed in space $O(\log|Q|)$, and*

(*3*) *If Q is a $\mathcal{M}^{[],\langle\cdot,\cdot\rangle}_{\cup}$ query, $|XQ(Q)| = O(|Q|)$.*

Proof. The proof of (1) is by induction, with (1) as the induction hypothesis, on the mapping XQ of Figure 3. Both in the figure and the proof, we will use expressions of the form "$\$x/\nu/\nu'$" as short syntax for "for $\$x'$ in $\$x/\nu$ return $\$x'/\nu'$". For example, for $XQ(A_i = A_j)(\$x)$, complex value v must be a tuple $\langle A_1 : v_1,\ldots,A_k : v_k\rangle$ and $T(v) = \langle\text{tup}\rangle\langle a_1\rangle T(v_1)\langle/a_1\rangle\ldots\langle a_k\rangle T(v_k)\langle/a_k\rangle\langle/\text{tup}\rangle$. But then $[\![\$x/a_i/*]\!](T(v)) = T(v_i)$ and $[\![\$x/a_j/*]\!](T(v)) = T(v_j)$. Thus

$$\begin{aligned} [\![XQ(A_i = A_j)(\$x)]\!]_1(T(v)) &= [\![\langle\text{list}\rangle\{\text{if (some } \$y \text{ in } \$x/a_i/* \text{ satisfies} \\ &\qquad\qquad \text{some } \$z \text{ in } \$x/a_j/* \text{ satisfies} \\ &\qquad\qquad (\$y = \$z)) \text{ then } \langle\text{tup}/\rangle\}\langle/\text{list}\rangle]\!]_1(T(v)) \\ &= \begin{cases} \langle\text{list}\rangle\langle\text{tup}/\rangle\langle/\text{list}\rangle & \ldots\ T(v_i) = T(v_j) \\ \langle\text{list}/\rangle & \ldots\ \text{otherwise} \end{cases} \\ &= T((A_i = A_j)(v)). \end{aligned}$$

The restriction of tuples to arity ≤ 2 in (3) is needed because $|XQ(\text{pairwith}_i)(\$x)|$ is linear in the arity of tuples. It is easy to verify that XQ satisfies (2) and (3). □

$$
\begin{array}{rcl}
XQ(\langle A_1 : f_1, \ldots A_k : f_k\rangle)(\$x) & := & \langle\text{tup}\rangle\langle a_1\rangle XQ(f_1)(\$x)\langle/a_1\rangle \ldots \\
 & & \qquad \langle a_k\rangle XQ(f_k)(\$x)\langle/a_k\rangle\langle/\text{tup}\rangle \\
XQ(\pi_i)(\$x) & := & \{\$x/a_i/*\} \\
XQ(\text{sng})(\$x) & := & \langle\text{list}\rangle\{\$x\}\langle/\text{list}\rangle \\
XQ(f \circ g)(\$x) & := & \{\text{for \$y in } XQ(f)(\$x) \text{ return } XQ(g)(\$y)\} \\
XQ(\text{map}(f))(\$x) & := & \langle\text{list}\rangle\{\text{for } \$y \text{ in } \$x/* \text{ return } XQ(f)(\$y)\}\langle/\text{list}\rangle \\
XQ(\text{id})(\$x) & := & \{\$x\} \\
XQ(\text{flatten})(\$x) & := & \langle\text{list}\rangle\{\$x/\text{list}/*\}\langle/\text{list}\rangle \\
XQ(\text{pairwith}_i)(\$x) & := & \langle\text{list}\rangle\{\text{for \$y in \$x/}a_i\text{/list/* return } \langle\text{tup}\rangle \\
 & & \qquad \langle a_1\rangle\{\$x/a_1/*\}\langle/a_1\rangle \ldots \langle a_{i-1}\rangle\{\$x/a_{i-1}/*\}\langle/a_{i-1}\rangle \\
 & & \qquad \langle a_i\rangle\{\$y\}\langle/a_i\rangle \\
 & & \qquad \langle a_{i+1}\rangle\{\$x/a_{i+1}/*\}\langle/a_{i+1}\rangle \ldots \langle a_k\rangle\{\$x/a_k/*\}\langle/a_k\rangle \\
 & & \langle/\text{tup}\rangle\}\langle/\text{list}\rangle \\
XQ(f \cup g)(\$x) & := & \langle\text{list}\rangle\{(XQ(f)(\$x))/*\}\{(XQ(g)(\$x))/*\}\langle/\text{list}\rangle \\
XQ(A_i = A_j)(\$x) & := & \langle\text{list}\rangle\{\text{if (some } \$y \text{ in } \$x/a_i/* \text{ satisfies} \\
 & & \qquad\qquad \text{some } \$z \text{ in } \$x/a_j/* \text{ satisfies} \\
 & & \qquad\qquad (\$y = \$z)) \text{ then } \langle\text{tup}/\rangle\}\langle/\text{list}\rangle \\
XQ(c)(\$x) & := & \begin{cases} \langle\text{list}/\rangle & \ldots\ c = [\,] \\ \langle\text{tup}/\rangle & \ldots\ c = \langle\rangle \\ \langle c/\rangle & \ldots\ \text{otherwise} \end{cases} \\
XQ(\text{true})(\$x) & := & \{\text{if \$x then } \langle\text{nonempty}/\rangle\}
\end{array}
$$

Fig. 3. Mapping from $\mathcal{M}_{\cup}^{[\,]}[=]$ to $XQ[=, \text{child}]$.

As mentioned before, many query languages for complex values that were developed earlier, such as nested relational algebra [Jaeschke and Schek 1982], complex value algebra without powerset [Abiteboul and Beeri 1995], and monad algebra [Tannen et al. 1992], share the same expressive power. Core XQuery is an interesting fragment of XQuery because it captures precisely this degree of expressiveness, which is commonly deemed "right" for nested, deeply structured data.

4. QUERY COMPLEXITY OF LANGUAGES WITH DEEP EQUALITY

Before we embark on our study of the query complexity of languages with deep equality, let us make an observation.

Proposition 4.1. *There are LOGLIN reductions that*

—*given a complex value v, compute an $\mathcal{M}_{\cup}$ ($\mathcal{M}_{\cup}^{\{\!|\,|\!\}}$, $\mathcal{M}_{\cup}^{[\,]}$) expression that evaluates to v on an arbitrary (e.g. empty) database.*

—*given an XML tree v, compute an XQ expression that evaluates to v on an arbitrary XML tree.*

That is, both monad algebra and Core XQuery have the power to construct arbitrary values from scratch. Since these languages are obviously closed under composition and all complexity classes we will consider for query complexity throughout this article are closed under LOGLIN-reductions, we may subsequently focus on query complexity; combined complexity is no harder.

4.1 Size Bounds on Values

It is possible to write queries in monad algebra (or equally in $\mathcal{M}_\cup^{\{\!|\,|\!\}}$ or $\mathcal{M}_\cup^{[\,]}$ and thus, by Lemma 3.3, in Core XQuery) that compute values of doubly exponential size.

Proposition 4.2. *There is an $\mathcal{M}_\cup$ query Q that computes a value of size $2^{2^{\Omega(|Q|)}}$.*

Proof. Consider the query Q

$$\phi_{\{0,1\}} \circ \underbrace{(\mathrm{id} \times \mathrm{id}) \circ \cdots \circ (\mathrm{id} \times \mathrm{id})}_{m \text{ times}}$$

where $\phi_{\{0,1\}} = (0 \circ \mathrm{sng}) \cup (1 \circ \mathrm{sng})$ computes the set $\{0, 1\}$ and m is linear in $|Q|$. Query Q computes the set of all nested pairs (=binary trees) of depth m with labels from $\{0, 1\}$ at the leaves. There are 2^{2^m} such nested pairs. $\square$

For the converse,

Proposition 4.3. *The values computable by $\mathcal{M}_\cup[=]$ queries are of size $2^{2^{O(n)}}$, where n is the size of the input (i.e., database and query).*

Proof. Let the function $C_f(n)$, for each $\mathcal{M}_\cup[=]$ expression f, be defined resp. bounded as follows: For constants, it is $O(1)$; for the operation id, it is $|n|$; for sng, it is $|n| + O(1)$; for flatten, σ, and π, it is $|n|$, for pair construction $\langle 1 : f, 2 : g\rangle$, it is $C_f(n) + C_g(n) + O(1)$; for union $f \cup g$, it is $C_f(n) + C_g(n)$; for pairwith, it is $n^2 + O(1)$; and finally, for $f \circ g$, it is $C_g(C_f(n))$.

It is easy to see that C_f provides us with an upper bound on the size of the value obtained by applying $\mathcal{M}_\cup[=]$ expression f on a value of size n.

For $n > 1$, pairwith is the locally costliest operation, so let us assume that Q consists of the composition of $|Q|$ operations with this cost as an upper bound. In particular, this will provide an overestimation of the size of the computed value because for $n > 1$, $C_f(n) + C_g(n) + O(1) < C_{\underbrace{\text{pairwith} \circ \cdots \circ \text{pairwith}}_{|f|+|g| \text{ times}}}(n) = (\cdots(((n^2 + O(1))^2 + O(1))^2 + O(1))\cdots)^2 + O(1)$. Now,

$$\begin{aligned} |[\![Q]\!](D)| \leq C_Q(|D|) &\leq (\cdots((\overbrace{|D|^2 + O(1))^2 + O(1))\cdots)^2 + O(1)}^{|Q| \text{ times}} \\ &\leq (\cdots(((|D| + O(|Q|))\overbrace{^2)^2)\cdots)^2}^{|Q| \text{ times}} \leq 2^{2^{O(|D|+|Q|)}} \end{aligned}$$

$\square$

Given an input value of size $2^{2^{O(n)}}$, each operation of $\mathcal{M}_\cup[=]$ can be evaluated on the input in time $2^{2^{O(n)}}$ on a random access machine. There are $|Q| \leq n$ operations, and $|Q| \cdot 2^{2^{O(n)}} = 2^{2^{O(n)}}$. Thus, COROLLARY 4.4. *$\mathcal{M}_\cup[=]$ is in 2ETIME w.r.t. combined complexity.*

4.2 A Space Bound

By Lemma 3.2 and Proposition 4.3, we cannot compute values of more than doubly exponential size in Core XQuery. Pointers into such values take only exponential space. It is not hard to design an algorithm which does not materialize intermediate results (trees) but uses pointers into such trees and counters to keep track of the current state of the computation and to recompute trees on demand, and which runs in exponential space.

THEOREM 4.5. *$XQ[=_{deep}$, child, descendant$]$ is in EXPSPACE w.r.t. query complexity.*

PROOF. Let *Symbol* be the set of opening and closing tags $\langle a \rangle, \langle /a \rangle$ for each label a of our labeling alphabet. We use a *list iterator* design pattern with methods getNext: $\rightarrow$ *Symbol* and atEnd: $\rightarrow$ *Boolean*. Such an iterator has an internal counter p initially set to (list position) 1; atEnd() is true if p is greater than the size of the list; if atEnd() is false, getNext() outputs the p-th element of the list and increments p by one; thus, getNext() can be used to iterate over the elements of the list and output them element by element. Given an iterator i, let $l(i)$ denote the list over which i iterates.

Consider the semantics definition of Figure 1. For each Core XQuery expression α with k free variables it defines a function $[\![\alpha]\!]_k$ that maps k-tuples $\vec{e}$ of trees to a list of trees, the semantics of expression α on environment $\vec{e}$.

For each expression α, let Iterator$[\![\alpha]\!]_k$ be a function that maps a k-tuple $\vec{j}$ of iterators with $l(j_m) = e_m$, for $1 \leq m \leq k$, to an iterator i such that $l(i) = [\![\alpha]\!]_k(\vec{e})$. Obviously it is possible to define such iterators: e.g. for getNext() we just compute $e_m = l(j_m)$, for each j_m, and store it in memory. Then we compute $[\![\alpha]\!]_k(\vec{e})$ and return its p-th symbol (the current position of the iterator).

This algorithm is not yet good because it takes too much space – we already know from Proposition 4.2 and Lemma 3.3 that trees produced by Core XQuery queries can be of doubly exponential size. However, we need to modify the direct functional implementation of our semantics definition of Figure 1 only moderately to obtain an iterator-based algorithm which does not need to explicitly store the subtrees $\vec{e}$ to compute $[\![\alpha]\!]_k(\vec{e})$ and return the p-th symbol. Instead, we can work directly using the iterators for $\vec{e}$ and never have to materialize large trees or lists.

This is rather straightforward; the most interesting case is probably the for-expressions, for which we can implement getNext() as follows.

```
*Symbol* Iterator$[\![$for $\$x_{k+1}$ in $\alpha$ return $\beta]\!]_k(\vec{e})$.getNext() **begin**
  *Integer* $p' := 1$;
  Iterator$[\![\alpha]\!]_k(\vec{e})$ $i$;

  **for** $m = 1$ **to** $i \rightarrow$count() **do begin**
    Iterator$[\![\beta]\!]_{k+1}(\vec{e}, i \rightarrow \text{get}(m))$ $j$;

    **while not** $j \rightarrow$atEnd() **do begin**
      *Symbol* $s := j \rightarrow$getNext();
      $p' := p' + 1$;
      **if** (this$\rightarrow p = p' - 1$) **then begin** $p := p + 1$; **return** $s$ **end**
    **end**
  **end**;
  **fail**
**end**
```

Here, given an iterator i over a list of trees represented by a well-formed string of opening and closing tags, we have assumed additional methods "count" and "get" on i such that count() returns the number of trees in the list $l(i)$ and get(m) returns an iterator over the m-th tree in that list. Both can be easily implemented using a counter that maintains the number of opening minus the number of closing tags seen left of the current position.

Now let us consider the space requirements of such an iterator-based evaluation technique. Observe first that there is again a doubly exponential upper bound on the size of Core XQuery results – this follows immediately from Lemma 3.2 and Proposition 4.3. But then, a position inside any such value – tree or list of trees – can be represented by a binary number of singly exponential length. For query Q, at any time, we need to store the state of no more than $O(|Q|)$ iterators in memory, one for each distinct variable, and each such state consists of only a constant number of exp-sized counters and of $O(|Q|)$ references to other iterators. Thus, our algorithm takes singly exponential space. $\Box$

Note that this EXPSPACE algorithm is quite robust and allows to add a number of XQuery features that we excluded from XQ, such as counting, (document position) arithmetics, and duplicate elimination.

Monad algebra with deep equality inherits the same upper bound.

THEOREM 4.6. *$\mathcal{M}_\cup[=_{deep}]$, $\mathcal{M}_\cup^{[\,]}[=_{deep}]$ and $\mathcal{M}_\cup^{\{\!|\,|\!\}}[=_{deep}, monus, unique]$ are in EXPSPACE w.r.t. query complexity.*

PROOF. The case of $\mathcal{M}_\cup^{[\,]}[=_{deep}]$ follows immediately from Lemma 3.3 and Theorem 4.5.

$\mathcal{M}_\cup[=_{deep}]$ and $\mathcal{M}_\cup^{\{\!|\,|\!\}}[=_{deep}, \text{monus}, \text{unique}]$ work analogously: We use the list-based algorithm for evaluating monad algebra on bags, and the only aspect that has to be modified is the checking of equality. Indeed, the remaining operations of $\mathcal{M}_\cup[=_{deep}]$ and $\mathcal{M}_\cup^{\{\!|\,|\!\}}[=_{deep}, \text{monus}, \text{unique}]$ are indifferent to orders in collection-typed values. When we want to check whether two bags identified by expressions e_1 and e_2 are equal, we can do this by checking whether for each member of $[\![e_1]\!]$ or $[\![e_2]\!]$, its multiplicity in $[\![e_1]\!]$ is the same as in $[\![e_2]\!]$. By previous arguments, a binary counter for counting members of a list requires only singly exponential space.

The operations "monus" and "unique" can be evaluated using similar counting techniques.

Deep set equality, for $\mathcal{M}_\cup[=_{deep}]$, can be implemented by modifying the equality check on bags to recursively verifying, for each element v of the two sets, whether at least one element equal to v occurs in the other set. $\Box$

5. QUERY COMPLEXITY OF LANGUAGES WITH ATOMIC EQUALITY

Next we consider the complexity of monad algebra and XQuery with atomic equality, both with and without negation. It is folklore that by extending $\mathcal{M}_\cup$ by equality on atomic values $=_{atomic}$, we still cannot express nonmonotone operations such as equality of sets or negation. We can safely generalize $=_{atomic}$ to equality of arbitrary complex values that do not include sets, $=_{mon}$, defined inductively as $=_{atomic}$ on atomic values and $v_1 =_{mon} w_1 \wedge \cdots \wedge v_k =_{mon} w_k$ on tuples $\langle v_1, \ldots, v_k\rangle$ and $\langle w_1, \ldots, w_k\rangle$. Of course this generalization does not improve upon the expressiveness of $\mathcal{M}_\cup[=_{atomic}]$.

