Forward Node-Selecting Queries over Trees

DAN OLTEANU
Universität des Saarlandes, Saarbrücken, Germany

Node-selecting queries over trees lie at the core of several important XML languages for the Web, e.g., the node-selection language XPath, the query language XQuery and the transformation language XSLT. The main syntactic constructs of such queries are the backward predicates, e.g., `ancestor` and `preceding`, and the forward predicates, e.g., `descendant` and `following`. Forward predicates are included in the depth-first, left-to-right preorder relation associated with the input tree, whereas backward predicates are included in the inverse of this preorder relation.

This work is devoted to an expressiveness study of node-selecting queries with proven theoretical and practical applicability, especially in the field of query evaluation against XML streams. The main question it answers positively is whether for each input query with forward and backward predicates there exists an equivalent forward-only output query. This question is then positively answered for input and output queries of varying structural complexity, using LOGLIN and PSPACE reductions.

Various existing applications based on the results of this work are reported, including query optimization and streamed evaluation.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Query Languages; I.7.2 [**Document Preparation**]: Markup Languages

General Terms: Theory, Languages, Rewriting

Additional Key Words and Phrases: Expressiveness, Streams, XML, XPath

1. INTRODUCTION

XPath [Clark and DeRose 1999] is the major language of choice for expressing node-selecting queries over ordered unranked trees representing XML documents. XPath is also at the core of several important languages for the Web, e.g., the query language XQuery [Boag et al. 2006], the transformation language XSLT [Clark 1999], the schema language XML-Schema [Fallside and Walmsley 2001], and the language for addressing fragments of XML documents XPointer [DeRose et al. 2002]. Therefore, the study of XPath was recognized early on to be of paramount importance and a significant body of research exists on this topic.

For selecting nodes in trees, XPath offers backward and forward navigation with a large palette of so-called axes. The axes are binary predicates and can be classified in *forward* and *reverse*, depending on whether they are included in the depth-first,

Author's address: Lehrstuhl für Informationssysteme, Universität des Saarlandes, Im Stadtwald, D-66123 Saarbrücken, Germany, `olteanu@infosys.uni-sb.de`.
A preliminary version of this article entitled "XPath: Looking Forward" is published in LNCS 2490, EDBT 2002 Workshops (selected papers) March 2002 [Olteanu et al. 2002].

left-to-right preorder relation (also called *document order*) or its inverse. Examples of forward predicates are `descendant` and `following-sibling`, and of reverse predicates are `ancestor` and `preceding-sibling`.

The number as well as the relevance of these backward and forward predicates has been challenged, e.g., in [Desai 2001; Boag et al. 2006; Kay 2004]. Additionally, the random access to XML data enabled by the interplay of both forward and reverse predicates, poses difficulties to efficient query evaluation against XML data being too large (or even unbounded) to be stored and accessed in main memory. This is the case of XML repositories encountered, e.g., in natural language processing [Ide et al. 2000], in biology [Bry and Kröger 2003], and astronomy [NASA 2004]. This is also the case of unbounded XML streams arising in various contexts:

—For selective dissemination of information (SDI), continuously generated streams of XML documents have to be filtered according to complex requirements specified as XPath queries before being distributed to the subscribers [Chan et al. 2002; Altinel and Franklin 2000]. The routing of data to selected receivers is also becoming increasingly important in the context of web service systems.

—To integrate data over the Internet, in particular from slow sources, it is desirable to progressively process the input before the full data is retrieved [Ives et al. 2002].

—As a general processing scheme for XML, several solutions for pipelined processing have been suggested, where the input is sent through a chain of processors each of which takes the output of the preceding processor as input, e.g., Apache Cocoon [Apache Project 2001a].

—There are efforts from the user community, e.g., Xalan [Apache Project 2001b], and requirements from the W3C standards committees, e.g., Requirement 19 of [Kay 2004], to support progressive XSL(T) rendering of large XML documents.

In all these contexts, sequential data access, as supported by forward-only queries, is preferred over random access, simply because backward navigation in the stream's history, as required by reverse predicates, can be very expensive, unaffordable or impossible. There are three principal options to evaluate queries with reverse predicates in such contexts:

(1) Store in memory sufficient information that allows access to past events, particularly when evaluating reverse predicates. This amounts to keeping in memory a (possibly pruned) representation of the data [Apache Project 2001b].

(2) Evaluate the queries in more than one pass over the stream, provided several passes are possible. With this approach, it is also necessary to store additional information to be used in successive runs. This information can be considerably smaller than what is needed in the first approach.

(3) Find equivalent forward queries, i.e., queries containing only forward predicates.

Contributions. This paper targets the last of the three aforementioned approaches and shows it to be possible for specific fragments of XPath. This approach is less time consuming than the second one and does not require to buffer unnecessary input fragments as the first approach does. More precisely, the main contributions of this paper are as follows.

—We establish a robust framework for rewriting queries with reverse predicates to equivalent forward queries.
Rather than dealing with XPath syntax, this framework uses an equivalent fragment of monadic non-recursive Datalog with negation [Abiteboul et al. 1995] and with built-in predicates for the XPath axes and nodetests. For convenience, we call this language LGQ (logical graph queries). Our motivation for using LGQ instead of XPath is twofold, though our study of LGQ remains also a study of XPath. First, the XPath syntax poses many (unnecessary) technical challenges and the rewriting rules of concern are more compact and fewer when expressed over LGQ. Second, whereas XPath allows path, tree, and forest queries, whose names remind of their query pattern analogues, LGQ allows also the structurally more complex DAG and graph queries. The main restriction of our LGQ and XPath queries is that they are absolute, that is, they are always evaluated from the root of the input data tree (in contrast to relative queries, which can be evaluated from any set of nodes).

—Any LGQ query can be effectively rewritten to an equivalent forward query, and the same forward query is obtained regardless of the order of rule applications. We support this statement by proving that our rewriting framework is sound and complete, terminates, and is confluent.

—The last mentioned contribution is in essence an expressiveness result: LGQ is as expressive as its forward fragment, i.e., the reverse predicates do not add to the LGQ expressiveness. Along this line, our rewriting framework sheds light on the equivalence of other LGQ fragments defined by the structural complexity of their queries. In particular, paths, trees, and forests can be rewritten to forward forests. If one additional rule is added to the framework, then arbitrary graphs can be rewritten to forward forests. Also, by using one single rule, paths, trees, and forests can be rewritten to a special form of forward DAGs.

—We study the complexity of rewriting for input and output queries of varying language fragments. We find PSPACE reductions for deriving structurally less (or equally) complex forward equivalents, e.g., forests. This case is accompanied unavoidably by an exponential blow-up of the size of the forward queries. We find also LOGLIN reductions (i.e., LOGSPACE reductions with linear output) for deriving structurally more complex forward equivalents, e.g., graphs.

—To better understand the relation between the complexity of rewriting and the structural complexity of the derived forward queries, we design the class of so-called simple graphs, which combine the advantages of efficient rewriting and of structurally less complex forward equivalents. Simple graphs are LOGLIN-reducible to forward forests. Such graphs forbid co-occurrences of vertical (horizontal) closure reverse predicates (immediately) after closure forward predicates along the same path.
Because each disjunct of a simple graph is rewritten to (at most) one forward tree, we notice an interesting query minimization byproduct: A tree query has the number of predicates bounded only in the number of their variables, thus independent of the number of predicates in the equivalent simple graph query.

—We report on related (theoretical and practical) work that also represents various application scenarios for the results of this paper.

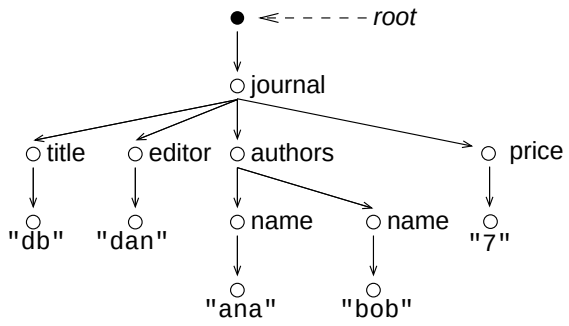
—An implementation of our rewriting framework adapted to XPath syntax is part of a publicly-available XPath rewriter used by the streaming XPath processor SPEX [Bry et al. 2005].

Structure of this article. Section 2 introduces the query language LGQ and the rewriting language $\text{LGQ}^{\rightarrow}$. Section 3 gives equivalence-preserving rewrite rules used in Section 4 to define three rewriting systems. These systems form the basis of our main expressiveness and complexity results given in Section 5. Section 6 details on work using our rewriting framework, and Section 7 concludes this article.

2. PRELIMINARIES

2.1 Trees

We consider finite unranked ordered trees with labeled nodes and one distinguished unlabeled *root* node. Such trees are abstractions of XML documents, as exemplified in Figure 1. The text content of XML documents is modeled as special nodes with quoted labels. All labels are words over a finite alphabet.




```
<journal>
  <title>db</title>
  <editor>dan</editor>
  <authors>
    <name>ana</name>
    <name>bob</name>
  </authors>
  <price>7</price>
</journal>
```

Fig. 1. Excerpt of a journal archive.

2.2 Queries

XPath is the prime language for expressing node-selecting queries on trees. In the following, we call XPath the navigational fragment of XPath 1.0 with additional axes (see below for the extensional predicates defining the supported axes), where each query is an absolute path or arbitrarily nested unions or differences of absolute paths [Olteanu 2004]. The qualifiers are restricted to only contain paths (thus no path comparisons, disjunctions or negations of paths). Occasionally, we further extend XPath with identity-based equality.

This language represents another syntax for a fragment of monadic non-recursive Datalog with negation over tree structures [Olteanu 2004]. For convenience, we call this fragment LGQ (logical graph queries). We prefer LGQ syntax over XPath syntax, because LGQ allows us to express rewriting rules more compactly.

We assume in the following familiarity with XPath and Datalog and introduce only some necessary technical machinery. Please refer to [Abiteboul et al. 1995] for

Datalog, to [Gottlob et al. 2004] for monadic Datalog over tree structures, and to [Clark and DeRose 1999] for XPath.

For defining LGQ, we restrict monadic non-recursive Datalog by (1) allowing negation only on intensional (or user-defined) unary predicates, (2) defining our own set of extensional (or built-in) predicates over trees, and (3) considering only absolute formulas as bodies of query rules. We detail next on the last two points.

LGQ Predicates and Atoms. We consider the base binary predicates $\mathsf{fstChild}$, $\mathsf{nextSibl}$, and self: for two nodes n and m, $\mathsf{fstChild}(n, m)$ holds if m is the first child of n, $\mathsf{nextSibl}(n, m)$ holds if m is the immediate next sibling of n, and $\mathsf{self}(n, m)$ holds if m is n.

For a base predicate α, its transitive closure α^+ and its reflexive transitive closure α^* are defined as usual by:

$$\alpha^0 = \mathsf{self}, \qquad \alpha^{n+1} = \alpha^n \circ \alpha, \qquad \alpha^+ = \bigcup_{n \in \mathbb{N} \setminus \{0\}} \alpha^n, \qquad \alpha^* = \bigcup_{n \in \mathbb{N}} \alpha^n.$$

Note that $\gamma = \alpha \circ \beta$ means $\gamma(x, z) = \alpha(x, y) \wedge \beta(y, z)$.

For the above base predicates and their closures, we consider also their inverses with appropriate names. Summing up, the following predicates are considered: self (equality), $\mathsf{nextSibl}$ (next sibling), $\mathsf{nextSibl}^+$ (next siblings), $\mathsf{nextSibl}^*$ (next siblings or self), $\mathsf{prevSibl} = \mathsf{nextSibl}^{-1}$ (previous sibling), $\mathsf{prevSibl}^+ = (\mathsf{nextSibl}^+)^{-1}$ (previous siblings), $\mathsf{prevSibl}^* = (\mathsf{nextSibl}^*)^{-1}$ (previous siblings or self), $\mathsf{fstChild}$ (first child), $\mathsf{child} = \mathsf{fstChild} \circ \mathsf{nextSibl}^*$ (children), child^+ (descendants), child^* (descendants or self), $\mathsf{par} = \mathsf{child}^{-1}$ (parent), $\mathsf{par}^+ = (\mathsf{child}^+)^{-1}$ (ancestors), $\mathsf{par}^* = (\mathsf{child}^*)^{-1}$ (ancestors or self).

We classify these predicates depending on the order and structural relations between the nodes of the contained pairs. If $\alpha(n, m)$ holds, then the predicate α is (1) *forward*, if m appears after n in document order, (2) *reverse*, if m appears before n in document order, (3) *horizontal*, if m is a sibling of n, or (4) *vertical*, if m is an ancestor or descendant of n. Exceptionally, the predicate self is considered forward.

Remark 2.1. Using these predicates, other existent XPath axes can be defined, e.g., $\mathsf{foll} = \mathsf{par}^* \circ \mathsf{nextSibl}^+ \circ \mathsf{child}^*$ (followings) and $\mathsf{prec} = \mathsf{foll}^{-1}$ (precedings). Note that $\mathsf{fstChild}$, $\mathsf{nextSibl}^*$, and $\mathsf{prevSibl}^*$ do not have corresponding XPath 1.0 axes. However, they can be expressed using disjunction or positional filters. □

The unary predicates of LGQ are represented by XPath nodetests and by intensional predicates. A *nodetest* is a construct from $\{n, n_{\neq}, \texttt{'t'}, \texttt{'t'}_{\neq}, \mathsf{root}\}$, where n and `'t'` are words over a finite alphabet and root is a special keyword. The nodetest predicate n ($n_{\neq}$) holds for nodes whose label is (not) n. The root nodetest holds only for the root node. Examples of nodetests are a, $\mathsf{a}_{\neq}$, $\texttt{'t'}$, $\texttt{'t'}_{\neq}$, where the latter two are written in quotes and refer to text content. An intensional predicate is defined by a rule, where the head is a unary atom with that predicate and the body is a formula.

Example 2.2. Consider the LGQ query

$$Q_1(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}(v_0, v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{par}(v_2, v) \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2)$$

that defines the intensional predicate Q_1 and makes use of three unary nodetest

predicates (root, journal, and editor) and of two distinct binary predicates (child and par). This query selects the parent nodes of editor nodes that are children of journal nodes, which in turn are children of the root node. For the tree of Figure 1, Q_1 selects the journal node. □

An LGQ formula is constructed using (nullary, unary, and binary) atoms and the standard connectives $\wedge$ (and), $\vee$ (or), and $\neg$ (not). LGQ only allows negation on intensional predicates. An atom is constructed from a predicate and a set of variables. For example, the binary atom $\mathsf{child}(v_1, v_2)$ uses the predicate child and the variables v_1 and v_2. Unary atoms are constructed using nodetest and intensional predicates. There are two nullary atoms $\bot$ and $\top$ useful for proofs and formula rewriting: $\bot$ holds for no input tree, whereas $\top$ holds for any input tree.

Given a binary atom $\alpha(v_1, v_2)$, v_1 is a *source* variable and v_2 is a *sink* variable. Given a conjunction of binary atoms, a *non*-source (non-sink) variable never appears as source (sink), and a *multi*-source (multi-sink) variable appears as source (sink) more than once. For a given rule, the variable occurring in the head is (implicitly) universally quantified, and all other variables in the body are (implicitly) existentially quantified.

Absolute Formulas. A formula is *absolute*, if (1) it has non-sink variables and (2) each non-sink variable has a root nodetest.

