View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Digitale Hochschulschriften der LMU

Evaluation of XPath Queries against XML Streams

Dan Olteanu

Dissertation
zur Erlangung des akademischen Grades des
Doktors der Naturwissenschaften
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität München

vorgelegt von
Dan Olteanu

München, Dezember 2004

Erstgutachter: François Bry
Zweitgutachter: Dan Suciu (University of Washington)
Tag der mündlichen Prüfung: 11. Februar 2005

To my wife Flori

Abstract

XML is nowadays the *de facto* standard for electronic data interchange on the Web. Available XML data ranges from small Web pages to ever-growing repositories of, e.g., biological and astronomical data, and even to rapidly changing and possibly unbounded streams, as used in Web data integration and publish-subscribe systems.

Animated by the ubiquity of XML data, the basic task of XML querying is becoming of great theoretical and practical importance. The last years witnessed efforts as well from practitioners, as also from theoreticians towards defining an appropriate XML query language. At the core of this common effort has been identified a navigational approach for information localization in XML data, comprised in a practical and simple query language called XPath [46].

This work brings together the two aforementioned "worlds", i.e., the XPath query evaluation and the XML data streams, and shows as well theoretical as also practical relevance of this fusion. Its relevance can not be subsumed by traditional database management systems, because the latter are not designed for rapid and continuous loading of individual data items, and do not directly support the continuous queries that are typical for stream applications [17].

The first central contribution of this work consists in the definition and the theoretical investigation of three term rewriting systems to rewrite queries with reverse predicates, like parent or ancestor, into equivalent forward queries, i.e., queries without reverse predicates. Our rewriting approach is vital to the evaluation of queries with reverse predicates against unbounded XML streams, because neither the storage of past fragments of the stream, nor several stream traversals, as required by the evaluation of reverse predicates, are affordable.

Beyond their declared main purpose of providing equivalences between queries with reverse predicates and forward queries, the applications of our rewriting systems shed light on other query language properties, like the expressivity of some of its fragments, the query minimization, or even the complexity of query evaluation. For example, using these systems, one can rewrite any graph query into an equivalent forward forest query.

The second main contribution consists in a streamed and progressive evaluation strategy of forward queries against XML streams. The evaluation is specified using compositions of so-called stream processing functions, and is implemented using networks of deterministic pushdown transducers. The complexity of this evaluation strategy is polynomial in both the query and the data sizes for forward forest queries and even for a large fragment of graph queries.

The third central contribution consists in two real monitoring applications that use directly the results of this work: the monitoring of processes running on UNIX computers, and a system for providing graphically real-time traffic and travel information, as broadcasted within ubiquitous radio signals.

Zusammenfassung

Heutzutage ist XML der *de facto* Standard für den Datenaustausch im Web. Dabei reicht die Spanne an verfügbaren XML Daten von kleinen Webseiten bis hin zu immer größer werdenden Sammlungen, beispielsweise an biologischen oder anstronomischen Daten und sogar, möglicherweise unbegrenzte, Datenströme mit schnellem Datenaufkommen, wie sie in publish-subscribe Systemen verwendet werden.

Getrieben durch die weite Verbreitung von XML Daten, bekommt die Anfragebearbeitung an XML Daten zunehmend größere theoretische und praktische Bedeutung. In den letzten Jahren konnten Initiativen sowohl von Seiten der Industrie als auch aus der Forschung beobachtet werden, die darauf abziehen eine angemessene XML Anfragesprache zu definieren. Das Kernergebnis dieser Initiativen ist die Identifikation eines navigationalen Ansatzes zur Lokalisierung von Informationen in XML Daten in der benutzer-orientierten Anfragesprache XPath.

Diese Arbeit bringt die zwei oben genannten Welten, die XPath Anfragebearbeitung und XML Ströme, zusammen und zeigt die sowohl praktische als auch theoretische Relevanz dieser Verbindung.

Der erste Hauptbeitrag dieser Arbeit besteht in der Definition und der theoretischen Untersuchung von drei Termersetzungssystemen, um Anfragen mit sogenannten “reverse” Predikaten, wie beispielsweise `parent` oder `ancestor`, in equivalente Anfragen, die keine solche Predikate enthalten, umzuschreiben. Unser Ansatz ist essentiell fuer die Auswertung von Anfragen mit “reverse” Predikaten gegen unbegrenzte XML Ströme, da weder die Speicherung von bereits verarbeiteten Stromfragmenten noch mehrere Durchläufe über den XML Strom erforderlich sind.

Neben diesem Hauptziel, die Anwendungen unserer Umschreibungssysteme werfen ein neues Licht auf andere Eigenschaften der Anfragesprache, wie die Ausdruckskraft einiger Fragmente, die Minimierung von Anfragen, und sogar die Komplexität der Anfrageauswertung. Man kann beispielsweise unter Nutzung dieser Umschreibungssysteme beliebige Graphanfragen in equivalente Waldanfragen ohne “reverse” Predikate umschreiben.

Der zweite Hauptbeitrag besteht in einer strom-basierten, progressiven Auswertungsstrategie für Waldanfragen ohne “reverse” Predikate gegen XML Ströme. Die Auswertung wird spezifiziert durch die Komposition von sogenannten Stromverarbeitungsfunktionen und implementiert unter Verwendung von Netzwerken aus deterministischen Kellerautomaten. Die Komplexität dieser Auswertungsstrategie ist polynomiell sowohl in der Grösse der Anfrage als auch der Daten fuer Waldanfragen ohne “reverse” Predikate und sogar für viele Graphanfragen.

Der letzte Hauptbeitrag besteht aus zwei praktisch verwendbaren Überwachungssystemen, die direkt auf den Resultaten dieser Arbeit aufsetzen: die Überwachung von auf einem UNIX System laufenden Prozessen und ein System, das Verkehrsinformationen aus Radiosignalen in Echtzeit überwacht und graphisch aufbereitet.

Acknowledgments

During the last three years, many people have contributed directly or indirectly to the development of this dissertation. I would like to express my gratitude to them.

First of all I am deeply indebted to my advisor François Bry, for his continuing trust and support during the evolution of this thesis. Further, I am grateful to Dan Suciu, whose work on XML query processing influenced constantly my research directions. This thesis and its author further benefitted from long and very useful discussions with two of my best supporters Tim Furche and Holger Meuss. Without their active commitment, this dissertation would not have been possible. I thank the students, whose theses I co-supervised, for their interest in my work and for bringing new relevant ideas to surface: Fatih Coskun, Serap Durmaz, Tim Furche, Tobias Kiesling, Sebastian Schaffert, Dominik Schwald, and Markus Spannagel. I thank also the members of our teaching and research group for creating a stimulating environment at the office and a pleasant stay in Munich: among others, Slim Abdennadher, Sacha Berger, Tim Geisler, Martin Josko, Michael Kraus, Ellen Lilge, Bernhard Lorenz, Hans Jürgen Ohlbach, Paula Pătrânjan, Stephanie Spranger, and Felix Weigel. I especially want to mention Norbert Eisinger for his always competent advises on various subjects ranging from easy ones, like confluence of rewriting systems, to complex ones, like teaching computer science topics.

Last, but definitely not least, I thank my wife, Flori, for her love and non-interrupting support, my parents and my brother for enduring the physical distance that separated us for such a long time, and all my friends for the weekends we spent together doing no research.

Contents

1 Introduction **1**
- 1.1 Data Streams: Use, Concepts, and Research Issues 2
- 1.2 Thesis Contributions and Overview 6

2 Preliminaries **9**
- 2.1 XML Essentials 9
- 2.2 Example Scenarios 11

3 LGQ (Logic Graph Query): An Abstraction of XPath **15**
- 3.1 Data Model 16
- 3.2 Syntax 19
- 3.3 Semantics 22
- 3.4 Digraph Representations 25
- 3.5 Path, Tree, DAG, Graph Formulas and Queries 26
- 3.6 Forward Formulas and their Specializations 28
- 3.7 Measures for Formulas 29
- 3.8 LGQ versus XPath 31
  - 3.8.1 XPath 31
  - 3.8.2 Conciseness of LGQ over XPath 36
  - 3.8.3 XPath=LGQ Forests 38

4 Source-to-source Query Transformation: From LGQ to Forward LGQ **45**
- 4.1 Problem Description 48
- 4.2 A Taste of Term Rewriting Systems 52
- 4.3 Rewrite Rules preserving LGQ Equivalence 56
  - 4.3.1 Rules adding single-join DAG-Structure 57
  - 4.3.2 Rules preserving Tree-Structure 59
  - 4.3.3 Rules removing DAG-Structure 67
  - 4.3.4 Rules for LGQ Normalization 69
  - 4.3.5 Rules for LGQ Simplification 70
- 4.4 Three Approaches to Rewrite LGQ to Forward LGQ Forests 72
  - 4.4.1 Rewriting Examples 73
  - 4.4.2 Soundness and Completeness 76

4.4.3 Termination . . . 79
4.4.4 Confluence . . . 80
4.5 Complexity Analysis . . . 81
4.6 Related Work . . . 89

5 Evaluation of Forward LGQ Forest Queries against XML Streams 95
5.1 Problem Description . . . 96
5.2 Specification . . . 101
5.2.1 Stream Messages . . . 102
5.2.2 Stream Processing Functions . . . 103
5.2.3 From LGQ to Stream Processing Functions . . . 105
5.2.4 Evaluation of Atoms . . . 108
5.2.5 Evaluation of Path Formulas . . . 110
5.2.6 Evaluation of Tree Formulas . . . 112
5.2.7 Answer Computation . . . 119
5.3 Implementation . . . 120
5.3.1 SPEX Transducers and Transducer Networks . . . 120
5.3.2 Transducers for Forward LGQ Predicates . . . 122
5.3.3 Processing Example with Transducers for LGQ Predicates . . . 127
5.3.4 Transducers for Other Stream Processing Functions . . . 128
5.4 Minimization Problems for SPEX Transducer Networks . . . 130
5.5 Complexity Analysis . . . 133
5.6 Experimental Results . . . 140
5.7 Related Work . . . 142
5.7.1 Query Evaluation against stored XML Data . . . 144
5.7.2 Query Evaluation against XML Data Streams . . . 147
5.7.3 Hybrid Approaches . . . 153

6 Applications 155
6.1 Monitoring Computer Processes . . . 155
6.2 Streamed Traffic and Travel Information . . . 157

7 Conclusion 159

A Proofs 161

Chapter 1

Introduction

XML is nowadays the *de facto* standard for electronic data interchange on the Web. Currently available XML data range from small Web pages, the primary use of XML some years ago, to ever-growing XML repositories and rapidly changing and possibly unbounded XML streams. In order to meet the requirements for storing and processing XML-based Web pages, the XML community proposed recently in-memory tree representations of XML data augmented with basic processing capabilities, e.g., the DOM-based application program interface [145]. However, the shift in the size and arrival rate of XML data has to be met also by a shift in appropriate techniques for processing it. XML repositories, as used in natural language processing [92], biological [28] and astronomical data [119], get beyond the barrier of main-memory capacities available on personal computers. Also, for continuously generated XML streams used, e.g., in publish-subscribe systems [37, 7] and in Web data integration [51], technologies like DOM based on in-memory representations of the entire XML data are not appropriate. Traditional database management systems are not designed for rapid and continuous loading of individual data items, and they do not directly support the continuous queries that are typical for stream applications [17].

Animated by the ubiquity of XML data, the basic task of XML querying is becoming of great theoretical and practical importance. The last years witnessed efforts as well from practitioners, as also from theoreticians towards the goal of defining an appropriate XML query language. Various working drafts of W3C, e.g., [46, 23, 45], and research papers, e.g., [5, 137], describe relevant work. As a core of this common effort has been identified a navigational approach for information localization in XML data, comprised in a practical and simple query language called XPath [46].

This work brings together the two aforementioned “worlds”, i.e., the XPath query evaluation and the XML data streams, and shows as well theoretical as also practical relevance of this fusion. After shaping next some directions of current research on stream processing in general, and XML stream processing in particular, this chapter names the contributions of this work with pointers to relevant chapters.

1.1 Data Streams: Use, Concepts, and Research Issues

Data streams [101, 26] are continuously sent data, whose size and arrival rate make difficult or even impossible their storage before being processed. The focus of current research on data streams is to provide techniques that allow, without delaying the arrival rate of the data streams, (1) to monitor data streams, i.e., to watch them for particular data patterns, and (2) to analyze and produce aggregate values from data streams. This section highlights some concepts, application domains, and research issues mainly related to data stream monitoring, though many characteristics also apply to the data stream analysis.

Application Domains

Data streams are encountered in many domains, ranging from analysis of scientific data to monitoring and filtering systems.

- Sensor-based monitoring systems, e.g., for traffic or atmospheric conditions.

  New techniques for monitoring data streams are developed to locate devices, like cars on highways or luggages in airports, that are equipped with position emitters (sensors). For example, sensors equipping trucks are used to monitor their traffic and highways usages, based on which appropriate fees are computed. Sensors can equip also highways to detect traffic parameters like average speed or congestion.

  In meteorology, streams of scalar values representing atmospheric conditions are gathered by sensors and used in monitoring systems that enable, e.g., early recognitions of tornados. The Sensor Web project [118] at NASA develops instruments for monitoring and exploring environments. The Sensor Web is an independent network of wireless, intra-communicating sensor pods, deployed to monitor and explore a limitless range of environments, and can be tailored to whatever conditions it is sent to monitor.

- Usage monitoring systems.

  Streams conveying transactional data are gathered over networks from credit card usages and phone calls for detecting usage patterns indicating possible frauds [52].

- Publish-subscribe systems, e.g., for press, media, or financial news.

  The world becomes increasingly information-driven and the natural need to find the desired information is associated to finding the needle in a haystack. To partially fulfill these needs, publish-subscribe systems [7] are used to selectively disseminate existing information gathered from various heterogeneous sources (e.g., newspapers) called publishers. The large amounts of users, which subscribe with particular profiles, are then notified across a network in real time on content matching their interest.

- XML packet routing.

  XML routers perform content-based routing of individual XML packets to other routers or clients based upon profiles (queries) that describe the information needs [6]. Industry trend towards development of XML messaging systems for business applications has already spawned a floury of start-up companies developing XML routing systems, e.g., Firano Software, Sarvega, Forum, Elitesecureweb, Knowhow, Xbridgesoft, XMLblaster, cf. [81].

- Video filtering based on XML content descriptions.

  The new generation video standards, e.g., MPEG-7 [107, 143], provide elaborate XML content descriptions that contain information ranging from "size" to the "current speaker in scene". Such metadata is to be transmitted as an XML stream separated yet related with the real video stream. The content-based video filtering and routing is needed, e.g., for jumping directly to or skipping certain scenes. First prototypes of MPEG-7 based systems, e.g., [136], point to the need of efficient filtering techniques for fast and continuous XML streams of highly structured metadata.

- Analysis of scientific data.

  The European Southern Observatory (ESO) [111] is confronted with the problem of processing weekly terabytes of astronomical data, as gathered by its Very Large Telescope (VLT). Such raw (pixel-based) data is usually accompanied by its content description (metadata) wrapped in XML. To some extend, the characteristics of such data are that of data streams: its arrival rate and size make a standard approach for storing, indexing, and processing it rather difficult. Current approaches for dealing with the metadata component are also based on novel techniques for stream processing[1].

Punctuate, Tuple, and XML streams

There are three kinds of data streams currently under consideration. A data stream can be a continuous sequence of

- points, i.e., scalar values like numbers or characters,
- tuples, and
- so-called XML elements that are well-formed fragments of XML documents.

Punctuate streams (see, e.g., [29]) and tuple streams (see, e.g., [77]) consist in sequences of data items that have the same length and are flat, like relational database tuples. Punctuate streams can be seen as a special case of tuple streams, because the constituent points are tuples of arity one. The XML documents, as conveyed by many XML streams

[1] Joint work of ESO Archive Centre and the author is planned to provide efficient processing techniques of such XML-based metadata.

especially in monitoring measurement data, have reduced text content. XML is used here as a formalism for specifying tree-like data, whose size and nesting depth can be unbounded and whose structure can have recursive definition. All these characteristics make the processing of XML data streams especially challenging (see, e.g., [124]).

Querying data streams

Data streams querying, also called data streams monitoring, is the search for specific data patterns in the continuously sent data, like news about a particular country in a stream of news reports, big exchange rate fluctuations in a stream of stock market data, or particular life-threatening value combinations in a stream of medical measurement data. Existing research on data streams adopted in the first place existing query languages like SQL for tuple streams, e.g., [1, 77], and XPath for XML streams, e.g., [37, 124]. For specific application domains special query languages are developed. At AT&T a programming language called Hancock [52] has been developed and used to detect changes in user behaviour with respect to dialed phone numbers, thus changes that can indicate possible misuses.

Data streams pose new challenges to query evaluation. New techniques are needed to enable a real-time evaluation of possibly complex queries using as a few as possible space for temporary results. Queries against data streams are sometimes called *continuous* queries, for they are evaluated steadily against incoming data. Independently arises also the question regarding the time when the user is informed about the answers. This can be done either progressively, i.e., as soon they are computed, or at given time intervals, or when particular events happen, or even on explicit user request.

The existing query evaluation methods for data streams share common characteristics:

- Only single-pass query evaluation techniques are considered that require no or limited storage of the input data stream.
- The evaluation techniques are often based on finite/pushdown automata, for such automata require simple and limited, consciously used, storage capabilities.
- So-called “window” techniques that process only excerpts of a given size from the input data stream are often used. These techniques can guarantee bounds on the memory needed for temporary results at the expense of computing approximate answers instead of exact answers, cf. [12, 17, 139].
- Changes to already generated query answers are in general not considered, thus the only kind of supported change to the answers is the addition of further data.

Data stream systems versus Database systems

Querying data streams represents a research field complementary to querying databases. From a practical view point, the result of querying data streams can be used to populate databases. Figure 1.1 gives a brief comparison between data stream systems and traditional

	Data streams	Databases
Data	transient	permanent
Queries	permanent	transient
Changes	(mostly) limited to appending	arbitrary
Answers	approximate	exact
Data access	single-pass	arbitrary
Indexing	of queries	of data

Figure 1.1: Comparison of data stream and database systems

database systems. While in general in databases, data is permanent and large and queries are transient and few at a time, in data stream systems the queries are permanent and numerous, whereas the data is transient. Publish-subscribe systems that filter data streams containing news reports according to queries of subscribers are based on possibly very large databases of queries. Hence, in a data stream context, traditional database techniques like indexing can be primarily applied for queries and not for data. In this respect, the data stream system XTrie [37] indexes XPath queries and therefore accelerates the evaluation of a large set of queries against the same XML stream.

Existing work on sequence [141] and temporal databases [142] has addressed some of the issues of stream-based evaluation in a relational database context, like time-sensitive queries and their related windows-based evaluation techniques. However, research in sequence databases has focused on the generation of efficient query plans evaluable in one-pass over the stream and with constant memory, independent of the data. This is possible if the database system controls, e.g., the sequence flows, and is unfortunately impossible in a data stream system. Also, research in temporal databases is concerned primarily with maintaining a full history of each data value over time, whereas in data stream systems the focus is on processing new data on the fly.

A cursory Review of Data Stream Systems

Data stream processing has become very active. We provide here an incomplete list of references to research contributions that describe stream processing techniques developed mostly within the last two years. More in-depth considerations are done, e.g., by [17] for relational data, and in Section 5.7 for XML data.

The existing relational (tuple) stream systems use SQL extended with constructs for sliding windows [40, 39, 17] or graphical "box and arrows" interface for specify data flow through the system [36]. All these systems focus on optimizations, adaptivity to unpredictable and volatile environments, and support for blocking operators, i.e., those operators that are unable to produce the first tuple of its output until it has seen its entire input. Additionally, NiagaraCQ [40] proposes rate-based optimizations based on stream-arrival and data-processing rates. Telegraph [39, 105] focuses on query execution strategies over data streams generated by sensors. Aurora [36, 1] proposes compile-time optimiza-

tions, like reordering operators and detection of common subqueries, and run-time optimizations, like load shedding based on application-specific measures of quality of service. STREAM [18, 17, 12, 117] investigates classes of queries that can be answered exactly using a given bound of memory, memory management issues in case of approximate answers, and scheduling decisions for multiple query plans based on rate synchronizations within operators.

The XML stream systems are divided in two main classes: given a set of queries, some systems report on *matched* queries against XML documents conveyed in the stream [7, 94, 37, 14, 57, 58, 76, 81], whereas others return the matched stream fragments [104, 127, 132, 131, 20, 67, 139, 21, 98, 124, 125, 100, 99, 25, 133]. All these systems are automata-based and, for processing large sets of queries, most of them employ various techniques for finding commonalities among queries. They differ mainly in the complexities of the employed query evaluation algorithm, which vary from linear [37, 20, 124, 125] to exponential [7, 14, 131] in the size of the queries, and in the degree of supporting XPath or XQuery fragments for specifying queries, which varies from simple XPath paths with child and descendant axes [7, 94, 37, 14, 131] to XQuery queries with child and descendant axes and result construction [104, 98, 100].

