Monadic Queries over Tree-Structured Data[*]

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

Abstract

Monadic query languages over trees currently receive considerable interest in the database community, as the problem of selecting nodes from a tree is the most basic and widespread database query problem in the context of XML. Partly a survey of recent work done by the authors and their group on logical query languages for this problem and their expressiveness, this paper provides a number of new results related to the complexity of such languages over so-called axis relations (such as "child" or "descendant") which are motivated by their presence in the XPath standard or by their utility for data extraction (wrapping).

1 Introduction

Speaking in terms of logic, the distinguishing features of a *monadic query* are its single free variable, by which it selects a subset of the domain of the input database, and that it is defined in a monadic logic (where all predicate variables are of arity one).

In this paper, we primarily study two query languages, *monadic conjunctive queries* and *monadic datalog* over trees, and do this in the context of unranked, node-labeled trees with an implicit ordering of sibling nodes, which are a convenient abstraction of Web documents and XML. Monadic conjunctive queries are conjunctive queries with unary head predicates. Monadic datalog programs are sets of such conjunctive queries interpreted as datalog rules. Over tree-structured data, both query languages are natural formalisms for selecting a subset of the nodes of a tree.

The significance of monadic query languages applied to tree-structured data (such as XML) is quite obvious: The primary purpose of XPath – a much-hyped and now heavily used W3C standard – and similar languages is to select nodes of a document (or, equivalently, subtrees rooted by these nodes). XPath is used as a free-standing query language, but is also incorporated into several other important XML-related W3C standards, including XPointer [36], XSLT [35], XML Schema [37], and the XML Query language [33], in which it constitutes much of the core XML data access functionality.

Apart from the selection of nodes, an interesting application of monadic query languages is the extraction of information from Web document trees (*tree wrapping*) [27, 19, 4, 3]. Wrappers can be defined as sets of *information extraction functions* (monadic queries) which filter or relabel tree nodes [16]. Monadic query languages used to define such wrappers require considerable expressive power. In an earlier paper [16], we studied the expressiveness of monadic datalog over trees, and discovered that this language exactly captures monadic second-order logic (MSO) over trees.

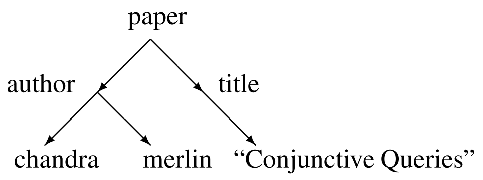
Our focus on (monadic) conjunctive queries and datalog as query languages in concert is not accidental. On one hand, monadic conjunctive queries are the building blocks (rules) of monadic datalog programs and can be studied and used independently. On the other hand, monadic datalog will serve us as a very natural formalism for encoding and efficiently evaluating monadic conjunctive queries, problems this paper will elaborate on.

We also define and study a language that constitutes the logical core of XPath (and which we thus name *Core XPath*). As we shall show, Core XPath nicely fits into our logical framework, as it compares to acyclic monadic conjunctive queries much in the same way as first-order queries compare to conjunctive queries in the classical relational database setting.

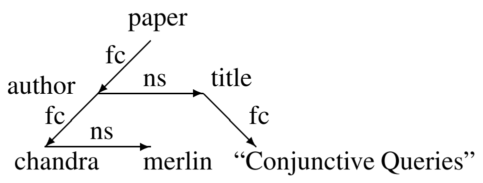
In this paper, we give an account of previous work on monadic queries over trees and extend the state of the art in several respects. We do not aim to surpass the expressiveness of monadic datalog (and thus MSO), but carry out further complexity studies for the languages proposed. While we have previously shown that monadic datalog over trees can be evaluated in time linear in the query and in the data, respectively, this result (as all linear-time results) heavily

[*]This work was supported by the Austrian Science Fund (FWF) under project No. Z29-INF and by the EU Research Training Network GAMES.

depends on the way the tree data is represented. To obtain our result, trees were represented in terms of the natural "firstchild", "nextsibling", and labeling built-in relations. For instance, the data tree



which models an XML document containing information about research papers is represented as



using firstchild ("fc") and nextsibling ("ns") relations.

An interesting question (studied in this paper) is whether we may generalize the built-in binary relations used in queries to XPath-like *axis relations*, which include transitive relations such as "descendant" of quadratic size, and still retain the very favorable (linear-time) computational characteristics. While such extensions do not further the expressive power of monadic datalog (it remains equivalent to MSO), they improve the practical usefulness of our query languages.

For instance, consider the query for the titles of papers co-authored by "chandra" and "merlin" in an appropriate XML document. In XPath, this query can be formulated as

paper[author[chandra and merlin]]/title

which, equivalently, we can write as

$$\begin{aligned} (\quad) \qquad \text{paper}(P),\ & \text{child}(P, A),\ \text{author}(A), \\ & \text{child}(A, \quad),\ \text{chandra}(\quad), \\ & \text{child}(A, M),\ \text{merlin}(M), \\ & \text{child}(P, \quad),\ \text{title}(\quad). \end{aligned}$$

as a monadic conjunctive query (using "child" and label predicates). "child" is an axis relation, and it would be painful (but possible, provided that we may use monadic datalog rather than a *single* monadic conjunctive rule to formulate our request) to go without it and use the predicates "firstchild" and "nextsibling" instead.

Unfortunately, our earlier linear-time result of [16] does not directly extend to the mentioned built-ins. Currently, it is not even known whether the query/combined complexity of the obtained query evaluation problem is polynomial.

Related Work

Based on a wealth of elegant equivalences between regular tree languages, logics (MSO), and automata (cf. [28, 29, 6]), there has recently been considerable research activity related to various theoretical aspects of XPath and XSLT-like query languages [25, 7, 5, 22, 24, 16].

However, except for [17], which shows that XPath has polynomial-time combined complexity, and the linear-time combined complexity result for monadic datalog over trees of [16], no research results on good or even reasonable algorithms for processing XPath and similar languages have been published which may serve as yardsticks for new algorithms. Our emphasis here is on the *combined complexity* of query evaluation (i.e., both query and data are considered variable); the linear data complexity of query languages captured by MSO follows from Courcelle's Theorem and has been known for a long time.

Note that unlike the setting of conjunctive queries in relational databases, query containment and query evaluation for XPath-like languages are problems that differ strongly in their nature and hardness. It has to be emphasized that in this paper, we address exclusively the query evaluation problem, and refer to [11, 21, 32] for work on query containment for (fragments of) XPath.

Contributions

- We give an overview of earlier results on monadic datalog, particularly concerning its expressive power.
- We show that acyclic monadic datalog rules over complex regular path expression relations can be efficiently rewritten into rules that use only the primitive relations of these expressions. Regular path expressions allow to conveniently express all of the XPath axis relations. Thus, all monadic datalog programs that consist solely of acyclic rules can be evaluated in linear time.
- We present the Core XPath language and, building on earlier work presented in [17], show that is has linear-time combined complexity. We describe analogies of Core XPath with monadic datalog over trees and CTL.
- We show that monadic datalog rules over "child", "firstchild", and "nextsibling" built-ins can be efficiently rewritten into acyclic rules (using transitive closure relations) with a simple *Chase*-like procedure. Consequently, we can give a most favorable answer to the previously open question concerning the combined complexity of Elog [16], a practically useful Web wrapping language, which is linear.
- In contrast to the previous contributions, we then point out limits to tractability. We show three closely related query evaluation problems NP-complete, namely

  - boolean conjunctive queries over regular path relations defined in terms of “firstchild” and “nextsibling” on trees,
  - acyclic boolean conjunctive queries with variables that may range over tree nodes *and* their labels (on trees), and
  - boolean conjunctive queries over the edge relation of DAGs.

- Finally, we show that monadic datalog (without the acyclicity restriction) over transitive axes such as descendant or ancestor-or-self has polynomial-time combined query evaluation complexity.