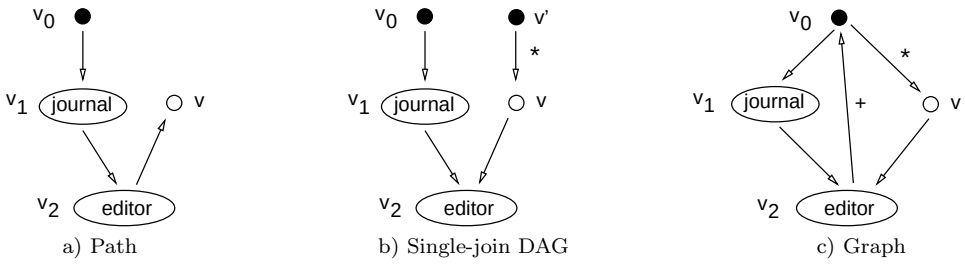
EXAMPLE 2.3. Consider again the query Q_1 of Example 2.2. The query body consists of a single disjunct of binary and unary atoms. The variables v_1 and v_2 are both source and sink, and the variable v is only sink. The remaining variable v_0 is the only non-sink variable and has a root nodetest. This makes the query body an absolute formula. Note that by dropping the root nodetest or any of the child atoms we obtain a formula that is not anymore absolute. □

Path, Tree, DAG, and Graph Formulas. A conjunction of LGQ atoms admits an intuitive graphical representation, where the variables induce nodes and the binary predicates directed edges with corresponding labels (in case of closure) and orientation (vertical/horizontal). For example, $\mathsf{nextSibl}^*$ induces a horizontal edge from left to right with label *, whereas par^+ induces a vertical bottom-up edge with label +. We fill in black the nodes corresponding to variables with root nodetests, and label each node with the nodetest (if any) of the corresponding variable.

The graphical representation of a formula with disjunction consists of the representations of each of its disjunct in disjunctive normal form. In case one of such conjunctions contains also a unary atom corresponding to a user-defined predicate, then the query body of that predicate is represented separately.

By analogy to their representations, formulas can be classified based on their structural complexities in paths, trees, forests, directed acyclic graphs (DAGs), and graphs. We also use the non-standard notion of single-join DAG to denote DAGs having distinct paths with at most one common sink node. Note that XPath queries are LGQ forests, and XPath queries with identity-based equality are single-join DAGs [Olteanu 2004].

EXAMPLE 2.4. Figure 2 shows graphically three formulas:
(below k is $\mathsf{root}(v_0) \wedge \mathsf{child}(v_0, v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2)$)



a) Path b) Single-join DAG c) Graph

Fig. 2. Examples of graphical representations for LGQ formulas.

—the path $p = k \wedge \mathsf{par}(v_2, v)$, which is the body of Q_1 in Example 2.2,
—the single-join DAG $d = k \wedge \mathsf{root}(v') \wedge \mathsf{child}^*(v', v) \wedge \mathsf{child}(v, v_2)$, and
—the graph with two cycles $g = k \wedge \mathsf{child}^*(v_0, v) \wedge \mathsf{child}(v, v_2) \wedge \mathsf{par}^+(v_2, v_0)$.

Each of the three formulas can induce a query with the distinguished variable v. Remarkably, although these queries have bodies with different structural complexities, they all select the same set of nodes as the query Q_1 in Example 2.2. □

Equivalence. We next define equivalence for LGQ formulas under the restriction that both formulas to test for equivalence have the same variables.

For a formula f and a tree T, an LGQ *valuation* maps each variable in f to a node in T, such that the formula obtained by replacing the variables in f by their mappings holds in T. A formula is *unsatisfiable* if for any tree it has no valuation. For example, the formula $\bot$ is unsatisfiable.

Consider two formulas l and r with the same variables. Then l is contained in r, noted $l \subseteq r$, if for any input tree any valuation of l is also a valuation of r. The formulas l and r are *equivalent*, noted $l \equiv r$, if for any input tree they have the same valuations. It is easy to see that two formulas identical up to equivalent subformulas are equivalent. Two queries $Q_1(v)$ and $Q_2(v)$ are equivalent if their bodies are equivalent.

Example 2.5. Consider the equivalent formulas l and r defined as follows:

$$l = \mathsf{nextSibl}(v_1, v_2) \wedge \mathsf{prevSibl}(v_2, v_3) \qquad r = \mathsf{nextSibl}(v_1, v_2) \wedge \mathsf{self}(v_3, v_1)$$

For both formulas, any non-empty valuation maps the variables v_1, v_2, and v_3 to nodes n_1, n_2, and n_1 respectively, such that n_2 is the next sibling of n_1. □

This syntactically restricted equivalence mostly suffices for the present article, because it is only used in conjunction with syntactic rewriting between equivalent formulas with the same variables. We further need two special equivalence cases between formulas and formulas either without variables (i.e., $\bot$ and $\top$), or with exactly one more non-sink variable. We next discuss these cases.

Example 2.6. Let $l_1 = \mathsf{nextSibl}(v_1, v_2) \wedge \mathsf{prevSibl}(v_1, v_2)$. Clearly, l_1 is unsatisfiable, because a node can not be at the same time a preceding and a following sibling of another node. Thus l_1 is equivalent to $\bot$.

The formula $l_2 \vee \top$ is equivalent to $\top$ for any l_2, because $\top$ is always satisfiable.

Finally, consider a formula l_3 and v one of its variables. Then, the formula $r_3 = l_3 \wedge \mathsf{child}^*(z, v) \wedge \mathsf{root}(z)$, where z is a new variable not occurring in l_3, is equivalent to l_3. This is because the formula $\mathsf{child}^*(z, v) \wedge \mathsf{root}(z)$ just states that v can be mapped to any node in the input tree, thus its addition does not change the set of valuations for l_3 and any input tree. □

2.3 Rewritings

Term rewriting systems are widely used as a model of computation to relate syntax and semantics. This article adopts the terminology for term rewriting systems used in [Baader and Nipkow 1998].

In order to express identities and rewritings of LGQ formulas, we define a language of rewriting rules and identities LGQ$^{\rightarrow}$, similar to LGQ. LGQ$^{\rightarrow}$ has two kinds of variables:

—variables ranging over LGQ formulas, written in upper case, e.g., X, Y, Z,

—variables ranging over LGQ variables, written in lower case and underlined, e.g., $\underline{x}$, $\underline{y}$, $\underline{z}$.

Note that the LGQ variables are written in lower case and not underlined, thus different from LGQ$^{\rightarrow}$ variables.

In LGQ$^{\rightarrow}$, the LGQ predicates are function symbols and LGQ formulas ground terms, i.e., terms without LGQ$^{\rightarrow}$ variables. LGQ$^{\rightarrow}$ has two binary predicates, the identity $\approx$ and the rewrite $\rightarrow$, both written in infix form. In the LGQ$^{\rightarrow}$ terms $s \approx t$ and $s \rightarrow t$, the term s is the left-hand side (lhs) and t is the right-hand side (rhs).

A LGQ$^{\rightarrow}$ *substitution* σ is a total mapping from LGQ$^{\rightarrow}$ variables to LGQ formulas or variables denoted by (1) $\{X_1 \mapsto s_1, \ldots, X_n \mapsto s_n\}$ indicating that the LGQ$^{\rightarrow}$ variable X_i maps to the LGQ formula s_i, or (2) $\{\underline{x_1} \mapsto s_1, \ldots, \underline{x_n} \mapsto s_n\}$ indicating that the LGQ$^{\rightarrow}$ variable $\underline{x_i}$ maps to the LGQ variable s_i. If σ maps an LGQ$^{\rightarrow}$ variable to an LGQ formula or variable, then that LGQ formula or variable is the *image* of the LGQ$^{\rightarrow}$ variable under σ. If an LGQ$^{\rightarrow}$ variable X (or $\underline{x_i}$) is not in the domain of σ, then $\sigma(X) = X$ (and $\sigma(\underline{x_i}) = \underline{x_i}$); if $f(t_1, t_2)$ is an LGQ$^{\rightarrow}$ term, then $\sigma(f(t_1, t_2)) = f(\sigma(t_1), \sigma(t_2))$. A substitution σ is a *matching* substitution of a LGQ$^{\rightarrow}$ term l to an LGQ formula t, if $\sigma(l) = t$. Under a matching substitution, the instances of lhs and rhs of a rewrite rule are LGQ formulas.

A *redex* is an instance of the lhs of a rewrite rule under a matching substitution. *Contracting* the redex means replacing it with the corresponding instance of the rhs of the rule. The *application* of a rewrite rule $lhs \rightarrow rhs$ to an LGQ formula s means contracting a redex $\sigma(lhs)$ in s to the rhs instance $\sigma(rhs)$, both under the matching substitution σ. A term s *derives* other term t, written $s \xrightarrow{*} t$, if t can be obtained from s after a finite (possibly empty) sequence of rewrites: $s \rightarrow \cdots \rightarrow t$. In this case, we say also that the term s is *reducible* (with respect to the relation $\rightarrow$). If there is no term t such that $s \xrightarrow{*} t$, then s is *irreducible*. If $s \xrightarrow{*} t$ and t is irreducible, then t is a *normal form* of s and we write $s \rightarrow^{!} t$.

A *term rewriting system* (LGQ$^{\rightarrow}$, $\rightarrow$) is a finite set of rewrite rules and (possibly) identities on LGQ$^{\rightarrow}$ terms. If identities are present, then they serve to specify rewriting modulo these identities (we detail later on this).

Termination. A term rewriting system is *terminating* if there are no infinite

derivations $s_0 \to s_1 \to \cdots$. The basic method to prove termination of $(\mathrm{LGQ}^{\rightarrow}, \to)$ is to embed it into another system $(A, >)$ that is known to terminate. This requires a monotone mapping ϕ from $\mathrm{LGQ}^{\rightarrow}$ to A, where $lhs \to rhs$ implies $\phi(lhs) > \phi(rhs)$.

Confluence. Two terms x and y are *joinable* for a relation $\to$, written x $\downarrow$ y, if there exists a term z such that $x \overset{*}{\to} z \overset{*}{\leftarrow} y$. A rewrite relation (and its system) is *confluent* if terms are joinable whenever they are derivable from a same term:

$$y_1 \overset{*}{\leftarrow} x \overset{*}{\to} y_2 \Rightarrow y_1 \downarrow y_2$$

In other words, confluent relations lead to the same irreducible term regardless of the order of rule applications.

A relation $\to$ is *locally confluent* if terms are joinable whenever they are derivable in one step from a same term

$$y_1 \leftarrow x \to y_2 \Rightarrow y_1 \downarrow y_2$$

A rewrite relation is locally confluent if (but not only if) no lhs unifies with a non-variable subterm (except itself) of any lhs, taking into account that variables appearing in two rules (or in two instances of the same rule) are always treated as disjoint. In cases when the above requirement is not fulfilled, we get so-called critical pairs, i.e., pairs of different terms derivable in one step from a same term. Local confluence can be shown now by proving joinability of all critical pairs.

Confluence can be reduced to local confluence only for rewrite relations that terminate [Newman 1942]. Termination and confluence ensure the *existence and uniqueness* of normal forms.

Rewriting modulo AC-theory. The LGQ predicates $\wedge$, $\vee$, and self are associative and commutative (AC). Such properties should be taken into account when applying rewrite rules. For example, the rewrite rule

$$\mathsf{child}(\underline{x}, \underline{y}) \wedge \mathsf{prevSibl}(\underline{y}, \underline{z}) \to \mathsf{child}(\underline{x}, \underline{z}) \wedge \mathsf{nextSibl}(\underline{z}, \underline{y})$$

should rewrite not only $\mathsf{child}(a, b) \wedge \mathsf{prevSibl}(b, c)$ to $\mathsf{child}(a, c) \wedge \mathsf{nextSibl}(c, b)$, but also, as highly desired, $\mathsf{prevSibl}(b, c) \wedge f \wedge \mathsf{child}(a, b)$ to $\mathsf{nextSibl}(c, b) \wedge \mathsf{child}(a, c) \wedge f$. Note that a syntactical matching fails in the latter case. What is needed is an equational matching that takes into account the AC properties of the $\wedge$ predicate.

The AC properties of the LGQ predicates $\wedge$, $\vee$, and self raise problems, because they can not be oriented into terminating rewrite rules. A common technique to accommodate them in the rewriting process is to consider rewriting modulo the AC-theory. More specifically, this article considers rewriting systems containing the following AC identities for $\wedge$, $\vee$, and self (α is any LGQ binary predicate):

$$X \wedge Y \approx Y \wedge X \qquad X \wedge (Y \wedge Z) \approx (X \wedge Y) \wedge Z$$

$$X \vee Y \approx Y \vee X \qquad X \vee (Y \vee Z) \approx (X \vee Y) \vee Z$$

$$\mathsf{self}(\underline{x}, \underline{y}) \approx \mathsf{self}(\underline{y}, \underline{x}) \qquad \mathsf{self}(\underline{x}, \underline{y}) \wedge \alpha(\underline{y}, \underline{z}) \approx \mathsf{self}(\underline{x}, \underline{y}) \wedge \alpha(\underline{x}, \underline{z})$$

3. EQUIVALENCE-PRESERVING REWRITE RULES

A query $Q(v) \leftarrow f$ is rewritten to an equivalent forward query $Q'(v) \leftarrow f'$ by rewriting the formula f, which represents the body of Q, to the equivalent forward

formula f', which represents the body of Q'. If f contains (possibly negated) user-defined predicates, then their bodies are rewritten as well.

We give next equivalence-preserving LGQ$^{\rightarrow}$ rules for rewriting LGQ formulas.

3.1 Rule adding single-join DAG Structure

We first consider a simple yet powerful equivalence-preserving rule, which can be used to rewrite any formula to an equivalent forward formula.

LEMMA 3.1. *The following rule rewrites formulas to equivalent formulas.*

$$\alpha(\underline{x},\underline{y}) \rightarrow \alpha^{-1}(\underline{y},\underline{x}) \wedge \mathsf{child}^*(z,\underline{y}) \wedge \mathsf{root}(z) \tag{1}$$

The variable z is a fresh LGQ variable and α is a reverse predicate.

PROOF. Let s be the input formula and t the result of rule application. Let also σ be an LGQ$^{\rightarrow}$ substitution, $l = \alpha(\sigma(\underline{x}),\sigma(\underline{y}))$ and $r = \alpha^{-1}(\sigma(\underline{y}),\sigma(\underline{x}))$. Then, $s \equiv t$ because (i) $l \equiv r$, (ii) s and t are identical up to the equivalent formulas l and r, and (iii) $\mathsf{child}^*(z,\sigma(\underline{y})) \wedge \mathsf{root}(z)$ does not restrict the possible mappings of $\sigma(\underline{y})$, for it just states that $\sigma(\underline{y})$ can be bound to any node in the input data tree. □

Using Rule (1), each reverse predicate is replaced by its forward counterpart and a dummy non-sink variable with a root nodetest is added to t to ensure that it is absolute. Note that in case the LGQ variable substituting $\underline{x}$ is a sink of one atom in s, then it becomes multi-sink in t. This makes that t has at least the structure of a single-join DAG, even if s is a forest.

EXAMPLE 3.2. Consider the query Q_1 from Example 2.2

$$Q_1(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}(v_0,v_1) \wedge \mathsf{child}(v_1,v_2) \wedge \mathsf{par}(v_2,v) \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2)$$

According to Rule (1), Q_1 is equivalent to

$$\begin{aligned} FQ_1(v) \leftarrow\ & \mathsf{root}(v_0) \wedge \mathsf{child}(v_0,v_1) \wedge \mathsf{child}(v_1,v_2) \wedge \mathsf{child}(v,v_2) \wedge \mathsf{child}^*(v',v) \wedge \\ & \mathsf{root}(v') \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2). \end{aligned}$$

The bodies of Q_1 and FQ_1 are depicted in Figures 2a) and 2b) respectively.