From the second class, the SPEX system [128, 127, 67, 139, 124, 125, 25], which processes with polynomial complexities a considerably large XPath fragment containing all axes, is the topic of this work. To the best of our knowledge, non-trivial approximation techniques for coping with XPath query evaluation under hard memory bounds were developed only within the SPEX system by [139].

1.2 Thesis Contributions and Overview

This work motivates two complementary facets of the problem of XPath query evaluation against XML streams and proposes practical solutions to them: the rewriting of queries with reverse predicates into queries with forward predicates only, and the evaluation of forward queries against XML streams conveying ordered trees. The combined solution proposed by this work is representative for the current trend of query evaluation techniques presented in the previous section, because it uses one pass over the input stream and it is based on pushdown automata.

The rewriting step proposed by this work is essential and its rationale lies in the problematic evaluation of queries with reverse predicates in a stream context, as explained next. XPath has binary predicates that relate source and sink nodes from the conveyed tree. The order of these nodes in the stream, which corresponds to the depth-first, left-to-right preorder traversal of the conveyed tree, gives the type of predicates: for forward predicates, e.g., child, the sink nodes appear after their source nodes, whereas for reverse predicates, e.g., par, the sinks appear before their sources. In a stream context, the one-pass evaluation of queries with reverse predicates is problematic, because at the time source nodes are encountered, the sink nodes belong to the stream fragment already seen and are not anymore accessible. Some stream-based systems attack unsatisfactorily this problem by storing nec-

essary fragments of the past stream, e.g., Xalan [11]. Most other systems, including the one described by this work, use our rewriting solution, e.g., [127, 84, 21, 138, 106, 131, 129, 124].

We present next the chapters of this thesis and highlight their contributions.

Chapter 2 recalls shortly widely accepted models and syntaxes of semistructured data, among which the tree model and the XML syntax are used further in this work. Two running application scenarios, a journal archive and a genealogical tree, are introduced that will serve various examples of the next chapters.

Chapter 3 introduces the language of logical graph queries (LGQ), an abstraction of the practical language XPath. LGQ is similar to non-recursive Datalog with built-in predicates. The data model of LGQ and XPath is that of unranked ordered trees with labeled nodes, where the built-in predicates on nodes in such trees are the intuitive binary relations first child, next sibling, and equality, as well as their reverses and the closures of them and their reverses. LGQ queries are more succinct than XPath queries, though semantically LGQ is equivalent to a strict fragment of it, called the language of forward LGQ forests, and also to (forward) XPath.

For an efficient evaluation of LGQ (and of XPath) queries against XML streams, Chapter 4 identifies as relevant the problem of rewriting LGQ queries with reverse predicates into equivalent LGQ queries without reverse predicates, also called forward LGQ queries. In this sense, Chapter 4 proposes three sound and complete term rewriting systems that terminate and are confluent modulo the associativity and commutativity of the LGQ predicates $\wedge$, $\vee$, and node-equality. The systems differ as well in the time and space complexities for rewriting LGQ queries, as also in the capability to yield forward LGQ queries of certain restricted types. For example, it is shown that LGQ graph queries can be rewritten into forward LGQ forest queries, whose sizes range from linear to exponential in the sizes of the input queries. Also, the size of each forward LGQ tree in the rewritten forest query is variable-preserving minimal, i.e., it is bounded by the number of variables of the initial LGQ query, and not by the number of its predicates, which can be significantly bigger. Finally, the chapter surveys related work on query minimization, containment, and answering queries using views, all relevant and directly connected to results of the chapter.

Using the aforementioned results of Chapter 4, Chapter 5 introduces a streamed and progressive evaluation strategy of forward LGQ forest queries against XML streams. The streamed aspect of the evaluation resides in the sequential (one-time) access to the nodes of the XML stream. A progressive evaluation delivers incrementally the query answers as soon as they are computed. The proposed evaluation strategy compiles queries in so-called stream processing functions consisting of sequential and parallel compositions of simpler functions specifying LGQ predicates with restricted access. Later on, it is shown how each such simple function is implemented efficiently by a deterministic pushdown transducer, and how each stream processing function specifying a query is realized by a network of transducers. When dealing with transducers networks, there are at least two minimization problems to address: the problem of finding the minimal network equivalent to a given network, and the problem of minimal stream routing within a given network. Both problems are discussed and for the latter problem, an effective solution is given that improves considerably the processing time of transducer networks. The time and space

complexities for processing of XML streams with networks of transducers for eight different forward LGQ fragments is investigated and showed to be polynomial in both the stream and the query sizes. It is shown also that only for particular queries the space complexity of the evaluation depends only on the depth of the tree conveyed in the XML stream, and not on its size. Furthermore, based on both the complexity results of Chapters 4 and 5, polynomial upper bounds for the complexities for the evaluation of a large LGQ fragment of graph queries are given. This chapter concludes with an overview on existing evaluation techniques for XML queries in various contexts like main-memory, relational databases, compressed data, and streamed data.

Two real monitoring applications that use directly the results of this work are shortly presented in Chapter 6: the monitoring of processes running on UNIX computers, and a system for providing graphically in real-time traffic and travel information, as broadcasted within ubiquitous radio signals.

Chapter 7 concludes this work, and the Appendix fuels the interest of a critical eye with some proofs that were skipped from the main body of this work.

Chapter 2

Preliminaries

2.1 XML Essentials

Much of today's data does not fit naturally into the traditional relational data model [3, 27]. Especially Web data and data produced from the integration of heterogeneous sources have *irregular*, *self-describing*, and *often changing* structure. These characteristics contrast pregnantly with the regularity of and the a priori existence of schema for relational data. Such neither raw nor fully structured data is called *semistructured data* [2].

Semistructured data has irregular structure. In contrast to a relational data item (i.e., a tuple), a semistructured data item may have missing attributes, multiple occurrences of the same attribute, or recursive definition. These properties make semistructured data items tree-like. Also, the same attribute may have different types in different items, and semantically related information may be represented differently in various items. The above characteristics are supported also by recent studies on the properties of publicly available semistructured data [44, 43]. These studies emphasize that the semistructured data used for information interchange between applications has in general recursive structure definition: a survey of 60 real datasets found 35 to be recursive, from which the ones for data interchange are all recursive.

Semistructured data is self-describing. The content of a semistructured data item can be tagged with labels that remind of attribute names in relational schemas. Explicit schemas, when available, provide powerful grammar-based mechanisms for specifying classes of flexible, possibly ordered and recursively nested structures.

Semistructured data has often changing structure, especially in dynamic environments, where data evolves over time and has various versions [41]. For example, in publish-subscribe systems [37, 7], subscribers are informed on particular topics to be found in data published from various sources. Due to the high number of sources and the abundance of data, it is expected that data on these topics may come with different structures from different sources and even at different moments in time.

Models for Semistructured Data. There are several data models proposed to capture the aforementioned properties of semistructured data. These models can be classified in two classes: graph-oriented, e.g., OEM [5], and tree-oriented, e.g., DOM [145]. The Object Exchange Model (OEM) represents semistructured data as an (unordered) edge-labeled graph, where additionally the nodes may have object identifiers. The Document Object Model (DOM) represents semistructured data as a (ordered) node-labeled tree, where additionally each node can have further properties, called attributes, of the form $name = value$. For both data models, nodes with text content are permitted. There is a direct correspondence between OEM and DOM models: the edge labels in OEM become the node labels in DOM, the identifiers in OEM become values of special attributes in DOM, and references to nodes in OEM become attributes having as values the identifiers of the referenced nodes in DOM. Thus, although DOM describes primarily tree-like structures, through attributes it can provide also a general mechanism for the realization of various “virtual” edges between nodes, if attributes of different nodes have the same value.

Syntaxes for Semistructured Data. There are several syntaxes for semistructured data, among which we mention the OEM [5] and the XML [24] syntaxes. The eXtensible Markup Language (XML) syntax, proposed by the World Wide Web Consortium (W3C), is nowadays generally accepted as the data description language for both web-based information systems and electronic data interchange.

OEM considers a BibTex-like syntax, where structures are represented as sets of semistructured data expressions. Each such expression stands for an OEM substructure starting with an edge and it is serialized as the edge label followed by the identifier of its sink node, possibly followed by the serialization of the set of subexpressions representing its edges, enclosed by curly braces. The graph structure is preserved in this serialization with the help of node identifiers, whose definitions and references are written syntactically different.

XML is a generic markup language. In contrast to other markup languages, like HyperText Markup Language (HTML) [134], XML does not have a fixed vocabulary, the semantics of its markup is not a priori given, and the markup is used to specify the semantic structure of the data rather than its layout. Using XML, one can define markup languages: there exists a plethora of XML-based languages developed mostly by W3C.

A serialization of a DOM (tree) structure in XML can be done as follows. A node is serialized as the concatenation of serializations of its children nodes in their order, enclosed by an opening and a closing tag. For a node with the label a, its opening tag is $\langle a \rangle$ and its closing tag is $\langle /a \rangle$. An attribute with a name *name* and a value *value* is serialized as $name = value$. The set of attributes of a node is serialized as a whitespace-separated list of serializations of the constituent attributes, and positioned in the opening tag of that node between its label and the closing angle bracket $\rangle$. The serialization of a text node is that text. Note that angle brackets are not allowed in node labels and text[1]. The XML serialization of a semistructured data is often called an XML document. Figure 2.1 shows later an XML document representing a journal archive and its associated tree.

[1]Angle brackets are allowed in text only if they are escaped; e.g., ⟨![CDATA[⟨a⟩]]⟩ is a valid text node.

An XML document is *well-formed* if it is either of the form *text*, or of the form $\langle a \rangle rest \langle /a \rangle$, where *text* is a text that does not contain angle brackets, a is a label, and *rest* is a sequence of well-formed XML documents[2]. Well-formed XML documents are important because they correspond to serializations of (DOM-like) trees that describe semistructured data. Therefore, tools developed for semistructured data, like query languages, can be easily adjusted to well-formed XML documents.

This work considers a DOM-like model and the XML syntax for semistructured data. The DOM structures and their XML serializations are further simplified by considering (1) the node attributes modeled as children nodes having the attribute name as label and the attribute value as text content, and (2) the text nodes modeled as labeled nodes where the content of the former become labels of the latter.

Grammars for XML Data. Although XML data has an implicit structure, given by the labels stored within the tags, it is often useful to specify further structural and content constraints for XML documents. Such constraints can be specified within grammars (often called schemata) that define languages of well-formed XML documents. The XML documents generated by a grammar G are *valid* with respect to G, or simply G-valid. The advantages offered by the existence of grammars for XML documents stem mainly from the data structure and content awareness that can be used, e.g., by basic services like storage and querying for improving efficiency.

There are several formalisms for specifying XML grammars. Among them, Document Type Definitions (DTDs) [24], XML-Schema [59], and Relax NG [48] are the most popular ones. All these formalisms are special subclasses of regular tree grammars [102], thus the theory of regular tree grammars and of tree automata [50], to which tree grammars are related, can be fruitfully used also for studying the properties of the practical aforementioned grammar languages. Directly derived from the membership problem for tree automata, [102] develops also validation tools for XML documents.

2.2 Example Scenarios

We introduce here two real-life scenarios of semistructured data exemplifying representative usages of semistructured data for expressing relational and tree structures. These scenarios are used in the next chapters as basis for various examples.

Journal Archive

Since the arrival of the XML syntax for semistructured data, the common practice in processing data across networks is to deal locally with robust database systems that handle relational data and to wrap it in XML, when it comes to exchange it. Our first scenario considers a natural relational structure expressed using semistructured data.

[2]Although omitted here, XML documents can start with *prologs* defining, e.g., their character encodings or links to external grammars or styles.

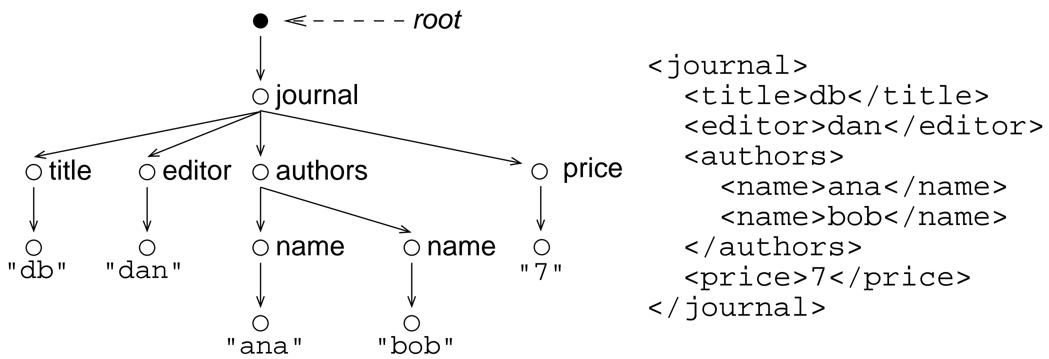


Figure 2.1: Excerpt of a journal archive

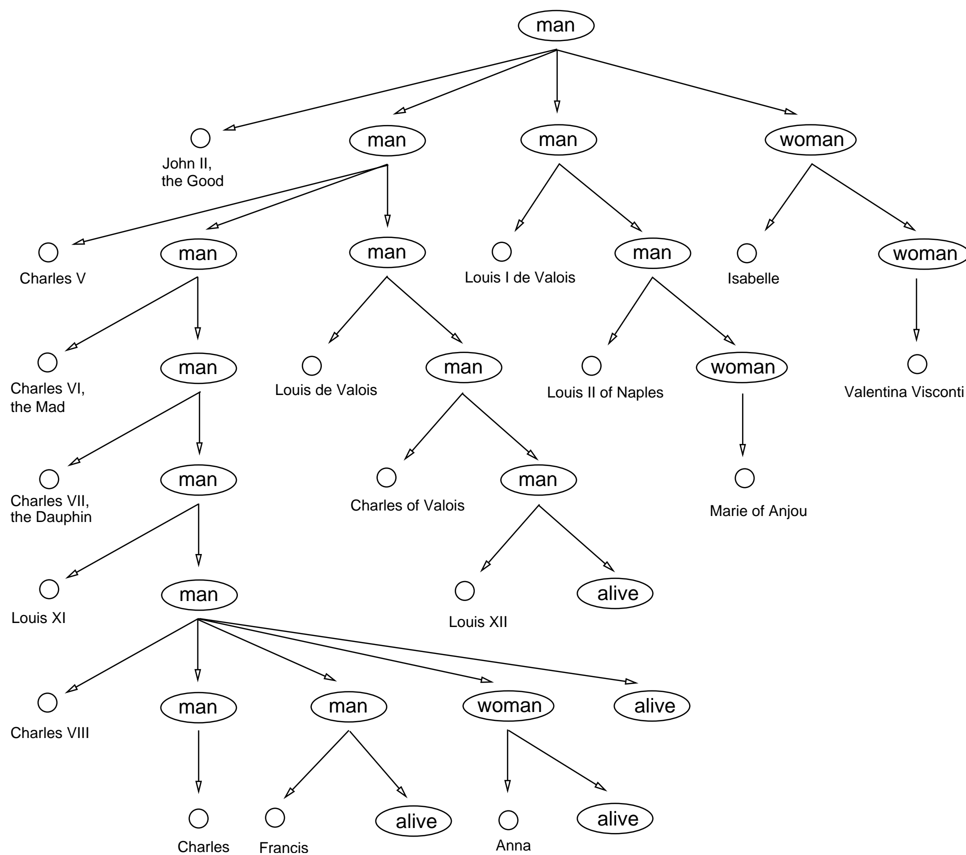


Figure 2.2: Excerpt of the family tree of John II, the Good

This scenario models a journal archive as a node-labeled tree, where each journal is represented as a node with label journal, and each of its properties, like title, editor, authors, and price, are represented as children nodes with corresponding labels. Figure 2.1 shows a possible journal entry and its XML serialization.

Genealogical Tree

Semistructured data is also used in practical cases to express tree structures with recursive definition [44, 43]. The second scenario considers a real-life case of semistructured data expressing the genealogical (or family) tree of important historical persons, like pharaohs, kings, or emperors. Such tree data were described since ancient times, and eventually used to decide on the successors at thrones.

This scenario models the genealogical tree of someone's folk (ancestors, descendants, brothers and sisters, nephews and nieces) as a node-labeled tree, where that person is represented as the root node, and each other person is represented as a node with label either man or woman and has a child text node consisting in its name, e.g., in the case of John this would be 'John'. The children of a person are represented also as children nodes of the node corresponding to that person, and the order between these nodes reflects the ascending order of the age of the corresponding children.

An interesting instance of this scenario is the family tree of the kings of France. An excerpt from its third dynasty, i.e., the Valois dynasty (1328-1589), is simplistically modeled in Figure 2.2 starting with the king John II, the Good, and ending shortly before the ascension to the throne by Louis XII in 1498[3].

Salic Law In older times, the decision on the succession to the throne of a kingdom (like of France), in case the king passes away, was sometimes defined by the so-called Salic Law. This law stipulates that the king is the first living man descending via exclusively a male line from the first king (in the case of the Valois dynasty, this is Charles, count of Valois), such that (1) all its male ancestors passed away, and (2) it has no older brother that lives or has a living male descendant via exclusively a male line. For the genealogical tree of Figure 2.2 of year 1498, the king is Charles VIII.

A question easily derived from the Salic Law, and, perhaps, posed by any pretendant to a throne in former times is: who must die, for someone to become a king? Considering the genealogical tree of Figure 2.2 of year 1498, Louis XII becomes a king, only if the current king Charles VIII, and also its male descendants that were alive at that time and descend exclusively via a male line (Francis in our case), die. This happened, indeed, in 1498, when Charles VIII died (in accident) and Francis also died.

As Chapter 3 shows next, such questions are not trivial ones and query languages like XPath 1.0 [46] are not expressive enough to pose them. The next chapter introduces a query language, called LGQ, that is expressive enough to capture such queries, and shows that a small extension of XPath makes it as expressive as LGQ.

[3]This family tree is in fact a graph: Louis XI is the son of Marie of Anjou and Charles VII, the Dauphin, and Charles of Valois is the son of Valentina Visconti and Louis of Valois.

Chapter 3

LGQ (Logic Graph Query): An Abstraction of XPath

XPath [46] is a key language among the plethora of W3C languages redefining the Web. The motivation for studying XPath stems from its importance as the prime language for expressing selection queries on XML documents, importance demonstrated especially by its usage in several W3C recommendations: the query language XQuery [23], the transformation language XSLT [45], the schema language XML-Schema [59], and the language for addressing fragments of XML documents XPointer [54]. The concepts of XPath are in fact not new. XPath is basically another syntax for a language of monadic queries (i.e., with a single free variable) having built-in binary predicates defining structural relations, like child or sibling, between nodes in ordered trees.

This thesis studies XPath through the more familiar glasses of a Datalog-like language, called LGQ. LGQ is the language of logic graph queries over tree-structured data, and can be seen as an abstraction of practical query languages for XML like XPath. It resembles closely the non-recursive Datalog with negation, or the language of conjunctive queries with union and negation [4].

The motivation for using LGQ instead of XPath is twofold. First, languages like the ones enumerated above and to which LGQ resembles, are well-studied and successfully researched in the literature [4]. Second, despite its growing importance, XPath is still not well-understood and its syntax poses many (unnecessary) technical challenges while doing more involved theoretical work (like query rewriting and answering).

The study of LGQ remains, however, also a study of XPath. This chapter shows that an LGQ fragment, representing the so-called LGQ forests, is equivalent to XPath. Chapter 4 investigates further the expressiveness of LGQ and shows that it is semantically not more expressive than its fragment of forward LGQ forests, thus than forward XPath. However, LGQ queries are in general more succinct than their equivalent XPath queries.

This chapter proceeds as follows. After introducing the common data model of LGQ and XPath, the LGQ syntax and semantics are provided, followed by the graphical representations and various measures for LGQ queries. At last, the connection between XPath and LGQ is established.

3.1 Data Model

As data model, LGQ and XPath use an abstraction of XML documents in from of finite unranked ordered trees, i.e., finite trees where a node can have an unbounded number of ordered children. This view upon XML documents is a simplification of that of XML Infoset [53], DOM [145], and XQuery 1.0 and XPath 2.0 Data Model [62], as explained next.

Such trees can have only two types of nodes: *root* and *element*. An element node is a labeled node that can have element nodes as children. A root node is a distinguished node without labels, and a tree has exactly one root node. The root node corresponds to the document node of DOM and of the XQuery 1.0 and XPath 2.0 Data Model.

Note that the data models of DOM, XQuery 1.0, and XPath 2.0 consider also other types of nodes, like attribute, text, processing instruction, and comment nodes. We consider the attributes of nodes modeled as children elements. Text nodes are also modeled as element nodes, where their text contents become labels. Both labels and text contents are words over a finite set of symbols. We may distinguish between text contents and labels by writing the text contents in quotes. We refer throughout this thesis to both of them as labels. The other kinds of nodes are not relevant for our primary issue of concern and their addition to the present formalism does not raise problems.