Structure of the Paper

In Section 2, we provide basic notions. Section 3 discusses earlier results on monadic datalog over trees. In Section 4, we present our results on acyclic queries. Section 5 addresses Core XPath. Section 6 presents a method for eliminating the “child” relation, leading to the linear-time combined complexity result for Elog . Section 7 discusses our three NP-completeness results. Finally, in Section 8, we conclude with a study of “descendant” queries.

2 Preliminaries and Basic Notions

2.1 Trees

Throughout this paper, only *finite unranked* trees will be considered. Trees are defined in the normal way and have at least one node. We assume that the children of each node are in some fixed order. Each node has a label taken from a finite nonempty set of symbols , the alphabet. Each unranked tree can be considered as a relational structure[1]

$$t \quad = \langle \text{dom}, \text{root}, \text{leaf}, (\text{label}\;)_{\;\in\;}, \text{firstchild}, \text{lastchild}, \text{nextsibling}\rangle$$

where dom is the set of nodes in the tree and the relations are defined according to their intuitive meanings. For instance, firstchild is binary and denotes the relation between nodes and their first children (according to our ordering) and nextsibling(n_1, n_2) is true iff n_1 and n_2 are the i-th and $(i+1)$-th children of a common parent node, respectively. $l \quad l\;(n)$ is true iff n is labeled in the tree.

Monadic second-order logic (MSO) over trees is a second-order logical language consisting of (1) variables (with lower-case names , , …) ranging over nodes, (2) set variables (written using upper-case names P, , …) ranging over sets of nodes, (3) parentheses, (4) boolean connectives and , (5) quantifiers $\forall$ and $\exists$ over both node and set variables, (6) the relation symbols of the model-theoretic tree structure in consideration, = (equality of node variables), and, as syntactic sugaring, possibly (7) the boolean operations , $\rightarrow$, and and the relation symbols $=$ and $\subseteq$ between sets.

Π_1-MSO refers to MSO sentences of the form

$$\forall P_1, \ldots, P_n : \quad (P_1, \ldots, P_n)$$

where the P are set variables and is a first-order formula. It is easy to define a natural total ordering of dom in MSO (obtained by depth-first left-to-right traversal of the tree, where, say, parents precede children), which is also called the *document order* in the context of wrapping HTML documents (see e.g. [34]). A unary MSO *query* is a unary predicate definable in MSO (i.e., by a formula with one free first-order variable). A tree language is definable in MSO iff there is a closed MSO formula φ over tree structures t such that $= \{t \mid t \qquad \varphi\}$.

The regular tree languages are precisely those recognizable by a number of finite automata, such as nondeterministic descending (or *top-down*) tree automata (NDTA), both nondeterministic (NATA) and deterministic (DATA) ascending (or *bottom-up*) tree automata [6], and deterministic (2DTA) as well as nondeterministic (2NTA) two-way tree automata [6, 25]. We provide a definition of deterministic bottom-up tree automata.

Definition 2.1 A deterministic bottom-up *unranked* tree automaton is a tuple $= \langle \;,\;,\;, F\rangle$ where is a finite set of states, is an (unranked) alphabet, is a (partial) transition function $\times \quad \rightarrow \quad ^*$ s.t. $(\;,\;)$ is a regular language over states in , and $F \subseteq$ is a set of final states.

The semantics of on a tree t, $^*(t)$, is defined inductively as follows: If t consists of only a leaf node labeled , and $(\;,\;) =$ for some , then $^*(t) =$. If t is labeled and has the children $t_1, \ldots, t$ with $^*(t_1) = \;_1, \ldots, \;^*(t\;) =$, and there is some such that $_1 \qquad \in \;(\;(\;,\;))$, then $^*(t) =$. As before, a run is called successful for some tree t (that is, the tree is *accepted*) iff $^*(t) \in F$. The set of -trees accepted by is denoted by $(\;)$.

Definition 2.2 A tree language is regular iff is is accepted by some deterministic bottom-up tree automaton.

The following is a classical result for ranked trees, which has been shown in [25] to hold for unranked trees as well (see also [6]).

Proposition 2.3 *A tree language is regular iff it is definable in MSO.*

[1] Of course, equally well-suited structures are obtained by adding predicates definable over the structure in some formalism or removing others that are redundant (w.r.t. definability).

Remark 2.4 In the context of representing HTML or XML documents in our data model of unranked trees, it is worthwhile to consider an *infinite* alphabet , which allows to merge both HTML tags and attribute assignments into labels. This requires a generalized notion of relational structures $\langle \mathrm{dom}, R_1, R_2, R_3, \ldots \rangle$ consisting of a domain dom and a countable (but possibly *infinite*) set of relations, of which only a finite number is nonempty. Even though all results cited or shown in this paper (such as Proposition 2.3) were proven for finite alphabets, it is trivial to see that they also hold for infinite alphabets in case the symbols of the alphabet (i.e., the node labels) are not part of the domain, and labels of domain elements are expressed via predicates (such as the label) only. Given these requirements, it is impossible to quantify over symbols of and any query in whatever language can only refer to a finite number of symbols of . (See the related discussion in the preliminaries of [24].) Note that MSO with an alternative, more adventuresome encoding via a tree structure with a binary relation "label" and a two-sorted domain of nodes and labels allows to encode problems hard for any level of the polynomial hierarchy. In this paper, we avoid this problem by assuming a finite set . Attribute assignments can be encoded, for instance, as lists of character symbols modeled as subtrees in our document tree.

2.2 Conjunctive Queries and Query Graphs

We re-state a few basic definitions from database theory. We are exclusively interested in *monadic conjunctive queries* over base predicates whose arity does not exceed two and refer to them simply as CQ's throughout this section. A CQ is of the form

$$\begin{aligned} (\;) \qquad & P_1(\;_1),\; \ldots,\; P_n(\;_n), \\ & R_1(\;_1,\;_1),\; \ldots,\; R\;(\;\;,\;\;). \end{aligned}$$

where $\;,\;_1,\ldots,\;_n,\;_1,\ldots,\;\;,\;_1,\ldots,$ are not necessarily distinct variables, $P_1, \ldots, P_n$ are (not necessarily distinct) unary predicates, and, finally, the $R_1, \ldots, R$ are (not necessarily distinct) binary predicates. Everything at the right of the arrow is called the *body* of , denoted Body( ). is called the *distinguished* or *head* variable and occurs in the body, and ( ) is called the head. The body consists of a *conjunction* of *atoms*, written as a comma-separated list.[2]

Informally, a satisfaction for query is a mapping : Vars( ) $\rightarrow$ dom such that when the variables of are replaced in its body using , the conjunction Body( ) becomes true. If is the head variable of , the query returns node n iff there exists a satisfaction for with $(\;) = n$.

[2]We assume that it is maintained in set fashion and we do not need to deal with duplicates.

The *query graph* of a CQ is the (undirected) graph obtained by taking the variables of , Vars( ), as nodes and the binary atoms of as edges. Optionally, we may annotate the nodes of this graph with the unary atoms $P\,(\;)$.

We assume the reader familiar with the common notions of graphs, cycles, forests, and trees in their *directed* as well as *undirected* forms. If not stated explicitly otherwise, we use the undirected notions of each of these terms *when talking about query graphs*. However, binary relations such as "child" or "nextsibling" used to represent trees have an intuitive "direction". To improve the information content of our figures, we will visualize this direction through arrows; all query graphs are to be read as undirected nevertheless.

Example 2.5 The query graph of the query from our introductory example (all edges are of type "child") is acyclic.

$$\begin{tikzcd}
 & & P \arrow[dl] \arrow[dr] & \\
 & A \arrow[dl] \arrow[dr] & & {} \\
{} & & M &
\end{tikzcd}$$

The arrows shown are used to visualize the meaning of the query; arrows point from parents to children.

Note that a directed acyclic graph w.r.t. such arrows would be cyclic in the undirected interpretation and would thus count as a cyclic query graph.