Using XPath syntax, Q_1 is /child::journal/child::editor/parent::node() and FQ_1 is /descendant-or-self::node()[child::node() == /child::journal/child::editor]. □

REMARK 3.3. Rule (1) can also be expressed using XPath with identity-based equality == [Olteanu et al. 2002]. Let P be a rule variable standing for a relative path, N a nodetest holder, R a reverse axis, and F its symmetrical forward.

$$\begin{aligned} /P/R::N &\rightarrow /\mathsf{descendant\text{-}or\text{-}self}::N[F::\mathsf{node}() == /P] \\ P[R::N] &\rightarrow P[/\mathsf{descendant\text{-}or\text{-}self}::N/F::\mathsf{node}() == \mathsf{self}::\mathsf{node}()] \end{aligned}$$

The above two rules using XPath syntax are harder to grasp than Rule (1): Due to XPath syntax, the rules have to consider (1) both cases of reverse predicates inside and outside filters (because XPath filters are syntactically delimited by square brackets), and (2) additional nodetest predicates. In LGQ the subformulas corresponding to XPath filters are not explicitly marked, and the nodetest predicates are not necessary for expressing the rule and therefore not present. □

3.2 Rules preserving Tree Structure

We next consider equivalence-preserving rules that can be used to rewrite any formula to an equivalent forward formula, such that the latter does not have more multi-sink variables than the former. This ensures that forest formulas are rewritten to forest forward equivalents.

In contrast to Rule (1), these new rules exploit the treeness of the input data and the relationships between the various forward and reverse predicates taken pairwise. Lemma 3.4 gives 20 such rules, representing each possible combination of the forward predicates $\mathsf{fstChild}$, child, child^+, $\mathsf{nextSibl}$, and $\mathsf{nextSibl}^+$, and the reverse predicates par, par^+, $\mathsf{prevSibl}$, and $\mathsf{prevSibl}^+$. Note that the combinations of the predicate self with other predicates are already covered by the AC-identities defined in Section 2.3. The reflexive transitive closure predicates are safely left out of discussion because they are reducible to the above cases as follows:

$$\alpha^*(\underline{x},\underline{y}) \rightarrow (\alpha^+(\underline{x},\underline{y}) \vee \mathsf{self}(\underline{x},\underline{y})). \tag{2}$$

LEMMA 3.4. *The rules of Figure 3 rewrite formulas to equivalent formulas.*

PROOF. Let s be the input formula and t the result of a rule application. For an instance $l \rightarrow r$ of each Rule (3) through (22) under an LGQ$^{\rightarrow}$ substitution $\sigma = \{\underline{x} \mapsto x, \underline{y} \mapsto y, \underline{z} \mapsto z\}$, we show that (i) $l \equiv r$, and (ii) $s \equiv t$, where t is obtained by replacing l by r in s.

We use the following implications and equivalences, where h is a horizontal predicate, $v_1, v_2 \in \{\mathsf{fstChild}, \mathsf{child}\}$, and α is any binary predicate:

$$\begin{aligned}
v_1(y,x) \wedge v_2(z,x) &\Rightarrow \mathsf{self}(y,z) && (Treeness) && (a)\\
\mathsf{nextSibl}(y,x) \wedge \mathsf{nextSibl}(z,x) &\Rightarrow \mathsf{self}(y,z) && (Treeness) && (b)\\
h(x,y) &\Rightarrow \mathsf{child}(z,x) \wedge \mathsf{child}(z,y) && (Siblings) && (c)\\
\alpha^+(x,y) &\equiv \alpha^*(x,z) \wedge \alpha(z,y) && (Closure) && (d)
\end{aligned}$$

Note that the variables appearing on the left-side of $\Rightarrow$ or $\equiv$ are universally quantified, whereas the others are existentially quantified.

Rules (3) and (4) are similar. We only prove Rule (3).

$$\begin{aligned}
\mathsf{child}(x,y) \wedge \mathsf{par}(y,z) &\equiv \mathsf{child}(x,y) \wedge \mathsf{child}(z,y) \overset{a}{\equiv} \mathsf{child}(x,y) \wedge \mathsf{child}(z,y) \wedge \mathsf{self}(x,z)\\
&\equiv \mathsf{child}(z,y) \wedge \mathsf{self}(x,z).
\end{aligned}$$

Rule (5).

$$\begin{aligned}
\mathsf{child}^+(x,y) \wedge \mathsf{par}(y,z) &\overset{d}{\equiv} \mathsf{child}^*(x,p) \wedge \mathsf{child}(p,y) \wedge \mathsf{child}(z,y)\\
&\overset{a}{\equiv} \mathsf{self}(p,z) \wedge \mathsf{child}^*(x,z) \wedge \mathsf{child}(z,y) \equiv \mathsf{child}^*(x,z) \wedge \mathsf{child}(z,y).
\end{aligned}$$

Rules (6) and (7) are similar. We only prove Rule (6).

$$\begin{aligned}
\mathsf{nextSibl}(x,y) \wedge \mathsf{par}(y,z) &\equiv \mathsf{nextSibl}(x,y) \wedge \mathsf{child}(z,y)\\
&\overset{c}{\equiv} \mathsf{nextSibl}(x,y) \wedge \mathsf{child}(z,y) \wedge \mathsf{child}(p,x) \wedge \mathsf{child}(p,y)\\
&\overset{a}{\equiv} \mathsf{nextSibl}(x,y) \wedge \mathsf{child}(z,y) \wedge \mathsf{child}(z,x) \wedge \mathsf{self}(p,y)\\
&\equiv \mathsf{nextSibl}(x,y) \wedge \mathsf{child}(z,x) \equiv \mathsf{nextSibl}(x,y) \wedge \mathsf{par}(x,z).
\end{aligned}$$

$$\mathsf{fstChild}(\underline{x},\underline{y}) \wedge \mathsf{par}(\underline{y},\underline{z}) \rightarrow \mathsf{self}(\underline{x},\underline{z}) \wedge \mathsf{fstChild}(\underline{z},\underline{y}) \tag{3}$$

$$\mathsf{child}(\underline{x},\underline{y}) \wedge \mathsf{par}(\underline{y},\underline{z}) \rightarrow \mathsf{self}(\underline{x},\underline{z}) \wedge \mathsf{child}(\underline{z},\underline{y}) \tag{4}$$

$$\mathsf{child}^+(\underline{x},\underline{y}) \wedge \mathsf{par}(\underline{y},\underline{z}) \rightarrow \mathsf{child}^*(\underline{x},\underline{z}) \wedge \mathsf{child}(\underline{z},\underline{y}) \tag{5}$$

$$\mathsf{nextSibl}(\underline{x},\underline{y}) \wedge \mathsf{par}(\underline{y},\underline{z}) \rightarrow \mathsf{nextSibl}(\underline{x},\underline{y}) \wedge \mathsf{par}(\underline{x},\underline{z}) \tag{6}$$

$$\mathsf{nextSibl}^+(\underline{x},\underline{y}) \wedge \mathsf{par}(\underline{y},\underline{z}) \rightarrow \mathsf{nextSibl}^+(\underline{x},\underline{y}) \wedge \mathsf{par}(\underline{x},\underline{z}) \tag{7}$$

$$\mathsf{fstChild}(\underline{x},\underline{y}) \wedge \mathsf{par}^+(\underline{y},\underline{z}) \rightarrow (\mathsf{fstChild}(\underline{x},\underline{y}) \wedge \mathsf{par}^+(\underline{x},\underline{z}) \vee \mathsf{fstChild}(\underline{x},\underline{y}) \wedge \mathsf{self}(\underline{x},\underline{z})) \tag{8}$$

$$\mathsf{child}(\underline{x},\underline{y}) \wedge \mathsf{par}^+(\underline{y},\underline{z}) \rightarrow (\mathsf{child}(\underline{x},\underline{y}) \wedge \mathsf{par}^+(\underline{x},\underline{z}) \vee \mathsf{child}(\underline{x},\underline{y}) \wedge \mathsf{self}(\underline{x},\underline{z})) \tag{9}$$

$$\mathsf{child}^+(\underline{x},\underline{y}) \wedge \mathsf{par}^+(\underline{y},\underline{z}) \rightarrow (\mathsf{child}^+(\underline{x},\underline{y}) \wedge \mathsf{par}^+(\underline{x},\underline{z}) \vee \mathsf{child}^*(\underline{x},\underline{z}) \wedge \mathsf{child}^+(\underline{z},\underline{y})) \tag{10}$$

$$\mathsf{nextSibl}(\underline{x},\underline{y}) \wedge \mathsf{par}^+(\underline{y},\underline{z}) \rightarrow \mathsf{nextSibl}(\underline{x},\underline{y}) \wedge \mathsf{par}^+(\underline{x},\underline{z}) \tag{11}$$

$$\mathsf{nextSibl}^+(\underline{x},\underline{y}) \wedge \mathsf{par}^+(\underline{y},\underline{z}) \rightarrow \mathsf{nextSibl}^+(\underline{x},\underline{y}) \wedge \mathsf{par}^+(\underline{x},\underline{z}) \tag{12}$$

$$\mathsf{fstChild}(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}(\underline{y},\underline{z}) \rightarrow \bot \tag{13}$$

$$\mathsf{child}(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}(\underline{y},\underline{z}) \rightarrow \mathsf{child}(\underline{x},\underline{z}) \wedge \mathsf{nextSibl}(\underline{z},\underline{y}) \tag{14}$$

$$\mathsf{child}^+(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}(\underline{y},\underline{z}) \rightarrow \mathsf{child}^+(\underline{x},\underline{z}) \wedge \mathsf{nextSibl}(\underline{z},\underline{y}) \tag{15}$$

$$\mathsf{nextSibl}(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}(\underline{y},\underline{z}) \rightarrow \mathsf{self}(\underline{x},\underline{z}) \wedge \mathsf{nextSibl}(\underline{z},\underline{y}) \tag{16}$$

$$\mathsf{nextSibl}^+(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}(\underline{y},\underline{z}) \rightarrow \mathsf{nextSibl}^*(\underline{x},\underline{z}) \wedge \mathsf{nextSibl}(\underline{z},\underline{y}) \tag{17}$$

$$\mathsf{fstChild}(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}^+(\underline{y},\underline{z}) \rightarrow \bot \tag{18}$$

$$\mathsf{child}(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}^+(\underline{y},\underline{z}) \rightarrow \mathsf{child}(\underline{x},\underline{z}) \wedge \mathsf{nextSibl}^+(\underline{z},\underline{y}) \tag{19}$$

$$\mathsf{child}^+(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}^+(\underline{y},\underline{z}) \rightarrow \mathsf{child}^+(\underline{x},\underline{z}) \wedge \mathsf{nextSibl}^+(\underline{z},\underline{y}) \tag{20}$$

$$\mathsf{nextSibl}(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}^+(\underline{y},\underline{z}) \rightarrow (\mathsf{nextSibl}(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}^+(\underline{x},\underline{z}) \vee \mathsf{nextSibl}(\underline{x},\underline{y}) \wedge \mathsf{self}(\underline{x},\underline{z})) \tag{21}$$

$$\mathsf{nextSibl}^+(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}^+(\underline{y},\underline{z}) \rightarrow (\mathsf{nextSibl}^+(\underline{x},\underline{y}) \wedge \mathsf{prevSibl}^+(\underline{x},\underline{z}) \vee \mathsf{nextSibl}^*(\underline{x},\underline{z}) \wedge \mathsf{nextSibl}^+(\underline{z},\underline{y})) \tag{22}$$

Fig. 3. Equivalence-preserving rules for paths of one forward and one reverse atom.

Rules (8) and (9) are similar. We only prove Rule (9).

$$\begin{aligned}\mathsf{child}(x,y) \wedge \mathsf{par}^+(y,z) \wedge &\equiv \mathsf{child}^+(z,y) \wedge \mathsf{par}(y,x) \overset{+}{\equiv} \mathsf{child}^*(z,x) \wedge \mathsf{child}(x,y)\\ &\overset{d}{\equiv} \mathsf{child}^+(z,x) \wedge \mathsf{child}(x,y) \vee \mathsf{self}(z,x) \wedge \mathsf{child}(x,y)\\ &\equiv \mathsf{par}^+(x,z) \wedge \mathsf{child}(x,y) \vee \mathsf{self}(x,z) \wedge \mathsf{child}(x,y).\end{aligned}$$

Equivalence (+) holds due to Rule (5).

Rule (10). We next denote the left-right depth-first preorder relation by $\ll$. Consider a valuation τ for $\mathsf{child}^+(x,y) \wedge \mathsf{par}^+(y,z)$. Then, the mappings of the variables x, y, and z are along a same path from the root and there is a partial order between them: $\tau(x) \ll \tau(y)$, $\tau(z) \ll \tau(y)$. The possibilities for the order between $\tau(x)$ and $\tau(z)$ are (1) $\tau(z) \ll \tau(x)$, (2) $\tau(x) = \tau(z)$, and (3) $\tau(x) \ll \tau(z)$. This reads also (1) $\tau(z)$ is an ancestor of $\tau(x)$, (2) $\tau(x)$ is the same as $\tau(z)$, (3) $\tau(x)$

is an ancestor of $\tau(z)$. Thus $\tau(z)$ lies between $\tau(x)$ and $\tau(y)$. The LGQ encoding of all these possibilities is:

$$\begin{aligned}&\mathsf{child}^+(x,y) \wedge \mathsf{par}^+(x,z) \vee \mathsf{child}^+(x,y) \wedge \mathsf{self}(x,z) \vee \mathsf{child}^+(x,z) \wedge \mathsf{child}^+(z,y)\\ &\equiv \mathsf{child}^+(x,y) \wedge \mathsf{par}^+(x,z) \vee \mathsf{child}^*(x,z) \wedge \mathsf{child}^+(z,y).\end{aligned}$$

Rules (11) and (12) are similar. We only prove Rule (11).

$$\begin{aligned}\mathsf{nextSibl}(x,y) \wedge \mathsf{par}^+(y,z) &\stackrel{b}{\equiv} \mathsf{nextSibl}(x,y) \wedge \mathsf{child}(p,x) \wedge \mathsf{child}(p,y) \wedge \mathsf{par}^+(y,z)\\ &\stackrel{d}{\equiv} \mathsf{nextSibl}(x,y) \wedge \mathsf{child}(p,x) \wedge \mathsf{child}(p,y) \wedge \mathsf{par}^*(r,z) \wedge \mathsf{par}(y,r)\\ &\equiv \mathsf{nextSibl}(x,y) \wedge \mathsf{child}(p,x) \wedge \mathsf{child}(p,y) \wedge \mathsf{par}^*(r,z) \wedge \mathsf{child}(r,y)\\ &\stackrel{a}{\equiv} \mathsf{self}(p,r) \wedge \mathsf{nextSibl}(x,y) \wedge \mathsf{child}(r,x) \wedge \mathsf{child}(r,y) \wedge \mathsf{par}^*(r,z)\\ &\stackrel{d}{\equiv} \mathsf{self}(p,r) \wedge \mathsf{nextSibl}(x,y) \wedge \mathsf{child}(r,y) \wedge \mathsf{par}^+(x,z)\\ &\equiv \mathsf{nextSibl}(x,y) \wedge \mathsf{par}^+(x,z).\end{aligned}$$

Rule (13) follows from the definition of $\mathsf{fstChild}(n,m)$: m is the first child of n, thus $\mathsf{fstChild}(x,y) \wedge \mathsf{prevSibl}(y,z) \equiv \bot$.