Proposition 5.1. *$\mathcal{M}_\cup[=_{atomic}]$ captures $\mathcal{M}_\cup[=_{mon}]$.*

Proof. Of course, every $\mathcal{M}_\cup[=_{atomic}]$ query is also a $\mathcal{M}_\cup[=_{mon}]$ query. For the other direction, we can define $=_{mon}$ using $=_{atomic}$ given the type τ of the values to compare. Viewing each such tuple type as a ranked tree t, we simply define $(A =^\tau_{mon} B)$ as the conjunction (implemented as the Cartesian product) of the equality predicates $(A.\pi =_{atomic} B.\pi)$ for each attribute path π in t from the root to a leaf. For example, for type $\tau = \langle C : \langle D : \text{Dom}, E : \langle F : \text{Dom}, G : \text{Dom}\rangle\rangle, H : \text{Dom}\rangle$,

$$\begin{aligned}(A =^\tau_{mon} B) := (A.C.D =_{atomic} B.C.D) \times (A.C.E.F =_{atomic} B.C.E.F) \times \\ (A.C.E.G =_{atomic} B.C.E.G) \times (A.H =_{atomic} B.H).\end{aligned}$$

(By definition, these types must be constructed from tuples and atomic values.) □

5.1 Upper Bound Results

We now come to address the observation made in the Introduction that while monad algebra queries may compute complex values of doubly exponential size (Proposition 4.2), resolution proofs for nonrecursive logic programs are always of only singly exponential size. The latter observation is known to yield a NEXPTIME upper bound for nonrecursive logic programming [Dantsin and Voronkov 1997]. The proof of the following theorem shows that monad algebra with atomic value equality is in NEXPTIME. The proof idea is a refinement of the one in [Dantsin and Voronkov 1997] – and in fact, as shown in the electronic appendix, monad algebra can be efficiently reduced to nonrecursive logic programming.

Theorem 5.2. *$\mathcal{M}_\cup[=_{atomic}]$ is in NEXPTIME w.r.t. query complexity.*

Proof. Without loss of generality, we may assume that all monad algebra operations are unary. This requires only a slight change of notation when we use the union operation $\cup$: Rather than writing $f \cup g$, we write $\langle A : f, B : g\rangle \circ \cup$.

We employ nested paths that can be thought of as terms constructed from constants and a single binary function symbol f. A constant c is written as c as a path. Inductively, if t, t' are terms and p, p' are their respective representations as paths, then the term $f(t, t')$ is represented as a path as $p.p'$ if t is atomic and as $(p).p'$ otherwise. Left f-term children are considered Skolem functions generating new path labels. For example, the term $f(f(x, y), f(z, f(u, v)))$ will be written as $(x.y).z.u.v$, and is understood as a path $w.z.u.v$ where w is a label generated from and identified by $x.y$.

$$
\begin{aligned}
[\![\mathrm{id}]\!](P) &:= P \\
[\![c]\!](P) &:= \{m.c \mid m.p \in P\} \\
[\![\pi_A]\!](P) &:= \{m.p \mid m.A.p \in P\} \\
[\![\mathrm{sng}]\!](P) &:= \{m.1.p \mid m.p \in P\} \\
[\![\mathrm{flatten}]\!](P) &:= \{m.(i.j).p \mid m.i.j.p \in P\} \\
[\![A =_{atomic} B]\!](P) &:= \{m.1.\langle\rangle \mid m.A.p, m.B.p \in P\} \cup \{m.1 \mid m.A.p \in P\} \\
[\![\pi_A \cup \pi_B]\!](P) &:= \{m.(1.i).p \mid m.A.i.p \in P\} \cup \\
&\quad\ \ \{m.(2.i).p \mid m.B.i.p \in P\} \\
[\![\mathrm{pairwith}_{A_j}]\!](P) &:= \{m.i.A_j.p \mid m.A_j.i.p \in P\} \cup \\
&\quad\ \ \{m.i.A_k.p' \mid m.A_j.i.p, m.A_k.p' \in P \wedge j \neq k\} \\
[\![f \circ g]\!](P) &:= [\![g]\!]([\![f]\!](P)) \\
[\![\mathrm{map}(f)]\!](P) &:= [\![\mathrm{map\_e}]\!]([\![f]\!]([\![\mathrm{map\_b}]\!](P))) \\
[\![\langle A_1 : f_1, \dots, A_k : f_k\rangle]\!](P) &:= \{m.A_1.p \mid m.p \in [\![f_1]\!](P)\} \cup \cdots \cup \\
&\quad\ \ \{m.A_k.p \mid m.p \in [\![f_k]\!](P)\}
\end{aligned}
$$

Fig. 4. A path-based alternative semantics for monad algebra.

We view every complex value as a deterministic tree, i.e., a tree in which each node v is uniquely identified by the path of labels from the root to v. We are able to uniquely assign such labels – even the elements of an index set to the elements of a set value, as we are considering query complexity and construct every set value from scratch (see Proposition 4.1). Such a deterministic tree is of course fully described by the set of root-to-leaf paths occurring in it.

Figure 4 shows an alternative semantics of $\mathcal{M}_\cup[=_{atomic}]$ in terms of deterministic trees. Each query maps a deterministic tree given as a set of paths to a deterministic tree given as a set of paths. Here, P always denotes a set of paths, $p, p', p_1, \dots, p_k$ denote paths, and m, i, j denote indexes of set members. The symbol $\langle\rangle$ (denoting an empty tuple) is to be understood as a constant and a path of length one. By map_b and map_e ("map-begin" and "map-end", respectively), we refer to the following two operations:

$$
\begin{aligned}
[\![\mathrm{map\_b}]\!](P) &:= \{(m.i).p \mid m.i.p \in P\} \\
[\![\mathrm{map\_e}]\!](P) &:= \{m.i.p \mid (m.i).p \in P\}.
\end{aligned}
$$

Let the mappings U^τ be a family of functions that map deterministic trees (represented by sets of paths) to complex values of type τ as follows: $U^{\mathrm{Dom}}(\{c\}) = c$,

$$
U^{\{\tau\}}(\{i_1.v_{1,1}, \dots, i_1.v_{1,n_1}, \dots, i_m.v_{m,1}, \dots, i_m.v_{m,n_m}\}) := \\
\{U^\tau(\{v_{1,1}, \dots, v_{1,n_1}\}), \dots, U^\tau(\{v_{m,1}, \dots, v_{m,n_m}\})\},
$$

$$
U^{\langle A_1:\tau_1,\dots,A_k:\tau_k\rangle}(\{A_1.v_{1,1}, \dots, A_1.v_{1,n_1}, \dots, A_k.v_{k,1}, \dots, A_k.v_{k,n_k}\}) := \\
\langle A_1 : U^{\tau_1}(\{v_{1,1}, \dots, v_{1,n_1}\}), \dots, A_k : U^{\tau_k}(\{v_{k,1}, \dots, v_{k,n_k}\})\rangle.
$$

Claim: Given a $\mathcal{M}_\cup[=_{atomic}]$ expression $f : \tau \to \tau'$ and a set of paths P which represents a complex value of type $\{\tau\}$,

$$U^{\{\tau'\}}([\![f]\!](P)) = \mathrm{map}(f)(U^{\{\tau\}}(P)).$$

Proof of Claim: By induction. For f not higher-order, that is, not of the form "map(g)", "$g \circ h$", or "$\langle A_1 : f_1, \ldots, A_k : f_k\rangle$", this follows immediately from the definition of $[\![\cdot]\!]$. For example, for sng : $\tau \to \{\tau\}$, $U^{\{\{\tau\}\}}([\![\mathrm{sng}]\!](P)) = U^{\{\{\tau\}\}}(\{m.1.p \mid m.p \in P\}) = \{\{U^{\tau}(\{p \mid m.p \in P\})\} \mid \exists p'\ m.p' \in P\} = \mathrm{map}(\mathrm{sng})(U^{\{\tau\}}(P))$.

For $f = g \circ h$ with $g : \tau \to \tau'$ and $h : \tau' \to \tau''$, let $P' = [\![g]\!](P)$ and $P'' = [\![h]\!](P')$. Then by the induction hypothesis, $U^{\tau'}(P') = \mathrm{map}(g)(U^{\{\tau\}}(P))$ and $U^{\{\tau''\}}(P'') = \mathrm{map}(h)(U^{\tau'}(P'))$. Thus $\mathrm{map}(h)(\mathrm{map}(g)(U^{\{\tau\}}(P))) = \mathrm{map}(g \circ h)(U^{\{\tau\}}(P)) = U^{\{\tau''\}}(P'') = U^{\{\tau''\}}([\![g \circ h]\!](P))$.

For $f = \mathrm{map}(g)$ and $g : \tau \to \tau'$, let $P' = [\![\mathrm{map\_b}]\!](P)$, let $P'' = [\![g]\!](P')$, and let $P''' = [\![\mathrm{map\_e}]\!](P'')$. Then, $U^{\{\tau\}}(P') = \mathrm{flatten}(U^{\{\{\tau\}\}}(P))$ and $(m.i) \mapsto (m, i)$ is a bijection: We can "undo" map_b using map_e. By the induction hypothesis, $U^{\{\tau'\}}(P'') = \mathrm{map}(g)(U^{\{\tau\}}(P'))$. Since $\mathrm{flatten} \circ \mathrm{map}(g) \equiv \mathrm{map}(\mathrm{map}(g)) \circ \mathrm{flatten}$ and P''' is obtained by nesting P'' using θ, $U^{\{\{\tau'\}\}}(P''') = \mathrm{map}(\mathrm{map}(g))(U^{\{\{\tau\}\}}(P))$.

For $f = \langle A_1 : f_1, \ldots, A_k : f_k\rangle : \tau \to \langle A_1 : \tau_1, \ldots, A_k : \tau_k\rangle$ with $f_l : \tau \to \tau_l$ ($1 \leq l \leq k$), assume the induction hypothesis holds for the f_l. For each m, each of the sets $\{m.A_l.p \mid m.p \in [\![f_l]\!](P)\}$ of the definition of $[\![\langle A_1 : f_1, \ldots, A_k : f_k\rangle]\!]$ contributes the paths for one attribute value A_l of the tuple to be constructed for the element with index m of $U^{\{\tau\}}(P)$. By a simple regrouping of the terms in these sets, we can verify that our hypothesis holds for f.

This establishes a formal connection between the semantics of monad algebra given in Section 2.2 and the one given by $[\![\cdot]\!]$. In particular, since each value v of type τ can also be read as a monad algebra expression $v : (\cdot) \to \tau$ which constructs it, $U^{\{\tau\}}([\![v]\!](\{1.\langle\rangle\})) = \mathrm{map}(v)(\{\langle\rangle\}) = \{v\}$, and for $Q : \tau \to \tau'$,

$$U^{\{\tau'\}}([\![v \circ Q]\!](\{1.\langle\rangle\})) = \mathrm{map}(v \circ Q)(\{\langle\rangle\}) = v \circ Q \circ \mathrm{sng} = (Q \circ \mathrm{sng})(v).$$

We have introduced map_b and map_e as shortcuts for studying the map operation. By definition,

$$[\![\mathrm{map\_b} \circ f \circ \mathrm{map\_e}]\!](P) = [\![\mathrm{map}(f)]\!](P)$$

for path-sets P. From now on, we assume that queries use map_b and map_e rather than map; thus we rewrite queries using this equivalence in the beginning.

An example demonstrating the construction of the deterministic tree for

$$\langle A : \{1, 2\}, B : \{2, 3\}\rangle \circ \mathrm{pairwith}_A \circ$$
$$\mathrm{map}(\mathrm{pairwith}_B \circ \mathrm{map}(A =_{atomic} B)) \circ \mathrm{flatten} \circ \mathrm{flatten}$$

is shown in Figure 5.

Now we can exploit the very restricted structure of the definition of $[\![\cdot]\!]$ to get an evaluation algorithm for Boolean queries that runs in NEXPTIME. Of course, a Boolean query is a set-typed query that is taken as true iff its result is nonempty. Thus, for evaluating query Q on input value v, we only need to guess a path p and check that it is in $[\![v \circ Q]\!](\{1.\langle\rangle\})$. We can do this recursively: To check whether path p is in $[\![f]\!]$, we need to guess and check *at most two* paths for subexpressions. Two paths need to be guessed and checked only for "pairwith" and $=_{atomic}$ – see

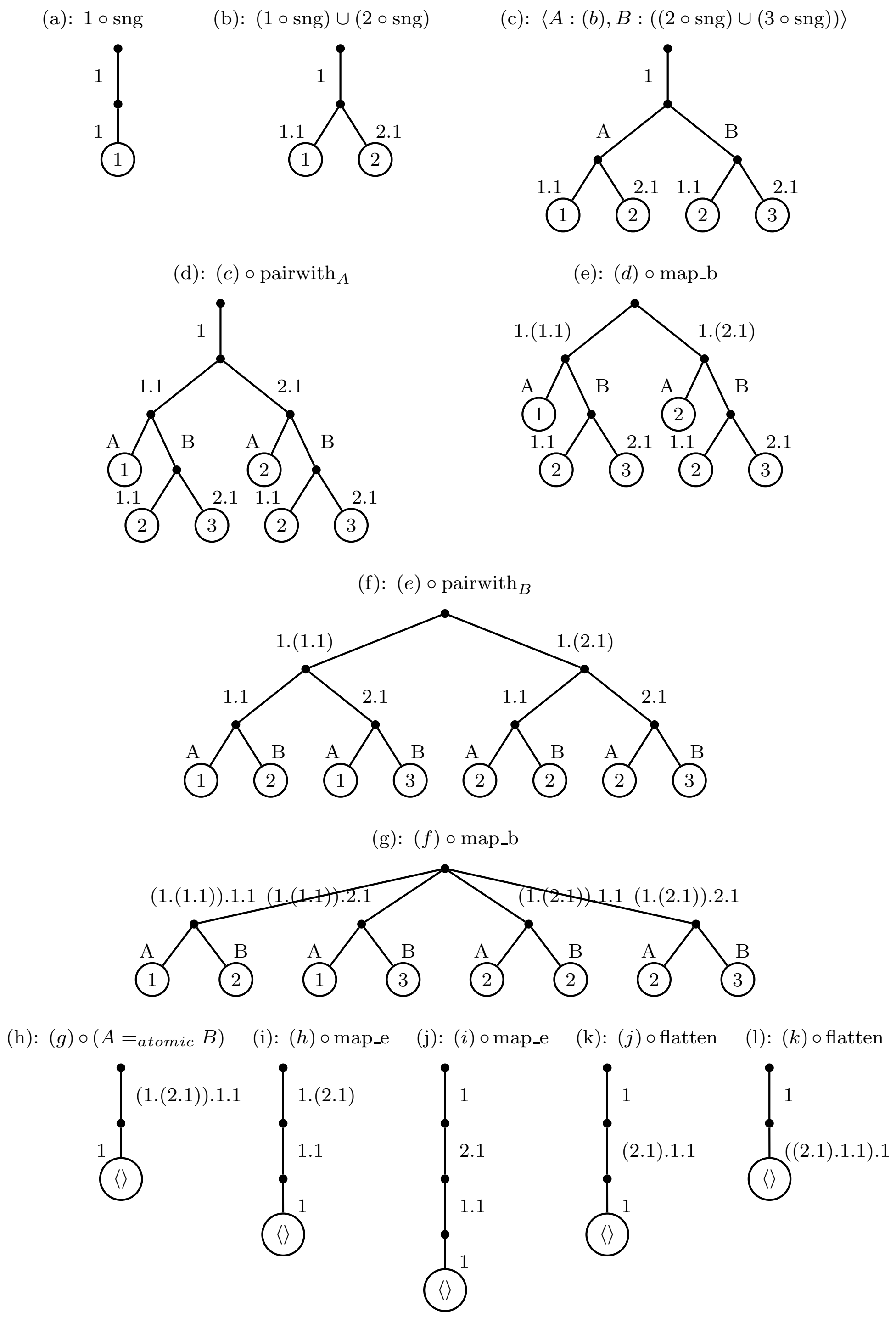


Fig. 5. Construction of deterministic tree for $\langle A : \{1, 2\}, B : \{2, 3\}\rangle \circ \text{pairwith}_A \circ \text{map}(\text{pairwith}_B \circ \text{map}(A =_{atomic} B)) \circ \text{flatten} \circ \text{flatten}$.