There are two functions of type Node → Boolean: *isRoot* and *isElement*. Applied to a same node, exactly one of them returns true. There is also a function *label* of type Node → String assigning to each element node its label and returning the empty string for the root node.

The nodes from a tree are represented in an XML document as described in Chapter 2. Figure 2.1 shows an XML document representing a journal archive and its associated tree.

There is a total order $\ll$ between the nodes in a tree that corresponds to the depth-first left-to-right preorder traversal of that tree. For two nodes n and m, $n \ll m$ means that n appears *before* m and m *after* n in the tree. For the corresponding XML document, n appears before m, if n is the root node and m is any element in the XML document, or if n is an element node and the representation of n has the first opening tag appearing before the representation of m. Because this order corresponds to the order of opening tags in XML documents, the order $\ll$ is also called the *document order*.

We consider Nodes(T) denoting the set of nodes in a tree T.

Nodetests

A *nodetest* is a construct permissible by the following grammar production

$$Nodetest ::= \; l \mid l_{\neq} \mid \texttt{*} \mid \textsf{root}$$

where l stands for a node's label, root is a special keyword, and * is a wildcard. A node has the nodetest l, if the node's label is l. A node has the nodetest $l_{\neq}$, if the node does not have the label l. Any node has the wildcard nodetest. Finally, only root nodes have the

nodetest root. Examples of nodetests are a, $a_{\neq}$, 't', $'t'_{\neq}$, where the latter two are written in quotes and refer to the text content of a node.

Let NodeTest be the set of all nodetests for a given tree instance. For a given node and nodetest, the function *test* returns true, if that node has that nodetest.

$$test : \text{Node} \times \text{NodeTest} \rightarrow \text{Boolean}$$

$$test(x,n) = \begin{cases} isElement(x) \wedge label(x) = name & \text{, if } n = name \\ isElement(x) \wedge label(x) \neq name & \text{, if } n = name_{\neq} \\ isElement(x) \wedge label(x) = \text{`}text' & \text{, if } n = \text{`}text' \\ isElement(x) \wedge label(x) \neq \text{`}text' & \text{, if } n = \text{`}text'_{\neq} \\ true & \text{, } n = * \\ isRoot(x) & \text{, } n = \mathsf{root}. \end{cases}$$

Note that the notion of nodetest introduced here deviates from the one of XPath 1.0 [46], where * holds only for element nodes, and there is no counterpart of $label_{\neq}$. Note also that other operations than equality and inequality, e.g., the less-than comparison $<$, can be incorporated in our nodetest formalism. In practical cases, such extensions make sense.

Binary Predicates

The base relations between two nodes in an ordered tree are the parent/child, sibling, and equality relations. Based on them, more complex relations can be defined. These base relations between nodes in a tree are supported by LGQ using the binary predicates fstChild, nextSibl, and self of type Node × Node → Boolean: for two nodes n and m, fstChild(n, m) holds if m is the first child of n, nextSibl(n, m) holds if m is the immediate next sibling of n, respectively self(n, m) holds if m is n. These predicates can be seen as specifications of basic services that the storage system or the XML document parser provide.

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

$$\alpha^0 = \mathsf{self}, \qquad \alpha^{n+1}(x,z) \Leftrightarrow \alpha^n(x,y) \wedge \alpha(y,z)$$

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

More convenient predicates can be defined further as the (composition of) transitive closures, and reflexive transitive closures, of the base predicates and their inverses [70]. For two nodes n and m,

- nextSibl$^+(n, m)$ holds if m is a following sibling of n, i.e., the next sibling of n, or the next sibling of the next sibling of n, and so on;

- nextSibl$^*(n, m)$ holds if m is a following sibling of n, or n itself;

- $\mathsf{prevSibl}(n, m) = \mathsf{nextSibl}^{-1}(m, n)$ holds if n is the preceding sibling of m, i.e., if m is the next sibling of n;
- $\mathsf{prevSibl}^{+}(n, m) = (\mathsf{nextSibl}^{+})^{-1}(m, n)$ holds if m is a preceding sibling of n, i.e., if n is a following sibling of m;
- $\mathsf{prevSibl}^{*}(n, m) = (\mathsf{nextSibl}^{*})^{-1}(m, n)$ holds if m is a preceding sibling of n, or n itself, i.e., if n is a following sibling of m, or m itself;
- $\mathsf{child}(n, m) = \mathsf{fstChild}(n, n') \wedge \mathsf{nextSibl}^{*}(n', m)$ holds if m is a child of n, i.e., if m is the first child of n, or a following sibling of the first child of n;
- $\mathsf{child}^{+}(n, m)$ holds if m is a descendant of n, i.e., if m is a child of n, or a child of a child of n, and so on;
- $\mathsf{child}^{*}(n, m)$ holds if m is a descendant of n, or n itself;
- $\mathsf{par}(n, m) = \mathsf{child}^{-1}(m, n)$ holds if m is the parent of n, i.e., if n is a child of m; Note that if $\mathsf{fstChild}(n, m)$ holds, then $\mathsf{par}(m, n)$ holds also;
- $\mathsf{par}^{+}(n, m) = (\mathsf{child}^{+})^{-1}(m, n)$ holds if m is an ancestor of n, i.e., if n is a descendant of m;
- $\mathsf{par}^{*}(n, m) = (\mathsf{child}^{*})^{-1}(m, n)$ holds if m is an ancestor of n, or n itself, i.e., if n is a descendant of m, or m itself;
- $\mathsf{foll}(n, m) = \mathsf{par}^{*}(n, n') \wedge \mathsf{nextSibl}^{+}(n', n'') \wedge \mathsf{child}^{*}(n'', m)$ holds if m follows n in document order, i.e., m is a following sibling of n, or of its ancestors, or descendant of a following sibling of either an ancestor of n or n itself;
- $\mathsf{prec}(n, m) = \mathsf{par}^{*}(n, n') \wedge \mathsf{prevSibl}^{+}(n', n'') \wedge \mathsf{child}^{*}(n'', m)$ holds if m precedes n in document order.

Note that for a given tree T and two nodes n and m in T, exactly one predicate $\alpha \in \{\mathsf{self}, \mathsf{par}^{+}, \mathsf{child}^{+}, \mathsf{prec}, \mathsf{foll}\}$ has $\alpha(n, m)$. This means also that, for any node n, these predicates divide the set of all nodes of T in disjunctive sets:

$$Nodes(T) = \mathsf{prec}(n) \cup \mathsf{foll}(n) \cup \mathsf{child}^{+}(n) \cup \mathsf{par}^{+}(n) \cup \mathsf{self}(n).$$

Predicate Classes. We classify the above built-in predicates depending on the order and structural relations between the nodes of the contained pairs. If for two nodes n and m $\alpha(n, m)$ 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. Based on this classification, we define the following predicate classes:

- the class F contains the **f**orward predicates $\{\mathsf{self}, \mathsf{fstChild}, \mathsf{child}, \mathsf{nextSibl}\}$,
- the class R contains the **r**everse predicates $\{\mathsf{par}, \mathsf{prevSibl}\}$,
- the class H contains the **h**orizontal predicates $\{\mathsf{nextSibl}, \mathsf{prevSibl}\}$,
- the class V contains the **v**ertical predicates $\{\mathsf{child}, \mathsf{par}\}$,
- the class X^+ for the transitive closures of predicates from $\mathsf{X} \in \{\mathsf{F}, \mathsf{R}, \mathsf{V}, \mathsf{H}\}$,
- the class X^* for the reflexive transitive closures of predicates from $\mathsf{X} \in \{\mathsf{F}, \mathsf{R}, \mathsf{V}, \mathsf{H}\}$,
- $\mathsf{X}^? = \mathsf{X} \cup \mathsf{X}^+ \cup \mathsf{X}^*$, where $X \in \{\mathsf{F}, \mathsf{R}, \mathsf{V}, \mathsf{H}\}$.

New classes can be created via intersection or union of aforementioned classes, e.g., $\mathsf{VF} = \mathsf{V} \cap \mathsf{F}$ contains the predicates belonging to both classes F and V, i.e., $\{\mathsf{fstChild}, \mathsf{child}\}$. Also, $\mathsf{VF}^? = \mathsf{V}^? \cap \mathsf{F}^?$ contains all forward vertical predicates, i.e., $\{\mathsf{fstChild}, \mathsf{child}, \mathsf{child}^+, \mathsf{child}^*\}$.

The restrictions of LGQ (or XPath) to allow only certain predicate classes define various LGQ (XPath) fragments. E.g., LGQ[$\mathsf{F}^?$] is the LGQ fragment without reverse predicates.

3.2 Syntax

As building blocks, LGQ has the built-in binary predicates of Section 3.1, unary predicates corresponding to nodetests, and unary predicates defined by the users using the built-in ones. The EBNF grammar for LGQ is given next.

$$
\begin{aligned}
LGQ &::= Id(Var) \ \leftarrow \ Formula. \\
Formula &::= Formula \wedge Formula \mid Formula \vee Formula \mid (Formula) \mid Atom. \\
Atom &::= Predicate(Var, Var) \mid Nodetest(Var) \mid Id(Var) \mid \neg Id(Var) \mid \bot \mid \top. \\
Predicate &::= Forward \mid Reverse. \\
Forward &::= Fwd_Base \mid Fwd_Base^+ \mid Fwd_Base^* \mid \mathsf{self} \mid \mathsf{fstChild} \mid \mathsf{foll}. \\
Fwd_Base &::= \mathsf{child} \mid \mathsf{nextSibl}. \\
Reverse &::= Rev_Base \mid Rev_Base^+ \mid Rev_Base^* \mid \mathsf{prec}. \\
Rev_Base &::= \mathsf{par} \mid \mathsf{prevSibl}.
\end{aligned}
$$

We explain next each LGQ syntactical construct.

Boolean Connectives

LGQ has two associative and commutative binary connectives $\wedge$ (and) and $\vee$ (or) and one unary connective $\neg$ (not). The connective $\neg$ has precedence over $\wedge$ that has precedence over $\vee$, i.e., $\neg$ binds stronger than $\wedge$ that binds stronger than $\vee$.

Atoms

A binary atom $\alpha(v_1, v_2)$, associates two sets of nodes identified by the variable v_1 and the variable v_2 according to the built-in predicate α.

For each possible nodetest predicate, e.g., a, $\mathsf{a}_{\neq}$, 'a', $\text{'a'}_{\neq}$, or root, there is a unary atom nodetest(v) that specifies the set of nodes with that nodetest. The set of nodes identified by v is restricted by a(v) to nodes with label a, and by $\mathsf{a}_{\neq}(v)$ to nodes that do not have label a. The set of nodes identified by v is restricted by 'a'(v) to nodes with text content 'a', and by $\text{'a'}_{\neq}(v)$ to nodes that do not have text content 'a'. In contrast to label nodetests, the text nodetests are enclosed in quotes. The nodetest root(v) restricts the nodes identified by v to the root node. To disambiguate the nodetest root from a possible label *root*, we consider the word *root* reserved and not allowed as a label.

For each user-defined predicate, e.g., Q, there is a unary atom $Q(v)$ that specifies the set of nodes contained by that predicate.

There are two special nullary atoms $\perp$ and $\top$ useful for proofs and formula rewriting. The atom $\perp$ selects no nodes regardless of the tree instance (i.e., it is unsatisfiable). The atom $\top$ selects always all nodes, regardless of the tree instance.

Using LGQ boolean connectives and atoms, one can construct formulas corresponding to conjuncts, disjuncts, and negations.

Types of Variables

LGQ variables are of two base types, depending at which position they appear in a binary atom $\alpha(v_1, v_2)$: the variables appearing at the first position are *source* variables, e.g., v_1 above, and the variables appearing at the second position are *sink* variables, e.g., v_2 above. A variable that never appears as source/sink is *non*-source/sink. A variable that appear more than one time as source/sink is called *multi*-source/sink. The amount of binary atoms having a given variable as source/sink defines the source/sink-arity of that variable. A variable with source/sink-arity n is also called a n-source/sink variable. The *forward sink-arity* of a variable in a disjunct is the amount of forward binary atoms that appear in that disjunct and have that variable as sink.

Formulas

An LGQ formula is defined recursively as follows

- a binary atom $\alpha(v_i, v_j)$ is a formula, where α is a built-in binary predicate,
- a unary atom nodetest(v) is a formula, where nodetest is a built-in unary predicate,
- a unary atom $Q(v)$ is a formula, where Q is a user-defined unary predicate,
- $\neg Q(v)$ is a formula, called also a negation, where Q is a user-defined unary predicate,
- $f_1 \wedge f_2$ is a formula, called also a disjunct, where f_1 and f_2 are formulas,

- $f_1 \vee f_2$ is a formula, called also a conjunct, where f_1 and f_2 are formulas.

The existence in a formula of conjuncts, disjuncts, and negations, is interpreted as the simultaneous existence, alternate existence, respectively absence, of the corresponding facts.

A formula f is in disjunctive normal form, if $f = f_1 \vee \cdots \vee f_n$ $(n \geq 1)$ and each f_i does not contain the connective $\vee$. The function *norm*: *Formula* $\rightarrow$ *Formula* brings a formula in disjunctive normal form.

A formula is *absolute*, if it has at least one non-sink variables, and all such non-sink variables have a root nodetest.

A disjunct is *connected*, if either (1) it does not contain binary atoms, or (2) it contains binary atoms and then each variable v_n from its body is either a non-sink variable or reachable from a non-sink variable v_0. A variable v_n is reachable from another variable v_0 in a disjunct f, if f contains $\alpha_1(v_0, v_1) \wedge \cdots \wedge \alpha_n(v_{n-1}, v_n)$ $(n \geq 0)$. A formula is connected, if all disjuncts from its disjunctive normal form are connected.

We use the following short-hand notations for formulas. To denote that f' is a subformula appears of f, we write $f' \subseteq f$. The set of variables from an LGQ formula f is denoted by $\mathrm{Vars}(f)$. If only a source v_0 and a sink v_h variable from a formula are of interest, such a formula can be abbreviated by $f(v_0, v_h)$ where f is an arbitrary identifier, e.g., $\mathsf{child}^+(v_0, v_1) \wedge \mathsf{b}(v_1) \wedge \mathsf{nextSibl}^+(v_1, v_2) \wedge \mathsf{d}(v_2)$ can be abbreviated to $p(v_0, v_2)$.

Rules and Queries

An LGQ *rule* has the form $Q(v) \leftarrow f$, and expresses the user-defined unary predicate Q. The left-hand side of $\leftarrow$ is the rule *head*, and the right-hand side is the rule *body*. The head of a rule has a single variable v, called the *head* variable, and therefore such rules are also called monadic [69]. A rule body is an LGQ formula.

An LGQ query is either a rule or a non-empty set of rules with one distinguished rule.

Restrictions on LGQ Queries

LGQ has syntactic restrictions for head variables, negation, and user-defined predicates. The head variable of each rule is syntactically restricted to appear in at least one non-negated atom of each disjunct in the query body, if the body is considered in disjunctive normal form. The negation can be applied only to user-defined predicates. Finally, recursive definitions of user-defined predicates are not allowed. In effect, LGQ is the language of non-recursive Datalog programs with negation and built-in predicates over tree structures, where additionally the negation is syntactically constrained to be applied only to user-defined predicates. This restriction eases various processings of LGQ queries, as developed in the next chapters, and comes, however, at the expense of writing larger queries than equivalent ones using unrestricted negation.

Throughout this thesis, we are interested in absolute and connected LGQ rules. An LGQ rule is absolute and connected, if its body formula is absolute and connected.

3.3 Semantics

The evaluation of an LGQ query consists in finding substitutions, or mappings, of its variables to nodes in the data tree instance, such that the predicates on such variables, as specified in that query, hold also on their substituting nodes in that tree instance. We define next the notion of LGQ substitution and consistent substitution and give the formal LGQ semantics based on such substitutions.

$$
\begin{aligned}
\mathcal{LQ} &: \text{Tree}\times \text{ Query} \rightarrow \text{Set(Node)}\\
\mathcal{LQ}_T[\![Q(v) \;\leftarrow\; f]\!] &= \pi_v(\mathcal{LF}_T[\![f]\!](subst(Q,T)))\\
\mathcal{LF} &: \text{Tree}\times\text{Formula} \times \text{Set(Substitution)} \rightarrow \text{Set(Substitution)}\\
\mathcal{LF}_T[\![Q(v)]\!](\beta) &= \{s \in \beta \mid s(v) \in \mathcal{LQ}_T[\![clause(Q)]\!]\}\\
\mathcal{LF}_T[\![\neg Q(v)]\!](\beta) &= \{s \in \beta \mid s(v) \notin \mathcal{LQ}_T[\![clause(Q)]\!]\} = \beta - \mathcal{LF}_T[\![Q(v)]\!](\beta)\\
\mathcal{LF}_T[\![f_1 \wedge f_2]\!](\beta) &= \mathcal{LF}_T[\![f_1]\!](\beta) \cap \mathcal{LF}_T[\![f_2]\!](\beta)\\
&= \mathcal{LF}_T[\![f_1]\!](\mathcal{LF}_T[\![f_2]\!](\beta)) = \mathcal{LF}_T[\![f_2]\!](\mathcal{LF}_T[\![f_1]\!](\beta))\\
\mathcal{LF}_T[\![f_1 \vee f_2]\!](\beta) &= \mathcal{LF}_T[\![f_1]\!](\beta) \cup \mathcal{LF}_T[\![f_2]\!](\beta)\\
\mathcal{LF}_T[\![(f)]\!](\beta) &= \mathcal{LF}_T[\![f]\!](\beta)\\
\mathcal{LF}_T[\![\alpha(v,w)]\!](\beta) &= \{s \in \beta \mid \alpha(s(v), s(w))\}\\
\mathcal{LF}_T[\![\mathsf{n}(v)]\!](\beta) &= \{s \in \beta \mid test(s(v), \mathsf{n})\}\\
\mathcal{LF}_T[\![\bot]\!](\beta) &= \emptyset\\
\mathcal{LF}_T[\![\top]\!](\beta) &= \beta
\end{aligned}
$$

Figure 3.1: LGQ Semantics

LGQ Substitutions

A *LGQ substitution* s is a total mapping from variables of an LGQ formula or query to nodes in a tree instance T: $s = \{v_1 \mapsto n_1, \ldots, v_k \mapsto n_k\}$ indicates that the variable v_i maps to the node n_i in T. If v is a variable and s a substitution, then $s(v)$ is the *image* of v under s, i.e., the node in T to which the variable v is mapped. An LGQ substitution s is *consistent* with an LGQ formula f and a tree T, if the predicates on variables of f hold also between the images of these variables under s in the tree T. More specifically, the consistency of an LGQ substitution is defined on the structure of LGQ formulas as follows:

- $f = \alpha(v, w)$, where α is a built-in predicate. Then, s is consistent with f and T if the images of v and w under s stand in predicate α in T, i.e., $\alpha(s(v), s(w))$.
- $f = \mathsf{n}(v)$, where n is a nodetest predicate. Then, s is consistent with f and T if the image of v under s is in the set of nodes in T with that nodetest n, i.e., $test(s(v), \mathsf{n})$.

- $f = Q(v)$, where Q is a user-defined predicate. Then, s is consistent with f and T if the image of v under s is contained in the predicate Q.
- $f = \neg Q(v)$, where Q is a user-defined predicate. Then, s is consistent with f and T, if the image of v under s is not contained in the predicate Q.
- $f = f_1 \wedge f_2$, where f_1 and f_2 are formulas. Then, s is consistent with f and T, if s is consistent with formulas f_1 and T, and also with f_2 and T.
- $f = f_1 \vee f_2$, where f_1 and f_2 are formulas. Then, s is consistent with f and T if s is consistent with at least one formula f_1 or f_2 and T.

LGQ Semantics

The LGQ semantics is given in Figure 3.1 by means of two functions $\mathcal{LQ}$ and $\mathcal{LF}$. The former function assigns meaning to LGQ rules, whereas the latter to LGQ formulas.

Applied on a tree T, an LGQ formula f, and a set of LGQ substitutions β of variables in f to nodes in T, the function $\mathcal{LF}$ computes the subset of β representing the substitutions consistent with f and T. If the tree T is understood from the context, then it can be ommited for simplification. The function $clause : Id \rightarrow$ Query used here delivers for a given user-defined predicate the rule defining it. Applied on an LGQ query $Q(v) \leftarrow f$ and a set of LGQ substitutions β of variables in f to nodes in T, the function $\mathcal{LQ}$ extracts the set of images of the head variable v under all substitutions from β consistent with f and T. If $\beta = \text{Set(Substitution)}$ is the set of all possible LGQ substitutions computable for a given query and a tree instance T, the function $\mathcal{LQ}$ defines the answer to that LGQ query. For a query Q and a tree T, Set(Substitution) is computed by $subst(Q, T)$.

LGQ Equivalence and Unsatisfiability

Definition 3.3.1 (LGQ Formula Equivalence). *Two LGQ formulas l and r are equivalent, noted $l \equiv r$, iff for any tree T the sets of all LGQ substitutions consistent with l and T, respectively r and T, restricted to the common variables of l and r, are the same:*

$$\pi_{vars}(\mathcal{LF}_T[\![l]\!](\beta_1)) = \pi_{vars}(\mathcal{LF}_T[\![r]\!](\beta_2))$$

where $vars = \text{Vars}(l) \cap \text{Vars}(r) \neq \emptyset$ and $\beta_1 = subst(l, T), \beta_2 = subst(r, T)$.