Rule (14).

$$\begin{aligned}\mathsf{child}(x,y) \wedge \mathsf{prevSibl}(y,z) &\equiv \mathsf{child}(x,y) \wedge \mathsf{nextSibl}(z,y)\\ &\stackrel{c}{\equiv} \mathsf{child}(x,y) \wedge \mathsf{nextSibl}(z,y) \wedge \mathsf{child}(p,z) \wedge \mathsf{child}(p,y)\\ &\stackrel{a}{\equiv} \mathsf{self}(p,x) \wedge \mathsf{child}(x,z) \wedge \mathsf{nextSibl}(z,y) \wedge \mathsf{child}(x,y)\\ &\equiv \mathsf{child}(x,z) \wedge \mathsf{nextSibl}(z,y).\end{aligned}$$

Rule (15).

$$\begin{aligned}\mathsf{child}^+(x,y) \wedge \mathsf{prevSibl}(y,z) &\stackrel{d}{\equiv} \mathsf{child}^*(x,p) \wedge \mathsf{child}(p,y) \wedge \mathsf{nextSibl}(z,y)\\ &\stackrel{c}{\equiv} \mathsf{child}^*(x,p) \wedge \mathsf{child}(p,y) \wedge \mathsf{nextSibl}(z,y) \wedge \mathsf{child}(r,z) \wedge \mathsf{child}(r,y)\\ &\stackrel{a}{\equiv} \mathsf{self}(p,r) \wedge \mathsf{child}^*(x,r) \wedge \mathsf{child}(r,y) \wedge \mathsf{nextSibl}(z,y) \wedge \mathsf{child}(r,z)\\ &\stackrel{d}{\equiv} \mathsf{child}^+(x,z) \wedge \mathsf{nextSibl}(z,y).\end{aligned}$$

Rule (16).

$$\mathsf{nextSibl}(x,y) \wedge \mathsf{prevSibl}(y,z) \equiv \mathsf{nextSibl}(x,y) \wedge \mathsf{nextSibl}(z,y) \stackrel{b}{\equiv} \mathsf{nextSibl}(z,y) \wedge \mathsf{self}(x,z).$$

All remaining rules have proofs similar to some other rules discussed above. More precisely, (17) is similar to (5), (18) to (13), (19) to (14), (20) to (15), (21) to (9), and (22) to (10).

The second part of the proof follows from the fact that s and t are either identical up to l and r respectively, or t is $\bot$. □

Example 3.5. Consider again the LGQ path query from Example 3.2

$$Q_1(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}(v_0,v_1) \wedge \mathsf{child}(v_1,v_2) \wedge \mathsf{par}(v_2,v) \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2)$$

According to Rule (4), Q_1 is equivalent to

$$FQ_2(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}(v_0, v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{self}(v_1, v) \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2)$$

or more compact, by replacing all occurences of v_1 by v

$$FQ'_2(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}(v_0, v) \wedge \mathsf{child}(v, v_2) \wedge \mathsf{journal}(v) \wedge \mathsf{editor}(v_2).$$

Note that the path FQ'_2 is structurally simpler and smaller than the DAG FQ_1 obtained by Rule (1) in Example 3.2.

Consider now the LGQ DAG query

$$Q_3(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{nextSibl}(v_1, v_2) \wedge \mathsf{par}(v_2, v) \wedge \mathsf{name}(v_1) \wedge \mathsf{name}(v_2) \wedge \mathsf{authors}(v)$$

that selects the authors nodes descendants of the root and parents of name nodes that immediately follow a name sibling node descendant of the root. For the tree of Figure 1, Q_3 selects the authors node. According to Rule (6), Q_3 is equivalent to

$$FQ_3(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{par}(v_1, v) \wedge \mathsf{nextSibl}(v_1, v_2) \wedge \mathsf{name}(v_1) \wedge \mathsf{name}(v_2) \wedge \mathsf{authors}(v)$$

because the parent of a sibling node (v_2) of a node (v_1) is also a parent of that node (v_1). Going further, Rule (5) can be applied now and yields

$$FQ'_3(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^*(v_0, v) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{child}(v, v_1) \wedge \mathsf{nextSibl}(v_1, v_2) \wedge \mathsf{name}(v_1) \wedge \mathsf{name}(v_2) \wedge \mathsf{authors}(v)$$

because the parent of a node descendant of the root is either the root or a descendant of the root, both having a child. Also, because between v_0 and v hold at the same time the relation child^* and a subset of it child^+, FQ'_3 can be further compacted to

$$FQ''_3(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{child}(v, v_1) \wedge \mathsf{nextSibl}(v_1, v_2) \wedge \mathsf{name}(v_1) \wedge \mathsf{name}(v_2) \wedge \mathsf{authors}(v)$$

Note that FQ''_3 is an LGQ path, thus expressible in XPath, whereas its equivalent original Q_3 is only expressible in XPath with identity-based equality. □

Remark 3.6. Rules (3) through (22) can be also expressed using XPath. E.g., Rule (5) can be expressed as (N and M are nodetest variables)

$$\mathsf{descendant::N/parent::M} \rightarrow \mathsf{descendant\text{-}or\text{-}self::N[child::M]}$$
$$\mathsf{descendant::N[parent::M]} \rightarrow \mathsf{descendant\text{-}or\text{-}self::N/child::M}.$$

As explained in Remark 3.3 for Rule (1), two rules are necessary in XPath to express Rule (5), for the case of reverse steps inside and outside filters. □

3.3 Rule removing DAG Structure

Rules (1) through (22) transform formulas to equivalent formulas that have in general more complex structure, due to additional disjunctions or multi-sink variables. Rule (23) given next trades multi-sink variables for reverse atoms in the hope of producing formulas with less complex structure. Section 4 shows later that any formula (including DAGs and graphs) can be rewritten to an equivalent forest forward formula, by interplaying the elimination of multi-sink variables, as done by

Rule (23), and the rewriting of the introduced reverse atoms to forest forward formulas, as done by Rules (3) through (22).

LEMMA 3.7. *The following rule rewrites formulas to equivalent formulas.*

$$fwd_1(\underline{x},\underline{y}) \wedge fwd_2(\underline{z},\underline{y}) \rightarrow fwd_1(\underline{x},\underline{y}) \wedge fwd_2^{-1}(\underline{y},\underline{z}). \tag{23}$$

The predicates fwd_1 and fwd_2 are forward.

REMARK 3.8. The rhs of Rule (23) can not be expressed in XPath, even extended with the identity-based equality: turning the formula $fwd_2(\underline{z},\underline{y})$ into $fwd_2^{-1}(\underline{y},\underline{z})$ would mean in XPath to loose the implicit context node corresponding to the LGQ variable instance of $\underline{z}$. □

EXAMPLE 3.9. Consider the LGQ DAG query

$$Q_4(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}(v_0,v_1) \wedge \mathsf{child}(v_1,v_2) \wedge \mathsf{child}(v,v_2) \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2)$$

that selects the parent node of an editor node that is child of a journal node, which is in turn a child of the root node. For the tree of Figure 1, Q_4 selects the journal node. According to Rule (23), Q_4 is equivalent to Q_1 of Example 3.2, which can be rewritten further to the equivalent forward path FQ_2'.

In other words, Rule (23) helps us to find a forward path formula, which is of course expressible in XPath, equivalent to the more complex DAG Q_4, expressible only in XPath with identity-based equality. □

3.4 Normalization and Simplification Rules

Some of Rules (2) through (22) introduce disjunctions nested in conjunctions, although these rules can only be applied to conjunctions of atoms, thus to formulas in disjunctive normal form (DNF). Therefore, we sometimes need to bring the input formula in DNF. Lemma 3.10 recalls the equivalence-preserving rules for DNF.

LEMMA 3.10. *The following rules rewrite formulas to equivalent formulas.*

$$X \wedge (Y \vee Z) \rightarrow X \wedge Y \vee X \wedge Z \tag{24}$$

$$X \vee (Y \vee Z) \rightarrow X \vee Y \vee Z \tag{25}$$

$$(Y \wedge Z) \rightarrow Y \wedge Z. \tag{26}$$

LGQ (and XPath) allows unsatisfiable and redundant formulas. Lemma 3.12 gives next a set of straightforward simplification rules. The benefit of such simplifications is twofold: They reduce the size of the formulas and usually lead to structurally simpler formulas.

EXAMPLE 3.11. The formula $\mathsf{child}(x,x)$ is an unsatisfiable formula, because no node is the child of itself. In the DAG formula $\mathsf{child}(x,y) \wedge \mathsf{child}^+(x,y)$ the atom $\mathsf{child}^+(x,y)$ is redundant, because under any non-empty valuation τ, both predicates child and child^+ hold on $(\tau(x),\tau(y))$. □

LEMMA 3.12. *Consider the pairs of different nodetests (l, r) and ($l_{\neq}$,$r_{\neq}$) and the non-reflexive predicates vh, reverse r, and forward f. Then, the following rules rewrite formulas to equivalent formulas.*

(Un)satisfiability Detection

$$vh(\underline{x},\underline{x}) \to \bot \tag{27}$$

$$l(\underline{x}) \wedge r(\underline{x}) \to \bot \qquad l_{\neq}(\underline{x}) \wedge r_{\neq}(\underline{x}) \to \bot \tag{28}$$

$$\mathsf{self}(\underline{x},\underline{x}) \to \top \tag{29}$$

$$\mathsf{root}(\underline{x}) \wedge r(\underline{x},\underline{y}) \to \bot \tag{30}$$

$$\mathsf{root}(\underline{x}) \wedge f(\underline{y},\underline{x}) \to \bot \tag{31}$$

(Un)satisfiability Propagation

$$X \wedge \bot \to \bot \tag{32}$$

$$X \vee \bot \to X \tag{33}$$

$$X \wedge \top \to X \tag{34}$$

$$X \vee \top \to \top \tag{35}$$

Duplicate elimination

$$X \wedge X \to X \tag{36}$$

$$X \vee X \to X \tag{37}$$

Note that more complex redundancies can be detected and eliminated by an interplay of the rules given in Section 3 and the rules of Lemma 3.12.

REMARK 3.13. Several other rules for navigation compaction and (un)satisfiability detection can be derived using the already introduced rules:

$$\alpha_1(\underline{x},\underline{y}) \wedge \alpha_2(\underline{x},\underline{y}) \to \alpha_1(\underline{x},\underline{y}) \tag{38}$$

$$\mathit{refl}(\underline{x},\underline{x}) \to \top \tag{39}$$

$$v(\underline{x},\underline{y}) \wedge h(\underline{x},\underline{y}) \to \bot \tag{40}$$

$$v(\underline{x},\underline{y}) \wedge h(\underline{y},\underline{x}) \to \bot \tag{41}$$

$$vh(\underline{x},\underline{y}) \wedge vh(\underline{y},\underline{x}) \to \bot \tag{42}$$

The above rules use the non-reflexive predicates vh, vertical v, and horizontal h, and the reflexive predicate *refl*. Additionally, $(\alpha_1, \alpha_2) \in \{(\mathsf{self}, \mathsf{child}^*), (\mathsf{self}, \mathsf{nextSibl}^*), (\mathsf{child}, \mathsf{child}^+), (\mathsf{child}, \mathsf{child}^*), (\mathsf{child}^+, \mathsf{child}^*), (\mathsf{nextSibl}, \mathsf{nextSibl}^+), (\mathsf{nextSibl}, \mathsf{nextSibl}^*), (\mathsf{nextSibl}^+, \mathsf{nextSibl}^*)\}$.

We next show how Rule (38) can be inferred using the existing ones, for the case that $(\alpha_1, \alpha_2) = (\mathsf{child}, \mathsf{child}^+)$.

$$\begin{aligned}
&\mathsf{child}(x,y) \wedge \mathsf{child}^+(x,y) \overset{(23)}{\to} \mathsf{child}(x,y) \wedge \mathsf{par}^+(y,x)\\
&\overset{(9)}{\to} \mathsf{child}(x,y) \wedge \mathsf{par}^+(x,x) \vee \mathsf{child}(x,y) \wedge \mathsf{self}(x,x)\\
&\overset{(27)}{\to} \mathsf{child}(x,y) \wedge \bot \vee \mathsf{child}(x,y) \wedge \mathsf{self}(x,x) \overset{(32)}{\to} \bot \vee \mathsf{child}(x,y) \wedge \mathsf{self}(x,x)\\
&\overset{(33)}{\to} \mathsf{child}(x,y) \wedge \mathsf{self}(x,x) \overset{(29)}{\to} \mathsf{child}(x,y) \wedge \top \overset{(34)}{\to} \mathsf{child}(x,y).
\end{aligned}$$

Additionally, the following rule for navigation compaction can not be derived from the existing ones and proves useful in practical cases

$$\alpha_1(\underline{x},\underline{y}) \vee \alpha_2(\underline{x},\underline{y}) \to \alpha_2(\underline{x},\underline{y}) \tag{43}$$

4. THREE REWRITING SYSTEMS

Using the rewrite rules defined in Section 3, we can rewrite LGQ formulas to forward LGQ formulas. These rules are distributed non-disjunctively in three sets that define three term rewriting systems:

—TRS_1 is defined by $\mathrm{LGQ}^{\rightarrow}$ and Rule (1),

—TRS_2 is defined by $\mathrm{LGQ}^{\rightarrow}$ and Rules (2) through (22) and (24) through (37),

—TRS_3 is TRS_2 extended with Rule (23).

Recall from Section 2.3 that these systems use AC-rewriting, because of the identities expressing the associativity and commutativity properties of $\wedge$, $\vee$, and self.

The three rewriting systems enjoy properties like termination, soundness and completeness, and confluence. Before elaborating on these properties, we demonstrate the benefits of such systems by means of an example.

4.1 Rewriting Example

We show how a complex graph formula is rewritten to equivalent forward formulas using each of the three rewriting systems. To help the reader, we use the graphical representation of formulas introduced in Section 2.2.

Consider the LGQ graph formula g

$$\begin{aligned}\mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{nextSibl}^*(v_1, v_2) \wedge \mathsf{par}^+(v_2, v_0) \wedge \mathsf{prevSibl}(v_2, v_3)\\ \wedge \mathsf{prevSibl}(v_3, v_1) \wedge \mathsf{child}(v_3, v_4) \wedge \mathsf{prevSibl}(v_5, v_4) \wedge \mathsf{child}^+(v_0, v_5).\end{aligned}$$

Figure 4 shows a possible sequence of rule applications for rewriting g to equivalent forward formulas d and t. The thick edges represent the predicates that are considered next in the rewriting process. Each thick rewrite arrow between representations of formulas is accompanied by the reference to the rule to apply next.