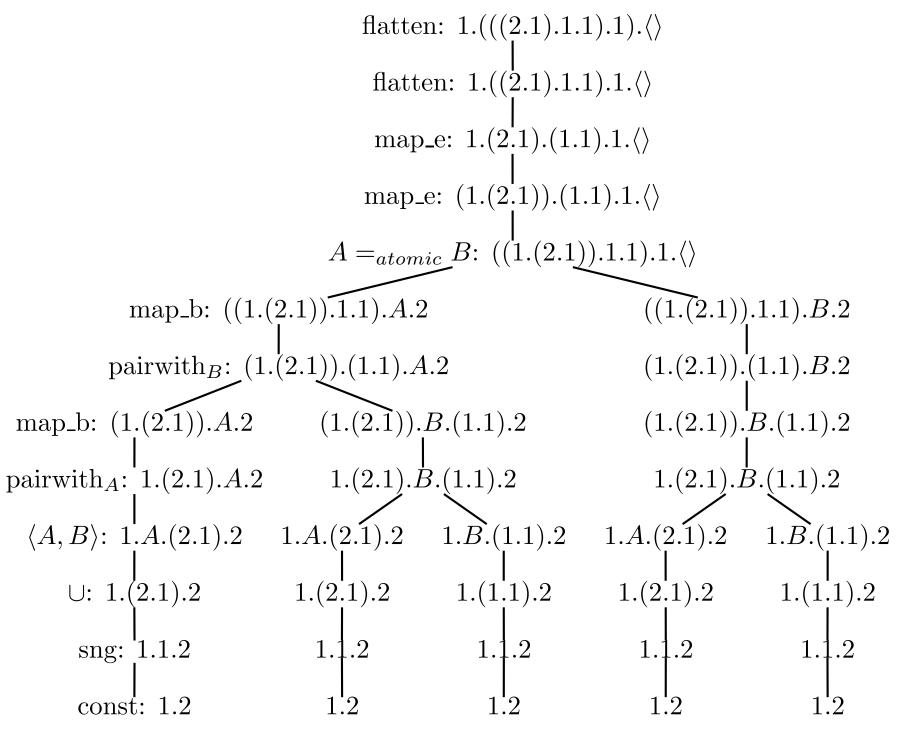


Fig. 6. Proof tree for query $\langle A:\{1,2\}, B:\{2,3\}\rangle \circ \text{pairwith}_A \circ \text{map}(\text{pairwith}_B \circ \text{map}(A =_{atomic} B)) \circ \text{flatten} \circ \text{flatten}$. The operation carried out at some node can be looked up at the node of the same depth in the leftmost path of the proof tree.

the definition of $[\![\cdot]\!]$ – and thus only in these two cases the computation branches out. For example, to verify that path $m.i.B.p'$ is in $[\![f \circ \text{pairwith}_A]\!](P)$, we have to guess a path p and check that $m.i.A.p, m.B.p' \in [\![f]\!](P)$. The remaining paths of $[\![f]\!](P)$ do not need to be computed. The "proof tree" for path $1.\langle\rangle$ thus computed has branching factor two and depth $O(|v|+|Q|)$. Each path grows by concatenation along a path of the proof tree. No copies of paths are concatenated, thus paths have polynomial size and the entire nondeterministic algorithm takes only exponential time. To illustrate this, Figure 6 shows the proof tree for the running example from above. □

THEOREM 5.3. *$\mathcal{M}_\cup[=_{atomic}, not]$ is in $TA[2^{O(n)}, O(n)]$ w.r.t. query complexity.*

PROOF. The proof is by a straightforward generalization of the previous proof. Consider a nondeterministic Turing machine implementation of the recursive algorithm for guessing and verifying that $[\![v \circ Q]\!](\{1.\langle\rangle\})$ is nonempty which was described in the proof of Theorem 5.2. To support the "not" operation as well, we just need to be able to check whether a given path $m.1.\langle\rangle$ is in $[\![f \circ \text{not}]\!](P)$. We do this by a universal computation – now of course on an alternating Turing machine – that verifies that for no i and p, the path $m.i.p$ is in $[\![f]\!](P)$. (That is, we verify that by making the assumption that $m.i.p$ is in $[\![f]\!](P)$, all possible computation paths reject.) Since there can be only linearly many occurrences of "not" in the input query, this can be done by an alternating Turing machine in exponential time with linearly many alternations. □

Our upper bounds are inherited by the other language dialects,

PROPOSITION 5.4. *With respect to query complexity, the languages*

(*1*) $\mathcal{M}_{\cup}^{\{\!|\,|\!\}}[=_{atomic}]$, $\mathcal{M}_{\cup}^{[\,]}[=_{atomic}]$, *and* $XQ[=_{atomic}, child]$ *are in NEXPTIME;*

(*2*) $\mathcal{M}_{\cup}^{\{\!|\,|\!\}}[=_{atomic}, not]$, $\mathcal{M}_{\cup}^{[\,]}[=_{atomic}, not]$, *and* $XQ[=_{atomic}, child, not]$ *are in* $TA[2^{O(n)}, O(n)]$ *under LOGLIN-reductions.*

PROOF. Regarding monad algebra on lists and bags, the proofs of Theorems 5.2 and 5.3 work without modifications. Actually, our encoding using deterministic trees treats collections as lists, and thus preserves both order and multiplicities of members. If only equality on atomic values is available, however, we cannot distinguish between sets, lists, and bags in queries.

The bounds on Core XQuery follow from those on monad algebra on lists established here and in Lemma 3.2. □

The complexity classes in which our XQuery evaluation problems reside are large enough that minor extensions, such as also supporting the descendant axis, do not matter.[10]

THEOREM 5.5. *With respect to combined complexity,*

—$XQ[=_{atomic}, child, descendant, not]$ *is in* $TA[2^{O(n)}, O(n)]$, *and*

—$XQ[=_{atomic}, child, descendant]$ *is in NEXPTIME.*

PROOF. (1) We first extend $\mathcal{M}_{\cup}^{[\,]}$ by a new operation "descmap" that, on a value $C(t)$ representing a tree t (using the representation function C of Section 3), computes the list of values $C(t')$ corresponding to the subtrees t' of t in document order. It is now easy to modify the LOGLIN-reduction of Lemma 3.2 to map any $XQ[=_{atomic}$, child, descendant] query to a corresponding $\mathcal{M}_{\cup}^{[\,]}[=_{atomic}$,descmap] query. Finally, we modify the algorithm of the proof of Theorem 5.2 to process queries involving descmap. Guessing a descendant of the root node of tree t, that is, guessing a subtree of t, simply requires to guess a prefix of a path in the deterministic tree representation of $C(t)$. This is easy, and the entire algorithm remains in NEXPTIME.

For (2), the proof is the same except that we use the algorithm of the proof of Theorem 5.3 and obtain a $TA[2^{O(n)}, O(n)]$ upper bound. □

5.2 Lower Bounds

In this section we establish lower bounds matching the upper bounds of Theorems 5.2 and 5.3. We first prove our results again for monad algebra on sets and then carry them over to the other languages. We start with the monotone fragment.

[10]It is folklore that computing the nodes of a tree reachable from a given node via any given axis can be done in LOGSPACE. For the descendant axis, this problem is LOGSPACE-complete [Cook and McKenzie 1987], cf. also [Gottlob et al. 2005].

THEOREM 5.6. *$\mathcal{M}_\cup[=_{atomic}]$ is NEXPTIME-hard w.r.t. query complexity.*

PROOF. The proof is by a LOGSPACE-reduction from NEXPTIME Turing machine acceptance. There are two main difficulties that we face in this reduction: We have to (i) deal with Turing machine tapes and configurations of exponential size and have to (ii) model the accepting computations of the Turing machine of exponential length succinctly – the $\mathcal{M}_\cup[=_{atomic}]$ query that must achieve this has to be computable in LOGSPACE and thus must be of polynomial size in the size of the input on the tape. Regarding (i) we basically reuse the query from the proof of Proposition 4.2 (used there to show that monad algebra queries can compute values of doubly exponential size) to generate a set of exponentially-sized nested pairs that contains encodings of all possible tape configurations. Nested pairs allow for an ordered representation whose elements can be uniquely addressed by short queries. The technically most involved part of the proof builds a query which defines the successor relation over these configurations. Difficulty (ii) is addressed using the reachability method of Savitch's theorem (cf. e.g. [Papadimitriou 1994]).

Let $M = (Q^M, q_0^M, \delta^M, F^M)$ be a nondeterministic Turing machine (NTM) that runs in time $2^{n^{O(1)}}$ on inputs of size n. We simulate the computation of M in $\mathcal{M}_\cup[=_{atomic}]$. Each run of M is a sequence of configurations of length $2^{K(n)}$, for a suitable k and $K(n) = n^k$. (We may assume w.l.o.g. that terminating computation paths of M remain in a final state until time $2^{K(n)}$ by appropriate design of M.) Each configuration of M consists of a read/write tape, a current state, and a position marker on the tape. Of course every $2^{K(n)}$ time NTM computation uses tape space bounded by $2^{K(n)}$.

Modeling configurations.

—Each tape of a configuration is modeled as a tuple of arity $2^{K(n)}$ (or more precisely, nested pairs of nesting depth $K(n)$) of tape symbols.
Let $\Sigma = \{s_1, \ldots, s_c\}$ be the (fixed) tape alphabet of M. Rather than representing the current position of the read/write head on the tape separately from the tape, we will assume a *valid tape* over extended tape alphabet $\Sigma' = \Sigma \cup \{\triangleright s \triangleleft \mid s \in \Sigma\}$ to contain a single symbol $\triangleright s \triangleleft$ (with $s \in \Sigma$) on the tape that indicates that this tape position stores symbol s and is the current position of the read/write head. We can compute the set of all $(2 \cdot c)^{2^{K(n)}}$ such tapes in $\mathcal{M}_\cup$ as

$$\mathit{Tapes} := \phi_{\Sigma'} \circ \underbrace{(\mathrm{id} \times \mathrm{id}) \circ \cdots \circ (\mathrm{id} \times \mathrm{id})}_{K(n) \text{ times}}$$

where $\phi_{\Sigma'}$ is an appropriate $\mathcal{M}_\cup$ expression that computes Σ'.[11]
As a result of this construction, some elements of set *Tapes* do not correspond to valid Turing tapes because they contain either zero or more than two markers indicating the current position of the read/write head on the tape. We will deal with this later.

—A superset of all possible configurations is

$$\mathit{Configs} := (\mathit{Tapes} \times Q^M) \circ \mathrm{map}(\langle t : \pi_1, q : \pi_2 \rangle).$$

[11]I.e., $\phi_{\Sigma'} := s_1 \circ \mathrm{sng} \cup \cdots \cup s_c \circ \mathrm{sng} \cup \triangleright s_1 \triangleleft \circ \mathrm{sng} \cup \cdots \cup \triangleright s_c \triangleleft \circ \mathrm{sng}$.

—The start configuration, consisting of the input tape, the start state, and the position marker at position 0 of the tape is obtained as follows.
We compute the start tape as the input x, with $|x| = n$, padded with $(2^{K(n)} - n)$ #-symbols (denoting unused tape space) and with the first position marked, but in our nested pairs representation.
Let query ϕ_x define the nested pair of depth $\lceil \log_2 n \rceil$ representing x padded by $(2^{\lceil \log_2 n \rceil} - n)$ #-symbols, and with the first position marked. (This is easy to compute in LOGSPACE.) E.g., for input $x = 01101$, the value computed[12] is

$$\langle\langle\langle \triangleright 0 \triangleleft, 1\rangle, \langle 1, 0\rangle\rangle, \langle\langle 1, \#\rangle, \langle \#, \#\rangle\rangle\rangle.$$

The start tape is

$$\phi_{start} := \langle 1 : \phi_x, 2 : \phi_{empty}\rangle \circ \underbrace{\phi_{pad} \circ \cdots \circ \phi_{pad}}_{K(n) - \lceil \log_2 n \rceil - 1 \text{ times}}$$

with

$$\phi_{pad} = \langle 1 : \text{id}, 2 : \langle 1 : \pi_2, 2 : \pi_2\rangle\rangle \quad \text{and} \quad \phi_{empty} := \# \circ \underbrace{\langle \text{id}, \text{id}\rangle \circ \langle \text{id}, \text{id}\rangle \circ \cdots \circ \langle \text{id}, \text{id}\rangle}_{\lceil \log_2 n \rceil \text{ times}}.$$

This takes the value computed by ϕ_x – which contains the input and some padding up to $2^{\lceil \log_2 n \rceil}$ symbols, pairs it with a sequence of #-symbols of the same length (computed by ϕ_{empty}), and then iteratively doubles the length of the tape by appending two copies of the second half of the already computed tape (because the second half consists exclusively of #-symbols). By this trick, there is a *fixed* expression ϕ_{pad} that we can compose our query with to double the length of the value produced.
The start configuration is

$$C_{start} := \langle t : \phi_{start}, q : q_0^M\rangle.$$

Observe that C_{start} is a valid configuration with precisely one tape head position marker.

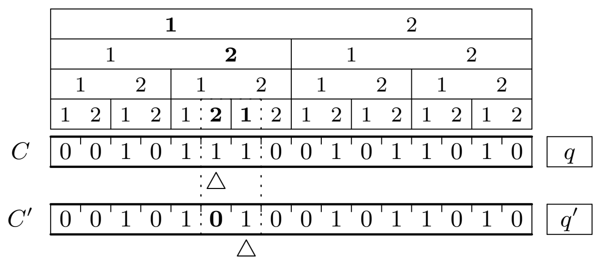
—The accepting configurations are those configurations in which the state is an element of the set $F^M = \{f_1, \ldots, f_{|F^M|}\}$ of accepting states of M:

$$\mathit{AcceptingConfigs} := \mathit{Configs} \circ (\sigma_{q =_{atomic} f_1} \cup \cdots \cup \sigma_{q =_{atomic} f_{|F^M|}}).$$

—In the following, we test equality of nested pairs (tape segments) and configurations of exponential size. We can define an equality test $=_{mon}$ of linear size on tapes and tape segments using only $=_{atomic}$ inductively as follows. On values of type Dom, $=_{mon}$ is $=_{atomic}$. Otherwise, on pairs $\langle 1 : \tau_1, 2 : \tau_2\rangle$,

$$(A =_{mon} B) := \big((\pi_A \circ \phi) \times (\pi_B \circ \phi)\big) \circ \sigma_{1.T =_{atomic} 2.T} \circ \sigma_{1.V =_{mon} 2.V} \circ \\ (\text{id} \times \text{id}) \circ \sigma_{1.1.T =_{atomic} \text{“1”}} \circ \sigma_{2.1.T =_{atomic} \text{“2”}} \circ \text{map}(\langle\rangle)$$

[12]It may be advisable to have a special symbol indicating the left end of the tape on its leftmost position to help the machine avoid running out of bounds. We assume such a symbol part of the input, rather than of our construction.



$$(q', 0, +1) \in \delta^M(q, 1)$$

Fig. 7. Zooming into the tapes to find a valid tape change resulting from a computation step of M.

where $\phi := \big(\langle T : 1, V : \pi_1\rangle \circ \mathrm{sng} \cup \langle T : 2, V : \pi_2\rangle \circ \mathrm{sng}\big)$. For configurations C, C',

$$(C =_{mon} C') \Leftrightarrow (C.t =_{mon} C'.t \wedge C.q =_{atomic} C'.q).$$

—Next we define an $\mathcal{M}_\cup[=_{atomic}]$ expression ϕ_{succ} that computes the pairs of configurations $\langle C, C'\rangle$ such that C' is a possible successor of C, i.e., computable using the transition relation δ^M of M in one step.

Here the exponential size of the configurations is a problem; ϕ_{succ} has to be chosen carefully in order not to be of exponential size. We achieve this as follows. We start with the Cartesian product of *Configs* – all pairs of configurations, even many that are invalid because they have zero or more than two head markers. For each pair, we make working copies w, w' of the tapes. We achieve this by the $\mathcal{M}_\cup$ expression

$$\phi_{prepare-succ} := \mathit{Configs} \circ (\mathrm{id} \times \mathrm{id}) \circ \mathrm{map}(\langle s : \mathrm{id}, w : \pi_{C.t}, w' : \pi_{C'.t}\rangle).$$

For w' to be a possible successor of w, the two tapes may differ at at most two consecutive tape positions (if the tape head moved, otherwise they may only differ at at most one position), and these positions must contain the read/write head position marker. We synchronously "zoom into" the working copies to find these two positions using the following three rules:

(1) If $w.2 = w'.2$ (i.e., the second halves of the tapes are equal), replace w by $w.1$ and w' by $w'.1$.

(2) If $w.1 = w'.1$ (i.e., the first halves of the tapes are equal), replace w by $w.2$ and w' by $w'.2$.

(3) If $w.1.1 = w'.1.1$ and $w.2.2 = w'.2.2$ (i.e., the first and last quarters of the tapes are equal) replace w by newly constructed pair $\langle 1 : w.1.2, 2 : w.2.1\rangle$ and w' by $\langle 1 : w'.1.2, 2 : w'.2.1\rangle$ (that is, by the second and third quarters).