Definition 3.3.2 (LGQ Query Equivalence). *Two LGQ queries Q_1 and Q_2 are equivalent, noted $Q_1 \equiv Q_2$, iff for any tree T they select the same set of nodes:*

$$\mathcal{LQ}_T[\![Q_1]\!] = \mathcal{LQ}_T[\![Q_2]\!].$$

Note that probing the equivalence of two formulas, as considered by Definition 3.3.1, requires that both formulas have common variables. This definition can be extended to formulas without common variables, if mappings between variables of both formulas are provided.

An important aspect of LGQ equivalence is that LGQ formulas that are identical up to equivalent subformulas are also equivalent.

Proposition 3.3.1 (Equivalence-preserving adjunction). *Let e_1 and e_2 be two LGQ formulas that are identical up to two subformulas l of e_1 and r of e_2. Then, e_1 and e_2 are equivalent, iff l and r are equivalent, and e_1 and e_2 do not contain variables that appear in one of l and r, and not in the other.*

Proof. Let $e_1 = l \wedge e$, $e_2 = r \wedge e$. The case with $e_1 = l \vee e$, $e_2 = r \vee e$ is dual. Then,

$$\text{Vars}(e) \cap (\text{Vars}(l) - \text{Vars}(r)) = \emptyset, \text{Vars}(e) \cap (\text{Vars}(r) - \text{Vars}(l)) = \emptyset.$$

Let us consider also

$$\beta_e = subst(e,T), \beta_l = subst(l,T), \beta_r = subst(r,T), \beta_{l,e} = subst(l \wedge e,T)\beta_{r,e} = subst(r \wedge e,T).$$

From the hypothesis (cf. Definition 3.3.1 of LGQ equivalence) we have (for any tree instance)

$$l \equiv r \Leftrightarrow \pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![l]\!](\beta_l) = \pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![r]\!](\beta_r).$$

The extension of the sets of substitutions β_l and β_r to $\beta_{l,e}$, respectively $\beta_{r,e}$, does not change the above equality because the projection is still done on common variables.

$$\pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![l]\!](\beta_{l,e}) = \pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![r]\!](\beta_{r,e}).$$

Because the variables from $\text{Vars}(l)$ contained in e appear also in $\text{Vars}(r)$, we have

$$\begin{aligned}
&\pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![e]\!](\beta_{l,e}) = \pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![e]\!](\beta_{r,e}) \Rightarrow\\
&\pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![l]\!](\beta_{l,e}) \cap \pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![e]\!](\beta_{l,e}) =\\
&\pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![r]\!](\beta_{r,e}) \cap \pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![e]\!](\beta_{r,e}) \Leftrightarrow\\
&\pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![l \wedge e]\!](\beta_{l,e}) = \pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![r \wedge e]\!](\beta_{r,e}).
\end{aligned}$$

The projection can be extended safely from the set of variables $\text{Vars}(l)\cap\text{Vars}(r)$ to $(\text{Vars}(l)\cap \text{Vars}(r))\cup\text{Vars}(e)$, because e has variables that either appear in both l and r, or not appear in any of them.

$$\begin{gathered}
\pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![l \wedge e]\!](\beta_{l,e}) = \pi_{\text{Vars}(l)\cap\text{Vars}(r)}\mathcal{LF}[\![r \wedge e]\!](\beta_{r,e}) \Leftrightarrow\\
\pi_{(\text{Vars}(l)\cap\text{Vars}(r))\cup\text{Vars}(e)}\mathcal{LF}[\![l \wedge e]\!](\beta_{l,e}) = \pi_{(\text{Vars}(l)\cap\text{Vars}(r))\cup\text{Vars}(e)}\mathcal{LF}[\![r \wedge e]\!](\beta_{r,e}) \Leftrightarrow\\
l \wedge e \equiv r \wedge e.
\end{gathered}$$

□

Definition 3.3.3 (LGQ unsatisfiability). *An LGQ formula e is unsatisfiable iff for any tree T the set of LGQ substitutions consistent with e is empty*

$$\mathcal{LF}_T[\![e]\!](subst(e,T)) = \emptyset$$

An LGQ query Q is unsatisfiable iff for any tree T it does not select any node

$$\mathcal{LQ}_T[\![Q]\!] = \emptyset$$

3.4 Digraph Representations

The *digraph representation* of an LGQ formula (query) is the directed multi-graph obtained by taking the formula (query) variables as nodes and the binary predicates on them as labels for edges. The node for the head variable (in the case of a query) is significantly represented using a box, the nodes for the other variables are represented using ellipses. Optionally, we may annotate the nodes with their unary predicates. The nodes corresponding to (non-sink) variables with the root nodetest predicate are filled-in in black.

In order to avoid cluttering, we may use a simplified digraph representation, where the edges are drawn in such a way so as to convey the type of their corresponding predicate: vertical (eventually inclined)/horizontal edges stand for vertical/horizontal predicates, and their direction conveys whether they are forward or reverse. Vertical forward predicates are then drawn as directed up-down edges, vertical reverse as down-up, horizontal forward as left-right, horizontal reverse as right-left[1]. The edges are then labeled only with the plus (+) sign, if they correspond to transitive closure predicates, and with the wildcard (*) sign, if they correspond to reflexive transitive closure predicates.

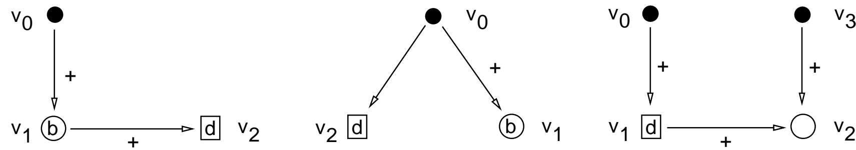


Figure 3.2: Digraph Representations of LGQ Queries Q_1, Q_2, and Q_3 of Example 3.4.1

Example 3.4.1. The following queries have the digraph representations from Figure 3.2

$$
\begin{aligned}
Q_1(v_2) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{b}(v_1) \wedge \mathsf{nextSibl}^+(v_1, v_2) \wedge \mathsf{d}(v_2) \\
Q_2(v_2) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{b}(v_1) \wedge \mathsf{child}(v_0, v_2) \wedge \mathsf{d}(v_2) \\
Q_3(v_1) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{d}(v_1) \wedge \mathsf{nextSibl}^+(v_1, v_2) \wedge \mathsf{root}(v_3) \wedge \mathsf{child}^+(v_3, v_2).
\end{aligned}
$$

□

There are formulas for which one can not derive a simplified digraph representation. For example, the formula $\mathsf{child}^+(v_0, v_1) \wedge \mathsf{nextSibl}(v_1, v_2) \wedge \mathsf{child}^+(v_2, v_0)$ enforces to draw from the descendant v_2 of v_0 a descendant edge back to v_0. However, from v_2, only edges having vertical reverse predicates as labels can be drawn to v_0, and the simplified digraph can not be derived. In fact, LGQ formulas that do not have a simplified digraph are unsatisfiable, i.e., their result is always empty regardless of the input data. Throughout the thesis, all formulas used in examples permit simplified digraphs Chapter 4 provides a rewriting-based mechanism that detects unsatisfiability of LGQ formulas and rewrites such formulas to the empty formula $\perp$.

[1]The predicates foll and prec are not supported by this simplified digraph representation. They can be, however, substituted by their definition based only on horizontal and vertical predicates, cf. Section 3.1.

If an LGQ formula contains several disjuncts, when brought in disjunctive normal form, then its digraph representation is the collection of the digraph representations of each such disjuncts. If an LGQ disjunct has also negation, i.e., it is of the form $f \wedge \neg Q(v)$ where f can contain again negation, then its digraph representation consists in the digraph representations of f, which is additionally marked with a plus (+) sign, and of Q, which is additionally marked with a minus (-) sign. In order to distinguish the variable v used to join Q and f, there is a dotted line between the box for the variable v in the digraph representation of Q and the representation of the same variable in the digraph of f.

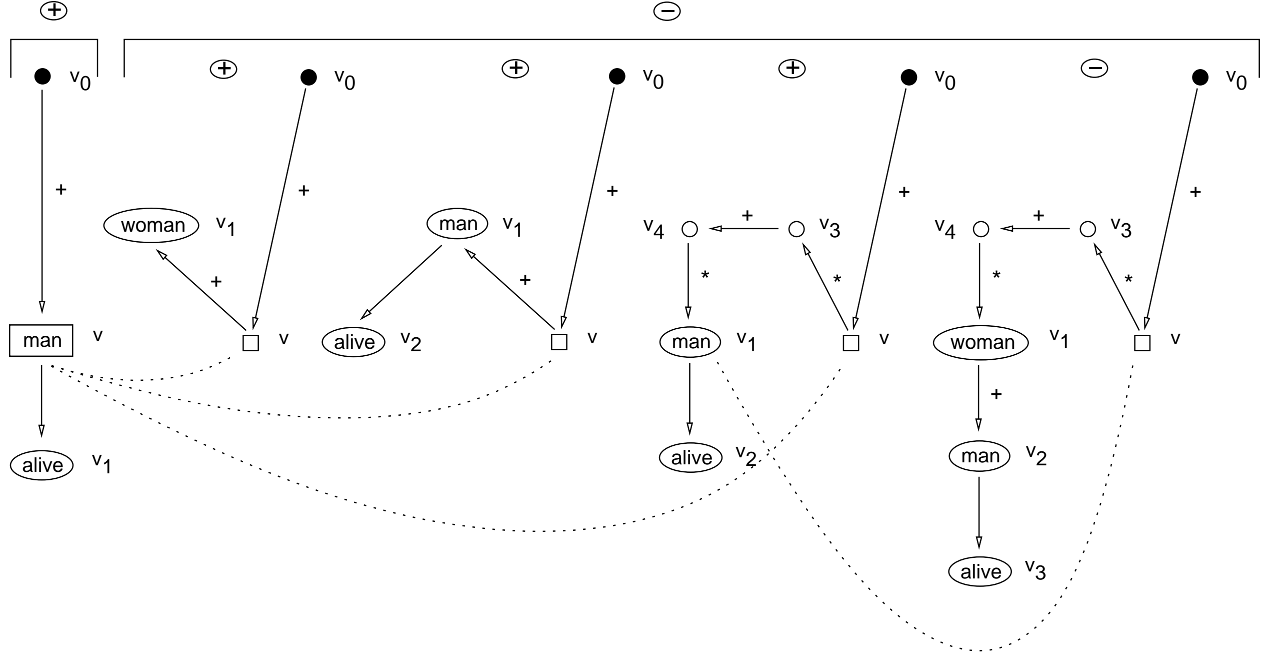


Figure 3.3: Digraph Representation of the LGQ Query of Example 3.4.2

Example 3.4.2. Consider an LGQ query asking for the current king after the Salic Law. For the genealogical tree of John II, the Good from Section 2.2, this man is Charles VIII. This query can be formulated in LGQ as (1) the selection of all living men, (2) the selection of all men that descend from at least one female descendant, or (3) have male ancestors that are alive, (4) or have male precedings that are alive, (4) but these precedings must descend via a male line from John.

$$
\begin{aligned}
Q(v) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{man}(v) \wedge \neg Q_1(v) \wedge \mathsf{child}(v, v_1) \wedge \mathsf{alive}(v_1) \\
Q_1(v) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{par}^+(v, v_1) \wedge \mathsf{woman}(v_1) \vee \\
&\qquad \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{par}^+(v, v_1) \wedge \mathsf{man}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{alive}(v_2) \vee \\
&\qquad \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{prec}(v, v_1) \wedge \mathsf{man}(v_1) \wedge \neg Q_2(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{alive}(v_2) \\
Q_2(v) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{prec}(v, v_1) \wedge \mathsf{woman}(v_1) \wedge \mathsf{child}^+(v_1, v_2) \wedge \mathsf{man}(v_2) \\
&\qquad \wedge \mathsf{child}(v_2, v_3) \wedge \mathsf{alive}(v_3).
\end{aligned}
$$

In order to use the simplified digraph representation for queries with horizontal and vertical predicates only, we replace in Q_1 and Q_2 $\mathsf{prec}(v, v_1)$ by $\mathsf{par}^*(v, v_3) \wedge \mathsf{prevSibl}^+(v_3, v_4) \wedge \mathsf{child}^*(v_4, v_1)$. The query Q has the digraph representation from Figure 3.3. □

3.5 Path, Tree, DAG, Graph Formulas and Queries

We introduce next the (rather intuitive) notions of path, tree, DAG, and graph formulas (queries), by analogy to their representation as graphs described in Section 3.4, where variables induce nodes and binary atoms induce oriented edges in that graph.

A *path* formula $p(v, w)$ is a connected formula representing a disjunct of atoms, where (1) each variable is neither multi-source nor multi-sink, and (2) it contains exactly one non-sink variable. The non-sink variable, denoted above by v, is called the path source, and the only non-source variable, denoted above by w, is called the path sink.

A *path query* is a query with an absolute path formula as body and the head variable as the path sink, i.e., $Q(t) \leftarrow p(v, t)$ with $p(v, t)$ a path formula. An example of a path query is $Q(v_2) \leftarrow root(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{b}(v_1) \wedge \mathsf{nextSibl}^+(v_1, v_2) \wedge \mathsf{d}(v_2)$, and its graphical representation is given in Figure 3.2.

A *tree* formula $t(v, w_1, \ldots, w_n)$ is a connected formula where (1) each variable can be multi-source but not multi-sink, and (2) it contains exactly one non-sink variable. The non-sink variable, denoted above by v, is called the tree source, and the non-source variables, denoted by $w_1, \ldots, w_n$, are called the tree sinks. A disjunction of tree formulas is a *forest* formula.

A *tree query* is a query with an absolute tree formula as body. An example of a tree query is $Q(v_2) \leftarrow root(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{b}(v_1) \wedge \mathsf{child}(v_0, v_2) \wedge \mathsf{d}(v_2)$, where v_0 is a multi-source variable. Its graphical representation is given later in Figure 3.2.

A *DAG* formula is a connected formula with multi-source and multi-sink variables, but without cycles. A formula is cyclic if, after bringing it in disjunctive normal form, it contains subformula paths with the same variable as path source and sink. *Single-join* DAG formulas are DAG formulas where additionally there are no two distinct path formulas with more than one non-sink common variable.

A *DAG query* (resp. single-join DAG query) is a query with an absolute DAG (resp. single-join DAG) formula as body. The single-join DAG query $Q(v_1) \leftarrow root(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{d}(v_1) \wedge \mathsf{nextSibl}^+(v_1, v_2) \wedge \mathsf{root}(v_3) \wedge \mathsf{child}^+(v_3, v_2)$ exemplifies a multi-sink variable v_2 occurring as sink in two binary atoms. Its graphical representation is given later in Figure 3.2.

A *graph* formula is a connected formula, without any restrictions concerning the cycles or types of variables. A *graph query* is a query with an absolute graph formula as body.

There is a specialization relation between classes of path, tree, single-join DAG, DAG, and graph formulas: one class is a specialization of the subsequent classes, in the order given above.

Variable-preserving Minimality of LGQ Trees and Forests

LGQ formulas can be quite complex, graph-like, where a variable can be reachable from another variable via several paths, or where cycles are allowed. The data instances to query are, however, trees. A natural question is whether this expressivity of LGQ graph formulas is not equivalent to that of LGQ tree formulas. Chapter 4 gives a positive answer to this question. The intuition is that even if in a disjunct a variable is reachable via several distinct paths from another variable, their images in a tree instance are connected using exactly one path. Thus a finite disjunction of all connection possibilities of the two variables consistent to the initial reachability constraints should be always doable. Each disjunct in the new formula would be then a LGQ tree subformula, where there is exactly one path from the non-sink variable to each sink variable v, thus one atom having v as sink.

The property of LGQ tree formulas to have each variable appearing only once as sink (except for the non-sink variable) is called variable-preserving minimality.

Definition 3.5.1 (Variable-preserving minimality). *A satisfiable formula e is variable-preserving minimal, if the number of binary atoms in each disjunct of the disjunctive normal form of e is equal to the number of variables of that disjunct minus one.*

Proposition 3.5.1 (Variable-preserving minimality of LGQ trees). *LGQ tree formulas are variable-preserving minimal.*

Proof. An LGQ tree formula does not have multi-sink variables, thus each variable appears exactly once as sink, except for the non-sink variable having the root nodetest. Therefore, the number of binary atoms in the tree formula is the number of sink variables of that formula, i.e., the number of all variables minus one. □

As a corollary, LGQ forest formulas are also variable-preserving minimal, because they are defined as disjunctions of LGQ tree formulas that are variable-preserving minimal.

3.6 Forward Formulas and their Specializations

This section gives a classification of formulas necessary for the results of Chapter 5.

A binary atom is forward, if its predicate is forward, and reverse otherwise. All unary atoms are considered forward. A formula (query) is forward if it contains only forward atoms, and it is reverse otherwise.

We give in the following the syntactical characterization of three types of forward formulas (queries): source-down, parent-down, and root-down forward formulas (queries).

A forward path formula is *source-down*, or simply sdown, if its path source is the source variable of an α-atom with the predicate $\alpha \in \mathsf{VF}^? \setminus \{\mathsf{self}\} = \{\mathsf{fstChild}, \mathsf{child}, \mathsf{child}^+\}$, and the formula contains no foll-atoms. Intuitively, for a (source) node being the image of the path source under an LGQ substitution, a sdown path selects only descendant nodes of that source node. A sdown formula can contain also atoms with $\mathsf{HF}^?$ predicates (but

their source variable must not be the path source), because the sibling nodes of children or descendants of the source node (as selected by the first atom child or child^+) are also children or descendants of the source node.

A forward path formula is *parent-down*, or simply pdown, if its path source is the source variable of an α-atom with $\alpha \in \mathsf{HF}^? = \{\mathsf{nextSibl}, \mathsf{nextSibl}^+, \mathsf{nextSibl}^*\}$, and it contains no foll-atoms. Intuitively, for a (source) node being the image of the path source variable under an LGQ substitution, a pdown path selects only descendant nodes of the parent of that context node. Other forward but foll-atoms are allowed, because the sibling nodes of the source node and their children or descendants are children or descendants of the parent of the source node.

A forward path formula is *root-down*, or simply rdown, if it contains at least a foll-atom and can contain any other forward atoms. Intuitively, for a (source) node being the image of the path source variable under an LGQ substitution, a rdown path selects only descendant nodes of the root of tree containing that source node. Note that an LGQ forward path formula is by default a rdown path formula.

The three types of path formulas allow also (sufficient) semantical characterizations:

$$p(v, w) \textit{ is sdown} \rightarrow \forall s \in \mathrm{Set(Substitution)} : \mathsf{child}^+(s(v), s(w))$$
$$p(v, w) \textit{ is pdown} \rightarrow \forall s \in \mathrm{Set(Substitution)} : \mathsf{par}(s(v), s(v')) \wedge \mathsf{child}^+(s(v'), s(w)) \wedge s(v) \ll s(w)$$
$$p(v, w) \textit{ is rdown} \rightarrow \forall s \in \mathrm{Set(Substitution)} : \mathsf{root}(s(v')) \wedge \mathsf{child}^+(s(v'), s(w)) \wedge t(v) \ll t(w).$$

Note that the above implications hold in both directions only for *some* given source nodes.

In general, a formula is (1) sdown if it contains only sdown path subformulas with a multi-source variable as the path source, (2) pdown if it contains at least a pdown path subformula with a multi-source variable as the path source and can contain sdown path subformulas, and (3) is rdown if it contains at least a rdown path subformula with a multi-source variable as the path source.

Example 3.6.1. Consider the path formulas $p_1(v_0, v_2) = \mathsf{child}(v_0, v_1) \wedge \mathsf{nextSibl}^+(v_1, v_2)$, $p_2(v_0, v_2) = \mathsf{nextSibl}^+(v_0, v_1) \wedge \mathsf{child}(v_1, v_2)$, and $p(v_0, v_2) = \mathsf{child}(v_0, v_1) \wedge \mathsf{foll}(v_1, v_2)$. Then, $p_1(v, w)$ is a sdown path formula, $p_2(v, w)$ is a pdown path formula, and $p(v, w)$ is a rdown path formula. Also, $p_1(v, w)$ is a sdown tree formula, $p_1(v, w_1) \wedge p_2(v, w_2)$ is a pdown tree formula, and $(p_2(v, w_1) \vee p_3(v, w_2)) \wedge p_1(v, w)$ is a rdown tree formula. □

3.7 Measures for Formulas

This section introduces measures for formulas necessary for the results of Chapter 4.

A useful relation computable for any two variables in a formula is the connectivity (or reachability) relation. Intuitively, two variables in a formula e are connected if there is (at least) one path between their corresponding nodes in the digraph representation of e. Additionally, we may also compute the length of such a path.