A CQ is called *acyclic* iff its query graph is acyclic. (Thus, our notion is a special case of but compatible to the wealth of previous work on hypergraph acyclicity, see e.g. [1].) Of course, an (undirected) graph is acyclic iff it is a forest. A forest is a tree if it is connected. Given a CQ , variable is an *ear* of iff occurs only in a single binary atom.

3 Monadic Datalog over Trees

In this paper, we will use a syntactically restricted fragment of standard datalog [30, 8]. We adhere to the usual minimal model (= least fixpoint) semantics, which can be defined using, say, the immediate consequence operator . We will also use the extension of datalog by stratified negation in the standard way.

Syntactically, a monadic datalog program is a set of monadic conjunctive queries (called rules). Predicates that appear in the head of some rule of a program are called *intensional*, all others are called *extensional*. An *extension* is a set of ground atoms that are assumed to be true. *Built-ins* are extensional predicates with a fixed extension given as input. By *signature* we denote a finite set of built-in predicates. By default, we will always use the signature

$$\tau \quad = \langle \text{root},\ \text{leaf},\ (\text{label}\quad)\ \in\quad ,\\ \text{firstchild},\ \text{lastchild},\ \text{nextsibling}\rangle$$

for representing trees. t denotes the extension of τ .

In order to be able to compare MSO with monadic datalog, we make a few assumptions. By (unary) *query*, for monadic datalog as for MSO, we denote a function that assigns a predicate to some nodes of a tree t (or, in other words, selects a subset of dom(t)).

The following result is part of the database folklore:

Proposition 3.1 *Over arbitrary finite structures, each monadic datalog query is Π_1-MSO-definable.*

In fact, MSO and monadic datalog are *equivalent* in their power to define unary queries. This was shown in [16] by simulating the query automata of [25], which are known to capture the expressiveness of MSO to define unary queries, in monadic datalog.

Proposition 3.2 [16] *For each unary MSO-definable query there is a monadic datalog query (over τ) s.t. for all trees, these two queries return the same result.*

We say that a monadic datalog program with some dedicated intensional predicate (say, "accept") accepts a tree t iff accept$(r) \in \quad (t)$ (i.e., the root node is in the inferred extension of "accept"). A monadic datalog program recognizes the tree language $L = \{t \mid \quad \text{accepts } t\}$.

The following result is similar to the folklore result that monadic fixpoint logic over trees expresses MSO (w.r.t. tree language acceptance), and can be shown by a straightforward simulation of bottom-up tree automata in monadic datalog.

Proposition 3.3 [16] *A tree language is regular iff it is definable in monadic datalog.*

Regarding the complexity of our language, it has been shown (as a generalization of Courcelle's theorem) that the data complexity of MSO *queries* over finite structures of bounded tree-width is in linear time [14]. Unranked labeled trees are of tree-width $\leq \quad + |\quad|$. Surprisingly, it can be shown that monadic datalog queries over trees can be evaluated in time linear in the size of the query and the data, respectively.

Proposition 3.4 [16] *Monadic datalog over τ (with stratified negation) has $(|\quad| * |dom(t)|)$ combined complexity (where $|\quad|$ is the size of the input program and $|dom(t)|$ is the size of the tree), linear-time data complexity, and its program complexity is complete for linear time.*

Proof Sketch Observe that all of the binary predicates in τ (that is, firstchild, lastchild, and nextsibling) have both a functional dependency from their first column to the second and one from the second column to the first.

Given a monadic datalog program , let us apply the following transformation. For each rule, we split off connected parts that do not contain the head variable, create a rule with a propositional head predicate for them, and add the propositional predicates to the original rule as replacements of the removed parts. For instance, $p(X) \quad p_1(X),\ p_2(\quad).$ is rewritten into $p(X) \quad p_1(X),\ \ .$ and $\quad p_2(\quad).$ It is easy to see that this transformation is linear in the size of the program.

Now, for each rule in the transformed program $'$, each variable functionally determines all others, as the variables are connected via binary predicates that are one-to-one only. (That is, in a graph of functional dependencies, each variable is reachable from each other variable.) Consequently, there is only a linear number of relevant ground instances (in the size of the data), which can be computed in linear time. It is known from [12, 23] (see also [10]) that the fixpoint of a ground program can be computed in time linear in the size of the program. This fact generalizes to ground programs with stratified negation, as we just need to process the strata one after another using, say, the algorithm of [23]. Thus, the composition of these steps requires $(|\quad| * |\text{dom}(t)|)$ time. This is an upper bound for the combined complexity of the problem, and we have both linear time data and program complexities.

Finally, the problem is P-hard with respect to program complexity, because it is a generalization of the propositional Horn-SAT problem, which is P-complete (cf. [26]).

4 Acyclic Conjunctive Queries

A *regular path expression* (cf. [2]) over a set of binary relations is a regular expression (using concatenation ".", the Kleene star "*", and disjunction "|") over alphabet . A regular path expression *with inversion* furthermore supports expressions of the form $R^{\ 1}$ and is inductively interpreted as a binary relation as follows.

$$\begin{array}{rcl} R_1.R_2 & := & \{\langle\ ,\ \rangle \mid \exists\ :\langle\ ,\ \rangle \in R_1, \langle\ ,\ \rangle \in R_2\} \\ R_1|R_2 & := & R_1 \quad R_2 \\ R^* & \ldots & \text{the reflexive and transitive closure of } R \\ R^{\ 1} & := & \{\langle\ ,\ \rangle \mid \langle\ ,\ \rangle \in R\} \end{array}$$

Finally, R^+ is a shortcut for $R.R^*$.

Example 4.1 For instance, the so-called *document order* relation $<$ used in several XML-related standards (such as XPath [34]) is defined as the order in which the opening tags of document tree nodes are first reached when reading the document (as a flat file) from left to right. We can encode the document order as a regular path expression over τ as

$((\text{nextsibling}\ \ ^{1})^{*}.\text{firstchild}\ \ ^{1})^{*}.\text{nextsibling}^{+}.$
$(\text{firstchild.nextsibling}^{*})^{*}$

Lemma 4.2 *Let be a set of binary relations, R be the relation defined by a regular path expression with inversion over , and let p be a unary predicate. Then, there is a linear-time algorithm for computing a monadic datalog program over which defines the unary predicate*

$$p.R := \{\ \ \mid \exists\ _{0} : p(\ _{0}) \text{ is true and } \langle\ _{0},\ \ \rangle \in R\}.$$

Proof (Sketch). Each regular (path) expression with inversion can be translated into a nondeterministic finite automaton with -transitions (denoted NFA) over the alphabet in linear time. The construction for the concatenation, star, and disjunction operators can be found in any introductory book on automata theory (e.g. [18]). We only present the construction for inversion here.

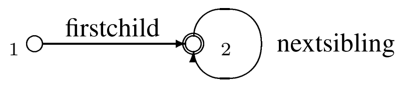
Let $= \langle\ ,\ , F,\ \rangle$ ($\in$, $F \subseteq$, $\subseteq$ $\times$ ($\{\ \}) \times$) be an NFA representing a regular path expression . To build an NFA for ^{1}, we proceed as follows. If $|F|$ 1, we first translate into an equivalent automaton $'$ with only a single final state (a new state f'), by linking each $f \in F$ with f' via an -transition. Then, we obtain an inverse automaton of $'$ by replacing each transition $\langle\ _{1}, r,\ _{2}\rangle$ of the transition relation of $'$ by $\langle\ _{2},\ ,\ _{1}\rangle$ if $r =$ and $\langle\ _{2}, r\ \ ^{1},\ _{1}\rangle$ otherwise ($(r\ \ ^{1})\ \ ^{1}$ is replaced by r). Moreover, we exchange the first and the final state.

Finally, we create a monadic datalog program defining $p.R$ on the basis of the NFA $= \langle\ ,\ , F,\ \rangle$ as follows.