All three cases may apply at the same time because the tapes of two valid configurations C, C', where C' is a successor of C, can be equal. An example of iterative zooming is shown in Figure 7. There, we look at a nested pair term of depth four (covering a tape of length 16) and the tape change occurs at positions 5 and 6 (if we count starting at 0). We first zoom into the left (using Rule 1) and from there into the right half (using Rule 2). Now both halves differ, but the first and fourth quarter do not, so we can use Rule 3 to zoom down to the

differing tape positions 6 and 7. In general, we obtain a tape sequence of length two by zooming into a tape (which is of length $2^{K(n)}$) $K(n) - 1$ times.
In our encoding in $\mathcal{M}_{\cup}[=_{atomic}]$, we compute the union of all triples (s, w, w') such that s is a pair of configurations with tapes $t = uwv$ and $t = uw'v$ and w and w' are of length 2 (i.e., w and w' are – if any – the only corresponding sequences in t, t' that differ). Now we have to make sure that w and w' contain the position marker.
This can be expressed in $\mathcal{M}_{\cup}[=_{atomic}]$ as follows:

$$\phi_{witness-succ} := \phi_{prepare-succ} \circ \underbrace{\phi_{zoom-in} \circ \cdots \circ \phi_{zoom-in}}_{K(n)-1\,\text{times}} \circ \phi_{marker}$$

where[13]

$$\begin{aligned}
\phi_{zoom-in} &:= (\sigma_{12\triangleright 34\triangleleft} \circ \pi_{12\triangleright 34\triangleleft} \cup \sigma_{\triangleright 12\triangleleft 34} \circ \pi_{\triangleright 12\triangleleft 34} \cup \sigma_{1\triangleright 23\triangleleft 4} \circ \pi_{1\triangleright 23\triangleleft 4}) \\
\sigma_{12\triangleright 34\triangleleft} &:= \sigma_{w.1=_{mon}w'.1} \\
\pi_{12\triangleright 34\triangleleft} &:= \text{map}(\langle s : \pi_s, w : \pi_{w.2}, w' : \pi_{w'.2}\rangle) \\
\sigma_{\triangleright 12\triangleleft 34} &:= \sigma_{w.2=_{mon}w'.2} \\
\pi_{\triangleright 12\triangleleft 34} &:= \text{map}(\langle s : \pi_s, w : \pi_{w.1}, w' : \pi_{w'.1}\rangle) \\
\sigma_{1\triangleright 23\triangleleft 4} &:= \sigma_{w.1.1=_{mon}w'.1.1} \circ \sigma_{w.2.2=_{mon}w'.2.2} \\
\pi_{1\triangleright 23\triangleleft 4} &:= \text{map}\big(\langle s : \pi_s, w : \pi_w \circ \langle 1 : \pi_{1.2}, 2 : \pi_{2.1}\rangle, w' : \pi_{w'} \circ \langle 1 : \pi_{1.2}, 2 : \pi_{2.1}\rangle\rangle\big)
\end{aligned}$$

and ϕ_{marker} selects those tuples for which w, w' are two tapes of length two that contain the read/write head marker:

$$\begin{aligned}
\phi_{marker} := (&\sigma_{w.1=_{atomic}\triangleright s_1 \triangleleft} \cup \cdots \cup \sigma_{w.1=_{atomic}\triangleright s_c \triangleleft} \cup \\
&\qquad \sigma_{w.2=_{atomic}\triangleright s_1 \triangleleft} \cup \cdots \cup \sigma_{w.2=_{atomic}\triangleright s_c \triangleleft} \cup).
\end{aligned}$$

Now, for each $\langle s : X, w : Y, w' : Z\rangle \in [\![\phi_{witness-succ}]\!]$, either $C = C'$ or Y, Z are precisely the at most two adjacent positions of the tapes of C and C' that can differ if C' is to be a successor of C. We encode the valid successors with respect to transition relation δ_M by a union of expressions that amount to selecting every pair of tapes that matches one of the transition rules encoded in δ^M:

$$\phi_{succ} := \phi_{witness-succ} \circ (\sigma_{\gamma_1} \cup \cdots \cup \sigma_{\gamma_m}) \circ \text{map}(\pi_s)$$

For instance, if $(q', b, +1) \in \delta(q, a)$, one σ_{γ_i} is to select the triples $\langle s : \langle S : \langle t : u \triangleright a \triangleleft sv, q : q\rangle, S' : \langle t : ub \triangleright s \triangleleft v, q : q'\rangle\rangle, w : \triangleright a \triangleleft s, w' : b \triangleright s \triangleleft\rangle \in [\![\phi_{witness-succ}]\!]$. (Details are omitted for lack of space, but it is important to note that the values that we are dealing with are atomic, so we only need equality on atomic values.) As for *Configs*, ϕ_{succ} contains pairs $\langle C, C'\rangle$ of invalid configurations. However, whenever C is a valid configuration, C' is indeed a possible successor configuration on M. It follows by induction that, starting from valid configuration C_{start}, we will only reach valid configurations via the successor relation ϕ_{succ}.

Modeling computations. Now we are ready to model accepting computations of M. Here the problem is the possibly exponential running time. We use a simple

[13]Here and later, $\pi_{A_1.\cdots.A_m} := \pi_{A_1} \circ \cdots \circ \pi_{A_m}$.

recursive divide-and-conquer approach in the spirit of the usual proof of Savitch's theorem (cf. e.g. [Papadimitriou 1994]). Let ψ_i be the pairs of configurations $\langle C, C' \rangle$ such that C is reachable from C' in 2^i steps. We define ψ_i as

$$\begin{aligned}\psi_0 &:= \phi_{succ}\\ \psi_{i+1} &:= \psi_i \circ (\mathrm{id} \times \mathrm{id}) \circ \sigma_{1.C'=2.C} \circ \mathrm{map}(\langle C : \pi_{1.C}, C' : \pi_{2.C'} \rangle)\end{aligned}$$

Note that the definition of ψ_{i+1} uses ψ_i only once, thus the formula remains computable in LOGSPACE.

There is an accepting computation path of length $2^{K(n)}$ iff there is a pair $\langle C, C' \rangle$ in $\psi_{K(n)}$ such that $C = C_{start}$ and the state of C' is in F^M. We can phrase this as

$$\begin{aligned}\phi_{accept} := \Big((\langle 1 : C_{start}, 2 : \psi_{K(n)} \rangle \circ \mathrm{pairwith}_2 \circ \sigma_{1 =_{mon} 2.C} \circ \\ \mathrm{map}(\pi_{2.C'})) \times \mathit{AcceptingConfigs} \Big) \circ \mathrm{map}([1 =_{mon} 2]) \circ \mathrm{flatten}.\end{aligned}$$

(Again we only employ equality on configurations.)

It is not difficult to see that the $\mathcal{M}_{\cup}[=_{atomic}]$ query ϕ_{accept} constructed is of polynomial size – a detailed discussion can be found below in the proof of Lemma 5.7 – and can be computed in LOGSPACE. The entire problem is formulated as the query (e.g., the input x is constructed from constants and pairs) and ϕ_{accept} does not make use of an input value. Thus we have shown that $\mathcal{M}_{\cup}[=_{atomic}]$ is NEXPTIME-hard with respect to query complexity (i.e., for a fixed database). □

LEMMA 5.7. *For the construction of the proof of Theorem 5.6, (a)* $|\phi_{accept}| = O(K(n))^2$*. (b) If* $=_{mon}$ *is available as a built-in,* ϕ_{accept} *can be defined such that* $|\phi_{accept}| = O(K(n))$*.*

PROOF. Part (a) of the lemma can be easily verified by inspection of the proof of Theorem 5.6:

$$\begin{aligned}|\mathit{Configs}| &= O\big(K(n)\big),\\ |C_{start}| &= O\big(K(n)\big),\\ |\mathit{AcceptingConfigs}| &= |\mathit{Configs}| + O(1) = O\big(K(n)\big),\\ | =_{mon} | &= O\big(K(n)\big),\\ |\phi_{prepare-succ}| &= O(|\mathit{Configs}|),\\ |\phi_{zoom-in}| &= O(| =_{mon} |),\\ |\phi_{witness-succ}| &= O(|\mathit{Configs}|) + O(|\phi_{zoom-in}| \cdot K(n)) = O\big(K(n)^2\big),\\ |\phi_{succ}| &= |\phi_{witness-succ}| + O(1) = O\big(K(n)^2\big),\\ |\psi_{K(n)}| &= |\phi_{succ}| + O\big(K(n) \cdot | =_{mon} |\big) = O\big(K(n)^2\big),\\ |\phi_{accept}| &= |C_{start}| + |\psi_{K(n)}| + |\mathit{AcceptingConfigs}| = O\big(K(n)^2\big)\end{aligned}$$

From this it is also clear that (b) if we use built-in $=_{mon}$ operation rather than our defined monotone equality operation, then $|\phi_{accept}| = O\big(K(n)\big)$. □

COROLLARY 5.8. $\mathcal{M}_{\cup}[=_{mon}]$ *is NETIME-hard under LOGLIN-reductions (with respect to query complexity).*

Theorem 5.9. *$\mathcal{M}_\cup[=_{mon}, not]$ is $TA[2^{O(n)}, O(n)]$-hard under LOGLIN-reductions (with respect to query complexity).*

Proof. The proof is by a LOGLIN-reduction from $\mathrm{TA}[2^{O(n)}, O(n)]$ Turing machine acceptance. Let $M = (Q_\exists^M, Q_\forall^M, q_0^M, \delta^M, F^M)$ be an alternating Turing machine (ATM) that runs in time $2^{O(n)}$ with $O(n)$ alternations on inputs of size n.

We simulate the computation of M in $\mathcal{M}_\cup[=_{mon}, \mathrm{not}]$. Each run of M is a tree of configurations of depth $2^{k\cdot n}$, for a suitable constant k. We may assume w.l.o.g. that terminating computation paths of M are no longer than $2^{k\cdot n}$, i.e., the depth of the computation tree of M is bounded by $2^{k\cdot n}$.

By Lemma 5.7, if we use a built-in operation $=_{mon}$, the sizes of all formulas of the proof of Theorem 5.6 are linear in the size of $K(n)$. Now, we fix $K(n) = k \cdot n$, for some constant k.

We use the formulae C_{start}, *Configs*, *AcceptingConfigs*, and ϕ_{succ} as constructed in the proof of Theorem 5.6. We define a modified version of $\psi_{k\cdot n}$ which computes the set of computation paths of length *up to* $2^{k\cdot n}$ (this can be realized by adding "stay transitions" $\langle C, C\rangle$, for $C \in \mathit{Configs}$ to ϕ_{succ}) and where the states of the intermediate configurations are all from $Q_\exists$ if the state of the first configuration is from $Q_\exists$ and are all from $Q_\forall$ otherwise. We can define this as

$$\psi_{i+1} := \psi_i \circ (\mathrm{id} \times \mathrm{id}) \circ \sigma_{1.C'=2.C} \circ \sigma_{1.C.q \in Q_\exists^M \Leftrightarrow 2.C.q \in Q_\exists^M} \circ \mathrm{map}(\langle C : \pi_{1.C}, C' : \pi_{2.C'}\rangle).$$

Now that we only need to consider tapes of size $2^{k\cdot n}$, the monad algebra expressions only occupy space $O(n)$, thus so far we have a LOGLIN reduction.

Let the sets of configurations A_i be inductively defined as

$$\begin{aligned} A_1 &:= \{C \mid \exists C'\ (C, C') \in \psi_{k\cdot n} \wedge \\ &\qquad C' \in \mathit{AcceptingConfigs} \wedge C.q \in Q_\exists^M\} \\ A_{i+1} &:= \{C \mid \exists C'\ (C, C') \in \psi_{k\cdot n} \wedge \\ &\qquad C' \in (\mathit{Configs} - A_i) \wedge C.q \in Q_\exists^M \Leftrightarrow C'.q \notin Q_\exists^M\} \end{aligned}$$

Clearly, $C \in A_i$ for odd i means that $C.q \in Q_\exists^M$ and that C is eventually accepting; $C \in A_i$ for even i means that $C.q \in Q_\forall^M$ and that C is not eventually accepting (both via i alternations and in 2^i steps).

W.l.o.g., we may assume that $F^M \subseteq Q_\exists^M$. By this assumption $F^M \subseteq A_1$, and thus the final transitions leading to accepting states may be universal, rather than just existential. We will now be somewhat sloppy and assume that the number of alternations $K(n) = O(n)$ we ask for is always odd. This is to keep the argument short, but a slight modification of the construction allows to eliminate the assumption.

Then, M accepts its input precisely if C_{start} is eventually accepting with $K(n)$ alternations, that is, iff $C_{start} \in A_{K(n)}$.

It is not difficult to construct $A_{K(n)}$ in monad algebra. We only remark that difference $A - B$ on sets of nested tuples can be defined using $=_{mon}$ and "not" as

$$\{a \in A \mid \nexists b\ b \in B \wedge a =_{mon} b\}$$

or, in monad algebra on pair $\langle 1 : A, 2 : B\rangle$,

$$\begin{aligned}&\mathrm{pairwith}_1 \circ \mathrm{flatmap}\big(\langle a : \pi_1, c : \langle a : \pi_1, B : \pi_2\rangle \circ \mathrm{pairwith}_B \circ \\ &\qquad\qquad \mathrm{flatmap}(a =_{mon} B) \circ \mathrm{not}\big\rangle \circ \mathrm{pairwith}_c \circ \mathrm{map}(\pi_a)\big)\end{aligned}$$

Formula ϕ_{accept} is obviously of linear size, and thus the construction in LOGLIN. □

Considering again $=_{atomic}$ as a built-in, our LOGSPACE-reduction of the proof of Theorem 5.6 for configurations and ϕ_{succ} (with $K(n) = n^k$) in combination with the construction for computations (A_i and ϕ_{accept}) of the proof of Theorem 5.9 yields

Corollary 5.10. *$\mathcal{M}_\cup[=_{atomic}, not]$ is TA$[2^{n^{O(1)}}, n^{O(1)}]$-hard under LOGSPACE-reductions (query complexity).*

We can give a more precise lower bound.

Theorem 5.11. *$\mathcal{M}_\cup[=_{atomic}, not]$ is TA$[2^{O(n)}, O(n)]$-hard under LOGLIN-reductions (query complexity).*

Proof. To allow for a query ϕ_{accept} of linear size overall, we have to rephrase both $\phi_{witness-succ}$ and $\psi_{K(n)}$ to use our formula defining $=_{mon}$ via $=_{atomic}$ only a constant number of times. We can do this now that we have negation and thus equality of a set of nested tuples available. We only sketch the idea here briefly, but it is the same for the two cases. Rather than testing equality linearly many times, we postpone the testing of equality on pairs of tuples until we have collected all the pairs in a set and we can test equality of them all at once. We demonstrate the idea for $\psi_{K(n)}$. Let

$$\begin{aligned}\psi'_0 &:= \phi_{succ} \circ \mathrm{map}\big(\langle 1 : \mathrm{id}, 2 : \emptyset\rangle\big)\\ \psi'_{i+1} &:= (\mathrm{id} \times \mathrm{id}) \circ \sigma_{1.C.q \in Q^M_\exists \Leftrightarrow 2.C.q \in Q^M_\exists} \circ\\ &\quad\ \mathrm{map}\Big(\big\langle 1 : \langle C : \pi_{1.1.C}, C' : \pi_{2.1.C'}\rangle,\\ &\quad\ 2 : \pi_{1.2} \cup \pi_{2.2} \cup \langle 1 : \pi_{1.1.C'}, 2 : \pi_{2.1.C}\rangle \circ \mathrm{sng}\big\rangle\Big)\end{aligned}$$

Now we are interested in those pairs of configurations (c, c') s.t.

$$\langle 1 : \langle C : c, C' : c'\rangle, 2 : S\rangle \in \psi'_{K(n)}$$

and for all $\langle 1 : t, 2 : t'\rangle \in S$, $t =_{mon} t'$. We can define this as

$$\psi_{K(n)} := \psi'_{K(n)} \circ \mathrm{map}\big(\langle 1 : \pi_1, 2 : \pi_2 \circ \text{all-equal}\rangle \circ \mathrm{pairwith}_2 \circ \mathrm{map}(\pi_1)\big) \circ \mathrm{flatten}$$

where all-equal $:= \mathrm{map}((1 =_{mon} 2) \circ [\mathrm{not}]) \circ \mathrm{flatten} \circ \mathrm{not}$.

For $\phi_{witness-succ}$, we proceed analogously. We define $\phi_{zoom-in}$ to be a mapping from sets of tuples $\langle s : (C, C'), w : t, w' : t', mbe : S\rangle$ (where $(s : (C, C'), w : t, w' : t')$ is as in the proof of Theorem 5.6 and S is a set of pairs yet to be checked to be equal – *mbe* is short for "must be equal") to sets of tuples of the same type. We replace e.g. $\sigma_{12\triangleright 34\triangleleft} \circ \pi_{12\triangleright 34\triangleleft}$ in $\phi_{zoom-in}$ by

$$\begin{aligned}&\mathrm{map}\big(\langle s : \pi_s, w : \pi_{w.2}, w' : \pi_{w'.2}, mbe : \langle w : \pi_{w.1}, w' : \pi_{w'.1}\rangle \circ \mathrm{sng} \cup\\ &\pi_{mbe} \circ \mathrm{map}\big(\langle w : \pi_{w.1}, w' : \pi_{w'.1}\rangle \circ \mathrm{sng}\ \cup \langle w : \pi_{w.2}, w' : \pi_{w'.2} \circ \mathrm{sng}\rangle\big) \circ \mathrm{flatten}\rangle\big)\end{aligned}$$

That is, in each such step we add the values that were checked to be equal using $=_{mon}$ in $\phi_{zoom-in}$ – here, for $\sigma_{12\triangleright 34\triangleleft} \circ \pi_{12\triangleright 34\triangleleft}$, $w.1$ and $w'.1$ – and add them to *mbe*. Before we do that, we split the pairs (t, t') of *mbe* into their immediate constituents (as shown in the bottom two lines of the monad algebra expression above). This is necessary to assure that all members of *mbe* are of the same type. However, it has a nice side-effect. By this restructuring, after the last zoom-in step, the members of *mbe* are pairs of *atomic values*, and we actually do not need $=_{mon}$ here at all and can use $=_{atomic}$ instead.