For LGQ formulas with cyclic digraphs, in the computation of paths connecting any two variables, we consider all cycles detected and not considered.

Definition 3.7.1 (Variable Connection). *The connection from variable a to variable b via a sequence of binary predicates p in a formula e is a 4-ary predicate $a \overset{p}{\leadsto}_e b$ defined as follows:*

- $a \overset{\alpha}{\leadsto}_e b$, *if* $\alpha(a,b) \subseteq e$,
- $a \overset{p.q}{\leadsto}_e b$, *if* $a \overset{p}{\leadsto}_{e_1} v \overset{q}{\leadsto}_{e_2} b$, *and* $e_1 \wedge e_2 \subseteq norm(e)$.

If the connection sequence is irrelevant, then it can be omitted, e.g., we may write $a \leadsto_e b$ instead of $a \overset{p}{\leadsto}_e b$.

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

Based on the variable connection relation, we define next the position-set of α-atoms in an LGQ formula, and the position-set of multi-sink variables in an LGQ formula.

Definition 3.7.2 (Position-set of α-atoms). *The position-set $pos^{\alpha}(e)$ of α-atoms in an LGQ formula e is the multiset of all lengths of connections from a non-sink variable and with its sequence ending with an α-atom (x is a possibly empty sequence of predicates):*

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

Example 3.7.1. Consider the formula $e = \mathsf{root}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge (\mathsf{child}(v_2, v_3) \vee \mathsf{child}(v_2, v_4))$ $\wedge\ \mathsf{child}^+(v_3, v_4)$. Then, e.g., $v_1 \overset{\mathsf{child}}{\leadsto}_e v_2$, $v_1 \overset{\mathsf{child.child}}{\leadsto}_e v_3$. The position-set of child-formulas is $pos^{\mathsf{child}}(e) = \{1, 2, 2\}$. □

Another important measure for LGQ formulas is their *size*.

Definition 3.7.3 (Size of LGQ Formulas). *The size $|e|$ of an LGQ formula e is the sum of 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.*

Reverse and DAG Factors of LGQ Formulas

We define next the positional and the type reverse factors of an LGQ formula, which are measures for the amount and position of reverse binary atoms of that formula.

Definition 3.7.4 (Reverse Position Factor of LGQ Formulas). *The reverse position factor $pos^{rev}(e)$ of an LGQ formula e is the union of positions-sets of all its reverse atoms:*

$$pos^{rev}(e) = \bigcup_{\alpha \in \mathsf{R}^?} (pos^{\alpha}(e)).$$

An LGQ formula can contain up to seven different types of reverse atoms, cf. Section 3.2. The amount and type of reverse atoms in a formula e is given by its reverse type factor $type^{rev}(e)$.

Definition 3.7.5 (Reverse Type Factor of LGQ Formulas). *The reverse type factor $type^{rev}(e)$ of an LGQ formula e containing br base reverse predicates, tcr transitive closure reverse predicates, and trcr reflexive transitive closure reverse predicates, is a multiset containing the number 1 br times, the number 2 tcr times, and number 3 trcr times.*

We define also the orders $>^{rev}_{pos}$ and $>^{rev}_{type}$ on LGQ formulas derived from the order $>_{mul}$ on multisets $\{pos^{rev}(e) \mid e \in LGQ\}$, respectively $\{type^{rev}(e) \mid e \in LGQ\}$:

$$s >^{rev}_{pos} t \Leftrightarrow pos^{rev}(s) >_{mul} pos^{rev}(t) \qquad s >^{rev}_{type} t \Leftrightarrow type^{rev}(s) >_{mul} type^{rev}(t).$$

For a given formula e, $type^{rev}(e) = \emptyset$ exactly when $pos^{rev}(e) = \emptyset$, and this means that e does not have reverse atoms at all.

Example 3.7.2. Consider the formulas $e = root(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge (\mathsf{par}(v_2, v_3) \wedge \mathsf{par}^+(v_3, v_4) \vee \mathsf{child}^+(v_2, v_3) \wedge \mathsf{self}(v_3, v_4)) \wedge \mathsf{par}^*(v_3, v_5)$ and $e' = root(v_2) \wedge \mathsf{par}^*(v_2, v_3) \wedge \mathsf{par}^*(v_2, v_4)$. The reverse factors are

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

□

We give in the following a measure for the amount of forward sink-arities of multi-sink variables in LGQ formulas. Recall that 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.

Definition 3.7.6 (DAG Type Factor of LGQ Formulas). *The dag type factor $type^{dag}(e)$ of an LGQ formula e is the multiset containing the forward sink-arity of each multi-sink variable in e.*

Example 3.7.3. Consider the formulas $e = root(v_1) \wedge \mathsf{child}(v_1, v_3) \wedge root(v_2) \wedge (\mathsf{child}^+(v_2, v_3) \vee \mathsf{child}^+(v_2, v_4) \wedge \mathsf{nextSibl}(v_4, v_3)) \wedge \mathsf{child}(v_3, v_5)$ that has a multi-sink variable v_3. The dag factor is the forward sink-arity of v_3, which is $type^{dag}(e) = \{2, 2\}$, because in each disjunct v_3 has a forward 2-arity.

□

As for reverse factors, we define also the order $>^{dag}_{type}$ on LGQ formulas derived from the order $>_{mul}$ on multisets $\{type^{dag}(e) \mid e \in LGQ\}$:

$$s >^{dag}_{type} t \Leftrightarrow type^{dag}(s) >_{mul} type^{dag}(t).$$

3.8 LGQ versus XPath

We introduce next the practical query language XPath and we show how it relates to LGQ.

3.8.1 XPath

The language for expressing node selection in tree considered in the following is the unabbreviated XPath fragment without functions, attribute handling, and value-based joins. This fragment extends Core XPath [71] with nodetests on text content and with some new axes and operators, as explained later.

Data Model

The data model of XPath considered here is the same as for LGQ, and given in Section 3.1. The binary predicates defined there are supported in XPath by means of binary relations called axes. For most built-in binary predicates, there is a corresponding XPath axis. XPath has six forward axes and five reverse built-in axes, cf. Figure 3.4. A forward/reverse axis relates a node to nodes that appear in the tree after/before in the document order. The axes of the following pairs are "symmetrical" of each other: parent – child, ancestor – descendant, descendant-or-self – ancestor-or-self, preceding – following, preceding-sibl – following-sibling, and self – self. For the binary predicates fstChild, nextSibl, $\mathsf{nextSibl}^*$, $\mathsf{prevSibl}^+$, and $\mathsf{prevSibl}^*$ there are no direct corresponding XPath axes. However, this section extends XPath with corresponding axes first-child, first-following-sibling, following-sibling-or-self, first-preceding-sibling, and preceding-sibling-or-self, as shown later.

LGQ Predicates	Corresponding XPath Construct
fstChild	first-child
child	child
child^+	descendant
child^*	descendant-or-self
nextSibl	first-following-sibling
$\mathsf{nextSibl}^+$	following-sibling
$\mathsf{nextSibl}^*$	following-sibling-or-self
foll	following
par	parent
par^+	ancestor
par^*	ancestor-or-self
prevSibl	first-preceding-sibling
$\mathsf{prevSibl}^+$	preceding-sibling
$\mathsf{prevSibl}^*$	preceding-sibling-or-self
prec	preceding
self	self .

Figure 3.4: Binary Predicates and corresponding XPath Constructs

The total function *pred*: *Predicate* $\rightarrow$ *Axis* is defined in Figure 3.4 and returns the corresponding XPath axis for a given binary predicate.

Syntax

An XPath query can be constructed following the productions of the grammar given below.

$$\begin{aligned}
path &::= path \mid path \mid /\, path \mid path\, /\, path \mid path\, [\, filter\,] \mid \\
&\quad forward_step \mid reverse_step \mid \top \mid \bot. \\
filter &::= filter\ \mathsf{and}\ filter \mid filter\ \mathsf{or}\ filter \mid \mathsf{not}(\, filter\,) \mid (\, filter\,) \mid path\,. \\
forward_step &::= forward_axis :: nodetest. \\
reverse_step &::= reverse_axis :: nodetest. \\
forward_axis &::= \mathsf{self} \mid \mathsf{child} \mid \mathsf{descendant} \mid \mathsf{descendant\text{-}or\text{-}self} \mid \mathsf{following\text{-}sibling} \mid \mathsf{following}\,. \\
reverse_axis &::= \mathsf{parent} \mid \mathsf{ancestor} \mid \mathsf{ancestor\text{-}or\text{-}self} \mid \mathsf{preceding\text{-}sibling} \mid \mathsf{preceding}\,.
\end{aligned}$$

Looking at an XPath query gives already an intuition for the ordered tree-like structure matched by the query and for the kind of nodes to select. Indeed, a query like descendant::a/child::b/preceding::c could be interpreted as a sequence of three navigations in a tree using the XPath axes descendant, child, and preceding. Considering a starting node, the descendants a-nodes are first reached, then their children b-nodes, and from the latter nodes, the set of their preceding c-nodes represents the result to the query.

Similar to LGQ queries, XPath queries can be also classified in XPath path, tree, and forest queries.

A *step* query is an expression *axis*::*nodetest*, where *axis* is either a forward or a reverse axis, and *nodetest* is a nodetest as defined in Section 3.1. A step is a "forward step", if its axis is a forward axis, or a "reverse step", if its axis is a reverse axis. For example, with the forward step descendant::a, one navigates from a node to its descendants a-nodes, with the reverse step ancestor::*, one navigates from a node to its ancestors.

The XPath steps are another syntax for LGQ formulas made out of one binary atom and one unary atom. The forward step descendant::a is expressed in LGQ as $\mathsf{child}^+(v_1, v_2) \land \mathsf{a}(v_2)$ and the ancestor step ancestor::* as $\mathsf{par}^+(v_1, v_2) \land *(v_2)$, or simpler as $\mathsf{par}^+(v_1, v_2)$.

A *path* query, called also a "location path", is a sequence of steps, like in /descendant::a/child::b. The previous path query selects all children b-nodes of a-nodes in the input tree. There are *absolute* and *relative* paths: an absolute path starts with a path constructor /, whereas a relative path does not. The intuition behind absolute and relative paths is that absolute paths start the navigation from the root node of the tree, whereas relative paths can be used to navigate also from other nodes. Note that the notion of absolute XPath queries is similar to the notion of absolute LGQ rules, as discussed in Section 3.2.

The XPath paths are another syntax for LGQ paths. The above XPath path is equivalent to the LGQ path $Q(v_2) \leftarrow \mathsf{root}(v_0) \land \mathsf{child}^+(v_0, v_1) \land \mathsf{a}(v_1) \land \mathsf{child}(v_1, v_2) \land \mathsf{b}(v_2)$.

A *filter* expression is defined recursively as a path, or an expression built up from paths and the connectives and, or, and not, together with parentheses. XPath filters are syntactically delimited by square brackets, and each step in a query can have none, one, or several such filters. Semantically, a filter conditions the selection of nodes. The query

/descendant::a[child::b], where [child::b] is a filter, selects from the input tree only those a-nodes that have at least one child b-node. The query /child::a[not(child::b)] selects from the input tree those a-nodes without b-nodes children. The more complex query /descendant::a[not(child::b[not(child::c)])] selects those a-nodes without b-nodes children that do not have c-nodes children.

The XPath queries with filters are another syntax for LGQ trees. The XPath query /descendant::a[child::b][child::c] is expressed in LGQ as $Q(v_1) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{a}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{b}(v_2) \wedge \mathsf{child}(v_1, v_3) \wedge \mathsf{c}(v_3)$.

Disjunctive queries are queries of the form $p_1 \mid \ldots \mid p_i \mid \ldots \mid p_k$, where for all $i = 1, \ldots, k$, p_i is a query and | is the set-union operator. For example, /descendant::a/child::* | /child::b selects all b-nodes children of the root and all children of every a-node from the input tree. With disjunctive paths, one can navigate from a node to select other nodes via several queries, thus the nodes selected by a disjunctive query is the union of the sets of nodes selected by each of the constituent query.

The XPath disjunctive queries are another syntax for LGQ forests. The above XPath disjunctive query can be expressed in LGQ as $Q(v_2) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{a}(v_1) \wedge \mathsf{child}(v_1, v_2) \vee \mathsf{root}(v_0) \wedge \mathsf{child}(v_0, v_2) \wedge \mathsf{b}(v_2)$.

The empty queries $\perp$ and $\top$ are the same as for LGQ. They are used as canonical equivalents to the XPath queries that select no nodes ($\perp$), or all nodes ($\top$) from any given tree. Thus, $\perp$ can be /parent::*, and $\top$ can be /descendant-or-self::*.

Other useful query constructs expressible in XPath

Other useful queries are expressible in XPath, although without dedicated syntactical constructs.

Universal quantification can be seen as a consequence of the existential quantification of filters and of allowing negation on such filters. For example, asking for nodes such that a filter ϕ holds for all its a-labeled children v can be encoded in XPath as [not(child::a[not(ϕ)])]. XPath supports, however, a restricted form of universal quantification: queries asking, e.g., for all nodes with a property ϕ_3 descendants of nodes with a property ϕ_1, such that between them there are only nodes with a property ϕ_2, can not be expressed in XPath.

Constructs if-then-else are expressible using unions of two paths with filters: if q then p_1 else p_2 can be expressed in XPath as [q]/p_1 | [not(q)]/p_2. Nested if-then-else constructs can then be also straightforwardly supported.

The logical implication $\rightarrow$ and equivalence $\leftrightarrow$ are also partially expressible in XPath: a filter $\mathsf{p}_1 \rightarrow \mathsf{p}_2$ is expressible as [not(p_1) or p_2], and a filter $\mathsf{p}_1 \leftrightarrow \mathsf{p}_2$ is expressible as [(p_1 and p_2) or (not(p_1) and not(p_2))]. Note that in both cases the nodes selected by p_1 or p_2 can not be answers, but they can rather condition the answers, because both paths are in filters.

Semantics

The semantics of XPath is given below by means of the two semantics functions $\mathcal{XQ}$ and $\mathcal{XF}$, inspired by [69]. Applied on an XPath query, the function $\mathcal{XQ}$ yields the set of pairs of source and answer nodes. Applied on an XPath filter, the function $\mathcal{XF}$ yields the set of nodes for which that filter is evaluated to true. Both functions are defined below using pattern matching on the structure of XPath queries, respectively filters.

$$
\begin{aligned}
&\mathcal{XQ} : \text{Tree} \times \text{Query} \rightarrow \text{Set}((\text{Node}, \text{Node}))\\
&\qquad \mathcal{XQ}_T[\![/p]\!] = Nodes(T) \times \{y \mid (x, y) \in \mathcal{XQ}_T[\![p]\!], test(x, \mathsf{root})\}\\
&\qquad \mathcal{XQ}_T[\![p_1 \mid p_2]\!] = \mathcal{XQ}_T[\![p_1]\!] \cup \mathcal{XQ}_T[\![p_2]\!]\\
&\qquad \mathcal{XQ}_T[\![p_1 - p_2]\!] = \mathcal{XQ}_T[\![p_1]\!] - \mathcal{XQ}_T[\![p_2]\!]\\
&\qquad \mathcal{XQ}_T[\![p_1 \texttt{==} p_2]\!] = \mathcal{XQ}_T[\![p_1]\!] \cap \mathcal{XQ}_T[\![p_2]\!]\\
&\qquad \mathcal{XQ}_T[\![p_1/p_2]\!] = \{(x, z) \mid (x, y) \in \mathcal{XQ}_T[\![p_1]\!], (y, z) \in \mathcal{XQ}_T[\![p_1]\!]\}\\
&\qquad \mathcal{XQ}_T[\![p_1[p_2]]\!] = \{(x, y) \mid (x, y) \in \mathcal{XQ}_T[\![p_1]\!], y \in \mathcal{XF}_T[\![p_2]\!]\}\\
&\qquad \mathcal{XQ}_T[\![(p)]\!] = \mathcal{XQ}_T[\![p]\!]\\
&\qquad \mathcal{XQ}_T[\![\alpha{::}\eta]\!] = \{(x, y) \mid (x, y) \in pred^{-1}(\alpha), test(y, \eta)\}\\
&\mathcal{XF} : \text{Tree} \times \text{Query} \rightarrow \text{Node}\\
&\qquad \mathcal{XF}_T[\![p_1 \text{ or } p_2]\!] = \mathcal{XF}_T[\![p_1]\!] \cup \mathcal{XF}_T[\![p_1]\!]\\
&\qquad \mathcal{XF}_T[\![p_1 \text{ and } p_2]\!] = \mathcal{XF}_T[\![p_1]\!] \cap \mathcal{XF}_T[\![p_1]\!]\\
&\qquad \mathcal{XF}_T[\![not(p)]\!] = Nodes(T) - \mathcal{XF}_T[\![p]\!]\\
&\qquad \mathcal{XF}_T[\![p]\!] = \{x_0 \mid \exists x : (x_0, x) \in \mathcal{XQ}_T[\![p]\!]\}
\end{aligned}
$$

The answer to an XPath query p is $\{y \mid \exists x : (x, y) \in \mathcal{XQ}_T[\![p]\!]\}$.

Extensions considered in this thesis

There are two XPath extensions that are considered throughout this thesis:, (1) the new axes first-child, first-following-sibling and first-preceding-sibling, and (2) the set difference operator.

The new axes first-child, first-following-sibling, following-sibling-or-self, first-preceding-sibling, and preceding-sibling-or-self express the selection from a given node n of (1) the first child of n, (2) the first sibling that immediately follows n in document order, (3) the first sibling that immediately precedes n in document order. All axes can be obtained using the XPath 1.0 positional filter [position()=1]. Applied to a set of nodes, this filter retains only the first node in document order (for the forward steps), or in reverse document order

(for the reverse steps). The axes are expressed as

first-child::*	=	child::*[position()=1]
first-following-sibling::*	=	following-sibling::*[position()=1]
following-sibling-or-self::*	=	following-sibling::* | self::*
first-preceding-sibling::*	=	preceding-sibling::*[position()=1]
preceding-sibling-or-self::*	=	preceding-sibling::* | self::*

The set difference operator $-$, similar to the `except` operator in XPath 2.0 [22], expresses the difference between two sets of nodes, as selected by two XPath queries: the answer to the query $p_1 - p_2$ consists in those nodes selected by the path p_1 and not by the path p_2, cf. XPath semantics defined above. Note that the XPath negation can be fully expressed using the set-difference operator: the query $p_1[\mathsf{not}(p_2)]$ is expressible as $p_1 - p_1[p_2]$. In contrast, the set difference operator can not be always expressed using XPath negation (see later Example 3.8.2).

Although not explicitly considered in this work, another interesting extension is represented by the filter with identity-based node equality `==` . For two XPath expressions p_1 and p_2, the filter p_1 `==` p_2 holds if there is a node selected by p_1 which is *identical* to a node selected by p_2. Note that the XPath fragment considered here extended with this filter is can express LGQ single-join DAG queries and a limited form of LGQ graph queries.

The equality `==` corresponds to built-in node equality operator (`==`) in XPath 2.0 and XQuery 1.0, but it can also be used for comparing node sets similar to general comparisons in XPath 2.0. XPath 1.0 has built-in support only for equality based on node values. The only implicit node-identity test ensured by XPath can be specified using the set union operator `|` , because the set of nodes specified by p_1 `|` p_2 consists in the nodes appearing at least in one set specified by p_1 or p_2. Then, the filter p_1 `==` p_2 can be expressed using the XPath 1.0 expression `count(`p_1 `|` p_2`) < count(`p_1`) + count(`p_2`)`, where $count(p)$ gives the number of nodes in the set specified by p.

Expressiveness

The XPath fragment defined by the grammar given above is a logical core of XPath 1.0 that extends the Core XPath of [69]. It is expressible in first-order logic when interpreted on ordered trees, and in Datalog with stratified negation [69]. XPath can be also extended to match the expressiveness of first-order logic when interpreted on ordered trees. In this sense, the above extension of positive (i.e., without negation) Core XPath with the set difference operator becomes first-order complete. In the same sense, [109, 108] propose CXPath, a first-order complete extension of XPath with conditional axis relations that allow queries of the kind "do a certain step (like child, descendant), while a condition is satisfied at the resulting node". It is also illuminating to think of Core XPath as a simple temporal logic [68], whereas CXPath extends XPath with (counterparts of) the *since* and *until* operators.

3.8.2 Conciseness of LGQ over XPath

Dealing with XPath syntax is sometimes cumbersome. Especially the explicit notation for filters using squared brackets rises various technical problems when translating XPath expressions to other query formalisms (or vice-versa), or when doing induction on the structure of XPath expressions.