TRS_2 rewrites g to d (see the box labeled d and TRS_2)

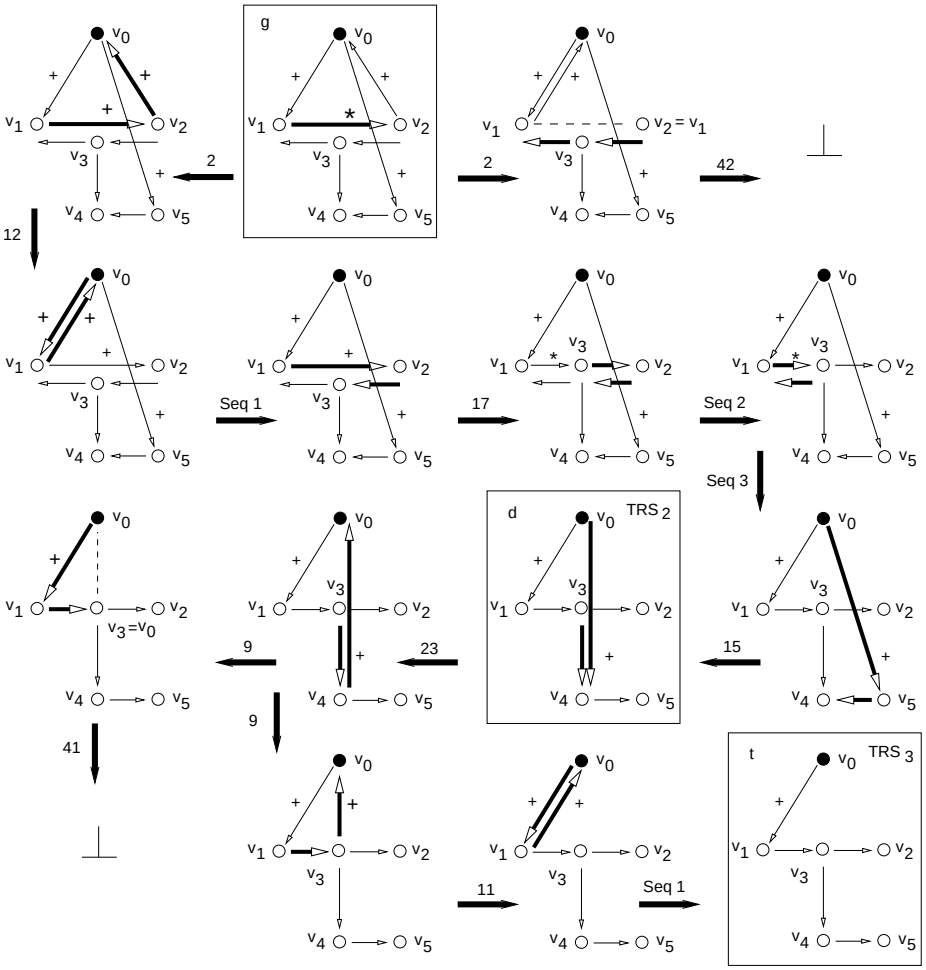
$$\begin{aligned}\mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{nextSibl}(v_1, v_3) \wedge \mathsf{nextSibl}(v_3, v_2) \wedge \mathsf{child}(v_3, v_4)\\ \wedge \mathsf{nextSibl}(v_4, v_5) \wedge \mathsf{child}^+(v_0, v_5).\end{aligned}$$

The formula d is forward, but still a (single-join) DAG. Using the additional rewrite rule of TRS_3, we obtain the forward tree formula t (see the box labeled t and TRS_3)

$$\begin{aligned}\mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{nextSibl}(v_1, v_3) \wedge \mathsf{nextSibl}(v_3, v_2) \wedge \mathsf{child}(v_3, v_4)\\ \wedge \mathsf{nextSibl}(v_4, v_5)\end{aligned}$$

It is worth noting that t is variable-preserving minimal, i.e., the amount of its binary atoms is exactly the number of its variables minus one. Also, the redundancies of g, mainly due to repeated up-down and left-right navigations, are detected and eliminated partly by TRS_2 and completely by TRS_3.

The Seq references on the rewrite arrows stand for sequences of rule applications, and they represent the compacted rules given at the bottom of Figure 4. Such



Seq 1	$\mathsf{child}^+(x,y) \wedge \mathsf{par}^+(y,x) \rightarrow \mathsf{child}^+(x,y)$
Seq 2	$\mathsf{nextSibl}(x,y) \wedge \mathsf{prevSibl}(y,x) \rightarrow \mathsf{nextSibl}(x,y)$
Seq 3	$\mathsf{nextSibl}^*(x,y) \wedge \mathsf{prevSibl}(y,x) \rightarrow \mathsf{nextSibl}(x,y)$

Fig. 4. Rewriting of the LGQ graph given in Section 4.1 using TRS_2 and TRS_3.

compacted rules can be derived from existing ones, as shown next for Seq 1.

$$\begin{aligned}
\mathsf{child}^+(x,y) \wedge \mathsf{par}^+(y,x) &\overset{(10)}{\rightarrow} \mathsf{child}^+(x,y) \wedge \mathsf{par}^+(x,x) \vee \mathsf{child}^*(x,x) \wedge \mathsf{child}^+(x,y) \\
&\overset{(27)}{\rightarrow} \mathsf{child}^+(x,y) \wedge \bot \vee \mathsf{child}^*(x,x) \wedge \mathsf{child}^+(x,y) \\
&\overset{(32)}{\rightarrow} \bot \vee \mathsf{child}^*(x,x) \wedge \mathsf{child}^+(x,y) \\
&\overset{(33)}{\rightarrow} \mathsf{child}^*(x,x) \wedge \mathsf{child}^+(x,y) \\
&\overset{(39)}{\rightarrow} \top \wedge \mathsf{child}^+(x,y) \overset{(34)}{\rightarrow} \mathsf{child}^+(x,y).
\end{aligned}$$

TRS_1 rewrites g to the following forward graph formula (each of lines 2 through 5 represent a subformula derived from one reverse atom of g):

$$\begin{aligned}\mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{nextSibl}^*(v_1, v_2) \wedge \mathsf{child}(v_3, v_4) \wedge \mathsf{child}^+(v_0, v_5)\\ \wedge \mathsf{child}^+(v_2, v_0) \wedge \mathsf{child}^*(v_2', v_2) \wedge \mathsf{root}(v_2')\\ \wedge \mathsf{nextSibl}(v_3, v_2) \wedge \mathsf{child}^*(v_3', v_3) \wedge \mathsf{root}(v_3')\\ \wedge \mathsf{nextSibl}(v_1, v_3) \wedge \mathsf{child}^*(v_1', v_1) \wedge \mathsf{root}(v_1')\\ \wedge \mathsf{nextSibl}(v_4, v_5) \wedge \mathsf{child}^*(v_4', v_4) \wedge \mathsf{root}(v_4').\end{aligned}$$

4.2 Termination

We first define some necessary measures and terminating orders on LGQ formulas.

The size $|e|$ of a formula e is the sum of sizes of all its constituent connectives and atoms, where the size of each boolean connective is one, and the size of an atom is given by its arity. The function $\#paren$ applied to a formula e returns the multiset containing the amount of parentheses nesting each atom of e.

The connection from variable a to variable b via a sequence of binary predicates p in a formula e is a 4-ary relation $a \overset{p}{\rightsquigarrow}_e b$ defined as follows:

—$a \overset{\alpha}{\rightsquigarrow}_e b$, if $\alpha(a, b)$ is included in e,

—$a \overset{p.q}{\rightsquigarrow}_e b$, if $a \overset{p}{\rightsquigarrow}_{e_1} v \overset{q}{\rightsquigarrow}_{e_2} b$, and $e_1 \wedge e_2$ is included in the DNF of e.

For a given connection $a \overset{p}{\rightsquigarrow}_e b$, the connection length is defined by the number of predicates in the connection sequence p, and denoted $|p|$.

The position-set $pos^\alpha(e)$ of α-atoms in a formula e is the multiset of all lengths of connections from a non-sink variable and with its sequence ending in an α-atom (x is a possibly empty sequence of predicates):

$$pos^\alpha(e) = \{l \mid a, b \text{ variables in } e, \mathsf{root}(a) \text{ included in } e, a \overset{x.\alpha}{\rightsquigarrow}_e b, l = |x.\alpha|\}$$

Example 4.1. The position-set of child-atoms $pos^{\mathsf{child}}(e)$ in

$$e = \mathsf{root}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge (\mathsf{child}(v_2, v_3) \vee \mathsf{child}(v_2, v_4)) \wedge \mathsf{child}^+(v_3, v_4)$$

is $\{1, 2, 2\}$, because e has three connections ending in child: $v_1 \overset{\mathsf{child}}{\rightsquigarrow}_e v_2$ with connection length 1, $v_1 \overset{\mathsf{child.child}}{\rightsquigarrow}_e v_3$ and $v_1 \overset{\mathsf{child.child}}{\rightsquigarrow}_e v_4$ with connection length 2. □

The reverse position factor $pos^{rev}(e)$ of a formula e is the union of position-sets of all its reverse atoms:

$$pos^{rev}(e) = \bigcup_{\alpha \text{ reverse}} (pos^\alpha(e)).$$

For a formula e, let br be the number of its base reverse predicates, tcr the number of transitive closure reverse predicates, and $trcr$ the number of reflexive transitive closure reverse predicates. The reverse type factor $type^{rev}(e)$ of e is a multiset containing the number 1 br times, the number 2 tcr times, and number 3 $trcr$ times.

EXAMPLE 4.2. Consider the formulas

$$\begin{aligned} e =& \mathsf{root}(v) \wedge \mathsf{child}(v, v_1) \wedge (\mathsf{par}(v_1, v_2) \wedge \mathsf{par}^+(v_2, v_3) \vee \mathsf{child}^+(v_1, v_2) \wedge \mathsf{self}(v_2, v_3)) \\ & \wedge \mathsf{par}^*(v_2, v_4) \\ e' =& \mathsf{root}(v) \wedge \mathsf{par}^*(v, v_1) \wedge \mathsf{par}^*(v, v_2). \end{aligned}$$

The reverse factors for e and e' are

$$\begin{aligned} pos^{rev}(e) &= \{|\mathsf{child.par}|, |\mathsf{child.par.par}^+|, |\mathsf{child.par.par}^*|, |\mathsf{child.child}^+\mathsf{.par}^*|\} \\ &= \{2, 3, 3, 3\} \\ type^{rev}(e) &= \{1, 2, 3\} \\ pos^{rev}(e') &= \{|\mathsf{par}^*|, |\mathsf{par}^*|\} = \{1, 1\} \\ type^{rev}(e') &= \{3, 3\}. \end{aligned}$$

□

The forward sink-arity of a variable is the number of forward binary atoms that appear in the same disjunct and have that variable as sink. The dag type factor $type^{dag}(e)$ of a formula e is the multiset containing the forward sink-arity of each multi-sink variable in e.

EXAMPLE 4.3. Consider the formula

$$\begin{aligned} e =& \mathsf{root}(v_1) \wedge \mathsf{child}(v_1, v_3) \wedge \mathsf{root}(v_2) \wedge \mathsf{child}(v_3, v_5) \\ & \wedge (\mathsf{child}^+(v_2, v_3) \vee \mathsf{child}^+(v_2, v_4) \wedge \mathsf{nextSibl}(v_4, v_3)) \end{aligned}$$

that has one multi-sink variable v_3. The dag factor is the multiset containing the forward sink-arity of v_3 in each of the two disjuncts of e: $type^{dag}(e) = \{2, 2\}$. □

We next recall standard definitions of strict and terminating orders on multisets of natural numbers and lexicographic products of such orders. Let $\mathcal{M}(\mathbb{N})$ denote the set of all finite multisets over $\mathbb{N}$. The order $>_{mul}$ is defined by

$$M >_{mul} N \Leftrightarrow \exists X, Y \in \mathcal{M}(\mathbb{N}), \emptyset \neq X \subseteq M, N = (M - X) \cup Y, \forall y \in Y : \exists x \in X : x > y.$$

The lexicographic product $>_1 \times \ldots \times >_n$ of strict orders $>_i$ $(1 \leq i \leq n)$ is

$$(x_1, \ldots, x_n) >_1 \times \ldots \times >_n (y_1, \ldots, y_n) \Leftrightarrow \exists k \leq n : (\forall i < k : x_i = y_i), x_k >_k y_k.$$

Using the above measures on LGQ formulas, we define the orders $>^{rev}_{pos}$, $>^{rev}_{type}$, $>^{dag}_{type}$, $>^{dnf}$, and $>^{size}$ as follows (s and t are LGQ formulas):

$$\begin{array}{llllll} s & >^{rev}_{pos} & t & \Leftrightarrow & pos^{rev}(s) & >_{mul} \quad pos^{rev}(t) \\ s & >^{rev}_{type} & t & \Leftrightarrow & type^{rev}(s) & >_{mul} \quad type^{rev}(t) \\ s & >^{dag}_{type} & t & \Leftrightarrow & type^{dag}(s) & >_{mul} \quad type^{dag}(t) \\ s & >_{dnf} & t & \Leftrightarrow & \#paren(s) & >_{mul} \quad \#paren(s) \\ s & >_{size} & t & \Leftrightarrow & |s| & > \quad |t| \end{array}$$

These orders are terminating because they are embeddings either into the strict and terminating order $>_{mul}$ on multisets over natural numbers, or into $>$ on natural numbers.

LEMMA 4.4. *TRS_1, TRS_2, and TRS_3 are terminating.*

PROOF. An order ω *holds* for a rule if $s\ \omega\ t$ holds for all formulas s and t such that t is derived by that rule from s. It can be easily seen that $>_{type}^{rev}$ holds for Rule (1), $>_{type}^{rev} \times >_{pos}^{rev}$ holds for Rules (2) through (22), $>_{type}^{dag}$ holds for Rule (23), $>_{dnf}$ holds for Rules (24) through (26), and $>_{size}$ holds for Rules (27) through (43). Additionally, applications of Rules (24) through (26) do not influence the order $>_{type}^{rev} \times >_{pos}^{rev}$, applications of Rules (27) through (43) do not influence the order $>_{type}^{rev} \times >_{pos}^{rev} \times >_{dnf}$, and applications of Rule (23) does not influence the order $>_{type}^{rev} \times >_{pos}^{rev} \times >_{dnf} \times >_{size}$.

The rewriting system TRS_i is terminating if there is a terminating order $>_i^{trs}$ that holds for all its rules ($1 \leq i \leq 3$). We define the orders $>_i^{trs}$ as follows:

—the order $>_1^{trs}$ is $>_{type}^{rev}$,
—the order $>_2^{trs}$ is $>_{type}^{rev} \times >_{pos}^{rev} \times >_{dnf} \times >_{size}$,
—the order $>_3^{trs}$ is $>_{type}^{dag} \times >_2^{trs}$.

□

4.3 Soundness and Completeness

A term rewriting system $(\mathrm{LGQ}^{\rightarrow}, \rightarrow)$ is *sound*, if for any LGQ formula s, any derivable LGQ formula t from s is equivalent to s, and if t is a normal form, then t is a forward LGQ formula.

A term rewriting system $(\mathrm{LGQ}^{\rightarrow}, \rightarrow)$ is *complete*, if for any equivalent LGQ formulas s and forward t, s is rewritten to a normal form forward LGQ formula t' that is equivalent to t.

LEMMA 4.5. *TRS_1, TRS_2, and TRS_3 are sound and complete.*

PROOF. Let s be the input formula and t the result of rule application. The rewriting systems are defined by rules of lemmata in Section 3, where $l \equiv r$ and $s \equiv t$ hold for any rule instance $l \rightarrow r$. Thus, s derives in one step an equivalent formula t: $s \rightarrow t \Rightarrow s \equiv t$. It easily follows by complete induction that $s \xrightarrow{*} t' \Rightarrow s \equiv t'$. Thus, if $s \equiv t$, then $t \equiv t'$.

We next show for each rewriting system that if $s \rightarrow^{!} t$, then t is forward.

$\mathbf{TRS}_1$ consists of Rule (1) that rewrites any reverse atom to an equivalent forward formula. Thus only a forward formula is irreducible.