Note that we could not have used this construction in the proof of Theorem 5.6 because now we need negation to check that for each pair (t, t') in *mbe*, $t =_{atomic} t'$. (This can be done using the "all-equal" predicate defined above, with $=_{atomic}$ replacing $=_{mon}$.) □

Since "not" is equivalent to (id $= \emptyset$),

COROLLARY 5.12. *$\mathcal{M}_\cup[=_{deep}]$ is TA$[2^{O(n)}, n]$-hard under LOGLIN-reductions (query complexity).*

The queries constructed in our lower bound proofs are from flat relations to flat relations (we may assume this since we actually use no input data value). Since relational algebra is in PSPACE w.r.t. combined complexity (cf. e.g. [Abiteboul et al. 1995]), PSPACE is closed under composition, and presumably PSPACE $\neq$ NEXPTIME, it seems unlikely that there is even a PSPACE reduction from $\mathcal{M}_\cup[=]$ on flat relations to relational algebra in the spirit of the Conservativity Theorem of Paredaens and Van Gucht ([Paredaens and Van Gucht 1988], Theorem 2.5). This gives a partial negative answer (or more precisely, a negative answer modulo a widely-held complexity-theoretic assumption) to an open question of some standing.

PROPOSITION 5.13. *With respect to query complexity, the languages*

(1) $\mathcal{M}_\cup^{\{\!|\,|\!\}}[=_{atomic}, not]$, $\mathcal{M}_\cup^{\{\!|\,|\!\}}[=_{deep}]$, $\mathcal{M}_\cup^{[\,]}[=_{atomic}, not]$, and $\mathcal{M}_\cup^{[\,]}[=_{deep}]$ are TA$[2^{O(n)}, O(n)]$-hard under LOGLIN-reductions;

(2) $\mathcal{M}_\cup^{\{\!|\,|\!\}}[=_{atomic}]$ and $\mathcal{M}_\cup^{[\,]}[=_{atomic}]$ are NEXPTIME-hard.

PROOF. The lower bound proofs for sets work without modifications on lists and bags – we actually do not compare collections except in the definition of A_i of the proof of Theorem 5.9, where we compute differences. But here, we define difference $R - S$ as a *filter* (using "map") that computes those elements of R for which no element of S with the same value exists. For lists, this will preserve order of the elements in R and for bags it will preserve their multiplicities. For the correctness of our reduction this does not matter, as long as we interpret nonempty collections of type $[\langle\rangle]$ resp. $\{\!|\langle\rangle|\!\}$ (possibly with duplicates) as true and empty collections as false. □

COROLLARY 5.14. *With respect to query complexity,*

—*XQ$[=_{deep}$, child$]$ and XQ$[=_{atomic}$, child, not$]$ are TA$[2^{O(n)}, O(n)]$-hard under LOGLIN-reductions;*

—*XQ$[=_{atomic}$, child$]$ is NEXPTIME-hard.*

PROOF. This follows immediately from the previous results on $\mathcal{M}_{\cup}^{[\,]}$ and the LOGLIN-reduction from $\mathcal{M}_{\cup}^{[\,]}[=]$ to $XQ[=, \text{child}]$ of Lemma 3.3. $\square$

6. DATA COMPLEXITY

Next we consider the data complexity of our query languages.

6.1 Data Complexity of Monad Algebra

It is quite easy to conclude from Theorem 2.5 (conservativity over relational algebra) that the data complexity of $\mathcal{M}_{\cup}[\sigma]$ must be rather low. For the following three results, we assume that the input is given as a string using symbols $\langle$, $\rangle$, {, }, ",", and additional characters to represent atomic values.

PROPOSITION 6.1 FOLKLORE, [SUCIU AND TANNEN 1997]. *$\mathcal{M}_{\cup}[=]$ is in TC_0 w.r.t. data complexity.*

Since the proof in [Suciu and Tannen 1997] is somewhat involved, an alternative direct proof is provided in the electronic appendix to this article.

For monad algebra on lists and bags,

PROPOSITION 6.2 (FOLKLORE). *$\mathcal{M}_{\cup}^{\{\!|\,|\!\}}[=, monus]$ and $\mathcal{M}_{\cup}^{[\,]}[=]$ are in TC_0 w.r.t. data complexity.*

"Parsing" and accessing nested data is described in the proof of Proposition 6.1 and implementing the various operations of monad algebra is not difficult. (See also the similar proof that Core XQuery is in TC_0 – Theorem 6.6). It is folklore that the majority gates of TC_0 circuits are powerful enough to support the arithmetics required to implement bag operations such as bag difference.

For a result that suggests that this is a good bound,

PROPOSITION 6.3 [GRUMBACH ET AL. 1996]. *There are $\mathcal{M}_{\cup}^{\{\!|\,|\!\}}[=, monus]$ queries that are not in AC_0.*

6.2 Data Complexity of XQ

By Proposition 6.2, $\mathcal{M}_{\cup}^{[\,]}[=]$ is in LOGSPACE with respect to data complexity. Since LOGSPACE is closed under composition (cf. [Papadimitriou 1994]) and the mapping C is clearly in LOGSPACE,

COROLLARY 6.4. *$XQ[=_{deep}, child]$ is in LOGSPACE w.r.t. data complexity.*

We can improve on this result. The data complexity of XQuery is so low that we need to be careful about how the XML data is represented. We distinguish the cases of representation by a DOM tree (i.e., a pointer structure) and representation by a string (an XML document). As we show, the complexity is (presumably) slightly lower on strings than on trees, even though the former require parsing the input. It turns out that the complexity bounds are precisely the same as for XPath [Gottlob et al. 2005].

THEOREM 6.5. *$XQ[=_{deep}, child, descendant]$ is LOGSPACE-complete under NC_1-reductions (data complexity) if the input is given as a DOM tree.*

PROOF. Let us assume that for a given XQuery, the results of all its subqueries are already given as input. Then we can evaluate the query in LOGSPACE because all the space we need is a fixed number of log-sized registers for the variables of the query. (We only need a logarithmic number of bits to store a node id of the input).

A fixed query consist of a fixed number of compositions. Since LOGSPACE is closed under compositions (cf. [Papadimitriou 1994]), we can compose the algorithm just discussed for the individual subqueries into a single LOGSPACE algorithm that intuitively computes the query by precomputing its subqueries bottom-up (w.r.t. the syntax tree of the query).

LOGSPACE-hardness follows from the fact that directed tree reachability is LOGSPACE-complete under NC_1-reductions [Cook and McKenzie 1987] and directed tree reachability (checking whether node w is reachable from node v in tree t) can be easily encoded by mapping t to a XML tree in which only v has label "v" and only w has label "w". Then the query

```
for $x in $root/descendant::v return
   for $y in $x/descendant::w return ⟨true/⟩
```

tests reachability of w from v. □

If the input is given as a character string, the query evaluation problem is (possibly) easier.[14]

THEOREM 6.6. *XQ*[$=_{deep}$, *child, descendant*] *is in* TC_0 *w.r.t. data complexity if the XML input is given as a character string.*

PROOF. We show a stronger result, that every Core XQuery expression can be encoded as a TC_0 reduction that transforms the input data into the query result.

By FOM, we denote first-order logic extended with majority quantifiers M [Barrington et al. 1990]. A formula $My\,\phi(\vec{x}, y)$ is true if $\phi(\vec{x}, y)$ is true for more than half of the positions y of the input. It is known that TC_0 is equivalent to the class of languages recognizable using FOM sentences [Barrington et al. 1990].

The reduction is encoded in FOM. A FOM reduction [Barrington et al. 1990] is a set of FOM formulae, consisting of a formula "size" s.t. size(s) iff the size of the string is s and a formula pos_a for each $a \in \Sigma$ s.t. $\text{pos}_a(i)$ iff the i-th symbol of the string is "a". It is known [Barrington et al. 1990] that FOM can express predicates $x = y + z$ and $x = \#y\,\phi(y)$, such that x is the number of positions y for which $\phi(y)$ holds. We use $\Sigma\{y \mid \phi(\vec{x}, y)\}$ as a shortcut for $\#u\ \exists y : \phi(\vec{x}, y) \wedge 1 \leq u \leq y$.

We assume the document to be encoded using an alphabet of opening and matching closing tags. For the sake of simplicity of this proof, but without loss of generality, we assume that base values (e.g. strings) are encoded as subdocuments, using opening and closing tags. The input is a well-formed sequence of opening and closing tags. We identify nodes by the position of their opening tag in the (input) string. The input is represented using a predicate $\text{size}[\![\$ROOT]\!]$ s.t. $\text{size}[\![\$ROOT]\!](n)$ iff n is the size of the input and predicates $\text{pos}_a[\![\$ROOT]\!]$ s.t. $\text{pos}_a[\![\$ROOT]\!](i)$ iff the

[14]The same observation has been made for XPath in [Gottlob et al. 2005].

$$
\begin{aligned}
\mathrm{size}\llbracket\epsilon\rrbracket_k(\vec{e},s) &:\Leftrightarrow s=0\\
\mathrm{pos}_l\llbracket\epsilon\rrbracket_k(\vec{e},i) &:\Leftrightarrow \mathrm{false}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{size}\llbracket\langle a\rangle\alpha\langle/a\rangle\rrbracket_k(\vec{e},s) &:\Leftrightarrow \exists s' : \mathrm{size}\llbracket\alpha\rrbracket_k(\vec{e},s') \wedge s = 2+s'\\
\mathrm{pos}_l\llbracket\langle a\rangle\alpha\langle/a\rangle\rrbracket_k(\vec{e},i) &:\Leftrightarrow \exists s : \mathrm{size}\llbracket\langle a\rangle\alpha\langle/a\rangle\rrbracket_k(\vec{e},s) \wedge\\
&\qquad (i=1 \Rightarrow l=\langle a\rangle) \wedge\\
&\qquad \big(1<i<s \Rightarrow \mathrm{pos}_l\llbracket\alpha\rrbracket_k(\vec{e},i-1)\big) \wedge\\
&\qquad (i=s \Rightarrow l=\langle/a\rangle)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{size}\llbracket\alpha\,\beta\rrbracket_k(\vec{e},s) &:\Leftrightarrow \exists s_1\exists s_2 : s=s_1+s_2 \wedge \mathrm{size}\llbracket\alpha\rrbracket_k(\vec{e},s_1) \wedge \mathrm{size}\llbracket\beta\rrbracket_k(\vec{e},s_2)\\
\mathrm{pos}_l\llbracket\alpha\,\beta\rrbracket_k(\vec{e},i) &:\Leftrightarrow \exists s : \mathrm{size}\llbracket\alpha\rrbracket_k(\vec{e},s) \wedge \big(i\le s \Rightarrow \mathrm{pos}_l\llbracket\alpha\rrbracket_k(\vec{e},i)\big) \wedge\\
&\qquad \big(i>s \Rightarrow \exists i' : i'=i-s \wedge \mathrm{pos}_l\llbracket\beta\rrbracket_k(\vec{e},i')\big)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{size}\llbracket\text{for } \$x_{k+1} \text{ in } \alpha \text{ return } \beta\rrbracket_k(\vec{e},s) &:\Leftrightarrow s=\Sigma\{s' \mid \exists j : \mathrm{item}\llbracket\alpha\rrbracket_k(\vec{e},j) \wedge \mathrm{size}\llbracket\beta\rrbracket_{k+1}(\vec{e},j,s')\}\\
\mathrm{pos}_l\llbracket\text{for } \$x_{k+1} \text{ in } \alpha \text{ return } \beta\rrbracket_k(\vec{e},i) &:\Leftrightarrow \exists s\exists j_0 : s=\Sigma\{s' \mid \exists j : j<j_0 \wedge\\
&\qquad \mathrm{item}\llbracket\alpha\rrbracket_k(\vec{e},j) \wedge \mathrm{size}\llbracket\beta\rrbracket_{k+1}(\vec{e},j,s')\} \wedge\\
&\qquad \exists s' : \mathrm{item}\llbracket\alpha\rrbracket_k(\vec{e},s+1) \wedge \mathrm{size}\llbracket\beta\rrbracket_{k+1}(\vec{e},s+1,s')\} \wedge\\
&\qquad s<i\le s+s' \wedge \mathrm{pos}_l\llbracket\beta\rrbracket_{k+1}(\vec{e},s+1,i-s)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{size}\llbracket\$x_i/\chi :: a\rrbracket_k(\vec{e},s) &:\Leftrightarrow s=\Sigma\{j'-j+1 \mid \mathrm{axis}_\chi\llbracket\$x_i\rrbracket_k(\vec{e},1,j) \wedge \mathrm{node}\llbracket\$x_i\rrbracket_k(\vec{e},j,j')\}\\
\mathrm{pos}_a\llbracket\$x_i/\chi :: a\rrbracket_k(\vec{e},i) &:\Leftrightarrow \exists s\exists j_0 : s=\Sigma\{j'-j+1 \mid \exists j : j<j_0 \wedge\\
&\qquad\qquad \mathrm{axis}_\chi\llbracket\$x_i\rrbracket_k(\vec{e},1,j) \wedge \mathrm{node}\llbracket\$x_i\rrbracket_k(\vec{e},j,j')\} \wedge\\
&\qquad \exists s'\exists j_0' : \mathrm{axis}_\chi\llbracket\$x_i\rrbracket_k(\vec{e},1,j_0) \wedge \mathrm{node}\llbracket\$x_i\rrbracket_k(\vec{e},j_0,j_0')\} \wedge\\
&\qquad s'=j_0'-j_0+1 \wedge\\
&\qquad s<i\le s+s' \wedge \mathrm{pos}_l\llbracket\$x_i\rrbracket_k(\vec{e},i-s+j_0-1)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{size}\llbracket\$x_i\rrbracket_k(x_1,\ldots,x_k,s) &:\Leftrightarrow \exists j : \mathrm{node}\llbracket\mathrm{expr}(\$x_i)\rrbracket_{i-1}(x_1,\ldots,x_{i-1},x_i,j) \wedge\\
&\qquad s=j-x_i+1\\
\mathrm{pos}_l\llbracket\$x_i\rrbracket_k(x_1,\ldots,x_k,i) &:\Leftrightarrow \mathrm{pos}_l\llbracket\mathrm{expr}(\$x_i)\rrbracket_{i-1}(x_1,\ldots,x_{i-1},x_i+i-1)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{size}\llbracket\$root\rrbracket_k(\vec{e},s) &:\Leftrightarrow \mathrm{size}(s)\\
\mathrm{pos}_l\llbracket\$root\rrbracket_k(\vec{e},i) &:\Leftrightarrow \mathrm{pos}_l(i)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{size}\llbracket\text{if } \Phi \text{ then } \alpha\rrbracket_k(\vec{e},s) &:\Leftrightarrow \big(\mathrm{cond}\llbracket\Phi\rrbracket_k(\vec{e}) \Rightarrow \mathrm{size}\llbracket\alpha\rrbracket_k(\vec{e},s)\big) \wedge \big((\neg\mathrm{cond}\llbracket\Phi\rrbracket_k(\vec{e})) \Rightarrow s=0\big)\\
\mathrm{pos}_l\llbracket\text{if } \Phi \text{ then } \alpha\rrbracket_k(\vec{e},i) &:\Leftrightarrow \mathrm{pos}_l\llbracket\alpha\rrbracket_k(\vec{e},i)
\end{aligned}
$$

Fig. 8. FOM encoding of the Core XQuery evaluation problem.

i-th symbol of the input is "a". Let

$$\begin{aligned}\mathrm{node}[\![\alpha]\!]_k(\vec{e}, i, i') :\Leftrightarrow \bigvee_{\langle a\rangle\in\Sigma} &\mathrm{pos}_{\langle a\rangle}[\![\alpha]\!]_k(\vec{e}, i) \wedge \mathrm{pos}_{\langle /a\rangle}[\![\alpha]\!]_k(\vec{e}, i') \wedge \\ \#u(i < u < i' \wedge \mathrm{pos}_{\langle a\rangle}[\![\alpha]\!]_k(\vec{e}, u)) &= \#u(i < u < i' \wedge \mathrm{pos}_{\langle /a\rangle}[\![\alpha]\!]_k(\vec{e}, u))\end{aligned}$$

That is, $\mathrm{node}[\![\alpha]\!]_k(\vec{e}, i, i')$ is true iff i and i' are the positions of an opening tag and a matching closing tag. Since we may assume that the document is well-formed, a sufficient condition for i' being the closing tag matching the opening tag i is that the number of opening tags $\langle a\rangle$ between i and i' is the same as the number of closing tags $\langle /a\rangle$ (i.e., other tags do not have to be considered).