1. $(\ )\quad p(\ ).$
2. for each tuple $\langle\ _{1},\ ,\ _{2}\rangle \in$,
$$_{2}(\ )\quad\ _{1}(\ ).$$
3. for each tuple $\langle\ _{1}, r,\ _{2}\rangle \in$, $r \in$,
$$_{2}(\ )\quad\ _{1}(\ _{0}),\ r(\ _{0},\ ).$$
4. for each tuple $\langle\ _{1}, r\ \ ^{1},\ _{2}\rangle \in$, $r \in$,
$$_{2}(\ )\quad\ _{1}(\ _{0}),\ r(\ ,\ _{0}).$$
5. for each $f \in F$,
$$p.R(\ )\quad f(\ ).$$

It is easy to verify that this encoding is indeed correct, i.e., that node n can be reached from a node n_0 s.t. $p(n)$ is true via a path matching R if and only if the datalog program produced by the above encoding computes $p.R(n)$.

Example 4.3 Let the relation "child" be defined by the regular path expression firstchild.nextsibling* over τ . Given a unary predicate p, the NFA for p.child is



Our monadic datalog encoding of p.child is

$$\begin{aligned} _{1}(X) &\quad p(X). \\ _{2}(X) &\quad\ _{1}(X_0),\ \text{firstchild}(X_0, X). \\ _{2}(X) &\quad\ _{2}(X_0),\ \text{nextsibling}(X_0, X). \\ p.\text{child}(X) &\quad\ _{2}(X). \end{aligned}$$

Example 4.4 We may also use Lemma 4.2 for regular path expressions *with labels*, such as

$$(\text{child.label}\ \ )^{+}.\text{child.label}\ \ ,$$

where label relations are defined as

$$\text{label}\ \ := \{\langle\ ,\ \ \rangle \mid\ \ \in \text{label}\ \ \}$$

Those are of linear size, but can even be eliminated completely by rewriting rules of the form

$$_{2}(\ )\quad\ _{1}(\ _{0}),\ \text{label}\ \ (\ _{0},\ ).$$

or

$$_{2}(\ )\quad\ _{1}(\ _{0}),\ \text{label}\ \ (\ ,\ _{0}).$$

obtained in our NFA encoding as

$$_{2}(\ )\quad\ _{1}(\ ),\ \text{label}\ \ (\ ).$$

The next proposition formalizes a technique for evaluating acyclic conjunctive queries that has long been known in the database theory community [20, 31, 1], applied to the special case that relations are at most binary.

Proposition 4.5 *Let be a monadic datalog program over unary and binary predicates and let r be a rule of with head variable . Assume that variable $\neq$ is an ear in r and*

$$P_1(\ ),\ \ldots,\ P\ \ (\ ),\ R(\ ,\ ')$$

is the set of all atoms over . Then, the altered program $'$ obtained by replacing the atoms over in r by atom $X.R(\ ')$ and adding

$$X(\ )\quad P_1(\ ),\ \ldots,\ P\ \ (\ ).$$

(where X is a new predicate) is equivalent to .

Example 4.6 Consider again our paper titles-by-authors example of the introduction. Using Proposition 4.5, the rule

$$\begin{aligned} (\;) \quad & \text{paper}(P), \\ & \text{child}(P,A),\ \text{author}(A), \\ & \text{child}(A,\;),\ \text{chandra}(\;), \\ & \text{child}(A,M),\ \text{merlin}(M), \\ & \text{child}(P,\;),\ \text{title}(\;). \end{aligned}$$

translates into

$$\begin{array}{ll} (\;) & \text{chandra}(\;). \\ M(M) & \text{merlin}(M). \\ A(A) & \text{author}(A),\ \;.\text{child}\ ^{1}(A),\ M.\text{child}\ ^{1}(A). \\ P(P) & \text{paper}(P),\ A.\text{child}\ ^{1}(P). \\ (\;) & \text{title}(\;),\ P.\text{child}(\;). \end{array}$$

Theorem 4.7 *Let τ^{+} be the signature consisting of τ and any number of binary predicates defined by regular path expressions over τ. Moreover, let be a monadic datalog program over τ^{+} in which each rule is acyclic. Then, can be rewritten into an equivalent monadic datalog program $'$ over τ in linear time.*

Proof (Sketch). Each non-empty tree has leaves, i.e., ears. The query graph of each acyclic rule of is a forest, whose arcs (with a emphasis on those defined by regular path expression atoms) can be iteratively eliminated using Proposition 4.5.

As a direct consequence, monadic datalog over the signature τ^{+} has linear time combined complexity (that is, given program and data tree t, we have $(|\;| * |\text{dom}|)$ as an upper bound). Since the reduction used in [16] to show that monadic datalog over τ captures MSO only requires acyclic rules, we conclude

Proposition 4.8 *Monadic datalog over τ^{+}, where all rules are acyclic, captures MSO over trees.*

We thus have a monadic query language for trees at our hands which is expressive yet has very low evaluation complexity. In the remainder of this paper, we will apply it to several practical database problems (namely, XPath processing and Web information extraction).

5 Core XPath

As pointed out in the introduction, XPath is an important language for querying XML. In this section, we propose a fragment of XPath which is the logical core of that language, but lacks many of the "bells and whistles" that account for little expressive power.

First we define a number of regular path expressions (over τ) which have found prominent use as the so-called *axis relations* of XPath. By inspection of the standards document [34], the reader can verify that the following definitions as regular path expressions over τ coincide with the those of [34]; Our list is complete except for axes (such as "attribute") which can be simulated by using the "child" axis in conjunction with a single check of a label.

Definition 5.1 (XPath axes)

$$\begin{array}{rcl} \text{self} & := & \\ \text{child} & := & \text{firstchild.nextsibling}^{*} \\ \text{parent} & := & \text{child}\ ^{1} \\ \text{descendant} & := & \text{child}^{+} \\ \text{ancestor} & := & \text{descendant}\ ^{1} \\ \text{descendant-or-self} & := & \text{child}^{*} \\ \text{ancestor-or-self} & := & \text{descendant-or-self}\ ^{1} \\ \text{following-sibling} & := & \text{nextsibling}^{*} \\ \text{preceding-sibling} & := & \text{following-sibling}\ ^{1} \\ \text{following} & := & \text{ancestor-or-self.nextsibling}^{+}. \\ & & \text{descendant-or-self} \\ \text{preceding} & := & \text{following}\ ^{1} \end{array}$$

Definition 5.2 Let the language of *Core XPath* be defined by its abstract EBNF syntax:

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

"cxp" is the start production, stands for an axis (see Definition 5.1), and t for a "node test" (either a label $l \in$ or "*", meaning "any label"). $(\;)$, for a label $\in$, is label . $(*)$ is dom. The semantics of Core XPath queries is defined by a function

$$\begin{array}{rcl} & : & (tfp) \rightarrow \ \text{dom}{\times}\text{dom} \\ [\![\ ::t[\;]]\!] & := & \{\langle\;,\;\rangle \mid \qquad \in (\;\;(t) \qquad [\![\;]\!])\} \\ [\![\quad]\!] & := & \text{dom} \times \{\;\mid \langle \text{root},\;\rangle \in \;[\![\;]\!]\} \\ [\![\ _1\quad _2]\!] & := & \{\langle\;,\;\rangle \mid \langle\;,\;\rangle \in \;[\![\ _1]\!] \\ & & \qquad \langle\;,\;\rangle \in \;[\![\ _2]\!]\} \end{array}$$

$$\begin{array}{rcl} & : & (pr\quad) \rightarrow \ \text{dom} \\ [\![\ _1 \text{ and } \ _2]\!] & := & [\![\ _1]\!] \qquad [\![\ _2]\!] \\ [\![\ _1 \text{ or } \ _2]\!] & := & [\![\ _1]\!] \qquad [\![\ _2]\!] \end{array}$$

$$\begin{aligned} [\![\text{not}(\ )]\!] &:= \text{dom} - [\![\ ]\!] \\ [\![\ ]\!] &:= \{\ _0 \mid \exists\ :\langle\ _0,\ \rangle \in\ [\![\ ]\!]\} \end{aligned}$$

The result of query is $\{\ \mid \exists\ :\langle\ ,\ \rangle \in\ [\![\ ]\!]\}$.