$\mathbf{TRS}_2$ consists of Rules (2) through (22) and (24) through (37). There are three cases concerning the type of binary atoms in s.

(A) s is already forward. Then simplification rules of Lemma 3.4 may apply, and yield an irreducible equivalent forward t, because no reverse binary atom appears on rhs but not on lhs of a rule.

(B) s has no forward atom. Then there must be paths from non-sink variables to each reverse atom, and each non-sink variable has a root atom (recall that we only consider absolute formulas). Applying repeteadly Rule (30) for unsatisfiability detection and Rules (32) and (33) for unsatisfiability propagation, the obtained normal form is $\bot$.

(C) s has reverse and forward atoms. Then, along a path in s, there are forward atoms occurring *before* reverse atoms, or no forward atoms occur before a reverse atom. The latter case is treated as no forward binary atoms appear in s (case

B above). In the former case, s has paths of one forward and one reverse atom matching lhs of one rule of Lemma 3.4. Such paths are rewritten either to (1) paths of two forward atoms, or to (2) trees of size two with one forward and one reverse branch, or to (3) forests of trees as in (2) and paths as in (1). In cases (2) and (3), some paths to reverse atoms change and become shorter. Next, cases (A), (B), or (C) apply.

The rules of Lemma 3.12, without (30) and (31), are simplification rules based on navigation compaction and unsatisfiability detection and propagation and can be left out without jeopardizing the reachability of an equivalent forward normal form. They are, however, relevant for removing cycles.

TRS$_3$ extends TRS$_2$ with Rule (23) that rewrites DAGs of two forward atoms to a path of one forward and one reverse atom. Therefore, a (forward) normal form for TRS$_2$ is not irreducible for TRS$_3$, if it contains multi-sink variables. There are two main cases for the elimination of multi-sink variables:

(A) A disjunct of s contains $\mathsf{root}(y) \wedge \mathit{fwd}(x,y)$ or $v(x,y) \wedge h(x,y)$ with v vertical and h horizontal predicates and is simplified to $\bot$, cf. Rules (27) through (35).

(B) Otherwise, Rule (23) is applied and yields a formula with one reverse atom, which is then rewritten by TRS$_2$ to a TRS$_2$ normal form without additional multi-sink variables. The procedure continues until s has no multi-sink variables. □

4.4 Confluence

LEMMA 4.6. *TRS$_1$ and TRS$_3$ are confluent for LGQ graphs. TRS$_2$ is confluent for LGQ forests, and not confluent for LGQ DAGs and graphs.*

PROOF. The rewriting systems are confluent if they are terminating and locally confluent [Newman 1942]. From Lemma 4.4 it follows that all three rewriting systems are terminating. Thus it only remains to show local confluence, i.e., that all critical pairs are joinable.

TRS$_1$ consists only of Rule 1, whose lhs is a single atom. There is no critical pair created by this rule with itself or the AC-identities.

TRS$_2$. We first show that for single-join DAGs (and also graphs), TRS$_2$ is not locally confluent. Consider the single-join DAG

$$\mathsf{root}(a) \wedge \mathsf{child}^+(a,b) \wedge \mathsf{prevSibl}(b,d) \wedge \mathsf{root}(e) \wedge \mathsf{child}^+(e,c) \wedge \mathsf{nextSibl}(c,b)$$

We follow the two different rewriting sequences A and B (underlined subformulas are rewritten in the next step)

A. $\mathsf{root}(a) \wedge \underline{\mathsf{child}^+(a,b) \wedge \mathsf{prevSibl}(b,d)} \wedge \mathsf{root}(e) \wedge \mathsf{child}^+(e,c) \wedge \mathsf{nextSibl}(c,b)$
$\rightarrow \mathsf{root}(a) \wedge \mathsf{child}^+(a,d) \wedge \mathsf{nextSibl}(d,b) \wedge \mathsf{root}(e) \wedge \mathsf{child}^+(e,c) \wedge \mathsf{nextSibl}(c,b).$

B. $\mathsf{root}(a) \wedge \mathsf{child}^+(a,b) \wedge \underline{\mathsf{prevSibl}(b,d)} \wedge \mathsf{root}(e) \wedge \mathsf{child}^+(e,c) \wedge \underline{\mathsf{nextSibl}(c,b)}$
$\rightarrow \mathsf{root}(a) \wedge \mathsf{child}^+(a,b) \wedge \mathsf{self}(c,d) \wedge \mathsf{root}(e) \wedge \mathsf{child}^+(e,c) \wedge \mathsf{nextSibl}(c,b).$

The final formulas can not be rewritten anymore and are different.

This concludes one half of the proof for TRS$_2$. We now prove that TRS$_2$ is locally confluent for LGQ forests (and trees and paths).

We next consider systematically all cases of interactions between various forward and reverse atoms that can lead to critical pairs. These are the cases when the lhs of each rule of Lemma 3.4 unifies with a subterm of the lhs of the built-in A-identity

for $\wedge$. The most general terms under consideration have a tree shape:

$$\alpha_1(a,b) \wedge \alpha_2(b,c) \wedge \alpha_3(b,d)$$

where α_1 is forward and α_2 and α_3 are reverse. DAGs are excluded by definition and forests are not considered, as their trees can be rewritten independently.

The following interactions are to be considered (note that $\wedge$ is commutative and therefore the symmetric cases for α_2 and α_3 are not necessary, as they are covered by identities):

Case	α_1	α_2	α_3
1	horizontal forward	vertical reverse	vertical reverse
2	horizontal forward	horizontal reverse	vertical reverse
3	vertical forward	horizontal reverse	horizontal reverse
4	vertical forward	horizontal reverse	vertical reverse

To easier follow the rewritings in each case, we rename the predicates α_1, α_2, α_3 to (a composition of) abbreviations of their type, e.g., v/h for vertical/horizontal, f/r for forward/reverse. We define the sets HF, HR, VF, VR containing the formulas that are horizontal forward, horizontal reverse, vertical forward, and horizontal reverse respectively.

Case 1. $hf = \alpha_1$, $vr_1 = \alpha_2$, $vr_2 = \alpha_3$. Then,

A. $\underline{hf(a,b)} \wedge \underline{vr_1(b,c)} \wedge vr_2(b,d) \rightarrow \underline{hf(a,b)} \wedge vr_1(a,c) \wedge \underline{vr_2(b,d)}$.

B. $\underline{hf(a,b)} \wedge vr_1(b,c) \wedge \underline{vr_2(b,d)} \rightarrow \underline{hf(a,b) \wedge vr_1(b,c)} \wedge vr_2(a,d)$.

Both branches are now reducible to $hf(a,b) \wedge vr_1(a,c) \wedge vr_2(a,d)$.

Case 2. Initially, interactions HF-VR (branch A) or HF-HR (branch B) are considered. Let $hf = \alpha_1$, $hr = \alpha_2$, $vr = \alpha_3$. Then,

A. $\underline{hf(a,b)} \wedge hr(b,c) \wedge \underline{vr(b,d)} \rightarrow \underline{hf(a,b) \wedge hr(b,c)} \wedge vr(a,d)$.

For both branches A and B, interactions HF-HR can be further considered for rewriting $hf(a,b) \wedge hr(b,c)$. This term is contracted to a term t containing only horizontal atoms. Interactions HF-VR of Case 1 are next considered, and after several rewritings either a or c become the source variable of vr, depending on the same predicate pair (hf,hr) for both branches A and B.

Case 3. Initially, interactions VF-HR are considered. Let $vf = \alpha_1$, $hr_1 = \alpha_2$, $hr_2 = \alpha_3$. In case $vf = \mathsf{fstChild}$, we have

A. $\underline{vf(a,b) \wedge hr_1(b,c)} \wedge hr_2(b,d) \rightarrow \bot \wedge hr_2(b,d) \rightarrow \bot$.

B. $\underline{vf(a,b)} \wedge hr_1(b,c) \wedge \underline{hr_2(b,d)} \rightarrow \bot \wedge hr_1(b,c) \rightarrow \bot$.

In case $vf \in \{\mathsf{child}, \mathsf{child}^+\}$, we have

A. $\underline{vf(a,b) \wedge hr_1(b,c)} \wedge hr_2(b,d) \rightarrow vf(a,c) \wedge \underline{hr_1^{-1}(c,b) \wedge hr_2(b,d)}$.

B. $\underline{vf(a,b)} \wedge hr_1(b,c) \wedge \underline{hr_2(b,d)} \rightarrow vf(a,d) \wedge \underline{hr_1(b,c) \wedge hr_2^{-1}(d,b)}$.

In case $hr_1 = hr_2 = \mathsf{prevSibl}$, we have

A. $vf(a,c) \wedge \underline{\mathsf{nextSibl}(c,b) \wedge \mathsf{prevSibl}(b,d)} \rightarrow vf(a,c) \wedge \mathsf{self}(c,d) \wedge \mathsf{nextSibl}(d,b)$.

B. $vf(a,d) \wedge \underline{\mathsf{prevSibl}(b,c) \wedge \mathsf{nextSibl}(d,b)} \rightarrow vf(a,d) \wedge \mathsf{self}(d,c) \wedge \mathsf{nextSibl}(c,b)$.

Both derived formulas are identical up to the variable equality $c = d$, which is ensured by the A-identity of self: $\mathsf{self}(v_1, v_2) \wedge \alpha(v_2, v_3) \approx \mathsf{self}(v_1, v_2) \wedge \alpha(v_1, v_3)$.

In case $hr_1 = \mathsf{prevSibl}^+$, $hr_2 = \mathsf{prevSibl}$, we have

A. $vf(a, c) \wedge \underline{\mathsf{nextSibl}^+(c, b) \wedge \mathsf{prevSibl}(b, d)}$
$\rightarrow vf(a, c) \wedge \underline{\mathsf{nextSibl}^*(c, d)} \wedge \mathsf{nextSibl}(d, b)$
$\rightarrow vf(a, c) \underline{\triangle} (\mathsf{nextSibl}^+(c, d) \underline{\vee} \mathsf{self}(c, d)) \wedge \mathsf{nextSibl}(d, b)$
$\rightarrow vf(a, c) \wedge \mathsf{nextSibl}^+(c, d) \wedge \mathsf{nextSibl}(d, b) \vee vf(a, c) \wedge \mathsf{self}(c, d) \wedge \mathsf{nextSibl}(d, b)$.

B. $vf(a, d) \wedge \underline{\mathsf{prevSibl}^+(b, c) \wedge \mathsf{nextSibl}(d, b)}$
$\rightarrow vf(a, d) \underline{\triangle} (\mathsf{prevSibl}^+(d, c) \wedge \mathsf{nextSibl}(d, b) \underline{\vee} \mathsf{self}(d, c) \wedge \mathsf{nextSibl}(d, b))$
$\rightarrow \underline{vf(a, d) \wedge \mathsf{prevSibl}^+(d, c)} \wedge \mathsf{nextSibl}(d, b) \vee vf(a, d) \wedge \mathsf{self}(d, c) \wedge \mathsf{nextSibl}(d, b)$
$\rightarrow vf(a, c) \wedge \mathsf{nextSibl}^+(c, d) \wedge \mathsf{nextSibl}(d, b) \vee vf(a, d) \wedge \mathsf{self}(d, c) \wedge \mathsf{nextSibl}(d, b)$.

Again, both derived formulas are identical up to $c = d$ in the second conjunct.

The remaining two cases with $hr_1 = \mathsf{prevSibl}$, $hr_2 = \mathsf{prevSibl}^+$ and $hr_1 = hr_2 = \mathsf{prevSibl}^+$ are similar to the last two cases. Case 4 is similar to Case 3..

TRS_3. Because TRS_3 includes TRS_2, the above discussion applies to TRS_3 as well. The new interaction cases are

A. $\alpha_1(a, b) \wedge \alpha_2(c, b) \wedge \alpha_3(d, b)$ with $\alpha_1, \alpha_2, \alpha_3$ forward
B. $\alpha_1(a, b) \wedge \alpha_2(c, b) \wedge \alpha_3(b, d)$ with α_1, α_2 forward, α_3 reverse.

We discuss case A (B is similar). There are three possible distinct contractions:

I. $\alpha_1(a, b) \wedge \alpha_2^{-1}(b, c) \wedge \alpha_3(d, b)$ $\rightarrow \alpha_1(a, b) \wedge \alpha_2^{-1}(b, c) \wedge \alpha_3^{-1}(b, d)$. (1) or
$\rightarrow \alpha_1^{-1}(b, a) \wedge \alpha_2^{-1}(b, c) \wedge \alpha_3(d, b)$. (2)

II. $\alpha_1(a, b) \wedge \alpha_2(c, b) \wedge \alpha_3^{-1}(b, d)$ $\rightarrow \alpha_1(a, b) \wedge \alpha_2^{-1}(b, c) \wedge \alpha_3^{-1}(b, d)$. (1) or
$\rightarrow \alpha_1^{-1}(b, a) \wedge \alpha_2(c, b) \wedge \alpha_3^{-1}(b, d)$. (3)

III. $\alpha_1^{-1}(b, a) \wedge \alpha_2(c, b) \wedge \alpha_3(d, b)$ $\rightarrow \alpha_1^{-1}(b, a) \wedge \alpha_2(c, b) \wedge \alpha_3^{-1}(b, d)$. (3) or
$\rightarrow \alpha_1^{-1}(b, a) \wedge \alpha_2^{-1}(b, c) \wedge \alpha_3(d, b)$. (1).

The terms (I) to (III) are pairwise joinable (the Arabic numbers on the right represent identical formulas). □

4.5 Structural Complexity of Derived Formulas

This section states the relationship between the structural complexities of the input formulas and the derived normal forms for each of our three rewriting systems.

LEMMA 4.7. *TRS_1 rewrites any single-join DAG to a forward single-join DAG, and any graph to a forward graph.*

PROOF. TRS_1 consists of Rule (1) that can create multi-sink variables. From Lemma 4.5 it follows that TRS_1 is sound and complete for graphs. Let s be the formula to rewrite and t the derived formula.

(A) If s is a forest, i.e., it has no multi-sink variables, then t is a single-join DAG with as many multi-sink variables as reverse atoms in s.

(B) If s is a single-join DAG, then t is a single-join DAG with at least as many multi-sink variables as there are in s plus the number of reverse atoms in s.
(C) If s is a graph, then t is a graph, because Rule (1) does not remove cycles. □

LEMMA 4.8. *TRS_2 rewrites any forest to a forward forest, any single-join DAG to a single-join DAG, and any graph to a forward graph.*

PROOF. TRS_2 consists of Rules (2) through (22) and (24) through (37) that preserve the (non-)sinkness of variables. From Lemma 4.5 it follows that TRS_2 is sound and complete for graphs. Let s be the formula to rewrite and t the derived formula.
(A) If s is a forest, i.e., it has no multi-sink variables, then t does not have multi-sink variables, hence t is a forest.
(B) If s is a single-join DAG, then t is a single-join DAG with the same multi-sink variables as there are in s.
(C) If s is a graph, then t is in general a graph, because multi-sinks and cycles are not necessarily removed. □

LEMMA 4.9. *TRS_3 rewrites any graph to a forward forest.*

PROOF. *TRS_3 extends TRS_2 with Rule* (23). *From Lemma 4.5 it follows that TRS_3 is sound and complete for graphs and that the normal forms of TRS_3 do not contain multi-sink variables, hence they are forests.* □

REMARK 4.10. For graphs with (without) closure predicates, TRS_3 yields forward forests with (without) closure predicates. Similar results are stated in [Benedikt et al. 2005] for forests restricted to vertical predicates. □

4.6 Analytical Complexity

This section studies the complexity of rewriting LGQ formulas for each of our three rewriting systems.
Complexity of AC-matching for $LGQ^{\rightarrow}$ rules. Although AC-matching is NP-complete in general [Lincoln and Christian 1989], is polynomial for the restricted case of $\text{TRS}_{1,2,3}$, as explained next.