Now, our $XQ[=_{deep}]$ query Q can be encoded by FOM formulas $\mathrm{pos}_l[\![Q]\!]_k$ and $\mathrm{size}[\![Q]\!]_k$ as shown in Figure 8 (for most XQ constructs), with

$$\begin{aligned}\mathrm{cond}[\![\$x_i =_{deep} \$x_j]\!]_k(\vec{e}) &:\Leftrightarrow \exists s : \mathrm{size}[\![\$x_i]\!]_k(\vec{e}, s) \wedge \mathrm{size}[\![\$x_j]\!]_k(\vec{e}, s) \wedge \\ &\quad \forall p : 1 \le p \le s \Rightarrow \bigwedge_{l\in\Sigma} \mathrm{pos}_l[\![\$x_i]\!](\vec{e}, p) \Leftrightarrow \mathrm{pos}_l[\![\$x_j]\!](\vec{e}, p)\end{aligned}$$

$$\begin{aligned}\mathrm{axis}_{\mathrm{descendant}}[\![\alpha]\!]_k(\vec{e}, i, j) &:\Leftrightarrow \exists i', j'\ \mathrm{node}[\![\alpha]\!]_k(\vec{e}, i, i') \wedge \mathrm{node}[\![\alpha]\!]_k(\vec{e}, j, j') \wedge \\ &\quad i < j \wedge j' < i' \\ \mathrm{axis}_{\mathrm{child}}[\![\alpha]\!]_k(\vec{e}, i, j) &:\Leftrightarrow \exists i', j'\ \mathrm{node}[\![\alpha]\!]_k(\vec{e}, i, i') \wedge \mathrm{node}[\![\alpha]\!]_k(\vec{e}, j, j') \wedge \\ &\quad i < j \wedge j' < i' \wedge \\ &\quad \nexists l, l' : \mathrm{node}[\![\alpha]\!]_k(\vec{e}, l, l') \wedge i < l < j \wedge j' < l' < i'.\end{aligned}$$

$$\begin{aligned}\mathrm{item}[\![\alpha]\!]_k(\vec{e}, i) &:\Leftrightarrow \exists i' : \mathrm{node}[\![\alpha]\!]_k(\vec{e}, i, i') \wedge\ \nexists j, j' : \mathrm{node}[\![\alpha]\!]_k(\vec{e}, j, j') \wedge \\ &\quad j < i \wedge i' < j'\end{aligned}$$

Note that in $\mathrm{pos}_l[\![\alpha]\!]_k(\vec{e}, i)$, $\mathrm{size}[\![\alpha]\!]_k(\vec{e}, s)$, $\vec{e}$ denotes the environment for k variables, indicating positions/nodes assigned to known variables.

Let the *defining expression* for a variable $\$x$, $expr(\$x)$, be α if $\$x$ is introduced in an XQ expression "for $\$x$ in α return β".

Considering the problem of deciding whether the root node of the query result has a child as the decision problem for query evaluation, we encode it in FOM (for query Q) as $\exists s : size[\![Q]\!]_1(1, s) \wedge s > 2$. □

REMARK 6.7. To get an intuition for the reduction of Figure 8, consider again our XQ semantics definition of Figure 1. There, the environments $\vec{e}$ are tuples of valuations of XQuery variables (i.e., trees). Consider the reformulation of the semantics $[\![\cdot]\!]$ of Figure 1 that we get if we assume that the value of each variable $\$x_i$ in an environment is an integer that indicates the position of the starting tag of the node it binds to in the value of $expr(\$x)$. To get a correct semantics definition along these lines, we make the following replacements:

$$\begin{aligned}
[\![\$x_i]\!]_k(t_1,\dots,t_n) &:= [\![\text{expr}(\$x_i)]\!]_{i-1}(t_1,\dots,t_{i-1})\\
[\![\text{for } \$x_{k+1} \text{ in } \alpha \text{ return } \beta]\!]_k(\vec{e}) &:= \text{let } i_1,\dots,i_n \text{ be the start positions of the}\\
&\ n \text{ elements in } [\![\alpha]\!]_k(\vec{e});\\
&\ \text{return } [\![\beta]\!]_{k+1}(\vec{e},i_1)+\cdots+[\![\beta]\!]_{k+1}(\vec{e},i_n)
\end{aligned}$$

The remaining definitions of Figure 1 stay the same. Here, concatenation is of course that of strings rather than that of lists.

We illustrate this semantics with an example. Consider the query "for \$x in \$root/a return \$x" on input string \$root = "$\langle c\rangle\langle d\rangle\langle /d\rangle\langle a\rangle\langle /a\rangle\langle a\rangle\langle c\rangle\langle /c\rangle\langle /a\rangle\langle /c\rangle$". We can evaluate the query by first binding \$x to the node identified by position 1 in $[\![expr(\$x)]\!]_0$ = "$\langle a\rangle\langle /a\rangle\langle a\rangle\langle c\rangle\langle /c\rangle\langle /a\rangle$" and outputting $[\![\$x]\!]_1(1)$ = "$\langle a\rangle\langle /a\rangle$". Second we bind \$x to position 3 in $[\![expr(\$x)]\!]_0$ and return $[\![\$x]\!]_1(3)$ = "$\langle a\rangle\langle c\rangle\langle /c\rangle\langle /a\rangle$".

The previous proof employs a direct encoding of this altered semantics in FOM.

7. COMPOSITION-FREE XQ

Composition-free Core XQuery, $XQ^-[\text{not}]$, is the fragment of Core XQuery obtained by the grammar

$$\begin{aligned}
query &::= () \mid \langle a\rangle query\langle /a\rangle \mid query\ query\\
&\mid\ var \mid var/axis :: \nu\\
&\mid\ \text{for } var \text{ in } var/axis :: \nu \text{ return } query\\
&\mid\ \text{if } cond \text{ then } query\\
cond &::= var = var \mid var = \langle a/\rangle \mid \text{true}\\
&\mid\ \text{some } var \text{ in } var/axis :: \nu \text{ satisfies } cond\\
&\mid\ cond \text{ and } cond \mid cond \text{ or } cond \mid \text{not } cond
\end{aligned}$$

The keyword "every" can again be obtained from "some" and "not". Testing whether condition $\$x/\chi :: \nu$ (where χ is an axis and ν is a node test) can be matched is of course possible as "some $\$y$ in $\$x/\chi :: \nu$ satisfies "$\$y = \$y$". *Positive* composition-free Core XQuery XQ^- is again obtained by removing negation "not" from the language.

For our expressiveness proof below, we will use a variant of XQ^- with less syntax, i.e. in which conditions are defined using the usual query operations rather than "some", "and", and "or".

Let $XQ^{\sim}$ denote the XQ queries

—which do not contain "let"-expressions,

—for which for each expression "for \$x in α return β", α is of the form $\$y/\nu$, and

—which in addition support conditions $\$z = \langle a/\rangle$.

$XQ^{\sim}$ and XQ^- are expressively equivalent.

Proposition 7.1. $XQ^{\sim} = XQ^-$.

Proof. $\Rightarrow$: For a mapping from $XQ^{\sim}$ to XQ^-, we define an appropriate translation function f that we use to rewrite all maximal if-conditions (i.e., conditions of if-expressions that are not subexpressions of if-expressions):

$$\begin{aligned}
f(\alpha\ \beta) &:= f(\alpha) \text{ or } f(\beta)\\
f(\text{for \$y in \$x/}\nu \text{ return } \alpha) &:= \text{some \$y in \$x/}\nu \text{ satisfies f}(\alpha)\\
f(\text{if } \phi \text{ then } \alpha) &:= f(\phi) \text{ and } f(\alpha)\\
f(\text{not } \phi) &:= \text{not } f(\phi)\\
f(\langle a\rangle\alpha\langle/a\rangle) &:= \text{true}
\end{aligned}$$

On all other kinds of expressions, f is the identity.

$\Leftarrow$: For a mapping from XQ^- to $XQ^\sim$, we only need to eliminate "true", "some", "and", and "or" using their definitions from Section 3. □

Example 7.2. It is easy to verify that the query

```
<result>
{ for $x in $root/a return
     if not(for $y in $x/b return if $y/c then ($y/d $y/e))
     then $x/f }
</result>
```

is $XQ^\sim$. The corresponding XQ^- query is

```
<result>
{ for $x in $root/a return
     if not(some $y in $x/b satisfies ($y/c and ($y/d or $y/e)))
     then $x/f }
</result>
```

□

The mappings from the proof of Proposition 7.1 can be implemented efficiently, thus our complexity results established below will hold for both XQ^- and $XQ^\sim$.

7.1 Complexity Results for XQ$^-$

We now provide our complexity characterization of composition-free Core XQuery. As announced in the Introduction, the query evaluation problem for XQ^- is in polynomial space with respect to combined complexity.

Proposition 7.3. *$XQ^-[=_{deep}$, all axes, not$]$ is in space $O(|Q|\cdot\log|t|)$, where $|Q|$ is the size of the query and $|t|$ is the size of the data tree.*

Proof. It is easy to check that by definition of the fragment, XQuery variables always range exclusively over nodes of the input tree. This can be verified by checking the invariant that each variable is introduced using a "for"-statement over a collection defined by an expression $\$x/\nu$, starting at the root node of the input tree. Thus there is a straightforward algorithm – direct nested-loop based evaluation – for XQ^- queries that only takes memory to store a pointer into the input tree (taking space $\log|t|$) for each of the $O(|Q|)$ variables in the query. □

For the remaining results, we study decision problems and thus Boolean queries. Since valid XML query results have to consist of at least a root node, we say that a Boolean (XQ^-) query $\langle a\rangle\alpha\langle/a\rangle$ returns true iff the root node of the result tree has at least one child.

PROPOSITION 7.4. *$XQ^-[=_{atomic}$, child, not$]$ is PSPACE-hard w.r.t. query complexity.*

PROOF. The problem is PSPACE-hard already with respect to query complexity (i.e., when the input tree is fixed). The proof is by reduction from the Quantified Boolean Formula evaluation problem (QBF), which is PSPACE-complete (cf. [Papadimitriou 1994]).

A QBF Ψ is a formula of the form $Q_1 x_1 \cdots Q_k x_k\ \Phi(x_1, \ldots x_k)$ where $Q_1, \ldots, Q_k \in \{\forall, \exists\}$ and $\Phi(x_1, \ldots, x_k)$ is a quantifier-free Boolean formula over the propositional variables $x_1, \ldots, x_k$ (that is, a formula constructed from $x_1, \ldots, x_k$ and the Boolean connectives $\wedge$, $\vee$, and $\neg$). Let Φ' be the expression obtained from formula Φ by replacing all occurrences of $\wedge$, $\vee$, and $\neg$ by "and", "or", and "not", respectively, and replacing each occurrence of variable x_i by ($\$x_i =_{atomic} \langle\text{true}/\rangle$). Let α be the *XQ* query

$\langle a\rangle$ { if Q'_1 $\$x_1$ in \$root/* satisfies $(\cdots(Q'_k$ $\$x_k$ in \$root/* satisfies $\Phi')\cdots)$
then $\langle\text{yes}/\rangle$} $\langle/a\rangle$

where Q'_i is "some" if $Q_i = \exists$ and "every" otherwise. Let t be the fixed data tree consisting of a root node with two children $t_{\text{true}}, t_{\text{false}}$, one with labels "true" and "false", respectively.

For a given valuation $\theta : x_i \to \{\text{true}, \text{false}\}$ of the propositional variables $x_1, \ldots, x_k$, $[\![\Phi']\!]_k(t_{\theta(x_1)}, \ldots, t_{\theta(x_k)})$ is true (i.e., a nonempty list) iff $\Phi(\theta(x_1), \ldots, \theta(x_k))$ is true. But then, since "some $\$x_i$ in \$root/*" and "every $\$x_i$ in \$root/*" provide existential and universal quantification, respectively, over t_{true} and t_{false}, $[\![\alpha]\!]_1(t) = \langle a\rangle\langle\text{yes}/\rangle\langle/a\rangle$ iff the QBF Ψ is true. □

EXAMPLE 7.5. Consider the QBF $\forall x \exists y((\neg x \vee y) \wedge (x \vee \neg y))$, which is true. This formula can be phrased as the query

```
⟨a⟩
{ if every $x in $root/* satisfies
        (some $y in $root/* satisfies
            (not $x =_atomic ⟨true/⟩ or $y =_atomic ⟨true/⟩) and
            ($x =_atomic ⟨true/⟩ or not $y =_atomic ⟨true/⟩)) then ⟨yes/⟩}
⟨/a⟩
```

□

While negation and universal quantification were redundant in *$XQ[=_{deep}]$*, and excluding them did not reduce the complexity of the language, it is interesting to consider the case of *XQ^-* without negation.

PROPOSITION 7.6. *$XQ^-[=_{deep}$, all axes$]$ is in NP w.r.t. combined complexity.*

PROOF. We show this for *$XQ^\sim$*, as it has less syntax than *XQ^-* and we have LOGSPACE-reductions between the two languages.

We define a nondeterministic procedure for computing part of the result of an *$XQ^\sim[=_{deep}$*, all axes] query by a modification of the *XQ* semantics function $[\![\cdot]\!]$ of Figure 1. Let $[\![\cdot]\!]'$ be $[\![\cdot]\!]$ restricted to the syntax of *$XQ^\sim$*, with the following exception: $[\![\text{for } \$x_{k+1} \text{ in } \$x_i/\chi :: \nu \text{ return } \alpha]\!]'_k(t_1, \ldots, t_k)$ guesses a subtree t_{k+1} of t_i such that t_{k+1} is in $[\![\$x_i/\chi :: \nu]\!]'_k(t_1, \ldots, t_k)$ and returns $[\![\beta]\!]'_{k+1}(t_1, \ldots, t_{k+1})$.

$$\begin{aligned}
\text{for \$x in () return } \alpha &\vdash \text{()} && (1)\\
\text{for \$x in } (\langle a\rangle\ \alpha\ \langle/a\rangle) \text{ return } \beta &\vdash \beta[\$x \Rightarrow (\langle a\rangle\ \alpha\ \langle/a\rangle)] && (2)\\
\text{for \$x in } (\alpha\ \beta) \text{ return } \gamma &\vdash (\text{for \$x in } \alpha \text{ return } \gamma)\ (\text{for \$x in } \beta \text{ return } \gamma) && (3)\\
\text{for \$y in (for \$x in } \alpha \text{ return } \beta) \text{ return } \gamma &\vdash \text{for \$x in } \alpha \text{ return for \$y in } \beta \text{ return } \gamma && (4)\\
\text{for \$x in (if } \phi \text{ then } \alpha) \text{ return } \beta &\vdash \text{for \$x in } \alpha \text{ return if } \phi \text{ then } \beta && (5)\\
\text{for \$y in \$x return } \alpha &\vdash \alpha[\$y \Rightarrow \$x] && (6)
\end{aligned}$$

Fig. 9. Rewrite rules for translating for-expressions to $XQ^{\sim}$.

Clearly, this algorithm runs in nondeterministic polynomial time, because each variable is only assigned a value once, and all other checks are polynomial. It is also easy to see that the result is sound: If, for a given query Q and data tree t, $[\![Q]\!]'_1(t)$ is nonempty, then so is $[\![Q]\!]_1(t)$. The converse can be verified by a straightforward induction on the syntax of $XQ^{\sim}$. Clearly, $[\![\text{for } \$x_{k+1} \text{ in } \alpha \text{ return } \beta]\!]_k$ computes a suitably ordered concatenation of the results $[\![\text{for } \$x_{k+1} \text{ in } \alpha \text{ return } \beta]\!]'_k$ would produce if all choices of assignment of a tree from $[\![\alpha]\!]_k$ to $\$x_{k+1}$ were tried. Therefore, emptiness of the former implies, since we may assume our NP algorithm is always lucky at guessing, emptiness of the latter. □

PROPOSITION 7.7. *$XQ^-[=_{atomic}, child]$ is NP-hard w.r.t. query complexity.*

PROOF. This follows immediately from the NP-hardness of conjunctive (relational) queries [Chandra and Merlin 1977], and a proof can be given e.g. by reduction from 3-Colorability: The fixed data tree consists of a root node and three children, which are labeled "red", "green", and "blue", respectively.

Given a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_m\}$ and $E = \{\{v_{i(1,1)}, v_{i(1,2)}\}, \ldots, \{v_{i(n,1)}, v_{i(n,2)}\}\}$ $(1 \le i(\cdot,\cdot) \le m)$, we construct the query

```
⟨result⟩ { for $x_1$ in $root/* return
              $\ddots$
                for $x_{m-1}$ in $root/* return
                   for $x_m$ in $root/* return
                   if (not $x_{i(1,1)}$ $=_{atomic}$ $x_{i(1,2)}$) and ... and
                      (not $x_{i(n,1)}$ $=_{atomic}$ $x_{i(n,2)}$) then ⟨yes/⟩ }
⟨/result⟩
```

It is easy to verify that indeed this query computes "yes" nodes precisely if G is 3-colorable. Obviously, the query can be computed from G in logarithmic space. □

7.2 Expressiveness of XQ$^-$

In this final section, we show that surprisingly, for an important case (atomic equality and "child" as the only supported axis), composition-free Core XQuery is actually just as expressive as full Core XQuery. This is true even though XQ^- is in PSPACE and XQ is hard for TA$[2^{O(n)}, O(n)]$. Thus under commonly-held complexity theoretic assumptions, XQ is exponentially more succinct than XQ^-.