In our examples throughout this paper, we often use the standard "simplified" syntax of XPath [34]. For instance, paper[author]/title is an abbreviated version of paper[child::author]/child::title (that is, if we do not make the axis explicit at some point, it is "child").

Clearly, Core XPath is a fragment of XPath, both syntactically and semantically. Moreover, Core XPath without "or" and "not" is equivalent to acyclic conjunctive queries over axis relations and Core XPath as a whole can be encoded in nonrecursive monadic datalog with stratified negation. We make this obvious by providing an encoding of Core XPath in that language.

Definition 5.3 (Monadic datalog encoding)

$$\begin{array}{rl} [\![\ ::l[\ ]]\!](\ ) & \text{node.}\ (\ ),\ l(\ ),\ \ [\![\ ]\!](\ ). \\ [\![\ \ ::l[\ ]]\!](\ ) & \text{root.}\ (\ ),\ l(\ ),\ \ [\![\ ]\!](\ ). \\ [\![\ \ \ ::l[\ ]]\!](\ ) & [\![\ ]\!].\ (\ ),\ l(\ ),\ \ [\![\ ]\!](\ ). \\ [\![\ ::l[\ ]]\!](\ ) & l\ \ \ [\![\ ]\!]\ .\ \ ^1(\ ). \\ \underline{[\![\ ]\!]\ \ l\ \ \ [\![\ ]\!]}(\ ) & [\![\ ]\!](\ ),\ l(\ ),\ \ [\![\ ]\!](\ ). \\ [\![\ ::l[\ ]\ \ ]\!](\ ) & \underline{[\![\ ]\!]\ \ l\ \ \ [\![\ ]\!]}.\ \ ^1(\ ). \\ [\![\ ]\!](\ ) & [\![\ ]\!](\ ). \\ [\![\ _1\ \ \ _2]\!](\ ) & [\![\ _1]\!](\ ),\ \ [\![\ _2]\!](\ ). \\ [\![\ \ \ ]\!](\ ) & \text{node}(\ ),\ \ \ [\![\ ]\!](\ ). \end{array}$$

$$\begin{array}{rl} [\![\ _1\ \ \ _2]\!](\ ) & [\![\ _1]\!](\ ). \\ [\![\ _1\ \ \ _2]\!](\ ) & [\![\ _1]\!](\ ). \end{array}$$

A Core XPath query returns node n on a given tree iff $[\![\ ]\!](n)$ is true in the fixpoint of the program obtained by recursively defining predicate $[\![\ ]\!]$ in terms of simpler predicates.

Thus, as first observed in [17] [3],

Proposition 5.4 *Core XPath has linear time (that is, $(|\ \ | * |dom|)$) combined complexity.*

[3]There however, this result is obtained using different techniques outside of a logic-based framework. In [17], the authors also report on experiments with the most widely used XPath engines, showing that these systems have query behavior that is considerably worse than ours. Moreover, the evaluation complexity of a number of more encompassing fragments of XPath (as well as full XPath) is studied, and further efficient algorithms for these problems are presented. All methods are covered by a pending patent.

Proof. The above encoding maps each Core XPath query into nonrecursive monadic datalog with negation over signature τ^+, which can be translated into monadic datalog with stratified negation in linear time (Theorem 4.7).

We conclude this section by putting Core XPath into the context of yet another logical language. As pointed out in [21] by Suciu and Miklau, a fragment of XPath is expressible in Computation Tree Logic (CTL, cf. [13]). We extend the (very restrictive) fragment of XPath treated in [21] (and called $XP^{\{\ \ *\ \ \}}$ there) to full Core XPath.

Syntactically, Core XPath queries consist of a so-called *location path* (a sequence of *location steps*, which are statements of the form $\ ::\ t[\ ]$) and a number of conditions enclosed in brackets. We assume that all location paths are of length one; this entails no loss of generality, as each Core XPath query can be rewritten to satisfy this requirement using additional conditions and the inverses of axes.

Example 5.5 The main location path of query

paper[author[chandra and merlin]]/title

is of length two (it is "paper/title"), as our goal is to select title nodes that are children of paper nodes. The query

title[parent::paper[author[chandra and merlin]]]

is equivalent but each location path only consist of one step. We obtain this query by in a sense re-rooting the query tree to make the "hot spot" of the query (which selects the nodes) the root.

Let us first consider the fragment of Core XPath that we can encode in plain CTL without any extensions. (Only the axes "self", "child", "descendant-or-self", and "descendant" may be used.)

Definition 5.6 (Mapping to CTL)

$$\begin{array}{rcl} [\![\text{self::}\ \ ]\!] & \Rightarrow & [\![\ ]\!] \\ [\![\text{child::}\ \ ]\!] & \Rightarrow & X[\![\ ]\!] \\ [\![\text{descendant::}\ \ ]\!] & \Rightarrow & F[\![\ ]\!] \\ [\![\text{descendant-or-self::}\ \ ]\!] & \Rightarrow & ([\![\ ]\!]\ \ \ F[\![\ ]\!]) \\ [\![t[\ ]]\!] & \Rightarrow & [\![t]\!]\ \ [\![\ ]\!] \\ [\![\ ]\!]\ (\text{where}\ \ \in\ ) & \Rightarrow & (\text{propositional predicate}) \\ [\![*]\!] & \Rightarrow & tr \\ [\![\ _1\ \text{and}\ \ _2]\!] & \Rightarrow & ([\![\ _1]\!]\ \ [\![\ _2]\!]) \\ [\![\ _1\ \text{or}\ \ _2]\!] & \Rightarrow & ([\![\ _1]\!]\ \ [\![\ _2]\!]) \\ [\![\text{not}\ \ ]\!] & \Rightarrow & [\![\ ]\!] \end{array}$$

Reverse axes (such as "parent", "ancestor", etc.) can be dealt with by using CTL with "past" operators [13]. We denote the past version of X by $X\ ^{1}$ and the past version of F by $F\ ^{1}$.

Example 5.7 The query

title[parent::paper[author[chandra and merlin]]]

translates into CTL (with past) as

title
$X\ ^{1}$(paper $\quad X$(author $\quad X$chandra $\quad X$merlin))

The remaining axes, such as "following" and "following-sibling", require multimodal CTL; using e.g. the modalities X to denote "child" (instead of X) and $X_{\rightarrow}$ to denote "following-sibling". Multimodal CTL with past operators can still be checked in linear time (i.e., in the query and the data, respectively), because it can be translated into a suitable fragment of Datalog LITE [15], which obeys this complexity bound. The translation required is a simple generalization of the one from CTL to Datalog LITE in [15].

6 Adding the "child" Axis to τ_u

The signature τ is a concise representation of labeled unranked trees, and monadic datalog over τ captures all of MSO (over such trees). Nevertheless, in a practical language it may be desirable or necessary to be able to state rules such as

$$(X) \quad \text{child}(X, \ ), \text{child}(\ , \ ), \text{nextsibling}^*(\ , \ ).$$

which make use of the child relation but are not necessarily acyclic. With the machinery introduced so far, we can neither rewrite such rules into rules over τ nor do we have a query complexity bound better than the general NP-completeness result for conjunctive queries (which holds even if predicates are of at most arity two [9]).

6.1 Complexity

In the next algorithm, we will use the signature

$$\tau' = \langle \text{dom}, \text{firstchild}, \text{nextsibling}, (\text{label}\ )\ _{\in}\ , \text{fs}, \text{ls}\rangle$$

where "fs" (nextsibling* relative to a first sibling) is

$$\text{fs} := \{\langle\ ,\ \rangle \mid \exists\ _0 : \text{firstchild}(\ _0,\ ), \text{nextsibling}^*(\ ,\ )\}$$

and has a functional dependency from the second to the first column (written as fs: $1 \rightarrow$, meaning that for each value , there is at most one value such that $\langle\ ,\ \rangle \in$ fs). "ls" (last sibling) is defined as

$$\text{ls} := \{\ \mid \exists\ _0 : \text{lastchild}(\ _0,\ )\}.$$

The *Chase* is a classical method for applying logical dependencies to conjunctive queries (cf. [1]) and takes linear time for functional dependencies.