The lhs of $\text{LGQ}^{\rightarrow}$ rewrite rules defining $\text{TRS}_{1,2,3}$ are of three kinds:

(1) a single binary $\text{LGQ}^{\rightarrow}$ atom with variables ranging over LGQ variables,
(2) an $\text{LGQ}^{\rightarrow}$ path made out of two atoms with different function symbols and with variables ranging over LGQ variables,
(3) an $\text{LGQ}^{\rightarrow}$ formula with two or three variables ranging over LGQ formulas.

In the first case, AC-matching boils down to syntactic matching, which is linear in the size of both participating terms. In the second case, AC-matching is reducible to syntactic matching, followed by checking whether the variable appearing in both atoms matched the same constant. This procedure takes at most quadratic time in the matched term. In the third case, the $\text{LGQ}^{\rightarrow}$ variables can match any subterms of an LGQ formula. The number of possible combinations of variable matchings is exponential in the number of variables, where the basis is the size of the matched term. Because the number of the $\text{LGQ}^{\rightarrow}$ variables is bounded in a constant (≤ 3), the time for AC-matching is at most cubic in the size of the term to match.

The aforementioned polynomial cases for AC-matching can be further reduced to linear, if the formulas and rules are represented more compactly. Consider the LGQ formulas given by their graphical representations. Then, rule applications can be performed in linear time as, e.g., matchings of paths of fixed length in graphs. The quadratic time of the second case is needed then only once for the construction of the graphical representation of the LGQ formula to rewrite.

The complexity results given next for rewriting LGQ formulas using $\text{TRS}_{1,2,3}$ ignore the complexity of AC-matching.

LEMMA 4.11. *TRS_1 is a LOGLIN reduction.*

PROOF. TRS_1 is complete for graphs. It consists of Rule (1) that rewrites locally each reverse predicate to two forward predicates and a root nodetest. To traverse the input graph, TRS_1 needs only a pointer to the current atom. □

LEMMA 4.12. *TRS_2 is a PSPACE reduction and for input forests derives forward forests with exponentially many trees.*

PROOF. Let s be the input graph and t the derived forest.

The exponential blow-up is due to repeated applications of Rules (10) and (22) or simpler variants that create disjunctions and double the size of rewritings. The other rules do not increase the size of rewritings.

The trees of s are rewritten independently of each other. We consider the rewriting of one tree of s with R reverse predicates and treat first the case of one path in the graph of s containing $r \leq R$ reverse atoms. Later we generalize to several paths containing all R reverse atoms.

The reverse atoms are intertwined with forward atoms. Consider that n_i forward atoms appear before the i^{th} reverse atom ($1 \leq i \leq r$), where the reverse atoms are indexed based on their appearance from right-to-left in the formula (thus, $n_i > n_j$ for $i < j$). For each set of rules that behave similarly, we define a family of functions ϕ_i, whose computation simulate a rewriting sequence where the rules are applied such that the first reverse atom is considered first. The time complexity of rewriting s is the number of steps required to compute ϕ_r, and the value computed by ϕ_r is the number of trees obtained by rewriting one tree of s.

Case 1. The applications of Rules (10) and (22) are simulated by

$$\phi_i(n_i,\ldots,n_1) = \begin{cases} \phi_i(n_i-1,\ldots,n_1-1) + \phi_{i-1}(n_{i-1},\ldots,n_1) & , i>1 \text{ and } n_i>1 \\ \phi_{i-1}(n_{i-1},\ldots,n_1) & , i>1 \text{ and } n_i=1 \\ 1+\phi_1(n_1-1) & , i=1 \text{ and } n_i>0 \\ 0 & , i=1 \text{ and } n_i=0. \end{cases}$$

The first branch leads to the exponential blow-up and encodes the creation of two trees. The first tree contains still the reverse atom i but has one forward atom fewer for all reverse atoms $j \leq i$ along the same path. The second tree does not contain anymore the reverse atom i.

The number of trees in t obtained from one tree in s, as also the number of computation steps, is exponential in r:

$$\phi_r(n_r,\ldots,n_1) = \sum_{i_r=0}^{n_r}\Big(\sum_{i_{r-1}=i_r}^{n_{r-1}}\big(\ldots\sum_{i_2=i_3}^{n_2}\phi_1(n_1-i_2)\big)\Big) \approx \prod_{i=1}^{r} n_i$$

Case 2. The applications of Rules (3)-(5) and (13)-(20) are simulated by

$$\phi_i(n_i,\ldots,n_1) = \begin{cases} \phi_{i-1}(n_{i-1},\ldots,n_1) & , i > 1 \\ 0 & , i = 1 \text{ and for Rules (13) and (18)} \\ 1 & , i = 1 \text{ and for the other rules.} \end{cases}$$

The computation of ϕ_r requires r steps and the number of trees in t obtained by rewriting one tree in s is 0 for Rules (13) and (18) and 1 otherwise. The entire formula s is traversed once and only one pointer to the current atom is needed, thus it requires extra logarithmic space in s.

Case 3. The applications of Rules (6), (7), (11), and (12) are simulated by

$$\phi_i(n_i,\ldots,n_1) = \begin{cases} \phi_i(n_i-1,\ldots,n_1-1) & , i > 1 \text{ and } n_i > 0 \\ \phi_{i-1}(n_{i-1},\ldots,n_1) & , i > 1 \text{ and } n_i = 0 \\ \phi_1(n_1-1) & , i = 1 \text{ and } n_i > 0 \\ 1 & , i = 1 \text{ and } n_i = 0. \end{cases}$$

The number of steps required for the computation of ϕ_r is

$$r + n_r + \sum_{i=1}^{r-1}(n_i - n_{i+1}) = r + n_1$$

The number of trees in t obtained from one tree in s is $\phi_r(n_r,\ldots,n_1) = 1$. As for the second case, only extra logarithmic space in s is needed.

Case 4. The applications of Rules (8), (9), and (21) are simulated by

$$\phi_i(n_i,\ldots,n_1) = \begin{cases} \phi_i(n_i-1,\ldots,n_1-1) + \phi_{i-1}(n_{i-1}-1,\ldots,n_1-1) & , i > 1 \text{ and } n_i > 0 \\ \phi_{i-1}(n_{i-1},\ldots,n_1) & , i > 1 \text{ and } n_i = 0 \\ 1 + \phi_1(n_1-1) & , i = 1 \text{ and } n_i > 0 \\ 0 & , i = 1 \text{ and } n_i = 0. \end{cases}$$

This case can be treated similarly to the first case above.

If $r < R$, then there are other reverse atoms along another path, containing, say, r' reverse atoms with n_i ($r < i \leq r' + r \leq R$) forward atoms before them. This new path is to be considered in each of the already derived trees. The number of trees in t, derived from the tree in s with R reverse atoms, becomes exponential in R.

The confluence of TRS_2 for forests ensures that any rewriting strategy leads to the same result (see Lemma 4.6). We consider the following rewriting strategy. We apply always first the second and third cases if possible. If not possible, we apply once the first or the last case. This leads to the creation of two trees. We continue rewriting only the tree with one reverse atom fewer and postpone the rewriting of the second one. After a number of rule applications linear in the number of its reverse atoms, this tree derives a forward tree. We output it and release the memory required to store it. Now, we rewrite the tree whose rewriting we postponed last. It is easy to see that at any time we need only space for polynomially many trees. □

REMARK 4.13. The trees in the yield of TRS_2 have linear size in the maximal size of a tree of the input forest, because (1) the rules do not introduce new variables nor unary predicates, and (2) the number of binary predicates of a tree is bounded in the number of its variables minus one. □

We now extend the result of Lemma 4.12 to graphs.

LEMMA 4.14. *TRS_3 is a PSPACE reduction.*

PROOF. Let s be the input graph and t the derived forest.

TRS_3 contains TRS_2 and Rule (23). Let us consider that Rule (23) is applied until no other applications are possible. This does not have influence on the shape of t, because TRS_3 is confluent for graphs. Then, each variable sink of n forward atoms creates $n-1$ reverse atoms. Thus, the number of reverse atoms remains linear in s. We further continue as for TRS_2. □

5. MAIN EXPRESSIVENESS AND COMPLEXITY RESULTS

This section finally gathers our main results concerning the expressiveness of LGQ (and XPath) with and without reverse predicates and relates the analytical complexity of rewriting queries and the structural complexity of the derived forward equivalents.

THEOREM 5.1. *LGQ and its fragment of forward forests are equally expressive.*

PROOF. From Lemma 4.9 it follows that TRS_3 rewrites any LGQ graph to an equivalent forward forest. □

The result of Theorem 5.1 applies also to the particular case of LGQ forests, which correspond to XPath.

COROLLARY 5.2. *XPath and its forward fragment are equally expressive.*

Theorem 5.1 and its Corollary 5.2 state that the reverse predicates do not increase the expressive power of LGQ and XPath. However, forward LGQ forests admit in general exponentially more succinct equivalent LGQ graphs with reverse predicates. As stated by Theorem 5.3, this holds even for LGQ trees with reverse predicates, thus also for XPath.

THEOREM 5.3. *There is an LGQ tree that does not admit an equivalent forward forest of polynomial size.*

PROOF. Consider the following LGQ tree Q having only vertical predicates

$$Q = \mathsf{root}(r) \wedge \mathsf{child}^+(r,c) \wedge l(c) \wedge \bigwedge_{i=1}^{n} (\mathsf{par}^+(c,v_i) \wedge l_i(v_i))$$

Consider the set of labels $L = \{l_1, \ldots, l_n, l\}$. Let $PS(n)$ denote the tree in Figure 5 with $n!$ different root-to-leaf paths over the alphabet L obtained by appending l to the permutations of $\{l_1, \ldots, l_n\}$. It is easy to see that Q admits exactly one valuation on each of the paths in $PS(n)$.

We first prove by contradiction there is no forward tree formula $T \subseteq Q$ that admits more than one valuation on $PS(n)$.

We assume there is a forward tree T with (a) $T \subseteq Q$ and with (b) more than one valuation in $PS(n)$. The formula T uses only forward vertical predicates and must have at least as many variables as Q (see Lemma 7.2 of [Gottlob et al. 2006]). There are two cases concerning the structure of T: T is (1) either a path (thus without multi-source variables), or (2) a tree (thus with multi-source variables). In the first

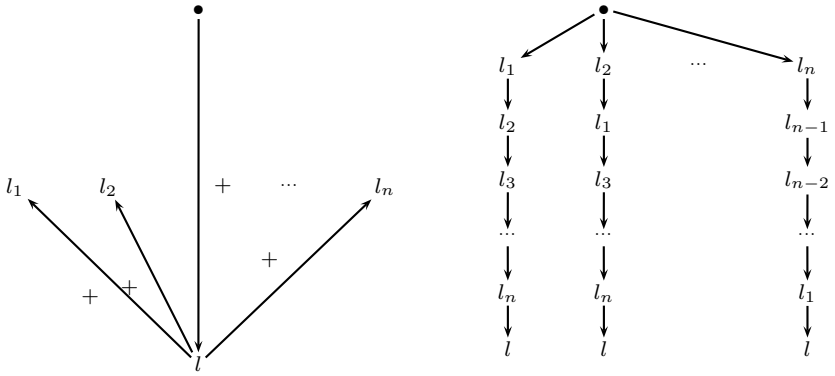


Fig. 5. Query Q (left) and $PS(n)$ tree structure (right) used in the proof of Theorem 5.3.

case, the variable mappings, under any valuation of T, lie on the same path in $PS(n)$, due to the semantics of forward vertical predicates. This is a contradiction with (b). In the second case, there are valuations that map variables to nodes not necessarily on the same path in $PS(n)$. This contradicts (a), because Q only admits one valuation on each of the paths in $PS(n)$.

In the remainder, the claim of the theorem is proven by contradiction. We assume Q admits an equivalent forward forest $F_m = \bigvee_{j=1}^{m} (T_j)$ with m polynomial in n.

There are $n!$ paths in $PS(n)$ and there is exactly one valuation of Q on each of them. Because F_m consists of polynomially many tree formulas, there must be at least one tree formula $T_j \subseteq F_m$ that admits valuations on exponentially many paths in $PS(n)$. However, this contradicts the first part of the proof, which states that a forward tree formula $T_j \subseteq Q \equiv F_m$ has at most one valuation on $PS(n)$. □

We further obtain an upper bound for rewriting LGQ graphs to equivalent LGQ forward forests.

THEOREM 5.4. *LGQ graphs are PSPACE-reducible to equivalent LGQ forward forests of exponential size.*

PROOF. From Lemma 4.14 it follows that TRS_3 is a PSPACE reduction and yields forward forests of exponential size. From Lemma 4.9 it follows that TRS_3 rewrites any graph to a forward forest. □

The above results concern the rewriting of LGQ graphs to the structurally less complex LGQ forward forests. The effort of both eliminating reverse predicates and reducing the structural complexity of the derived forward formulas has a rather high analytical complexity. Theorem 5.5 ensures that only the elimination of reverse predicates is not the source of this high complexity.

THEOREM 5.5. *LGQ graphs are LOGLIN-reducible to equivalent LGQ forward graphs. LGQ single-join DAGs are LOGLIN-reducible to equivalent LGQ forward single-join DAGs.*

PROOF. From Lemma 4.11 it follows that TRS_1 is a LOGLIN reduction. From

Lemma 4.7 it follows that TRS_1 rewrites any graph to a forward graph and any single-join DAG to a forward single-join DAG. □

Since LGQ single-join DAGs capture XPath with identity-based equality, it follows immediately that

COROLLARY 5.6. *XPath with identity-based equality is LOGLIN-reducible to its forward fragment.*

The exponential lower bound of Theorem 5.3 is in essence bad news. To better delimit the source of this exponentiality, we define the class of so-called simple LGQ graphs, which admit equivalent forward forests of linear size, and thus polynomial evaluation [Gottlob et al. 2002].

DEFINITION 5.7. *A simple graph is an LGQ graph in disjunctive normal form with no path having vertical (horizontal) closure reverse predicates (immediately) after vertical (horizontal) forward predicates. Additionally, no variable is sink of several paths with vertical (horizontal) closure forward predicates.*

PROPOSITION 5.8. *Simple graphs are LOGLIN-reducible to equivalent LGQ forward forests.*

PROOF. It immediately follows from the proof of Lemma 4.14. To rewrite simple graphs, it suffices to apply locally only rules that keep the size of rewritings linear in the size of the input formulas. □

The reduction of simple graphs to forward forests of linear size exhibits a minimization aspect. The number of predicates is bounded in the number of variables for trees in the resulting forests, and in the number of all possible different binary predicates times the square of the number of variables for disjuncts of the input simple graphs.

6. RELATED WORK AND APPLICATIONS

Our initial results on the equal expressiveness of XPath with or without identity-based equality and its forward fragment [Olteanu et al. 2002] found applications to query evaluation and optimization in various contexts. This section overviews some work that relates to and/or uses the proposed rewriting systems.

Streamed Query Evaluation. The initial motivation of our work stems in the inherent difficulty to evaluate queries with reverse predicates against XML streams, as also explained in Section 1. Trading queries with reverse predicates for equivalent forward queries is primarily used by SPEX [Olteanu et al. 2002; Olteanu et al. 2004]. Other XPath processors use TRS_1, e.g., [Barton et al. 2003], and TRS_2, e.g., [Helmer et al. 2002; Schott and Noga 2003; Marian and Siméon 2003].