We use the shortcut $(\langle a\rangle\alpha\langle/a\rangle)/\chi{::}\nu$ for $\$x/\chi{::}\nu$ such that $\$x$ has been defined using "let" as $(\langle a\rangle\alpha\langle/a\rangle)$. Below, "dos" is a shortcut for the "descendant-or-self"

$$\begin{array}{rl}
\text{(let } \$x := \langle a\rangle\{ \text{ for } \$w \text{ in } \$\text{root}/\text{*} \text{ return } \langle b\rangle\{\$w\}\langle/b\rangle \ \}\langle/a\rangle) \text{ for } \$y \text{ in } \$x/b \text{ return } \$y/* & \overset{elim.let}{\vdash} \\
\text{for } \$y \text{ in } (\langle a\rangle\{ \text{ for } \$w \text{ in } \$\text{root}/\text{*} \text{ return } (\langle b\rangle\{\$w\}\langle/b\rangle) \ \}\langle/a\rangle)/b \text{ return } \$y/* & \overset{Lem.\ 7.8}{\vdash} \\
\text{for } \$y \text{ in } (\text{for } \$w \text{ in } \$\text{root}/\text{*} \text{ return } (\langle b\rangle\{\$w\}\langle/b\rangle)) \text{ return } \$y/* & \overset{4}{\vdash} \\
\text{for } \$w \text{ in } \$\text{root}/\text{*} \text{ return for } \$y \text{ in } (\langle b\rangle\{\$w\}\langle/b\rangle) \text{ return } \$y/* & \overset{2}{\vdash} \\
\text{for } \$w \text{ in } \$\text{root}/\text{*} \text{ return } (\langle b\rangle\{\$w\}\langle/b\rangle)/* & \overset{Lem.\ 7.8}{\vdash} \\
\text{for } \$w \text{ in } \$\text{root}/\text{*} \text{ return } \$w &
\end{array}$$

Fig. 10. Example rewriting.

axis; it will be redundant because $\$x/\text{dos}::\nu$ is equivalent to

$$(\text{if } \$x/\text{self}::\nu \text{ then } \$x)\ \$x//\nu.$$

LEMMA 7.8. *Let a be a label, χ an axis, ν a nodetest, and α an expression from $XQ^{\sim}[=_{atomic}$, child, descendant, self, dos, not$]$. Then there is an expression from $XQ^{\sim}[=_{atomic}$, child, descendant, self, dos, not$]$ equivalent to $(\langle a\rangle\alpha\langle/a\rangle)/\chi::\nu$.*

PROOF. Rules to rewrite each such expression $(\langle a\rangle\alpha\langle/a\rangle)/\chi::\nu$ into an equivalent $XQ^{\sim}[=_{atomic}$, child, descendant, self, not$]$ expression are easy to specify:

$$\begin{array}{rcl}
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\nu & \vdash & \alpha/\text{self}::\nu \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\text{self::b} & \vdash & () \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\text{self::a} & \vdash & \langle a\rangle\ \alpha\ \langle/a\rangle \\
(\langle b\rangle\ \alpha\ \langle/b\rangle)/\text{self::*} & \vdash & \langle b\rangle\ \alpha\ \langle/b\rangle \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)//\nu & \vdash & \alpha/\text{dos}::\nu \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\text{dos}::* & \vdash & \langle a\rangle\ \alpha\ \langle/a\rangle\ (\alpha//*) \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\text{dos}::a & \vdash & \langle a\rangle\ \alpha\ \langle/a\rangle\ (\alpha//a) \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\text{dos}::b & \vdash & \alpha//b \\
()/\chi::\nu & \vdash & () \\
(\alpha\ \beta)/\chi::\nu & \vdash & (\alpha/\chi::\nu)\ (\beta/\chi::\nu) \\
(\text{for } \$\text{x in } \alpha \text{ return } \beta)/\chi::\nu & \vdash & \text{for } \$\text{x in } \alpha \text{ return } (\beta/\chi::\nu) \\
(\text{if } \phi \text{ then } \alpha)/\chi::\nu & \vdash & \text{if } \phi \text{ then } (\alpha/\chi::\nu) \\
(\$x/\chi::\nu)/\chi'::\nu' & \vdash & \text{for } \$y \text{ in } \$x/\chi::\nu \text{ return } \$y/\chi'::\nu'
\end{array}$$

(Note that $(\$x/\chi::\nu)/\chi'::\nu'$ in the final rule is really equivalent to the for-expression on the right-hand side of that rule, and is in general not equivalent to $\$x/\chi::\nu/\chi'::\nu'$, as the former may produce duplicates if both χ and χ' are "descendant".) □

THEOREM 7.9. *$XQ^{\sim}[=_{atomic}$, child, desc, self, not$]$ captures the $XQ[=_{atomic}$, child, desc, self, not$]$ queries.*

PROOF. We first replace each expression of the form "(let $\$x := \langle a\rangle\alpha\langle/a\rangle)\ \beta$" by an expression $\beta' := \beta[\$x \Rightarrow \langle a\rangle\alpha\langle/a\rangle]$ obtained by substituting each occurrence of variable $\$x$ in β by $\langle a\rangle\alpha\langle/a\rangle$.

We now need to consider where such a replacement of a variable $\$x$ by an expression $\langle a\rangle\alpha\langle/a\rangle$ can occur:

(1) Inside an equality $\$x =_{atomic} \alpha$ (with α either a variable or a constant $\langle b/\rangle$). To rewrite $\$x$ with $\langle a\rangle\alpha\langle/a\rangle$, we may assume that α is (); otherwise, we could not type $\langle a\rangle\alpha\langle/a\rangle$ to be an atomic value. Thus we obtain $\langle a/\rangle =_{atomic} \alpha$, which is $XQ^{\sim}$. Conditions $\langle a/\rangle =_{atomic} \langle a/\rangle$ and $\langle a/\rangle =_{atomic} \langle b/\rangle$ are rewritten into "true" and "not(true)", respectively.

(2) Inside an expression $\$x$ or $\$x/\chi{::}\nu$ (either in the "in"- expression of a for-loop or as an expression constructing "output").
Here rewriting may lead to expressions of the form $(\langle a\rangle\alpha\langle/a\rangle)/\chi{::}\nu$, which is not XQ syntax. We can eliminate such expressions using Lemma 7.8.

Now the query obtained is already an $XQ^{\sim}$ query if in all expressions "for $\$x$ in α return β", α is of the form $\$z$ or $\$z/\chi{::}\nu$. Otherwise, we apply the rewrite rules from Figure 9. This may again produce expressions $(\langle a\rangle\alpha\langle/a\rangle)/\chi{::}\nu$, by rule (2). We eliminate such cases again using Lemma 7.8.

It can be verified that the rewrite system thus specified indeed maps any query from language $XQ[=_{atomic}, \text{child, desc, self, not}]$ to an equivalent query in the language $XQ^{\sim}[=_{atomic}, \text{child, desc, self, not}]$. An example mapping to $XQ^{\sim}$ illustrating our rewrite system is given in Figure 10. □

Acknowledgments

I am indebted to Michael Benedikt, Peter Buneman, Leonid Libkin, Nicole Schweikardt, Val Tannen, and Andrei Voronkov for their remarks that helped to improve an earlier version of this article [Koch 2005a]. I thank Dan Olteanu and Stefanie Scherzinger for proofreading [Koch 2005b]. Finally, I thank Jan Van den Bussche for pointing out to me that the question whether there is a *polynomial-time* mapping from nested-relational algebra on flat relations to flat relational algebra is considered a longstanding open problem.

REFERENCES

Abiteboul, S. and Beeri, C. 1995. "The Power of Languages for the Manipulation of Complex Values". *VLDB J.* **4**, 4, 727–794.

Abiteboul, S. and Hillebrand, G. G. 1995. "Space Usage in Functional Query Languages". In *Proc. of the 5th International Conference on Database Theory (ICDT)*. 439–454.

Abiteboul, S., Hull, R., and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley.

Barrington, D. A. M., Immerman, N., and Straubing, H. 1990. "On Uniformity within NC1". *Journal of Computer and System Sciences* **41**, 3, 274–306.

Berman, L. 1980. "The Complexity of Logical Theories". *Theor. Comput. Sci.* **11**, 216–224.

Buneman, P., Naqvi, S. A., Tannen, V., and Wong, L. 1995. "Principles of Programming with Complex Objects and Collection Types". *Theor. Comput. Sci.* **149**, 1, 3–48.

Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. 1981. "Alternation". *Journal of the ACM* **28**, 1, 114–133.

Chandra, A. K. and Merlin, P. M. 1977. "Optimal Implementation of Conjunctive Queries in Relational Data Bases". In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing (STOC'77)*. Boulder, CO, USA, 77–90.

Codd, E. F. 1970. "A Relational Model of Data for Large Shared Data Banks". *Commun. ACM* **13**, 6 (June), 377–387.

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

Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. 2001. "Complexity and Expressive Power of Logic Programming". *ACM Computing Surveys* **33**, 3 (Sept.), 374–425.

Dantsin, E. and Voronkov, A. 1997. "Complexity of Query Answering in Logic Databases with Complex Values". In *Proc. LFCS'97, LNCS 1234*. 56–66.

Dantsin, E. and Voronkov, A. 2000. Expressive Power and Data Complexity of Query Languages for Trees and Lists. In *Proceedings of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'00)*. ACM Press, Dallas, Texas, USA, 157–165.

Fernandez, M., Siméon, J., and Wadler, P. 2000. An Algebra for XML Query. In *Proc. FSTTCS 2000*. LNCS 1974. Springer-Verlag, 11–45.

Fernandez, M. F. and Siméon, J. 2004. "Building an Extensible XQuery Engine: Experiences with Galax (Extended Abstract)". In *Proc. XSYM*. 1–4.

Ferrante, J. and Rackoff, C. 1975. "A Decision Procedure for the First Order Theory of Real Addition with Order". *SIAM J. Comput.* **4**, 1, 69–76.

Florescu, D., Hillery, C., Kossmann, D., Lucas, P., Riccardi, F., Westmann, T., Carey, M. J., Sundararajan, A., and Agrawal, G. 2003. "The BEA/XQRL Streaming XQuery Processor". In *Proc. VLDB 2003*. 997–1008.

Gottlob, G., Koch, C., Pichler, R., and Segoufin, L. 2005. "The Complexity of XPath Query Evaluation and XML Typing". *Journal of the ACM* **52**, 2 (Mar.), 284–335.

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

Grumbach, S., Libkin, L., Milo, T., and Wong, L. 1996. "Query Languages for Bags – Expressive Power and Complexity". *SIGACT News* **27**, 2, 30–44.

Grumbach, S. and Milo, T. 1996. "Towards Tractable Algebras for Bags". *Journal of Computer and System Sciences* **52**, 3, 570–588.

Grumbach, S. and Vianu, V. 1995. "Tractable Query Languages for Complex Object Databases". *Journal of Computer and System Sciences* **51**, 2, 149–167.

Hartmanis, J., Lewis II, P. M., and Stearns, R. E. 1965. "Hierarchies of Memory Limited Computations". In *Proc. Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design*. 179–190.

Hidders, J., Paredaens, J., Verkammen, R., and Demeyer, S. 2004. "A Light but Formal Introduction to XQuery". In *Proc. XSYM*. 5–20.

Hull, R. and Su, J. 1989. "On Accessing Object-oriented Databases: Expressive Power, Complexity, and Restrictions". In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD'89)*. 147–158.

Hull, R. and Su, J. 1993. "Algebraic and Calculus Query Languages for Recursively Typed Complex Objects". *Journal of Computer and System Sciences* **47**, 1, 121–156.

Jaeschke, G. and Schek, H.-J. 1982. "Remarks on the Algebra of Non First Normal Form Relations". In *Proc. PODS'82*. 124–138.

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.

Koch, C. 2005a. "On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values". In *Proc. PODS'05*.

Koch, C. 2005b. "On the Role of Composition in XQuery". In *Proc. WebDB*.

Koch, C., Scherzinger, S., Schweikardt, N., and Stegmaier, B. 2004. "Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams". In *Proc. VLDB 2004*. Toronto, Canada.

Kuper, G. and Vardi, M. Y. 1993a. "On the Complexity of Queries in the Logical Data Model". *Theor. Comput. Sci.* **116**, 1&2, 33–57.

Kuper, G. and Vardi, M. Y. 1993b. "The Logical Data Model". *ACM Transactions on Database Systems* **18**, 3, 379–413.

Libkin, L. 2004. *Elements of Finite Model Theory.* Springer.

Libkin, L. and Wong, L. 1997. "Query Languages for Bags and Aggregate Functions". *Journal of Computer and System Sciences* **55***,* 2, 241–272.

Ludäscher, B., Mukhopadhyay, P., and Papakonstantinou, Y. 2002. "A Transducer-Based XML Query Processor". In *Proc. VLDB 2002.* 227–238.

Marian, A. and Siméon, J. 2003. "Projecting XML Documents". In *Proc. VLDB 2003.* 213–224.

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

Paredaens, J. and Van Gucht, D. 1988. "Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions". In *Proc. PODS.* 29–38.

Stockmeyer, L. J. 1974. The Complexity of Decision Problems in Automata Theory. Ph.D. thesis, Dept. Electrical Engineering, MIT, Cambridge, Mass., USA.

Suciu, D. and Tannen, V. 1997. "A Query Language for NC". *Journal of Computer and System Sciences* **55***,* 2, 299–321.

Tannen, V., Buneman, P., and Wong, L. 1992. "Naturally Embedded Query Languages". In *Proc. of the 4th International Conference on Database Theory (ICDT).* 140–154.

Van den Bussche, J. 2005. Personal communication.

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

Vorobyov, S. and Voronkov, A. 1998. "Complexity of Nonrecursive Logic Programs with Complex Values". In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'98).* 244–253.

Voronkov, A. 2004. Personal communication.

Wong, L. 1996. "Normal Forms and Conservative Extension Properties for Query Languages over Collection Types". *Journal of Computer and System Sciences* **52***,* 3, 495–505.

World Wide Web Consortium. 2005. "XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Candidate Recommendation (3 November 2005). http://www.w3.org/TR/xquery-semantics/.

XQueryUseCases 2005. "XML Query Use Cases. W3C Working Draft 15 September 2005". http://www.w3.org/TR/xquery-use-cases/.

A. ELECTRONIC APPENDIX

A.1 Reduction from Monad Algebra to Nonrecursive Logic Programming

PROOF OF THEOREM 5.2, SECOND VERSION. The proof is by a LOGSPACE-reduction to the *Success Problem* of nonrecursive logic programming with function symbols (but without sets), i.e. the problem of deciding whether a distinguished Boolean predicate evaluates to true. This problem is known to be NEXPTIME-complete [Dantsin and Voronkov 1997].

We show that every $\mathcal{M}_{\cup}[=_{atomic}]$ query can be reduced to a nonrecursive logic program with a single binary function symbol f. This is the function symbol used to build paths in the first proof of Theorem 5.2, and we use the same notation again.

Our predicates are binary and of the form $p(X, v)$, where X is a path prefix identifying a node w of the deterministic tree representation of our complex value, and v denotes one of the paths to leaves emanating from w, which together fully specify the complex value below node w.

—We translate an expression map(f) on path X represented by predicate $[\![Q]\!]$ into

$$\begin{aligned} [\![Q;\text{start\_map}]\!](X.i, v) &\leftarrow [\![Q]\!](X, i.v). \\ [\![Q;\text{map}(f)]\!](X, i.v) &\leftarrow [\![Q;\text{start\_map};f]\!](X.i, v). \end{aligned}$$

plus the translation of f mapping from predicate $[\![Q;\text{start\_map}]\!]$ to predicate $[\![Q;\text{start\_map; f}]\!]$. That is, on a value identified by path prefix X, we move down to the set member children of X, the $X.i$. Then we apply f on the values $X.i$, and finally, we return to X.

—We translate an expression $\langle A_1 : f_1, \ldots, A_k : f_k\rangle$ on path X represented by predicate $[\![Q]\!]$ into

$$\begin{aligned} [\![Q;\langle A_1 : f_1, \ldots, A_k : f_k\rangle]\!](X, A_1.v) &\leftarrow [\![Q;f_1]\!](X, v) \\ &\vdots \\ [\![Q;\langle A_1 : f_1, \ldots, A_k : f_k\rangle]\!](X, A_k.v) &\leftarrow [\![Q;f_k]\!](X, v) \end{aligned}$$

plus, for each $1 \leq i \leq k$, the translation of f_i mapping from predicate $[\![Q]\!]$ to predicate $[\![Q;f_i]\!]$.

—Compositions $f \circ g$, are read as $f;g$ and f and g are translated separately. The result predicate of f is used as the input predicate of g.