Arguably, the Datalog-like syntax of LGQ is intuitive. Moreover, it allows more freedom than XPath in writing concise queries with help of variables. Also, the syntax sugaring of XPath permits to write a bunch of queries quite differently, although they all impose rather similar constraints on the trees to be queried. For example, the XPath queries

/descendant::man[child::man[child::woman]][child::woman],
/descendant::man[child::man[child::woman] and child::woman],
/descendant::man[child::man/child::woman and child::woman]

select the same set of nodes representing men having daughters and sons that have also daughters. They differ only in some syntactical sugaring for expressing filters and are equivalent between them and also to the LGQ query

$$Q(v_1) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{man}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{man}(v_2) \\ \wedge \mathsf{child}(v_2, v_3) \wedge \mathsf{woman}(v_3) \wedge \mathsf{child}(v_1, v_4) \wedge \mathsf{woman}(v_4).$$

Note that in the above LGQ query there is no explicit notation for filters *à* la XPath, but only the distinguished variable v_1 is written explicitly in the head.

No explicit notation for XPath filters leads to an even greater advantage of the LGQ syntax. Consider now the XPath queries

/descendant::man[child::man[child::woman]]/child::woman,
/descendant::man[child::woman]/child::man[child::woman],
/descendant::man[child::woman]/child::man/child::woman

They are not equivalent though their structure is the same, and only the answer nodes are selected by another step. Because they have the same structure, LGQ queries equivalent to them have the same body as before, but only different distinguished variables: in the first case the distinguished variable is v_4 (and selects daughters of men with sons having daughters), in the second v_2 (and selects sons of men such that both have daughters), and in the third v_3 (and selects daughters of men having sisters and fathers).

LGQ allows also graph queries that are not expressible directly in XPath, though the results of Chapter 4 ensure the existence of equivalent XPath queries. consider the LGQ DAG query that selects for each man all their female descendants that follow at least one of their male descendants

$$G(v_3) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{man}(v_1) \wedge \mathsf{child}^+(v_1, v_2) \wedge \mathsf{man}(v_2) \wedge \mathsf{child}^+(v_1, v_3) \\ \wedge \mathsf{woman}(v_3) \wedge \mathsf{foll}(v_1, v_3).$$

Note that v_3 is a multi-sink variable in G. XPath can simulate the use of multi-source variables in LGQ by means of filters. However, XPath can not simulate the use of multi-sink variables, as needed in the above LGQ query G. There exists, however, the equivalent XPath query

/descendant::man/descendant::man/following-sibling::*/descendant-or-self::woman |
/descendant::man/descendant::*[descendant::man]/following-sibling::*/descendant-or-self::woman.

The rationale behind the existence of equivalent XPath queries to any LGQ query is twofold: (1) XPath queries are as expressible as LGQ forests, and (2) LGQ graphs are as expressible as LGQ forests. The latter assertion is discussed in the next chapter, whereas the former is discussed next.

3.8.3 XPath=LGQ Forests

This section shows that XPath queries are equivalent to LGQ forest queries. After discussing the relation between XPath negation and LGQ negation, we give the encodings of XPath into LGQ forests and vice versa. Chapter 4 shows further that any LGQ query can be reduced to forward LGQ forest queries, thus making XPath as expressive as LGQ.

LGQ Negation versus XPath Negation

The XPath negation can be expressed in LGQ.

Example 3.8.1. Consider the XPath query /descendant::*[not(child::man)]/child::woman selecting the daughters of all persons that have no sons. For the tree of Figure 2.2 depicting an excerpt of the genealogical tree of John II the Good, the result is the person Louis II of Naples. The same query can be expressed also in LGQ as

$$\begin{aligned} Q(v) &\leftarrow \mathsf{root}(v_0) \land \mathsf{child}^+(v_0, v_1) \land \neg Q'(v_1) \land \mathsf{child}(v_1, v) \land \mathsf{woman}(v), \\ Q'(v) &\leftarrow \mathsf{root}(v_0) \land \mathsf{child}^*(v_0, v) \land \mathsf{child}(v, v_1) \land \mathsf{man}(v_1). \end{aligned}$$

The query Q specifies the selection of woman children of persons that are not selected also by Q'. In turn, Q' selects all persons that has at least one son. □

In the previous example, the negative part of the XPath query is expressed in LGQ using a separate rule Q' that is a counterpart of the XPath negated filter together with the atoms $\mathsf{root}(v_0)$ and $\mathsf{child}^*(v_0, v)$ that ensure Q' absolute and connected. Note that the added atoms do not constrain the bindings to v, because a query like $Q''(v) \leftarrow \mathsf{root}(v_0) \land \mathsf{child}^*(v_0, v)$ selects all nodes from the input tree. This general scheme is used for encoding XPath queries with negation into LGQ.

The negation is used in XPath also for supporting universal quantification. However, this support is limited, and renders the XPath negation weaker than the LGQ negation, as shown in the next example.

Example 3.8.2. Consider again the genealogical tree of Figure 2.2 and the query Q selecting all men that have only male ancestors. After the Salic law, these men are pretendants to the throne of France. The result of Q consists in all men in that tree.

$$\begin{aligned} Q(v) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{man}(v) \wedge \neg Q'(v), \\ Q'(v) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^*(v_0, v_1) \wedge \mathsf{woman}(v_1) \wedge \mathsf{child}^+(v_1, v). \end{aligned}$$

The positive part of Q selects all men, whereas the subquery Q' selects all persons (including) descendants of at least one female person. Therefore, the query Q selects all men having only male ancestors. This query is not expressible in Core XPath [109, 108], but in XPath extended with the set difference operator:

/descendant::man - /descendant::woman/descendant::man.

□

Encoding XPath into LGQ Forests

The function $\overrightarrow{\mathcal{XL}}$ gives the encoding of XPath into LGQ forests: it takes as parameters an XPath construct and an LGQ variable, called the working variable, and produces a pair (v, f) consisting of the new working variable v and the LGQ formula f corresponding to that XPath construct.

$$\begin{aligned}
\overrightarrow{\mathcal{XL}}[\![/p]\!](_) &= (v, \mathsf{root}(v_0) \wedge f) : (v, f) = \overrightarrow{\mathcal{XL}}[\![p]\!](v_0), v_0 = \mathsf{fresh_var()} \\
\overrightarrow{\mathcal{XL}}[\![\alpha{::}\eta]\!](v) &= (v_1, pred^{-1}(\alpha)(v, v_1) \wedge \eta(v_1)) : v_1 = \mathsf{fresh_var()} \\
\overrightarrow{\mathcal{XL}}[\![p_1[p_2]]\!](v) &= (v_1, f_1 \wedge f_2) : (v_1, f_1) = \overrightarrow{\mathcal{XL}}[\![p_1]\!](v), (v_2, f_2) = \overrightarrow{\mathcal{XL}}[\![p_2]\!](v_1) \\
\overrightarrow{\mathcal{XL}}[\![p_1/p_2]\!](v) &= (v_2, f_1 \wedge f_2) : (v_1, f_1) = \overrightarrow{\mathcal{XL}}[\![p_1]\!](v), (v_2, f_2) = \overrightarrow{\mathcal{XL}}[\![p_2]\!](v_1) \\
\overrightarrow{\mathcal{XL}}[\![p_1 \mid p_2]\!](v) &= (v_1, f_1 \vee f_2) : (v_1, f_1) = \overrightarrow{\mathcal{XL}}[\![p_1]\!](v), (v_1, f_2) = \overrightarrow{\mathcal{XL}}[\![p_2]\!](v) \\
\overrightarrow{\mathcal{XL}}[\![p_1 \text{ or } p_2]\!](v) &= (v, f_1 \vee f_2) : \ (v_1, f_1) = \overrightarrow{\mathcal{XL}}[\![p_1]\!](v), (v_2, f_2) = \overrightarrow{\mathcal{XL}}[\![p_2]\!](v) \\
\overrightarrow{\mathcal{XL}}[\![p_1 \text{ and } p_2]\!](v) &= (v, f_1 \wedge f_2) : \ (v_1, f_1) = \overrightarrow{\mathcal{XL}}[\![p_1]\!](v), (v_2, f_2) = \overrightarrow{\mathcal{XL}}[\![p_2]\!](v) \\
\overrightarrow{\mathcal{XL}}[\![not(p)]\!](v) &= (v, \neg Q(v)) : \ (v_1, f) = \overrightarrow{\mathcal{XL}}[\![p]\!](v), \\
&\qquad Q(v) \ \leftarrow \ \mathsf{root}(v_0) \wedge \mathsf{child}^*(v_0, v) \wedge f, Q = \mathsf{fresh_id()} \\
\overrightarrow{\mathcal{XL}}[\![p_1 - p_2]\!](v) &= (v_1, f_1 \wedge \neg Q(v_1)) : (v_1, f_1) = \overrightarrow{\mathcal{XL}}[\![p_1]\!](v), (v_1, f_2) = \overrightarrow{\mathcal{XL}}[\![p_2]\!](v), \\
&\qquad Q(v_1) \ \leftarrow \ \mathsf{root}(v_0) \wedge \mathsf{child}^*(v_0, v) \wedge f_2, Q = \mathsf{fresh_id()} \\
\overrightarrow{\mathcal{XL}}[\![(p)]\!](v) &= \overrightarrow{\mathcal{XL}}[\![p]\!](v).
\end{aligned}$$

Finally, the encoding of an absolute XPath query $/p$ is the LGQ rule $Q(v) \ \leftarrow \ f$, where $(v, f) = \overrightarrow{\mathcal{XL}}[\![/p]\!](_)$.

The function $\overrightarrow{\mathcal{XL}}$ is defined using pattern matching on the structure of XPath queries, which are restricted syntactically to be absolute (i.e., with the leading /) and without absolute paths in filters[2].

XPath does not have variables. Given two XPath location steps $\alpha_1{::}\eta_1$ and $\alpha_2{::}\eta_2$, one can construct either (1) the path $\alpha_1{::}\eta_1/\alpha_2{::}\eta_2$ by using the path constructor /, or (2) the step with filter $\alpha_1{::}\eta_1[\alpha_2{::}\eta_2]$ by using the filter constructor [ ].

LGQ has variables. The above XPath path and filter expressions can be simply encoded in LGQ by annotating with variables the positions before and after each XPath syntactical construct. For example, the path $\alpha_1{::}\,\eta_1/\alpha_2{::}\,\eta_2$ becomes ${}^{v_1}\alpha_1^{v_2}{::}\,{}^{v_3}\eta_1^{v_4}/{}^{v_5}\alpha_2^{v_5}{::}\,{}^{v_6}\eta_2^{v_7}$. The XPath constructs ::, /, [ ] ensure that the same variable must appear before and after them, thus obtaining ${}^{v_1}\alpha_1^{v_2}{::}\,{}^{v_2}\eta_1^{v_4}/{}^{v_4}\alpha_2^{v_5}{::}\,{}^{v_5}\eta_2^{v_7}$. By considering further that each XPath axis α has a corresponding extensional LGQ binary predicate $pred^{-1}(\alpha)$ (cf. Figure 3.4), and that each XPath nodetest has a corresponding LGQ unary predicate, we finally obtain ${}^{v_1}\alpha_1^{v_2}{::}\,{}^{v_2}\eta_1^{v_2}/{}^{v_2}\alpha_2^{v_5}{::}\,{}^{v_5}\eta_2^{v_5}$, or as an LGQ formula $pred^{-1}(\alpha_1)(v_1, v_2) \wedge \eta_1(v_2) \wedge pred^{-1}(\alpha_2)(v_2, v_5) \wedge \eta_2(v_5)$. The case of XPath expressions with filters is similar.

We explain now some particularities of the encoding of or-filters and unions. The other XPath constructs are encoded similarly. The encoding of an or-filter of two XPath expressions (which can be on their turn also filters) is a disjunction of the encodings of each expression with the same working variable. The encoding of a union of two XPath expressions (which can be also unions) is similar to that of an or-filter, except that the new working variables obtained from the encodings of both XPath expressions are the same.

In the definition of the encoding of XPath into LGQ, the LGQ variables and the names for the LGQ intensional predicates are created fresh using the functions fresh_var(), respectively fresh_id(). For a given LGQ formula or query, a *fresh* variable (predicate name) is a new variable (predicate name) that does not appear already in that formula or query.

Note that such an encoding of XPath does not create multi-sink variables, i.e., variables that appear more than one time at the second position in binary atoms. In effect, for any XPath query, the function $\overrightarrow{\mathcal{XL}}$ creates an LGQ forest.

Example 3.8.3. Consider the XPath queries

$$\begin{aligned} p_1 &= /\mathsf{descendant::man[child::man]/child::woman} \\ p_2 &= /\mathsf{descendant::man[child::woman]/child::man} \\ p_3 &= /\mathsf{descendant::man[child::woman\ and\ child::man]}. \end{aligned}$$

that select (1) all daughters of men having also sons, (2) all sons of men having also daughters, and (3) all men having daughters and sons. From the genealogical tree of Figure 2.2, the queries select (1) the (nodes corresponding to the) persons Isabelle and Anna, (2) the persons Charles V, Charles, Francis, and Louis I de Valois, and (3) the persons John II the Good and Charles VIII. The LGQ-encodings of them are three LGQ queries $Q_1(v_3) \leftarrow e$ and $Q_2(v_2) \leftarrow e$, and $Q_3(v_1) \leftarrow e$, where their body e is

$$\mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{man}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{man}(v_2) \wedge \mathsf{child}(v_1, v_3) \wedge \mathsf{woman}(v_3).$$

[2]XPath queries with absolute paths in filters can be rewritten to equivalent queries without such filters.

We show next how p_1 can be encoded in e bottom-up.

$$(v_1, f_1) = \overrightarrow{\mathcal{XL}}[\![\mathsf{descendant::man}]\!](v_0, \mathsf{root}(v_0)) = (v_1, \mathsf{root}(v_0) \land \mathsf{child}^+(v_0, v_1) \land \mathsf{man}(v_1))$$
$$(v_2, f_2) = \overrightarrow{\mathcal{XL}}[\![\mathsf{child::man}]\!](v_1, f_1) = (v_2, f_1 \land \mathsf{child}(v_1, v_2) \land \mathsf{man}(v_2))$$
$$(v_3, f_3) = \overrightarrow{\mathcal{XL}}[\![\mathsf{descendant::man[child::man]}]\!](v_0, \mathsf{root}(v_0)) = (v_1, f_2)$$
$$(v_4, f_4) = \overrightarrow{\mathcal{XL}}[\![\mathsf{child::woman}]\!](v_3, f_3) = (v_3, f_3 \land \mathsf{child}(v_1, v_3) \land \mathsf{woman}(v_3))$$
$$(v_5, f_5) = \overrightarrow{\mathcal{XL}}[\![\mathsf{descendant::man[child::man]/child::woman}]\!](v_0, \mathsf{root}(v_0)) = (v_4, f_4).$$

The final encoding of p_1 is $Q_1(v_5) \leftarrow f_5$, where $v_5 = v_3$ and $f_5 = e$. □

Encoding LGQ Forests into XPath

LGQ is not more expressive than XPath. However, there is no straightforward encoding of the entire LGQ into XPath. Chapter 4 elaborates on this non-trivial encoding. We give here only an encoding of LGQ forests into XPath using two functions $\mathcal{X}$ and $\overrightarrow{\mathcal{LX}}$ and some simplifications rules. This encoding is more involved than for XPath into LGQ, and this is the price to pay for the explicit notation of filters in XPath.

The LGQ forest queries are brought first in disjunctive normal form and without $\top$ and $\perp$ atoms in disjuncts. Chapter 4 gives later rewriting rules that yield queries in this form. The function $\mathcal{X}$ encodes such queries into unions of XPath absolute queries and uses the function $\overrightarrow{\mathcal{LX}}$ for encoding the body of each LGQ rule into an XPath union term. The function $\overrightarrow{\mathcal{LX}}$ takes as parameters the head variable v and the body b of the current rule, the working formula to encode and the working variable, and produces an XPath expression.

$$\mathcal{X}[\![Q(v) \leftarrow f_1 \lor f_2]\!] = \mathcal{X}[\![Q(v) \leftarrow f_1]\!] \mid \mathcal{X}[\![Q(v) \leftarrow f_2]\!]$$
$$\mathcal{X}[\![Q(v) \leftarrow \mathsf{root}(v_0) \land b]\!] = /\overrightarrow{\mathcal{LX}}_{v,b}[\![b \land \top]\!](v_0)$$

$$\overrightarrow{\mathcal{LX}}_{v,b}[\![\alpha(x,y) \land f]\!](x) = \begin{cases} [\mathit{left}_x]/\mathit{step}\ \mathit{left}_y & , y \rightsquigarrow_b v \text{ or } y = v \\ [\mathit{step}\ \mathit{left}_y]\ \mathit{left}_x & , y \not\rightsquigarrow_b v \text{ and } y \neq v \end{cases},$$
$$\mathit{left}_x = \overrightarrow{\mathcal{LX}}_{v,b}[\![f]\!](x), \mathit{step} = \mathit{pred}^{-1}(\alpha)::*, \mathit{left}_y = \overrightarrow{\mathcal{LX}}_{v,b}[\![b]\!](y)$$
$$\overrightarrow{\mathcal{LX}}_{v,b}[\![\eta(x) \land f]\!](x) = [\mathsf{self}::\eta]\ \overrightarrow{\mathcal{LX}}_{v,b}[\![f]\!](x)$$
$$\overrightarrow{\mathcal{LX}}_{v,b}[\![\neg Q(x) \land f]\!](x) = [\mathsf{self}::* \text{ - } \mathcal{X}[\![\mathit{clause}(Q)]\!]]\ \overrightarrow{\mathcal{LX}}_{v,b}[\![f]\!](x)$$
$$\overrightarrow{\mathcal{LX}}_{v,b}[\![\eta(y) \land f]\!](x) = \overrightarrow{\mathcal{LX}}_{v,b}[\![\alpha(y,z) \land f]\!](x) = \overrightarrow{\mathcal{LX}}_{v,b}[\![\neg Q(y) \land f]\!](x) = \overrightarrow{\mathcal{LX}}_{v,b}[\![f]\!](x)$$
$$\overrightarrow{\mathcal{LX}}_{v,b}[\![\top]\!](x) = [\mathsf{self}::*].$$

The function $\overrightarrow{\mathcal{LX}}$ is defined using pattern matching on the structure of the working formula ϕ to encode and on the working variable x. If ϕ starts with an atom that does not have x as source, then that atom is skipped at this encoding stage, and the encoding continues with the rest of ϕ in the same way until an atom with x as source is encountered. If ϕ is $\top$, i.e., it is exhausted and no atoms with x as source are found, then the filter [self:: $*$] is added to the generated query. Note that this filter, like the LGQ atom $\top$, does not add further constraints to the answers and can be safely removed afterwards.

If $\phi = \eta(x) \wedge f$, i.e., ϕ starts with the unary atom $\eta(x)$, then ϕ is encoded as an XPath path where the first step self::η encodes that unary atom, and the rest of the path remains to encode f, where x is the working variable.

If $\phi = \alpha(x, y) \wedge f$, i.e., ϕ starts with the binary atom $\alpha(x, y)$, then the encoding of ϕ depends on whether the head variable v is reachable in b from y. Let left_y be the XPath expression representing the encoding of the subformula of b containing the atoms reachable from x via y, and left_x the encoding of the subformula of b containing atoms reachable from x via other variables than y, where in both cases x is the working variable. Their encodings are detailed below. In case v is reachable in b from y, then left_y contains necessarily the path leading to the answers, and is written in XPath outside filters. Also in this case, left_x is a filter, because there can not be another atom $\alpha'(x, y')$ with v reachable from $y' \neq y$ in b (b is a tree formula). In the other case, left_y becomes a filter and left_x can contain the path leading to the answers.

The expressions $\mathit{step}\ \mathit{left}_y$ and left_x are generated as follows. The former expression consists in the step $\mathit{step} = \mathit{pred}^{-1}(\alpha)::*$ representing the encoding of $\alpha(x, y)$, followed by the expression left_y representing the encoding of the subformula of b containing atoms that have y as source variable (thus y becomes now the working variable). The latter expression left_x is the encoding of the working formula without $\alpha(x, y)$, where x is still the working variable. Note that each atom is considered exactly once, because b is a tree formula.

If $\phi = \neg Q(x) \wedge f$, i.e., ϕ starts with the unary atom $\neg Q(x)$, then ϕ is encoded as an XPath expression consisting in a filter that encodes that atom, and an expression left_x that encodes f, in both cases with x as the working variable. The filter represents the difference between the expression generated until now and the encoding of the rule Q, done by the function $\mathcal{X}$. The encoding left_x is done like in the above cases.