XPath Optimization. [Grust et al. 2004] proposes efficient index-based access methods to XML data stored in relational databases and shows how TRS_2 rules can be used to further optimize query evaluation in such contexts by pruning index regions or trading queries for their (forward or even reverse) equivalents. For example, the optimized access to descendant nodes favors descendant(child) over ancestor(parent) predicates, as considered also by TRS_2. However, the optimized access

to text nodes can favor parent over child predicates, as in

`descendant::n/child::text()` → `descendant::text()[parent::n]`.

[Grust et al. 2004] points also out that TRS_2 can be used to establish the space of equivalent XPath queries out of which a cost-based optimizer would pick candidates based on cost measures. This idea is considered by the VAMANA XPath query processor [Raghavan et al. 2005].

Coping with Ordering and Duplicates. XPath semantics requires that the answers to XPath queries are sets of nodes (thus without duplicates) sorted in document order. This requirement renders some well-known XPath evaluation techniques, e.g., Xalan [Apache Project 2001b], very inefficient, because they can accumulate exponentially many duplicates to be removed at later evaluation stages (see the discussion of [Gottlob et al. 2002]). For example, a possible evaluation of the XPath query `/descendant:a/ancestor::b` would first compute the set of b-nodes in wrong order and, if several `a`-nodes have common `b`-ancestors, with duplicates.

A possibility to overcome this exponentiality is to sort intermediate results and prune duplicates after the evaluation of each subquery. However, there are queries that do not require to sort and do not create duplicates. For example, the set of nodes selected by the equivalent forward query `/descendant:b[descendant::a]`, as obtained by TRS_2, is already in document order and has no duplicates. Static inference of these properties is investigated in [Hidders and Michiels 2003]. A technique largely based on TRS_2 is proposed in [Helmer et al. 2002], where queries are translated to sequences of algebraic operations that do not generate duplicate nodes. [Olteanu et al. 2004] proposes a streaming evaluation for XPath forward queries that does not require sorting and avoids the creation of duplicates.

XPath Expressiveness. By proposing TRS_1 and TRS_2, our previous work [Olteanu et al. 2002] gives two important expressiveness results: XPath is as expressive as its forward fragment augmented with identity-based equality, as ensured by TRS_1, and even without identity-based equality, as ensured by TRS_2. Indirectly, [Olteanu et al. 2002] points out the exponential succinctness of forward XPath with identity-based equality (captured by LGQ forward single-join DAGs) over forward XPath.

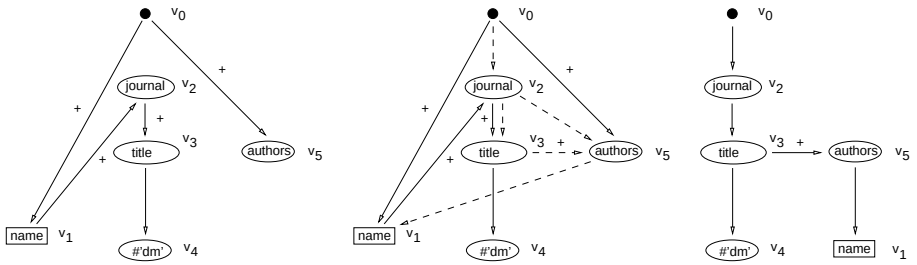
TRS_2 is used by [Gottlob et al. 2006] to show that the language of conjunctive queries over the LGQ predicates is as expressive as XPath. Motivated by preliminary versions of [Gottlob et al. 2006], this article shows how the rewriting framework of [Olteanu et al. 2002] extended only with the trivial Rule (23) can rewrite arbitrary LGQ graph queries (not directly expressible in XPath) to LGQ forest forward queries (expressible in XPath).

The work [Benedikt et al. 2005], subsequent to ours [Olteanu et al. 2002], investigates closure properties of various fragments of XPath restricted to vertical predicates, focusing on the ability to perform basic Boolean operations while remaining within the fragments. In particular, it refinds the expressiveness results of [Olteanu et al. 2002] for the case of queries with vertical predicates. Additionally, [Benedikt et al. 2005] shows the equal expressiveness of XPath with closure (non-closure) vertical predicates and XPath with closure (non-closure) forward vertical predicates. Note that this expressiveness result can be also inferred directly from

the rewrite rules of our term rewriting systems.

Complexity of Graph Queries over Trees. Although the evaluation of graph queries is NP-hard in general, there are non-trivial classes of graph queries that admit polynomial evaluation. [Gottlob et al. 2006] characterizes some polynomial classes depending on the presence of particular predicates in the queries. The current article gives the strictly larger polynomial class of simple graph queries, characterized by co-occurrences of closure predicates in a particular order along a same path in the query. More precisely, Proposition 5.8 states that simple graph queries are LOGLIN-reducible to LGQ forward forest queries, which admit linear XPath equivalents and thus polynomial evaluation [Gottlob et al. 2002]. Moreover, the polynomial evaluation of LGQ forward single-join DAG queries [Olteanu et al. 2004] implies there is an even larger polynomial class of graph queries, containing those queries polynomially-reducible to LGQ forward single-join DAG queries.

Schema-Aware Queries. TRS_3 can be used to rewrite queries under constraints using a technique similar to the chase&backchase of [Popa et al. 1999]. First, structural constraints, as specified, e.g., by schemata of XML documents to be queried, can be expressed as new binary predicates on the already existing variables of the query (the chase). Then, the enriched query, possibly a big graph, is rewritten by TRS_3 to a forward forest query (the backchase).



(a) Query Q (b) Query Q and Constraints (c) Forward Query FQ

Fig. 6. Query Rewriting under Schema Constraints.

Example 6.1. Consider the LGQ tree query Q asking for names appearing within journals whose titles are different from dm (Data Mining), only if the XML document speaks about authors. Figure 6 (a) shows Q using our graphical representation, where additionally the node corresponding to the distinguished variable v_1 is represented by a box.

$$Q(v_1) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{name}(v_1) \wedge \mathsf{par}^+(v_1, v_2) \wedge \mathsf{journal}(v_2) \wedge \mathsf{child}^+(v_2, v_3)$$
$$\wedge\, \mathsf{title}(v_3) \wedge \mathsf{child}(v_3, v_4) \wedge \text{'dm'}_{\neq}(v_4) \wedge \mathsf{child}^+(v_0, v_5) \wedge \mathsf{authors}(v_5)$$

Consider now given a schema with the content models $authors(name+)$ and $journal(title, editor, authors, price?)$ and text content for the other nodes.

From this schema we infer that (1) authors nodes can appear only as children of journal nodes, (2) title nodes can appear only as children of journal nodes and

precede authors nodes, and (3) name nodes can appear only as children of authors nodes. Formulated in LGQ and using the variables of Q, these constraints become $\mathsf{child}(v_2, v_3) \wedge \mathsf{child}(v_2, v_5) \wedge \mathsf{nextSibl}^+(v_3, v_5) \wedge \mathsf{child}(v_5, v_1)$. Figure 6 (b) shows Q together with these constraints displayed as dashed edges. The new obtained query is a graph with cycles. By rewriting it, we obtain the forward tree query FQ from Figure 6 (c).

The new query FQ is simpler and more efficient than Q, because it does not contain child^+ predicates and it restricts considerably the search space for possible name node answers. More important, the evaluation of FQ requires no buffering, because all other query constraints must be met *before* the name nodes are encountered in the stream. □

More principled investigation of query rewriting under constraints in our framework is subject to future research.

Query Minimization. The query minimization problem is to find for a given query a minimal-sized equivalent one. Current approaches to query minimization, e.g., [Wood 2001; Amer-Yahia et al. 2002; Flesca et al. 2003], consider tree queries with child and child^+ predicates and wildcard nodetests, and have at their core the observation that for such restricted queries minimal-sized equivalents can be found among their subqueries. Thus, the minimal query is obtained by pruning redundant subqueries until no subquery can be removed while preserving equivalence.

The query minimality aspect touched by the present article complements the above efforts by removing semantic redundancies of graph queries. The "minimal" query is not necessarily a subquery of the original one, and, besides dropping on some predicates, it may contain new ones. More precisely, the current article finds that simple graph queries are LOGLIN-reducible to forward forest queries, whose trees have the number of predicates independent of the number of predicates of the original queries and bounded in the number of variables minus one.

ReXP. An implementation of TRS_2 adapted to XPath syntax and called ReXP is hosted at `http://spex.sourceforge.net` as part of the publicly available streaming XPath processor called SPEX. In addition to a library dedicated to query rewriting and the specification of rewriting rules, ReXP offers a Java 1.5 API for parsing XPath queries and accessing their abstract syntax trees. A useful feature of ReXP is its graphical interface that allows to visualize the input and outcome of each rule application (see `http://spex.sourceforge.net` for ReXP snapshots).

7. CONCLUSION

This work has been primarily motivated by the need to enable on-the-fly (or streamed) evaluation of node-selecting queries on large XML repositories and unbounded XML streams. For this, (1) it identifies the reverse predicates as problematic for on-the-fly query evaluation, and (2) it proposes a robust rewriting framework for finding forward queries equivalent to queries with "problematic" reverse predicates. Properties of the proposed rewriting framework, like soundness and completeness, termination, confluence, and complexity of rewriting are also investigated. The second point above is essentially an expressiveness result and inves-

tigating along the same line, this work identifies various classes of queries equally expressive to their forward fragments.

Finally, various applications of the rewriting framework to streaming, main-memory, and RDBMS-based query evaluation are presented.

ACKNOWLEDGMENTS

I thank Christoph Koch for useful discussions on preliminary versions of this article. Also, the proof of Theorem 5.3 is a joint work with him.

REFERENCES

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

Altinel, M. and Franklin, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*. 53–64.

Amer-Yahia, S., Cho, S., Lakshmanan, L. V. S., and Srivastava, D. 2002. Tree pattern query minimization. *VLDB Journal 11,* 4, 315–331.

Apache Project 2001a. *Cocoon 2.0: XML publishing framework*. Apache Project. `http://xml.apache.org/cocoon/index.html`.

Apache Project 2001b. *Xalan-Java Version 2.2*. Apache Project. `http://xml.apache.org/xalan-j/index.html`.

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That*. Cambridge University Press.

Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M., and Josifovski, V. 2003. Streaming XPath processing with forward and backward axes. In *Proc. of Int. Conf. on Data Engineering (ICDE)*. 455–466.

Benedikt, M., Fan, W., and Kuper, G. M. 2005. Structural properties of XPath fragments. *Theor. Comput. Sci 336,* 1, 3–31.

Boag, S., Chamberlin, D., Fernandez, M. F., Florescu, D., Robie, J., and Siméon, J. 2006 XQuery 1.0: An XML query language. Candidate Recommendation, World Wide Web Consortium.

Bry, F., Coskun, F., Durmaz, S., Furche, T., Olteanu, D., and Spannagel, M. 2005. The XML stream query processor SPEX (demo). In *Proc. of Int. Conf. on Data Engineering (ICDE)*. SPEX prototype available at `http://spex.sourceforge.net`.

Bry, F. and Kröger, P. 2003. A computational biology database digest: Data, data analysis, and data management. *Distributed and Parallel Databases 13,* 1, 7–42.

Chan, C.-Y., Felber, P., Garofalakis, M. N., and Rastogi, R. 2002. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. of Int. Conf. on Data Engineering (ICDE)*. 235–244.

Clark, J. 1999. XSL Transformations (XSLT) version 1.0. W3C Recommendation, World Wide Web Consortium.

Clark, J. and DeRose, S. 1999. XML path language (XPath) version 1.0. W3C Recommendation, World Wide Web Consortium.

DeRose, S., Jr., R. D., Grosso, P., Maler, E., Marsh, J., and Walsh, N. 2002. XML pointer language (XPointer). W3C Recommendation, World Wide Web Consortium. `http://www.w3.org/TR/xptr/`.

Desai, A. 2001. Introduction to Sequential XPath. In *Proc. IDEAlliance XML Conference*.

Fallside, D. C. and Walmsley, P. 2001. XML-Schema. W3C Recommendation, World Wide Web Consortium. `http://www.w3.org/XML/Schema`.

Flesca, S., Furfaro, F., and Masciari, E. 2003. On the minimization of XPath queries. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*. 153–164.

Gottlob, G., Koch, C., and Pichler, R. 2002. Efficient algorithms for processing XPath queries. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*. 95–106.

Gottlob, G. and Koch, C. 2004. Monadic Datalog and the Expressive Power of Languages for Web Information Extraction. In *Journal of the ACM, 51(1):74-113*.

Gottlob, G., Koch, C., and Schulz, K. 2006. Conjunctive queries over trees. In *Journal of the ACM 53(2)*. to appear.

Grust, T., van Keulen, M., and Teubner, J. 2004. Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems (TODS)* 29, 91–131.

Helmer, S., Kanne, C.-C., and Moerkotte, G. 2002. Optimized translation of XPath into algebraic expressions parameterized by programs containing navigational primitives. In *Proc. of Int. Conf. on Web Information Systems Engineering (WISE)*. 215–224.

Hidders, J. and Michiels, P. 2003. Avoiding unnecessary ordering operations in XPath. In *Proc. of Int. Conf. on Data Base Programming Languages (DBPL)*. 54–70.

Ide, N., Bonhomme, P., and Romary, L. 2000. XCES: An XML-based standard for linguistic corpora. In *Proc. Annual Conf. on Language Resources and Evaluation (LREC)*. 825–830.

Ives, Z. G., Halevy, A. Y., and Weld, D. S. 2002. An XML query engine for network-bound data. *VLDB Journal 11,* 4, 380–402.

Kay, M. 2004. XSL Transformations (XSLT) Version 2.0. Working draft, World Wide Web Consortium.

Lincoln, P. and Christian, J. 1989. Adventures in associative-commutative unification. *Journal of Symbolic Computation 8,* 1–2, 217–240.

Marian, A. and Siméon, J. 2003. Projecting XML documents. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*. 213–224.

Marx, M. 2005. Conditional XPath. In *ACM Transactions on Database Systems (TODS)*. to appear.

NASA. 2004. *XML Group Resources Page*. `http://xml.gsfc.nasa.gov/`.

Newman, M. H. A. 1942. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics 43,* 2, 223 – 243.

Olteanu, D. 2004. Evaluation of XPath queries against XML streams. Ph.D. thesis, University of Munich.

Olteanu, D., Furche, T., and Bry, F. 2004. Evaluating complex queries against XML streams with polynomial combined complexity. In *Proc. of Annual British National Conference on Databases (BNCOD)*. 31–44.

Olteanu, D., Meuss, H., Furche, T., and Bry, F. 2002. XPath: Looking forward. In *Proc. of EDBT Workshop XMLDM*. 109–127. LNCS 2490.

Popa, L., Deutsch, A., and Tannen, V. 1999. Physical data independence, constraints, and optimization with universal plans. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*. 459–470.

Raghavan, V., Deschler, K., and Rundensteiner, E. A. 2005. Vamana: A scalable cost-driven XPath engine. In *Int. Workshop on XML Schema and Data Management (XSDM)*.

Schott, S. and Noga, M. L. 2003. Lazy XSL transformations. In *Proc. of ACM Symposium on Document Engineering*. 9–18.

Wood, P. T. 2001. Minimizing simple XPath expressions. In *Proc. of Int. Workshop on Web and Databases (WebDB)*. 13–18.