—The remaining operations are translated as follows.

$$[\![Q;c]\!](X, c) \leftarrow [\![Q]\!](X, v).$$

$$\begin{aligned} [\![Q;\text{pairwith}_B]\!](X, i.B.v) &\leftarrow [\![Q]\!](X, B.i.v). \\ [\![Q;\text{pairwith}_B]\!](X, i.A.v) &\leftarrow [\![Q]\!](X, A.v), [\![Q]\!](X, B.i.w). \end{aligned}$$

$$[\![Q;\text{flatten}]\!](X, (i.j).v) \leftarrow [\![Q]\!](X, i.j.v).$$

$$[\![Q;(A =_{atomic} B)]\!](X, 1.\langle\rangle) \leftarrow [\![Q]\!](X, A.v), [\![Q]\!](X, B.v).$$

$$\begin{aligned}
[\![Q;\pi_A \cup \pi_B]\!](X,(1.i).v) &\leftarrow [\![Q]\!](X,A.i.v)\\
[\![Q;\pi_A \cup \pi_B]\!](X,(2.i).v) &\leftarrow [\![Q]\!](X,B.i.v)
\end{aligned}$$

$$[\![Q;\pi_{A_i}]\!](X,v) \leftarrow [\![Q]\!](X,A_i.v)$$

$$[\![Q;\mathrm{sng}]\!](X,1.v) \leftarrow [\![Q]\!](X,v).$$

By Proposition 4.1, we may assume that our query ignores the input data; so we assume a predicate $[\![\epsilon]\!]$ and a fact $[\![\epsilon]\!](\epsilon,\mathrm{dummy}) \leftarrow .$ as part of our logic program.

It is not hard to verify that this translation of a query Q in $\mathcal{M}_\cup[=_{atomic}]$ into a nonrecursive logic program can be effected in LOGSPACE and that indeed the goal $[\![Q]\!](\epsilon, i.\langle\rangle)$ is true iff Q evaluates to true. $\Box$

We consider two examples to illustrate the construction of the logic programs. To save some space, however, we use short predicate names p_i.

Example A.1. The logic program for the query $\langle 1:0\circ\mathrm{sng}, 2:1\circ\mathrm{sng}\rangle\circ\cup$ is

$$\begin{array}{rcll}
[\![\epsilon]\!](\epsilon,\mathrm{dummy}) & \leftarrow & . & \\
p_1(X,0) & \leftarrow & [\![\epsilon]\!](X,v). & \#\ \text{constant 0}\\
p_2(X,1.v) & \leftarrow & p_1(X,v). & \#\ \text{sng}\\
p_3(X,1) & \leftarrow & [\![\epsilon]\!](X,v). & \#\ \text{constant 1}\\
p_4(X,1.v) & \leftarrow & p_3(X,v). & \#\ \text{sng}\\
p_5(X,1.v) & \leftarrow & p_2(X,v). & \#\ \text{create\_tuple}\\
p_5(X,2.v) & \leftarrow & p_4(X,v). & \#\ \text{create\_tuple}\\
p_6(X,(1.i).v) & \leftarrow & p_5(X,1.i.v). & \#\ \text{union}\\
p_6(X,(2.i).v) & \leftarrow & p_5(X,2.i.v). & \#\ \text{union}
\end{array}$$

The goal predicate p_6 computes the sets of paths of the deterministic tree representation of the result value, that is, $\{\pi \mid p_6(\epsilon,\pi) \text{ is true}\} = \{(1.1).0, (2.1).1\}$. $\Box$

Example A.2. On values of type $\{\langle A:\mathrm{Dom}, B:\mathrm{Dom}\rangle\}$ represented by predicate p_{input}, the query

$$\mathrm{map}(\langle C:\pi_A, D:\pi_B\circ\mathrm{sng}\rangle)$$

is encoded as the logic program

$$\begin{array}{rcll}
p_1(X.i,v) & \leftarrow & p_{input}(X,i.v). & \#\ \text{begin\_map}\\
p_2(X,v) & \leftarrow & p_1(X,A.v). & \#\ \pi_A\\
p_3(X,v) & \leftarrow & p_1(X,B.v). & \#\ \pi_B\\
p_4(X,1.v) & \leftarrow & p_3(X,v). & \#\ \text{sng}\\
p_5(X,C.v) & \leftarrow & p_2(X,v). & \#\ \text{create\_tuple}\\
p_5(X,D.v) & \leftarrow & p_4(X,v). & \#\ \text{create\_tuple}\\
p_6(X,i.v) & \leftarrow & p_5(X.i,v). & \#\ \text{end\_map}
\end{array}$$

with goal p_6. $\Box$

The reduction to nonrecursive logic programming of the proof of Theorem 5.2 (second version) can be rather easily extended to a reduction from $\mathcal{M}_\cup[=_{atomic}, \mathit{not}]$

to nonrecursive normal logic programming (that is, with negation). All we need to do is encode the operation "not" as

$$\begin{aligned} [\![Q;\text{not}]\!](X, 1.\langle\rangle) &\leftarrow \text{set}[\![Q]\!](X), \text{not nonempty}[\![Q]\!](X). \\ \text{nonempty}[\![Q]\!](X) &\leftarrow [\![Q]\!](X, v). \end{aligned}$$

where the "set$[\![Q]\!]$" predicates are defined alongside the $[\![Q]\!]$ predicates such that set$[\![Q]\!](X)$ is true iff X is the path prefix of a set, empty or not. This reduction is not in LOGLIN because of the size of the predicates generated. Even if we replace the predicate names by shorter ones of the form p_i (where i is an integer), they are of size $\log n$ each (where n is the size of the input query in monad algebra) and the overall size of the logic program is $O(n \cdot \log n)$. (There are linearly many rules.) But since we can compose this preparation with an ATM run and nonrecursive range-restricted normal logic programming is known to be in TA$[2^{O(n)}, O(n)]$ [Vorobyov and Voronkov 1998], this shows that

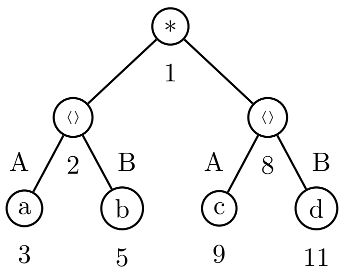
Corollary A.3. *$\mathcal{M}_{\cup}[=_{atomic}, not]$ is in TA$[2^{O(n \cdot \log n)}, O(n \cdot \log n)]$ w.r.t. query complexity.*

A.2 Data Complexity

Proof of Proposition 6.1. For simplicity, we will here assume that all tuples are pairs, but the proof immediately generalizes to tuples of higher arity.

We assume complex values given as strings constructed using symbols from alphabet Σ consisting of $\langle$, $\rangle$, $\{$, $\}$, ",", and character symbols for atomic values.

For example, the value



of type $\{\langle A : \text{Dom}, B : \text{Dom}\rangle\}$ is represented as string "$\{\langle a, b\rangle, \langle c, d\rangle\}$".

Given a complex value v, we identify (set-, pair-, and atomic) terms of v (i.e., nodes of the tree shown above) by the index of the first symbol of the term in the input string. For instance, the root node is identified with index 1 because the opening curly brace is the first symbol of the input string. Conversely, if i is the index of term t in value v, then we use $t_v(i)$ to denote t.

Let $I_v = \{1, \ldots, |v|\}$. Let *flat* be a function that maps every complex value v of type τ to a relational structure $\langle Set, Pair, Atomic\rangle$ with relations $Set \subseteq I_v^2$, $Pair \subseteq I_v^3$, and $Atomic \subseteq I_v \times Dom$, where $\langle x, y\rangle \in Set$ iff there is a set-term $t_v(x)$ in v that has a term $t_v(y)$ as member, $\langle x, y, z\rangle \in Pair$ iff there is a pair-term $t = \langle t_1, t_2\rangle$ in v with $t = t_v(x)$, $t_1 = t_v(y)$, and $t_2 = t_v(z)$, and $\langle x, w\rangle \in Atomic$ iff there is an atomic term $t = w$ in v with $t = t_v(x)$.

For the example value discussed above,

$$\begin{aligned} \mathit{Atomic} &= \{\langle 3, a\rangle, \langle 5, b\rangle, \langle 9, c\rangle, \langle 11, d\rangle\} \\ \mathit{Set} &= \{\langle 1, 2\rangle, \langle 1, 8\rangle\} \\ \mathit{Pair} &= \{\langle 2, 3, 5\rangle, \langle 8, 9, 11\rangle\} \end{aligned}$$

We have the power of TC_0 at hand to define a reduction from our input strings to the flat relations. We will not go into the details of a TC_0 reduction (cf. [Barrington et al. 1990]), these are technical but in this case easy. The only point worth mentioning is that we can check whether two indexes i, j are the left and right delimiters of a set or tuple. We show this in FO logic with majority quantifiers (FOM). By [Barrington et al. 1990], $\mathrm{TC}_0 = \mathrm{FOM}$. It is also known [Barrington et al. 1990] that FOM can express predicates $x = y + z$ and $x = \#y\, \phi(y)$, such that x is the number of positions y for which $\phi(y)$ holds.

$$\begin{aligned} \text{set-node}(i, j) &:= Q_{\{}(i) \wedge Q_{\}}(j) \wedge \\ &\quad x = \#u\big(Q_{\{}(u) \wedge i < u < j\big) \wedge \\ &\quad y = \#u\big(Q_{\}}(u) \wedge i < u < j\big) \wedge x = y \\ \text{tuple-node}(i, j) &:= Q_{\langle}(i) \wedge Q_{\rangle}(j) \wedge \\ &\quad x = \#u\big(Q_{\langle}(u) \wedge i < u < j\big) \wedge \\ &\quad y = \#u\big(Q_{\rangle}(u) \wedge i < u < j\big) \wedge x = y \end{aligned}$$

where $(Q_a)_{a \in \Sigma}$ represents the input string and $Q_a(i)$ is true iff symbol a is at position i of the input string. (That is, these formulae state that i, j are positions of symbols with matching opening and closing delimiters and the number of opening delimiters occurring between i and j is the same as the number of closing delimiters.)

Atomic nodes can already be defined in FO. Let "node" denote nodes of any of the three kinds. Now, for instance,

$$\begin{aligned} \phi_{\mathit{Set}}(i, j) := \exists i', j'\ &\text{set-node}(i, i') \wedge \text{node}(j, j') \wedge i < j \wedge j' < i' \wedge \\ &\neg \exists k, k'\ \text{node}(k, k') \wedge i < k < j \wedge j' < k' < i'. \end{aligned}$$

This formula states that i is the identifier of a set-node and j the identifier of one of its children.

Let the $\mathcal{M}_{\cup}[\sigma]$ query V_τ for the type of the input data be defined inductively as

$$\begin{aligned} V_{\mathrm{Dom}} &:= \mathit{Atomic} \circ \mathrm{map}(\langle 1 : \pi_1, 2 : \pi_2 \circ \mathrm{sng}\rangle) \\ V_{\langle A:\tau_1, B:\tau_2\rangle} &:= \mathit{Pair} \circ \mathrm{map}(\langle 1 : \pi_1, 2 : V_{\tau_1}|\pi_2 \times V_{\tau_2}|\pi_3\rangle) \\ V_{\{\tau\}} &:= \mathit{Set} \circ \langle 1 : \mathrm{map}(\pi_1), 2 : \mathrm{id}\rangle \circ \mathrm{pairwith}_1 \circ \\ &\quad \mathrm{map}\big(\langle 1 : \pi_1, 2 : \pi_2|\pi_1 \circ (\mathrm{id} \times V_\tau) \circ \\ &\quad \sigma_{1=2.1} \circ \mathrm{map}(\pi_{2.2}) \circ \mathrm{flatten} \circ \mathrm{sng}\rangle\big) \end{aligned}$$

where $S|v = \langle 1 : v, 2 : S\rangle \circ \mathrm{pairwith}_S \circ \sigma_{1=2.1} \circ \mathrm{map}(\pi_{2.2})$.

For our example, we get V_τ as shown in Figure 11.

It is not difficult to verify that for every complex value v of type τ, $V_\tau(\mathit{flat}(v)) = \{\langle 1 : i, 2 : \{v\}\rangle\}$, where i is the identifier of the root of v, and that $V' := V_\tau \circ \mathrm{map}(\pi_2) \circ \mathrm{flatten}$ computes $\{v\}$.

$$
\begin{aligned}
V_{\mathrm{Dom}} &= \{\langle 3, \{a\}\rangle, \langle 5, \{b\}, \langle 9, \{c\}\rangle, \langle 11, \{d\}\rangle\rangle \\
V_{\langle A:\mathrm{Dom}, B:\mathrm{Dom}\rangle} &= \{\langle 2, 3, 5\rangle, \langle 8, 9, 11\rangle\} \circ \mathrm{map}(\langle 1 : \pi_1, 2 : V_{\mathrm{Dom}}|\pi_2 \times V_{\mathrm{Dom}}|\pi_3\rangle) \\
&= \{\langle 2, V_{\mathrm{Dom}}|3 \times V_{\mathrm{Dom}}|5\rangle, \langle 8, V_{\mathrm{Dom}}|9 \times V_{\mathrm{Dom}}|11\rangle\} \\
&= \{\langle 2, \{a\} \times \{b\}\rangle, \langle 8, \{c\} \times \{d\}\rangle\} \\
&= \{\langle 2, \{\langle a, b\rangle\}\rangle, \langle 8, \{\langle c, d\rangle\}\rangle\} \\
V_{\{\langle A:\mathrm{Dom}, B:\mathrm{Dom}\rangle\}} &= \{\langle 1, 2\rangle, \langle 1, 8\rangle\} \circ \langle 1 : \mathrm{map}(\pi_1), 2 : \mathrm{id}\rangle \circ \mathrm{pairwith}_1 \circ \\
&\quad \mathrm{map}\big(\langle 1 : \pi_1, 2 : \pi_2|\pi_1 \circ (\mathrm{id} \times V_{\langle A:\mathrm{Dom}, B:\mathrm{Dom}\rangle}) \circ \sigma_{1=2.1} \circ \\
&\quad \mathrm{map}(\pi_{2.2}) \circ \mathrm{flatten} \circ \mathrm{sng}\rangle\big) \\
&= \langle 1 : \{1\}, 2 : \{\langle 1, 2\rangle, \langle 1, 8\rangle\}\rangle \circ \mathrm{pairwith}_1 \circ \\
&\quad \mathrm{map}\big(\langle 1 : \pi_1, 2 : \pi_2|\pi_1 \circ (\mathrm{id} \times V_{\langle A:\mathrm{Dom}, B:\mathrm{Dom}\rangle}) \circ \sigma_{1=2.1} \circ \\
&\quad \mathrm{map}(\pi_{2.2}) \circ \mathrm{flatten} \circ \mathrm{sng}\rangle\big) \\
&= \{\langle 1 : 1, 2 : \{\langle 1, 2\rangle, \langle 1, 8\rangle\}\rangle\} \circ \\
&\quad \mathrm{map}\big(\langle 1 : \pi_1, 2 : \pi_2|\pi_1 \circ (\mathrm{id} \times V_{\langle A:\mathrm{Dom}, B:\mathrm{Dom}\rangle}) \circ \sigma_{1=2.1} \circ \\
&\quad \mathrm{map}(\pi_{2.2}) \circ \mathrm{flatten} \circ \mathrm{sng}\rangle\big) \\
&= \{\langle 1 : 1, 2 : \{2, 8\} \circ (\mathrm{id} \times V_{\langle A:\mathrm{Dom}, B:\mathrm{Dom}\rangle}) \circ \sigma_{1=2.1} \circ \\
&\quad \mathrm{map}(\pi_{2.2}) \circ \mathrm{flatten} \circ \mathrm{sng}\rangle\} \\
&= \{\langle 1 : 1, 2 : (\{2, 8\} \times \{\langle 2, \{\langle a, b\rangle\}\rangle, \langle 8, \{\langle c, d\rangle\}\rangle\}) \circ \sigma_{1=2.1} \circ \\
&\quad \mathrm{map}(\pi_{2.2}) \circ \mathrm{flatten} \circ \mathrm{sng}\rangle\} \\
&= \{\langle 1 : 1, 2 : \{\langle 2, \langle 2, \{\langle a, b\rangle\}\rangle\rangle, \langle 8, \langle 8, \{\langle c, d\rangle\}\rangle\rangle\} \circ \\
&\quad \mathrm{map}(\pi_{2.2}) \circ \mathrm{flatten} \circ \mathrm{sng}\rangle\} \\
&= \{\langle 1 : 1, 2 : \{\{\langle a, b\rangle\}, \{\langle c, d\rangle\}\} \circ \mathrm{flatten} \circ \mathrm{sng}\rangle\} \\
&= \{\langle 1 : 1, 2 : \{\{\langle a, b\rangle, \langle c, d\rangle\}\}\rangle\}
\end{aligned}
$$

Fig. 11. V_τ for the running example.

By Theorem 2.5, for every $\mathcal{M}_\cup[\sigma]$ query from flat relations to flat relations there is an equivalent relational algebra query. Thus, for any Boolean $\mathcal{M}_\cup[\sigma]$ query Q, there is a relational algebra query $Q' \equiv V' \circ \mathrm{map}(Q) \circ \mathrm{flatten}$. Of course,

$$Q(v) \Leftrightarrow (V' \circ \mathrm{map}(Q) \circ \mathrm{flatten})(\mathit{flat}(v)) \Leftrightarrow Q'(\mathit{flat}(v)).$$

For a fixed query Q (and thus a fixed type τ), Q' is fixed and can be evaluated on a (flat relational) database in AC_0 (cf. e.g. [Libkin 2004; Abiteboul et al. 1995]) and thus in TC_0. Preprocessing function *flat* is in TC_0, so we can compose these two steps and get a TC_0 overall bound. $\square$