Simplifications. The encoding of LGQ forests into XPath using $\mathcal{X}$ and $\overrightarrow{\mathcal{LX}}$ generates as many filters as binary predicates that are not on the connection sequence from the non-sink variable to the head variable. Also, each atom with a built-in predicate is encoded into one distinct XPath step, although XPath steps comprise both a unary and a binary extensional predicate. The following simple rewritings can be applied to the encoding of LGQ forests in order to simplify them (p stands for an XPath expression):

$$p[\mathsf{self}::*] \rightarrow p \qquad\qquad \alpha::*[\mathsf{self}::\eta] \rightarrow \alpha::\eta.$$

Example 3.8.4. Consider the query Q_1 of Example 3.8.3 that selects all daughters of men having also sons

$Q_1(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{man}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{man}(v_2) \wedge \mathsf{child}(v_1, v) \wedge \mathsf{woman}(v)$

$Q_1(v) \leftarrow b$ is encoded into XPath as follows. We compute and label first some expressions:

$$e_1 = \overrightarrow{\mathcal{LX}}_{v,b}[\![\mathsf{man}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{man}(v_2) \wedge \mathsf{child}(v_1, v) \wedge \mathsf{woman}(v)]\!](v_0) = \mathsf{self::*}$$

$$e_2 = \overrightarrow{\mathcal{LX}}_{v,b}[\![\mathsf{man}(v_2) \wedge \mathsf{child}(v_1, v) \wedge \mathsf{woman}(v)]\!](v_2) = \mathsf{[self::man][self::*]}$$

$$e_3 = \overrightarrow{\mathcal{LX}}_{v,b}[\![\mathsf{man}(v_2) \wedge \mathsf{child}(v_1, v) \wedge \mathsf{woman}(v)]\!](v_1) = \overrightarrow{\mathcal{LX}}_{v,b}[\![\mathsf{child}(v_1, v) \wedge \mathsf{woman}(v)]\!](v_1)$$

$$= [\overrightarrow{\mathcal{LX}}_{v,b}[\![\mathsf{woman}(v)]\!](v_1)]/\mathsf{child::*}\ \overrightarrow{\mathcal{LX}}_{v,b}[\![\mathsf{woman}(v)]\!](v) = \mathsf{[self::*]/child::*[self::woman]}.$$

Then, the encoding of Q_1 is

$$\mathcal{X}[\![clause(Q_1)]\!]$$

$$= /\overrightarrow{\mathcal{LX}}_{v,b}[\![\mathsf{child}^+(v_0, v_1) \wedge \mathsf{man}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{man}(v_2) \wedge \mathsf{child}(v_1, v) \wedge \mathsf{woman}(v)]\!](v_0)$$

$$= /[e_1]/\mathsf{descendant::*}\ \overrightarrow{\mathcal{LX}}_{v,b}[\![b]\!](v_1) = /\mathsf{[self::*]/descendant::*}\ \overrightarrow{\mathcal{LX}}_{v,b}[\![b]\!](v_1)$$

$$= /\mathsf{[self::*]/descendant::*[self::man]}\ \overrightarrow{\mathcal{LX}}_{v,b}[\![\mathsf{child}(v_1, v_2) \wedge \mathsf{man}(v_2) \wedge \mathsf{child}(v_1, v) \wedge \mathsf{woman}(v)]\!](v_1)$$

$$= /\mathsf{[self::*]/descendant::*[self::man][child::*}\ e_2]\ e_3$$

$$= /\mathsf{[self::*]/descendant::*[self::man][child::*[self::man][self::*]][self::*]/child::*[self::woman]}.$$

This XPath query can be further simplified to /descendant::man[child::man]/child::woman.

□

XPath=LGQ Forests

The encodings of XPath into LGQ forests and vice-versa are correct, as ensured by the following lemma. As a corollary, it follows that LGQ forest queries are as expressive as XPath queries.

Lemma 3.8.1 (Correctness of $\overrightarrow{\mathcal{XL}}$ and $\overrightarrow{\mathcal{LX}}$ encodings). *The following holds:*

1. *Given any XPath query p and tree T, the semantics of p is the semantics of the LGQ formula f representing the encoding of p using $\overrightarrow{\mathcal{XL}}$:*

$$\pi_{v,v_1}(\mathcal{LF}_T[\![f]\!](\beta)) = \mathcal{XQ}[\![p]\!],\ \text{where } \beta = subst(f, T), \overrightarrow{\mathcal{XL}}[\![p]\!](v) = (v_1, f).$$

2. *Given any LGQ forest $Q(v) \leftarrow f$ and tree T, the semantics of Q is the semantics of the XPath query p representing the encoding of Q using $\mathcal{X}$:*

$$\pi_{v,v_1}(\mathcal{LF}_T[\![f]\!](\beta)) = \mathcal{XQ}[\![p]\!],\ \text{where } \beta = subst(f, T), p = \mathcal{X}[\![Q(v) \leftarrow f]\!].$$

Proof. The proof is given in the Appendix. □

Chapter 4

Source-to-source Query Transformation: From LGQ to Forward LGQ

The language of logical graph queries LGQ, as well as XPath, allows the specification of structural constraints for the nodes to be selected by means of binary predicates between nodes in trees. These structural constraints can be intuitively seen as "navigations" in trees, and are enabled by a large number of LGQ "navigational" predicates: seven forward predicates ($\mathsf{self}, \mathsf{fstChild}, \mathsf{child}, \mathsf{child}^+, \mathsf{child}^*, \mathsf{nextSibl}^+, \mathsf{foll}$) and five reverse predicates ($\mathsf{par}, \mathsf{par}^+, \mathsf{par}^*, \mathsf{prevSibl}, \mathsf{prec}$). The number as well as the relevance of these navigational predicates for querying XML has been challenged in [55, 23, 96].

The random access to XML data that is enabled by the various LGQ predicates (corresponding to navigational axes of XPath) has proven particularly difficult for an efficient query evaluation against XML streams, where only one-pass over the stream is affordable (or possible). Processing of XML has seen the widespread use of the W3C document object model (DOM) [145], where a main-memory representation of the entire XML data is used. As DOM has been developed with focus on document processing in user agents (e.g., browsers), this approach has several shortcomings for other application areas.

First, a considerable amount of XML applications, in particular data-centric applications, handle XML documents too large to be processed in main memory. Such XML documents are often encountered in natural language processing [92], in biology [28] and astronomy [119]. This aspect is exacerbated by expensive main-memory representations of XML documents. E.g., DOM-like main-memory structures for XML documents tend to be four-five times larger than the original XML document [91].

Second, the need for progressive processing (also referred to as sequential processing) of XML has emerged: Stream-based processing generating partial results as soon as they are available gives rise to a more efficient evaluation in certain contexts, e.g.,:

- 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 [37, 7]. 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 [94].

- 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 taking the output of the preceding processor as input, e.g., Apache Cocoon [10].

- For progressive rendering of large XML documents, e.g., by means of XSL(T), cf. Requirement 19 of [96]. There have been attempts to solve this problem [11].

There is a great interest in the identification of a subset of XPath (and of LGQ) that allows efficient streamed and progressive processing, cf. [55] and Requirement 19 of [96].

For stream-based processing of XML data, the Simple API for XML (SAX) [110] has been specified that allows sequential access to the content of XML documents with low memory footprint. Of particular concern for progressive SAX-like processing are the LGQ reverse predicates (corresponding to reverse axes of XPath), i.e., those predicates (e.g., par, prec) that contain pairs of source nodes and sink nodes occurring *before* these source nodes in document order. A restriction to forward predicates, i.e., those predicates that contain pairs of source nodes and sink nodes appearing *after* these source nodes, is a straightforward consideration for an efficient stream-based evaluation of XPath-like queries [55].

There are three principal options how to evaluate an LGQ query with reverse predicates in a stream-based context:

- Storing in memory sufficient information that allows to access past events particularly when evaluating a reverse predicate. This amounts to keeping in memory a (possibly pruned) DOM representation of the data [11].

- Evaluating queries in more than one pass over the stream, provided several passes are affordable/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.

- Rewriting queries with reverse predicates into equivalent ones without those reverse predicates.

In this chapter, we target the last approach and we show it to be always possible. It is less time consuming than the second approach and does not require the in-memory storage of fragments of the input as the first approach does.

We accomplish this goal by making use of the theory of term rewriting systems. We define first equivalence-preserving rewrite rules for LGQ formulas, and use them in

three distinct term rewriting systems. We show that all rewriting systems enjoy important properties like soundness and completeness, termination, confluence, and the existence and uniqueness of normal forms modulo the equational theory AC (associativity-commutativity) for predicates $\wedge$, $\vee$, and self. Using these systems, queries of various LGQ fragments can be rewritten into forward queries within the same or smaller fragments, and with complexities varying from linear to exponential in the size of the input queries.

The first term rewriting system (TRS_1) rewrites any LGQ single-join DAG into a forward LGQ single-join DAG, and any LGQ graph into a forward LGQ graph. The complexities of TRS_1 are linear for time and logarithmic for space, and the size of the output query is bounded in the size of the input query.

The second term rewriting system (TRS_2) rewrites any LGQ forest into a forward LGQ forest, any LGQ single-join DAG into a forward LGQ single-join DAG, and any LGQ graph into a forward LGQ graph. For arbitrary queries, the complexities are exponential for time and space, and the size of the output can be exponential in the number of reverse predicates in the input (i.e., the size of the reverse type factor of the input). It is shown that, in general, LGQ forests can not be rewritten into forward LGQ forests that have the size smaller than exponential (worst case). However, for queries without closure forward predicates appearing before closure reverse predicates, both either vertical or horizontal, along a connection sequence, the complexities of rewriting are polynomial for time, and logarithmic for space, and the size of the output is bounded in the size of the input.

The third term rewriting system (TRS_3) rewrites any LGQ graph into a forward LGQ forest. TRS_3 includes TRS_2 and inherits the complexities of TRS_2. For LGQ graph containing only closure predicates, respectively only non-closure predicates, TRS_3 yields LGQ forests containing also only closure predicates, respectively non-closure predicates. In particular, it rewrites any LGQ graph containing neither disjunctions nor closure predicates into a forward tree, which is variable-preserving minimal (cf. Proposition 3.5.1).

Beyond their declared main purpose of providing equivalences between forward queries and queries with reverse predicates within various LGQ fragments, the applications of our rewriting systems shed light on other LGQ properties, like the expressivity of some LGQ fragments as mentioned above, the minimization of LGQ queries, or even the complexity of LGQ query evaluation.

In this respect, the rewriting systems detect and eliminate non-trivial redundancies within queries (see Example 4.4.2 for an immediate impression). Also, they render evaluation strategies designed only for forward queries of particularly restricted LGQ fragments as sufficient to cover the whole language LGQ, equivalent to these restricted fragments. Indeed, Chapter 5 gives later an evaluation strategy only for forward LGQ forest and single-join DAG queries with polynomial complexity. The complexities of the evaluation of other LGQ queries follows then from both the complexities of rewriting them into forward queries and of the evaluation of these forward queries. Besides the polynomial complexities of the evaluation of LGQ forest and single-join DAG queries (and thus of XPath queries), the most interesting result obtained from the joint work of both this chapter and Chapter 5 is that there is a considerably large fragment of LGQ graph queries that admits evaluation with polynomial complexities, although in general their complexities are exponential.

4.1 Problem Description

The *equivalence-preserving removal of LGQ reverse atoms (ERRA) problem* is: given an LGQ query (formula) containing reverse atoms, is there always an equivalent forward LGQ query (formula)? And in the positive case, deliver the forward LGQ query (formula).

This chapter gives a positive answer to the ERRA problem for the whole LGQ language and for some of its fragments. Furthermore, this chapter reveals several flavours of this problem, dependant on the type (i.e., path, tree, etc.) of the given LGQ query that contains reverse atoms, and of its equivalent LGQ forward query obtained. Keeping an eye on XPath, this chapter particularly studies for which LGQ queries there are equivalent forward LGQ queries that correspond to XPath queries and that can be obtained as solutions to the ERRA problem.

The salient characteristics of the ERRA problem are (1) the preservation of equivalence between the initial query and the obtained forward query, and (2) the removal of constituent reverse atoms. Therefore, this problem has strong connections to the well-known problems of query equivalence and view-based query rewriting, though the ERRA problem still remains different. Section 4.6 discusses in more depth related problems.

The query equivalence problem is to decide whether two queries deliver the same answers for any data instance, thus approaches to the query equivalence problem assume the queries given and deliver yes/no answers regarding their equivalence. Such approaches can not, however, apply to solve the ERRA problem, where an equivalent forward query has to be *found*. Guessing a forward query and then testing whether it is equivalent to a given query (containing reverse formulas) is not affordable, for there are infinitely many forward queries[1]. This observation gives also the direction for how to construct automated solvers to the ERRA problem: if one *provides* equivalent forward queries for a finite number of etalon queries containing reverse formulas, and then *shows* that any other query is in fact just a combination of such etalon queries for which there are already forward equivalents, then one could use the finite set of forward equivalents to *rewrite* any query to a forward equivalent, like putting together the pieces of a puzzle. Using this approach for the ERRA problem, a sound (but not necessarily complete) approach to the equivalence problem would be then: given two queries, check whether one query can be obtained from the other by using a finite number of rewrite steps.

The rewriting approach opens the door to the next strongly related problem: the view-based query rewriting and answering problem (AQUV) [35]. The AQUV problem is to find efficient methods to answer a query using a set of previously defined materialized views over the database, rather than accessing the database relations. Looking at ERRA through AQUV glasses, one could see the etalon formulas containing reverse atoms as view bodies and their corresponding forward equivalents as view heads; a rewriting of a query using such views replaces all occurrences of instances of any of the views bodies with the corresponding instance of the views heads, thus delivering at the end an equivalent and

[1] Similar to the ideas of [65], it may be of interest to study if there is only a finite (though large) number of *canonical* forward queries that do not contain redundancies, that are equivalent to a given input, and that depend on the input's properties (like structure, size, etc.).

forward query. This is, indeed, the way it is proceeded also in this chapter, with some minor observations. First, the views must not be materialized. Second, we are interested to rewrite only the problematic reverse atoms, thus rewriting the given query only in terms of the given views is not an issue. Third, there can be more than one rewriting step, for there can be views that map queries to equivalent queries that still contain reverse atoms.

Warm-up Examples

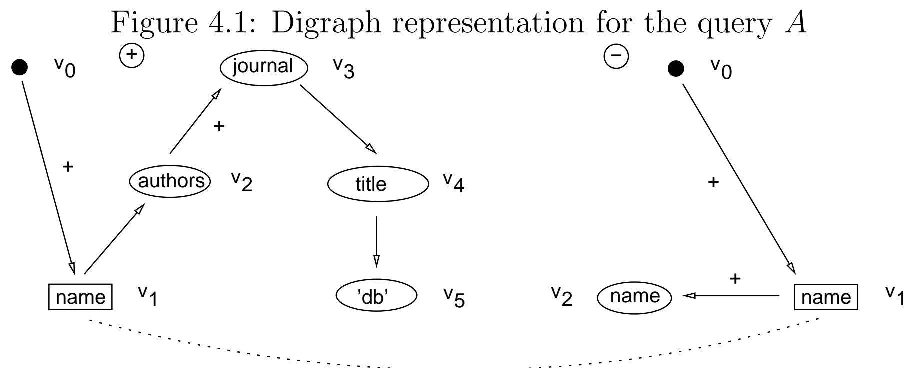
Many real-world XML queries, formulated in XPath or LGQ, use reverse predicates. A common practice in writing XML queries is to first specify the nodes to be selected, and then to further add structural constraints for these nodes. Arguably, such additional structural constraints use as well forward as also reverse predicates.

For the impatient reader, this section gives a bit of the taste of rewriting reasonably complex LGQ queries into equivalent forward LGQ queries.

Example 4.1.1. Consider the journal archive example of Section 2.2 and the query

$$\begin{aligned} A(v_1) &\leftarrow \mathsf{root}(v_0) \land \mathsf{child}^+(v_0,v_1) \land \mathsf{name}(v_1) \land \neg A'(v_1) \land \mathsf{par}(v_1,v_2) \land \mathsf{authors}(v_2) \\ &\quad \land \mathsf{par}^+(v_2,v_3) \land \mathsf{journal}(v_3) \land \mathsf{child}(v_3,v_4) \land \mathsf{title}(v_4) \land \mathsf{child}(v_4,v_5) \land \text{`}\mathsf{db}\text{'}(v_5). \\ A'(v_1) &\leftarrow \mathsf{root}(v_0) \land \mathsf{child}^+(v_0,v_1) \land \mathsf{name}(v_1) \land \mathsf{prevSibl}^+(v_1,v_2) \land \mathsf{name}(v_2). \end{aligned}$$

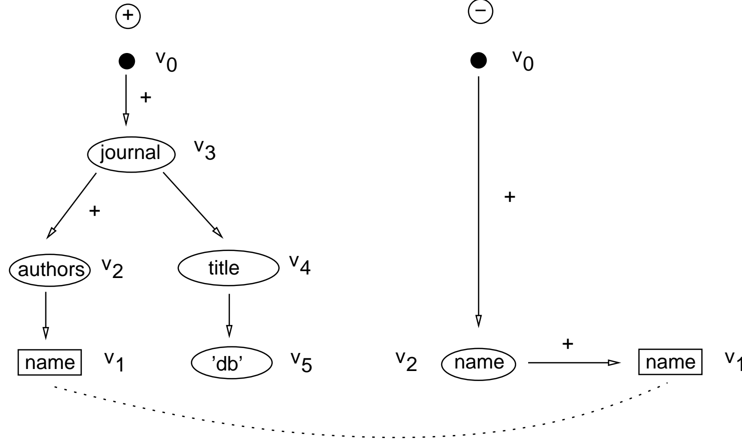
Figure 4.1: Digraph representation for the query A



The query A selects the first author of a journal with the title 'db', or more precisely, the first name child (v_1) of authors-nodes (v_2) that have journal ancestors (v_3) with a title child containing the text 'db' (v_4). Note that the same answer can be obtained by using the fstChild predicate that contains the pairs of nodes and their first children. This alternative shows that fstChild is also redundant in LGQ and can be obtained using child and negation.

For the tree instance of Figure 2.1 representing a journal archive, this query selects the first name-node in document order (i.e., the node containing the text 'ana'). The digraph representation for this query is given in Figure 4.1. The same answer can be selected also

Figure 4.2: Digraph representation for the forward query FA equivalent to A



using the following forward LGQ query FA with the digraph representation of Figure 4.2

$$FA(v_1) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_3) \wedge \mathsf{journal}(v_3) \wedge \mathsf{child}^+(v_3, v_2) \wedge \mathsf{authors}(v_2) \wedge \mathsf{child}(v_2, v_1) \wedge \mathsf{name}(v_1) \wedge \mathsf{child}(v_3, v_4) \wedge \mathsf{title}(v_4) \wedge \mathsf{child}(v_1, v_5) \wedge \text{`}\mathsf{db}\text{'}(v_5).$$

$$FA'(v_1) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_2) \wedge \mathsf{name}(v_2) \wedge \mathsf{nextSibl}^+(v_2, v_1) \wedge \mathsf{name}(v_1).$$

It is easy to see that FA and A are equivalent. In the following, we proceed step by step with the rewriting of A, by first identifying path formulas made out of one forward and one reverse atom, and then rewriting them to forward path formulas (if possible). In the following, we make use of a substitution s consistent with the query A and with the tree instance.