Definition 6.1 (Chase rule for functional dependencies of binary relations) Let be a CQ.

1. For the functional dependency R: \$1 $\rightarrow$ \$2 and atoms $R(\ ,\ _1), R(\ ,\ _2)$ in , replace every occurrence of $_2$ in by $_1$.

2. For the functional dependency R: \$2 $\rightarrow$ \$1 and atoms $R(\ _1,\ ), R(\ _2,\ )$ in , replace every occurrence of $_2$ in by $_1$.

Algorithm 6.2 (Rule Transformation)
Input: a monadic datalog rule r over τ {child}.
Output: a rule over τ'.

1. Replace each occurrence of an atom child(,) by firstchild(, $_0$), fs($_0$,), where $_0$ is a new variable.

2. Replace each occurrence of an atom lastchild(,) by firstchild(, $_0$), fs($_0$,), ls( ), where $_0$ is a new variable.

3. while one of the following rules is applicable, apply it:

   - the Chase rules for the fd's

   $$\begin{array}{rlrl} \text{firstchild}: & 1 \rightarrow & \text{firstchild}: & \rightarrow 1 \\ \text{nextsibling}: & 1 \rightarrow & \text{nextsibling}: & \rightarrow 1 \\ \text{fs}: & \rightarrow 1 & & \end{array}$$

   - replace atoms

   $$\text{fs}(\ ,\ ), \text{nextsibling}(\ ,\ )$$

   by

   $$\text{fs}(\ ,\ ), \text{nextsibling}(\ ,\ ).$$

Theorem 6.3 *Let r be a rule over the signature τ {child}. The rule obtained by applying Algorithm 6.2 to r is equivalent to r. Moreover, Algorithm 6.2 runs in time* $(|r|)$.

None of the steps 1 to 3 changes the meaning of the query, and Chase procedures of this kind can be implemented to run in linear time.

Theorem 6.4 *Let r be a rule over τ $\{child\}$ and r' the rule obtained by applying Algorithm 6.2 to r. If the (undirected) graph of rule r' has a cycle then r' is unsatisfiable.*

Proof Sketch Three binary predicates need to be considered, "firstchild", "nextsibling", and "fs". To show our theorem, we need to distinguish a number of cases.

- *Directed cycles*: "fs" atoms define a *reflexive* and transitive closure. Directed cycles consisting exclusively of "fs" atoms are satisfiable, but are never created by our algorithm.

  All other kinds of directed cycles, for any permutation of "firstchild", "nextsibling" and "fs" atoms, entail unsatisfiability of the rule, as both "firstchild" and "nextsibling" are antisymmetric. This is also true for point loops firstchild(,) and nextsibling(,).

$$\text{firstchild} \quad {}_{1} \quad \text{firstchild} \quad {}_{2}$$

- *Undirected cycles containing "firstchild" atoms*:

  1. If there is an atom firstchild(,) where and are siblings, the rule is unsatisfiable.

  $$\text{firstchild} \quad {}_{1} \quad \text{nextsibling} \quad {}_{2}$$

  2. If there are two atoms firstchild($_1$, $_1$) and firstchild($_2$, $_2$), where $_1$ and $_2$ are distinct siblings, the rule is unsatisfiable.

  $$\text{nextsibling} \quad {}_1 \rightarrow {}_2 \quad \text{firstchild} \quad \text{firstchild} \quad {}_3 \rightarrow \quad \text{nextsibling}$$

  Algorithm 6.2 cannot produce (undirected) cycles containing "firstchild" which belong to neither of these two classes.

- All that is now left to consider is undirected cycles consisting exclusively of "nextsibling" and "fs" atoms. By virtue of the Chase, if we just consider the "nextsibling" and "fs" atoms of our rule, the graph either contains a directed cycle or is a forest. Thus, our theorem is shown.

Thus, given a program, we apply Algorithm 6.2 to each rule. Then we eliminate those rules with cycles, as they are unsatisfiable. To obtain a program over τ $\{$nextsibling$^*\}$, we rename each atom fs(,) to nextsibling*(,) and replace each ls( ) by lastchild($_0$,), where $_0$ is a new variable not appearing elsewhere.

Thus, what we have shown in this section was

Theorem 6.5 *Every monadic datalog program over τ $\{child\}$ can be rewritten into an equivalent program over τ $\{nextsibling^*\}$ in which each rule is acyclic. This transformation can be effected in linear time in the size of the program.*

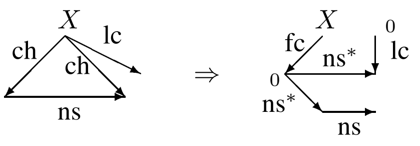
Example 6.6 Consider the following query

$$\text{child}(X, \ ),\ \text{nextsibling}(\ , \ ),$$
$$\text{child}(X, \ ),\ \text{lastchild}(X, \ ).$$

which rewrites into the acyclic query

$$\text{firstchild}(X, \ _0),$$
$$\text{nextsibling}^*(\ _0, \ ),\ \text{nextsibling}(\ , \ ),$$
$$\text{nextsibling}^*(\ _0, \ ),\ \text{lastchild}(\ _0, \ ).$$

In our graphical notation, the translation is



As a consequence of the previous theorem, we have

Corollary 6.7 *There is an evaluation algorithm for monadic datalog over τ $\{child\}$ which takes time* $(|dom(t)| * | \ |)$.

6.2 Application to Tree Wrapping

An important application in which rules over τ with the "child" axis are required is wrapping [4, 3]. In [16], we defined a wrapping language called Elog , which was basically simply monadic datalog over τ $\{$child$\}$[4]. (Note that the "before" predicate of Elog is defined as equivalent to "nextsibling" modulo a check of one label.) Elog is a simplified but functional fragment of Elog, the kernel wrapping language used in the LiXto system[5]. By the previous result, monadic datalog over τ $\{$child$\}$ (and thus Elog) is still in linear time in terms of query and data, respectively. Consequently,

Corollary 6.8 *The Elog wrapper language has linear time query complexity.*

[4]Actually, Elog$^-$ rules obey some syntactic restrictions to facilitate visual specification, but Elog$^-$ still captures MSO. Indeed, the visual wrapper specification process of the LiXto system [4, 3], by which Elog$^-$ was motivated, only creates wrapper programs that are acyclic or can be made so using very simple means.

[5]See `http://www.lixto.com`.

7 Limits of Tractability

Unfortunately, while acyclic conjunctive queries over regular path expressions defined over the relations of τ had linear-time evaluation complexity, the complexity for such queries without the acyclicity restriction is NP-complete.

Theorem 7.1 *The query complexity of boolean conjunctive queries over binary predicates defined by regular path expressions (with inverse) over "firstchild" or "nextsibling" on trees is NP-complete.*

Proof. Membership in NP is clear, and hardness follows from a reduction from 3-Colorability (3COL). Indeed, the NP-completeness result is already true on the tree which is a simple path of length three (no branching), i.e. the structure

$$P_3 := \langle \text{dom} = \{n_1, n_2, n_3\},\ \text{fc} = \{\langle n_1, n_2\rangle, \langle n_2, n_3\rangle\}\rangle$$

(where "fc" is short for "firstchild"). The 3-colorability of a graph $= \langle\ ,\ \rangle$ can be encoded as a boolean conjunctive query with a body atom $v \neq w$ for each edge $\langle v, w\rangle \in$ of the graph. By the natural definition of 3COL, is 3-colorable iff is true on a set of three nodes, e.g. dom of P_3. Now, $\neq$ on P_3 can be defined as

$$\text{fc} \mid \text{fc.fc} \mid \text{fc}^{\ 1} \mid \text{fc}^{\ 1}.\text{fc}^{\ 1}$$

as a regular path relation over "fc".

While our complexity result for conjunctive queries using the child relation over trees is linear (and thus robustly polynomial), this is not so for the edge relation of DAGs.

Theorem 7.2 *The evaluation of boolean conjunctive queries over the edge relation of DAGs is NP-complete.*

Proof. By reduction from 3COL for a graph $= \langle\ ,\ \rangle$: We assume an arbitrary total ordering of the nodes of . The query contains a body atom for each edge $\{v\ , v\ \} \in$. If $v\ \leq v\ $, we add (,) to , otherwise (,).

Let $n = |\ |$. The data DAG consists of $3 * n$ nodes r , g , and for $1 \leq i \leq n$ and edges

$$\{\langle\ ,\ \rangle \mid\ ,\ \in \{r, g,\ \},\ \neq\ ,\ 1 \leq i < j \leq n\}.$$

is satisfiable over this DAG iff is 3-colorable.

Note that the NP-completeness of conjunctive queries over the edge relation of graphs in general has long been known [9].

As the final part of this section, and as a follow-up to and illustration of Remark 2.4, we show that the evaluation of (even just boolean) acyclic conjunctive queries extended with *label variables* is NP-hard. By label variables, we refer to variables ranging exclusively over the labels of document nodes. We use a binary predicate label′ instead of the | | unary predicates label of τ , where $\langle n, l\rangle \in$ label′ iff node n is labeled l in the tree. A conjunctive query with label variables matches a tree iff the same query without the label′ atoms has a satisfaction : Vars( ) $\to$ dom such that for each pair of atoms label′(X,), label′(,) (i.e., with the same label variable), (X) and ( ) have the same label.

Theorem 7.3 *The query complexity of acyclic conjunctive queries with label variables over trees is NP-complete.*

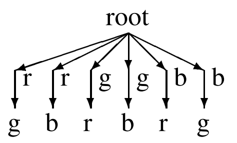
Proof. Membership in NP is clear. Hardness follows from the following LOGSPACE reduction from 3COL. Given the undirected graph $= \langle\ ,\ \rangle$ to be colored, with

$$= \{\{v_1, w_1\},\ \ldots,\ \{v_n, w_n\}\}.$$

Without loss of generality, we assume that all nodes of appear in . (For each node v not in any arc, we add a *new* node v' to and arc $\{v, v'\}$ to .) We create the query

$$\begin{array}{l} \text{child}(R, X_1),\ \text{child}(X_1,\ _1), \\ \text{label}(X_1, v_1),\ \text{label}(\ _1, w_1), \\ \ldots, \\ \text{child}(R, X_n),\ \text{child}(X_n,\ _n), \\ \text{label}(X_n, v_n),\ \text{label}(\ _n, w_n). \end{array}$$

It is easy to verify that is 3-colorable if and only if our query is satisfied on data tree



where "root", r, g, and are the node labels.

8 Descendant Queries

After reporting a number of favorable complexity results on acyclic queries and queries with the "child" axis, we showed the opposite side of the medal and provided three NP-completeness results on general (possibly cyclic) conjunctive queries. This motivates the search for interesting fragments of cyclic conjunctive queries which are tractable. In this section, we report on a significant fragment with polynomial-time query evaluation complexity, namely, one that supports transitive closure axes such as "descendant". We are currently investigating for additional such classes and are working towards charting the tractability frontier.

The goal of this section is to find efficient algorithms for evaluating queries over the signature

$$\tau\ = \langle (P\ )_1\ \ ,\ \text{child}^*,\ \text{child}^+\rangle$$

where the P are unary predicates.

Algorithm 8.1 (Elimination Algorithm)
Input: A boolean conjunctive query over τ and a tree.
Output: The truth value of .

```
for each $n \in$ dom do $(n)$ := Vars( );

while there is a change do
{
    if $P($ $)$ and $\in$ $(n)$ but $n \notin P$ then
        $(n) :=$ $(n) - \{$ $\}$;
    if $R($ , $)$, $\in$ $(n)$, and
      there is no $n'$ s.t. $\langle n, n' \rangle \in R$ and $\in$ $(n')$ then
        $(n) :=$ $(n) - \{$ $\}$;
    if $R($ , $)$, $\in$ $(n)$, and
      there is no $n_0$ s.t. $\langle n_0, n \rangle \in R$ and $\in$ $(n_0)$ then
        $(n) :=$ $(n) - \{$ $\}$;
}
if $\bigcup_n$ $(n) =$ Vars(Q) then return true;
else return false;
```

where R is either child* or child$^+$.

Theorem 8.2 *Given a boolean conjunctive query over unary predicates, child* and child$^+$ and a tree t, Algorithm 8.1 returns true iff is true on t.*

Proof (Sketch). We only need to show the soundness of Algorithm 8.1. Its completeness follows from the fact that its rules for eliminating assignments enforce requirements locally that must hold in unison in a solution satisfaction. Therefore, Algorithm 8.1 never eliminates a single assignment needed in an embedding of into t.

We restrict ourselves to the case that the query graph of is connected. The general case can be shown with the same argument used in the proof of Proposition 3.4.

We extract a satisfaction : Vars( ) $\rightarrow$ dom for from as follows. Initially, is empty and is undefined everywhere. We traverse the tree depth-first; whenever we encounter a variable assignment $\in$ (n) and $\notin$, we add to and set $($ $) := n$.

Induction Claim: We show by induction that in every step, the boolean query $-$ consisting of those atoms of that only contain variables in is satisfied by the embedding computed so far. Thus, after the final step, is a satisfaction for .

Induction: Clearly, each variable assignment $\in$ (n) not eliminated by Algorithm 8.1 satisfies all unary atoms over , thus we only need to consider binary atoms.

When the first variable is added to , $-$ contains no binary atoms and our claim holds trivially.

Assume variable is added in the current step. All atoms that must now hold in addition to those of the previous steps (for which we assume satisfies them) are of the form $R(w,$ $)$, where R is either child* or child$^+$.

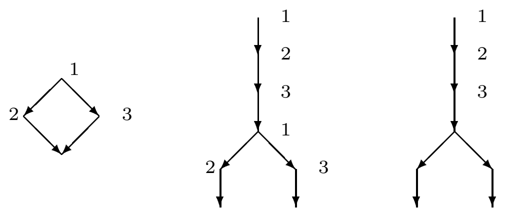
By the elimination rules of Algorithm 8.1,

1. As $\in$ (n), there must be a node n_0 on the path from the root to n s.t. $w \in$ (n_0) and $\langle n_0, n \rangle \in R$. This node n_0 has already been visited (as we proceed using depth-first traversal), and thus (w) has already been defined.

2. Moreover, there is an in the subtree of every node assigned w.

Since n is the first node in document order that is assigned (otherwise n would not be the first such node we encounter by depth-first traversal), there is no occurrence of w at the left of the path from the root to n. Thus, n is in the subtree of (w). As no node above (w) is assigned w, $\langle$ (w), $($ $)\rangle \in R$.

Example 8.3 Consider the query shown below on the left (all binary atoms are, say, of the form child*). Moreover, assume that for each variable , the query contains a unary atom label $_i($ $)$. Below, in the center, We show a data tree and the labels the nodes are assigned. It is easy to verify that the Elimination Algorithm returns $\in$ (n) exactly if n is labeled .



At the right, we show a mapping obtained by traversing the tree depth-first and always adding the first encounter of each variable to . Clearly, is a satisfaction for our query.

Note also that Algorithm 8.1 cannot be used to answer monadic queries: For instance, if $_1$ were the head variable, the branching node of the tree is *incorrectly* labeled $_1$ by Algorithm 8.1, so to say.

Theorem 8.4 *Boolean conjunctive queries over τ have $(|$ $| * |dom|)$ time combined complexity.*

Proof Sketch. By Theorem 8.2, a connected boolean conjunctive query is true over a tree iff $\bigcup_{n \in \text{dom}}$ $(n) \neq$. We encode the fixpoint computation of Algorithm 8.1 as a (semipositive) monadic datalog program, which consists of the following rules.

We use regular path expressions (see Lemma 4.2) to keep this proof concise, and use the abbreviations

$$\text{firstsib} := (\text{dom.firstchild} \mid \text{root})$$
$$\text{lastsib} := (\text{dom.lastchild} \mid \text{root}).$$

For instance, nodes for which all children have property p can be defined as

$$\text{leaf} \mid \text{lastsib}.p.(\text{nextsibling}\ ^{1}.p)^{*}.\text{firstchild}\ ^{1}$$

That is, such nodes n are either leaves or there is a node (which happens to coincide with n) whose last child has property p, each of the earlier siblings up to the first have property p, and n is the parent of that first sibling.

- Atom "$P(\ )$" is in :

$$\not\in \ (v) \qquad P(v).$$

- Atom "child$^{*}(\ ,\ )$" is in :

$$\begin{array}{lll}
\not\in \ (v) & & \not\in \ \ tr\ \ (v). \\
\not\in \ \ tr\ \ (v) & & \not\in \ (v), \\
 & & \text{lastsib}.\ \not\in \ \ tr\ \ . \\
 & & (\text{nextsibling}\ ^{1}.\ \not\in \ \ tr\ \ )^{*}. \\
 & & \text{firstchild}\ ^{1}(v). \\
\not\in \ \ tr\ \ (v) & & \text{leaf}(v),\ \ \not\in \ (v). \\
\not\in \ (v) & & \text{root}. \\
 & & (\ \not\in \ .\text{child})^{*}.\ \ \not\in \ (v).
\end{array}$$

- Atom "child$^{+}(\ ,\ )$" is in :

$$\begin{array}{lll}
\not\in \ (v) & & \text{lastsib}.\ \not\in \ \ tr\ \ . \\
 & & (\text{nextsibling}\ ^{1}.\ \not\in \ \ tr\ \ )^{*}. \\
 & & \text{firstchild}\ ^{1}(v). \\
\not\in \ (v) & & \text{leaf}(v). \\
\not\in \ (v) & & \text{root}.(\ \not\in \ .\text{child})^{*}(v).
\end{array}$$

This program computes the complement of , as obtained by Algorithm 8.1. The construction exploits the order information which is available in the tree to check whether a predicate (asserting that a certain variable is not assigned to a given node n) is true *for all* nodes from a certain region of the tree (e.g., the descendants of n). The program can be generated in time $(|\ \ |)$ and monadic datalog programs (over τ) can be evaluated in time linear in their size (times the size of the data). Our theorem follows.

Corollary 8.5 *Monadic datalog over τ has polynomial-time combined complexity.*

Proof. We evaluate a monadic datalog program as follows. A monadic conjunctive query with body Body( ) and head variable is evaluated by running Algorithm 8.1 for boolean conjunctive queries on query

$$'\qquad (\ ),\ \text{Body}(\ ).$$

$|\text{dom}|$ times, each time fixing the desired value n of the head variable by setting the value of auxiliary relation to $\{n\}$.

A fixpoint for is obtained by iteratively applying this evaluation method for monadic rules to the atoms derived so far. Initially, all ground atoms over IDB predicates are false, and an atom is set to true if a monadic rule entails it. (However, no true atom is ever set to false later on.) Thus, the overall algorithm is monotonic and the fixpoint is reached after at most $(|\ \ | * |\text{dom}|)$ steps.

References

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

[2] S. Abiteboul and V. Vianu. Regular Path Queries with Constraints. *JCSS*, **58**(3):428–452, 1999.

[3] R. Baumgartner, S. Flesca, and G. Gottlob. "Declarative Information Extraction, Web Crawling, and Recursive Wrapping with Lixto". In *Proc. LPNMR'01*, Vienna, Austria, 2001.

[4] R. Baumgartner, S. Flesca, and G. Gottlob. "Visual Web Information Extraction with Lixto". In *Proc. VLDB'01*, 2001.

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

[6] A. Brüggemann-Klein and D. Wood. "Regular Tree Languages over Non-ranked Alphabets". Unpublished manuscript, 1998.

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

[8] S. Ceri, G. Gottlob, and L. Tanca. *"Logic Programming and Databases"*. Springer-Verlag, Berlin, 1990.

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

[10] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. "Complexity and Expressive Power of Logic Programming". To appear in *ACM Computing Surveys*.

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

[12] W. F. Dowling and J. H. Gallier. "Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae". *Journal of Logic Programming*, **1**(3):267–284, 1984.

[13] E. A. Emerson. "Temporal and Modal Logic". In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, chapter 16, pages 995–1072. Elsevier Science Publishers B.V., 1990.

[14] J. Flum, M. Frick, and M. Grohe. "Query Evaluation via Tree-Decompositions". In *Proc. of the International Conference on Database Theory*, 2001.

[15] G. Gottlob, E. Grädel, and H. Veith. "Datalog LITE: A Deductive Query Language with Linear-time Model Checking". *ACM Transactions on Computational Logic*, **3**(1):42–79, 2002.

[16] G. Gottlob and C. Koch. "Monadic Datalog and the Expressive Power of Web Information Extraction Languages". In *Proc. 21st ACM Symposium on Principles of Database Systems (PODS)*, Madison, Wisconsin, 2002. to appear.

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

[18] J. E. Hopcroft and J. D. Ullman. *"Introduction to Automata Theory, Languages, and Computation"*. Addison-Wesley Publishing Company, Reading, MA USA, 1979.

[19] L. Liu, C. Pu, and W. Han. "XWRAP: An XML-Enabled Wrapper Construction System for Web Information Sources". In *Proceedings of the 16th International Conference on Data Engineering*, 1998.

[20] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[21] G. Miklau and D. Suciu. "Containment and Equivalence for an XPath Fragment". In *Proc. 21st ACM Symposium on Principles of Database Systems (PODS)*, Madison, Wisconsin, 2002. (to appear).

[22] T. Milo, D. Suciu, and V. Vianu. "Typechecking for XML Transformers". In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS' 00)*, pages 11–22, 2000.

[23] M. Minoux. "LTUR: A Simplified Linear-Time Unit Resolution Algorithm for Horn Formulae and Computer Implementation". *Information Processing Letters*, **29**(1):1–12, 1988.

[24] F. Neven and T. Schwentick. Expressive and Efficient Pattern Languages for Tree-Structured Data. In *Proc. 19th Symp. on Principles of Database Systems (PODS 2000)*, pages 145–156, Dallas, Texas, USA, 2000. ACM Press.

[25] F. Neven and T. Schwentick. "Query Automata on Finite Trees". *Theoretical Computer Science (to appear)*, 2001.

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

[27] A. Sahuguet and F. Azavant. "Building Intelligent Web Applications Using Lightweight Wrappers". *Data and Knowledge Engineering*, **36**(3):283–316, 2001.

[28] W. Thomas. "Automata on Infinite Objects". In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, chapter 4, pages 133–192. Elsevier Science Publishers B.V., 1990.

[29] W. Thomas. "Languages, Automata, and Logic". In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 7, pages 389–455. Springer Verlag, 1997.

[30] J. D. Ullman. *Principles of Database & Knowledge-Base Systems Vol. 1*. Computer Science Press, 1988.

[31] J. D. Ullman. *Principles of Database & Knowledge-Base Systems Vol. 2: The New Technologies*. Computer Science Press, 1989.

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

[33] World Wide Web Consortium. "XML Query". http://www.w3c.org/XML/query/.

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

[35] World Wide Web Consortium. XSL Transformations (XSLT). W3C Recommendation Version 1.0. http://www.w3.org/TR/xslt.

[36] World Wide Web Consortium. "XML Pointer Language Version 1.0. W3C Candidate Recommendation", Sept. 2001. http://www.w3.org/TR/2001/CR-xptr-20010911.

[37] World Wide Web Consortium. "XML Schema Part 0: Primer. W3C Recommendation", May 2001. http://www.w3c.org/XML/Schema.