Step 1. We rewrite at this step $\mathsf{child}^+(v_0, v_1) \wedge \mathsf{par}(v_1, v_2)$ into $\mathsf{child}^*(v_0, v_2) \wedge \mathsf{child}(v_2, v_1)$. Because the root node $s(v_0)$ can not be an authors-node $s(v_2)$, we further simplify it to $\mathsf{child}^+(v_0, v_2) \wedge \mathsf{child}(v_2, v_1)$. The intuition behind this rewriting is the following: if the authors-node $s(v_2)$ is the parent of a name-node $s(v_1)$, which is a descendant of the root node $s(v_0)$, then that $s(v_2)$ is also a descendant of the root $s(v_0)$ and has a name-node child $s(v_1)$. Then, A is rewritten into an equivalent query A_1 (A' remains the same)

$$A_1(v_1) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_2) \wedge \mathsf{authors}(v_2) \wedge \mathsf{child}(v_2, v_1) \wedge \mathsf{name}(v_1) \wedge \neg A'(v_1) \wedge \mathsf{par}^+(v_2, v_3) \wedge \mathsf{journal}(v_3) \wedge \mathsf{child}(v_3, v_4) \wedge \mathsf{title}(v_4) \wedge \mathsf{child}(v_4, v_5) \wedge \text{'}\mathsf{db}\text{'}(v_5).$$

Step 2. We rewrite at this step $\mathsf{child}^+(v_0, v_2) \wedge \mathsf{par}^+(v_2, v_3)$ into $\mathsf{child}^*(v_0, v_3) \wedge \mathsf{child}^+(v_3, v_2) \vee \mathsf{child}^+(v_0, v_2) \wedge \mathsf{par}^+(v_0, v_3)$. In our case, $s(v_0)$ is the root node, and because the root does not have ancestors, the second disjunct is dropped. In the first disjunct, a journal-node $s(v_3)$ can not be the root node $s(v_0)$. Therefore, the rewritten formula remains $\mathsf{child}^+(v_0, v_3) \wedge \mathsf{child}^+(v_3, v_2)$. The intuition is the following: if an authors-node $s(v_2)$ is a descendant of the root node $s(v_0)$ and also a descendant of a journal-node $s(v_3)$, then that journal-node lies on the path between the root node and the authors-node. Then, A_1 is

rewritten into an equivalent query A_2 (A' remains the same)

$$\begin{aligned} A_2(v_1) \leftarrow & \ \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_3) \wedge \mathsf{journal}(v_3) \wedge \mathsf{child}^+(v_3, v_2) \wedge \mathsf{authors}(v_2) \wedge \mathsf{child}(v_2, v_1) \\ & \wedge \mathsf{name}(v_1) \wedge \neg A'(v_1) \wedge \mathsf{child}(v_3, v_4) \wedge \mathsf{title}(v_4) \wedge \mathsf{child}(v_4, v_5) \wedge \text{'}\mathsf{db}\text{'}(v_5). \end{aligned}$$

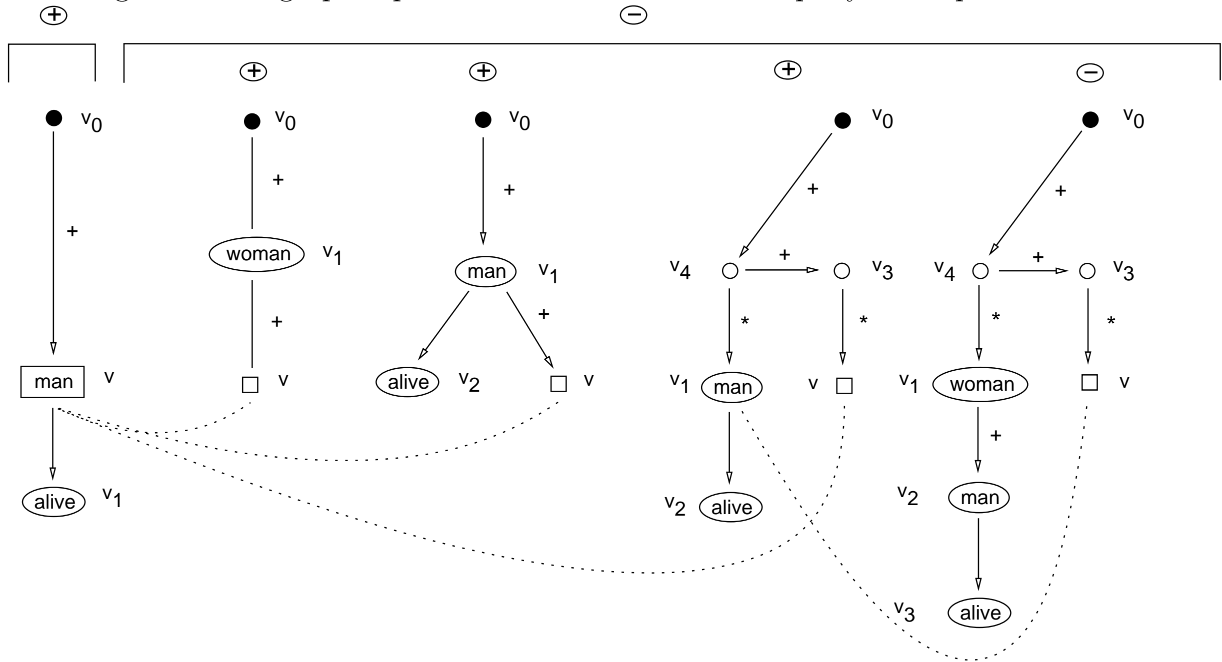
Step 3. A_2 is a forward query. It remains to rewrite A' into an equivalent forward query. We rewrite at this step $\mathsf{child}^+(v_0, v_1) \wedge \mathsf{prevSibl}^+(v_1, v_2)$ into $\mathsf{child}^+(v_0, v_2) \wedge \mathsf{nextSibl}^+(v_2, v_1)$. The intuition is the following: if a node $s(v_1)$ precedes a sibling node $s(v_2)$ that is a descendant of the node $s(v_0)$, then $s(v_1)$ is a descendant of the node $s(v_0)$ and is followed by the sibling $s(v_2)$. Then, A' is rewritten into the equivalent query FA'. □

Example 4.1.2. Consider the genealogical tree example of Section 2.2 and the query

$$\begin{aligned} G(v) \leftarrow & \ \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{man}(v) \wedge \neg G_1(v) \wedge \mathsf{child}(v, v_1) \wedge \mathsf{alive}(v) \\ G_1(v) \leftarrow & \ \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge ((\mathsf{prec}(v, v_1) \wedge \neg G_2(v) \vee \mathsf{par}^+(v, v_1)) \wedge \mathsf{man}(v_1) \\ & \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{alive}(v_2) \vee \mathsf{par}^+(v, v_1) \wedge \mathsf{woman}(v_1)) \\ G_2(v) \leftarrow & \ \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{prec}(v, v_1) \wedge \mathsf{woman}(v_1) \wedge \mathsf{child}^+(v_1, v_2) \wedge \mathsf{man}(v_2) \\ & \wedge \mathsf{child}(v_2, v_3) \wedge \mathsf{alive}(v_3). \end{aligned}$$

specifying the Salier law for the succession at the throne of a kingdom, as explained in Section 2.2.

Figure 4.3: Digraph representation for the forward query FG equivalent to G



For the tree instance of Figure 2.2 representing the genealogical tree of John II the Good, this query selects the male person named Charles VIII. The digraph representation

of G is given in Example 3.4.2. The same answer can be selected using the following forward LGQ query FG with the digraph representation of Figure 4.3

$$\begin{aligned} FG(v) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{man}(v) \wedge \neg FG_1(v) \wedge \mathsf{child}(v, v_1) \wedge \mathsf{alive}(v) \\ FG_1(v) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{child}^+(v_1, v) \wedge \mathsf{woman}(v_1) \vee \\ &\quad \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{child}^+(v_1, v) \wedge \mathsf{man}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{alive}(v_2) \vee \\ &\quad \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_4) \wedge \mathsf{nextSibl}^+(v_4, v_3) \wedge \mathsf{child}^*(v_3, v) \wedge \neg FG_2(v) \\ &\quad \wedge \mathsf{child}^*(v_4, v_1) \wedge \mathsf{man}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{alive}(v_2). \\ FG_2(v) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_4) \wedge \mathsf{child}^*(v_3, v) \wedge \mathsf{nextSibl}^+(v_4, v_3) \wedge \mathsf{child}^*(v_4, v_1) \\ &\quad \wedge \mathsf{woman}(v_1) \wedge \mathsf{child}^+(v_1, v_2) \wedge \mathsf{man}(v_2) \wedge \mathsf{child}(v_2, v_3) \wedge \mathsf{alive}(v_3). \end{aligned}$$

In the following, we highlight only the rewriting of G_2 into FG_2 (the rewriting of G_1 into FG_1 is similar). We make use of a substitution s consistent with the query G_2 and the tree instance. First, each $\mathsf{prec}(v, v_1)$ atom is replaced by the equivalent formula $\mathsf{par}^*(v, v_3) \wedge \mathsf{prevSibl}^+(v_3, v_4) \wedge \mathsf{child}^*(v_4, v_1)$. The query G_2 becomes (in disjunctive normal form)

$$\begin{aligned} G'_2(v) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge \mathsf{par}^*(v, v_3) \wedge \mathsf{prevSibl}^+(v_3, v_4) \wedge \mathsf{child}^*(v_4, v_1) \\ &\quad \wedge \mathsf{woman}(v_1) \wedge \mathsf{child}^+(v_1, v_2) \wedge \mathsf{man}(v_2) \wedge \mathsf{child}(v_2, v_3) \wedge \mathsf{alive}(v_3). \end{aligned}$$

Step 1. We rewrite the formula $\mathsf{child}^+(v_0, v) \wedge \mathsf{par}^*(v, v_3)$. Clearly, the nodes $s(v_0)$, $s(v)$, and $s(v_3)$ lie on the same path. Moreover, $s(v_0)$ appears before $s(v)$ on this path, and $s(v_3)$ can appear either before $s(v_0)$, or between $s(v_0)$ and $s(v)$, or can be exactly $s(v)$. The formula specifying all these possibilities is

$$\mathsf{child}^+(v_0, v) \wedge \mathsf{par}^+(v_0, v_3) \vee \mathsf{child}^*(v_0, v_3) \wedge \mathsf{child}^+(v_3, v) \vee \mathsf{child}^+(v_0, v) \wedge \mathsf{self}(v, v_3).$$

The first disjunct is unsatisfiable because $s(v_0)$ is the root node and there are no ancestors of the root node. The second disjunct becomes $\mathsf{child}^+(v_0, v_3) \wedge \mathsf{child}^+(v_3, v)$, because $s(v_0)$ is the root node and $s(v_3)$ is an alive-node. Concluding, the above formula becomes $\mathsf{child}^+(v_0, v_3) \wedge \mathsf{child}^*(v_3, v)$, and G'_2 is rewritten into the equivalent query

$$\begin{aligned} G''_2(v) &\leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_3) \wedge \mathsf{child}^*(v_3, v) \wedge \mathsf{prevSibl}^+(v_3, v_4) \wedge \mathsf{child}^*(v_4, v_1) \\ &\quad \wedge \mathsf{woman}(v_1) \wedge \mathsf{child}^+(v_1, v_2) \wedge \mathsf{man}(v_2) \wedge \mathsf{child}(v_2, v_3) \wedge \mathsf{alive}(v_3). \end{aligned}$$

Step 2. The only reverse atom remained is $\mathsf{prevSibl}^+(v_3, v_4)$, which is rewritten together with $\mathsf{child}^+(v_0, v_3)$ into the formula $\mathsf{child}^+(v_0, v_4) \wedge \mathsf{nextSibl}^+(v_4, v_3)$, because if a node $s(v_4)$ precedes a sibling node $s(v_3)$ that is a descendant of the node $s(v_0)$, then the node $s(v_4)$ is also a descendant of $s(v_0)$ and is followed by the sibling node $s(v_3)$. With this last rewriting, G''_2 becomes FG_2. □

The goal of this chapter is to automate the above process of finding an equivalent forward query to any LGQ query. We do this by making use of the theory of term rewriting systems.

4.2 A Taste of Term Rewriting Systems

Term rewriting systems are widely used as a model of computation to relate syntax and semantics. This section introduces basic notions on term rewriting systems [16] necessary to rewrite LGQ formulas.

Identities and Rewrite Rules

In order to express identities and rewritings of LGQ formulas, we define a language of rewriting rules and identities $\mathrm{LGQ}^{\rightarrow}$, similar to LGQ. $\mathrm{LGQ}^{\rightarrow}$ 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., $\underline{x}$, $\underline{y}$, $\underline{z}$.

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

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

Example 4.2.1. The $\mathrm{LGQ}^{\rightarrow}$ term $X \wedge Y \approx Y \wedge X$ is an identity that expresses the commutativity property of the $\wedge$ LGQ predicate. The $\mathrm{LGQ}^{\rightarrow}$ term $\mathsf{child}(\underline{x}, \underline{y}) \wedge \mathsf{prevSibl}(\underline{y}, \underline{z}) \rightarrow \mathsf{child}(\underline{x}, \underline{z}) \wedge \mathsf{nextSibl}(\underline{z}, \underline{y})$ specifies a rewriting for LGQ formulas. □

A $\mathrm{LGQ}^{\rightarrow}$ *substitution* σ is a total mapping from $\mathrm{LGQ}^{\rightarrow}$ variables to LGQ formulas or variables denoted by (1) $\{X_1 \mapsto s_1, \ldots, X_n \mapsto s_n\}$ indicating that the $\mathrm{LGQ}^{\rightarrow}$ 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 $\mathrm{LGQ}^{\rightarrow}$ variable $\underline{x_i}$ maps to the LGQ variable s_i. If σ maps an $\mathrm{LGQ}^{\rightarrow}$ variable to an LGQ formula or variable, then that LGQ formula or variable is the *image* of the $\mathrm{LGQ}^{\rightarrow}$ variable under σ. If an $\mathrm{LGQ}^{\rightarrow}$ 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 $\mathrm{LGQ}^{\rightarrow}$ term, then $\sigma(f(t_1, t_2)) = f(\sigma(t_1), \sigma(t_2))$.

A substitution σ is a *matching* substitution of a $\mathrm{LGQ}^{\rightarrow}$ 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.

If u is the *most general unifier* of a set of terms, then any other unifier v can be expressed as $v = uw$, where w is another substitution.

A **term rewriting system** $(T, \rightarrow)$ is a finite set of rewrite rules and (possibly) identities on terms of T. If identities are present, then they serve to specify rewriting modulo these identities, as detailed in the following.

The next section proposes three rewriting systems ($\mathrm{LGQ}^{\rightarrow}$,$\rightarrow$) that contain rewrite rules with lhs instances equivalent to corresponding rhs instances. These rewriting systems can be used to transform LGQ formulas into equivalent forward LGQ formulas.

Example 4.2.2. The LGQ$^{\rightarrow}$ rule $\mathsf{child}(\underline{x}, \underline{y}) \wedge \mathsf{prevSibl}(\underline{y}, \underline{z}) \rightarrow \mathsf{child}(\underline{x}, \underline{z}) \wedge \mathsf{nextSibl}(\underline{z}, \underline{y})$ can be used to rewrite the formula $e_1 = \mathsf{child}(a, d) \wedge \mathsf{child}(a, b) \wedge \mathsf{prevSibl}(b, c)$ into the formula $e_2 = \mathsf{child}(a, d) \wedge \mathsf{child}(a, b) \wedge \mathsf{nextSibl}(c, b)$. Note that $e_1 \equiv e_2$. □

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. The result of such an application is written $s[\sigma(rhs)/\sigma(lhs)]$, and the entire application is written similar to a rule: $s \rightarrow s[\sigma(rhs)/\sigma(lhs)]$. A term s *derives* other term t, written $s \stackrel{*}{\rightarrow} t$, 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 $s \stackrel{*}{\rightarrow} t$, then s is *irreducible*. If $s \stackrel{*}{\rightarrow} t$ and t is irreducible, then t is a *normal form* of s, and we may write $s \rightarrow^! t$.

When dealing with term rewriting systems, there are (at least) two important questions to be asked:

Termination: Is it always the case that after finitely many rule applications an irreducible term is reached?

Confluence: If there are different ways of applying rules to a given term t, leading to different derived terms t_1 and t_2, can t_1 and t_2 be joined, i.e., can we always find a common term s that can be reached both from t_1 and t_2 by rule applications?

Both aforementioned properties ensure the existence and uniqueness of normal forms.

Termination

A rewriting relation $\rightarrow$ is *terminating* if there are no infinite derivations $s_0 \rightarrow s_1 \rightarrow \cdots$. Terminating relations are also called *well-founded*. By extension, a rewrite system $(T, \rightarrow)$, whose relation $\rightarrow$ is terminating, is also terminating.

The termination problem for rewriting systems is in general undecidable, i.e., there can not be a general procedure that, given an arbitrary finite rewriting system, answers "yes" if the system terminates, and "no" otherwise. However, it is useful to show that a particular rewriting system terminates. The basic method to prove termination of a rewriting system $(T, \rightarrow)$ is to embed it into another rewriting system $(A, >)$ that is known to terminate. This requires a monotone mapping $\phi : T \rightarrow A$, where monotone means that $lhs \rightarrow rhs$ implies $\phi(lhs) > \phi(rhs)$. The most popular choice for termination proofs is an embedding into $(\mathsf{N}, >)$, which is known to terminate, because the $>$ order on natural numbers is well-founded[2].

Because some rewriting systems need more complex orders, it is often useful to build them as *lexicographic products* of simpler ones. From a number n of strict orders $>_i$, i.e.,

[2]Recall that the order $>$ on rational (and also real) numbers is not well-founded, because there can be an infinitely descending chain of rational numbers between two rational numbers.

transitive and irreflexive relations, one can build the lexicographic product $>_{1\dots n}$ as

$$(x_1, \dots, x_n) >_{1\dots n} (y_1, \dots, y_n) \Leftrightarrow \exists k \leq n : (\forall i < k : x_i = y_i), x_k >_k y_k.$$

Properties like strictness and termination carry over from orders to their lexicographic products.

A useful and simple method for constructing terminating orders is multisets (or bags), i.e., sets with repeated elements.

Definition 4.2.1. *A multiset M over a set A is a function $M : A \to \mathbb{N}$. Intuitively, $M(x)$ is the number of copies of $x \in A$ in M.* $\mathrm{M}(A)$ *denote the set of all finite multisets over A.*

We use standard set notation like $\{x, y, y\}$ as an abbreviation of the function $\{x \mapsto 1, y \mapsto 2, z \mapsto 0\}$ over the set $A = \{x, y, z\}$.

Some basic operations and relations on $\mathrm{M}(A)$ are:

Element : $x \in M \Leftrightarrow M(x) > 0$.
Inclusion : $M \subseteq N \Leftrightarrow \forall x \in A : M(x) \leq N(x)$.
Union : $(M \cup N)(x) = M(x) + N(x)$.
Difference: $(M - N)(x) = M(x) \dot{-} N(x)$, where $m \dot{-} n$ is $m - n$ if $m \geq n$, else is 0.

The order on multisets M over a finite set A can be derived from an order on A.

Definition 4.2.2 (Multiset Order). *Given a strict order $>$ on a set A, the corresponding multiset order $>_{mul}$ is defined as follows:*

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

Properties like strictness and termination carry over from $(A, >)$ to $(\mathrm{M}(A), >_{mul})$.

Example 4.2.3. Consider the multisets $M = \{8, 1\}$ and $N = \{7, 7, 1\}$. Then, $M >_{mul} N$ because $N = (M - X) \cup Y$ with $X = \{8\}$ and $Y = \{7, 7\}$. Note that X and Y are not uniquely determined: $X = M$ and $Y = N$ do work here too. □

Throughout this chapter, we use strict orders on terms derived from the order $>_{mul}$ on (finite) multisets over finite sets of natural numbers.

Confluence

Definition 4.2.3 (Joinable Terms). *Two terms x and y are joinable for a relation $\to$, written $x \downarrow y$, iff there exists a term z such that $x \xrightarrow{*} z \xleftarrow{*} y$.*

Definition 4.2.4 (Confluence). *A rewrite relation is confluent iff terms are joinable whenever they are derivable from a same term*

$$y_1 \xleftarrow{*} x \xrightarrow{*} y_2 \Rightarrow y_1 \downarrow y_2$$

Checking confluence can be hard, because it requires to test the joinability of all possible terms derivable from a same term. A strictly weaker variant of confluence, called local confluence, can be, however, easier to check.

Definition 4.2.5 (Local confluence). *A relation $\rightarrow$ is locally confluent iff terms are joinable whenever they are derivable in one step from a same term*

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

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:

Definition 4.2.6 (Critical Pairs). *If $l \rightarrow r$ and $s \rightarrow t$ are two rewrite rules (with variables made distinct) and μ a most general unifier of l and a non-variable subterm s' of s, then the equation $\mu(t) = \mu(s[\mu(r)/\mu(s')])$, where $\mu(r)$ has replaced $\mu(s')$ $(= \mu(l))$ in $\mu(s)$, is a critical pair.*

A finite rewrite system has a finite number of critical pairs. Local confluence can be obtained also in the case of existence of critical pairs.

Theorem 4.2.1 ([16]). *A rewrite relation is locally confluent iff all its critical pairs are joinable.*

Confluence can be reduced to local confluence only for rewrite relations that terminate.

Lemma 4.2.1 ([122]). *A termination relation is confluent if it is locally confluent.*

We say also that the system is confluent when its relation is confluent.

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.

Example 4.2.4. The rewrite rule

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

can rewrite not only $\mathsf{child}(a, b) \wedge \mathsf{prevSibl}(b, c)$ into $\mathsf{child}(a, c) \wedge \mathsf{nextSibl}(c, b)$, but also, as highly desired, $\mathsf{prevSibl}(b, c) \wedge f \wedge \mathsf{child}(a, b)$ into $\mathsf{nextSibl}(c, b) \wedge \mathsf{child}(a, c) \wedge f$. Note that a syntactical substitution 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 LGQ predicates raise serious problems in rewriting systems, because such properties can not be oriented into terminating rewrite rules.

Example 4.2.5. Consider the rule $X \wedge Y \rightarrow Y \wedge X$ expressing the commutativity property of the $\wedge$ connective. The repeated application of this rule to the LGQ formula $\mathsf{child}(a, b) \wedge \mathsf{prevSibl}(b, c)$ yields an infinite number of contractions

$$\mathsf{child}(a, b) \wedge \mathsf{prevSibl}(b, c) \rightarrow \mathsf{prevSibl}(b, c) \wedge \mathsf{child}(a, b) \rightarrow \mathsf{child}(a, b) \wedge \mathsf{prevSibl}(b, c) \rightarrow \cdots$$

□

A common technique to accommodate AC properties in the rewriting process is to consider rewriting modulo the AC-theory. More specifically, this chapter considers rewriting systems containing the set AC of identities expressing the commutativity and associativity properties of $\wedge$, $\vee$, and self ($\alpha \in \mathsf{F}^? \cup \mathsf{R}^?$):

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

For the unification of terms, the syntactic unification does not suffice anymore and unification modulo AC-equations (or simply AC-unification) has to be considered. Also, AC matching substitutions must be used to detect applicability of rules.

Several important notions applicable to syntactical rewriting have to be reconsidered now in the light of rewriting modulo an equational theory. Let us consider the "problematic" identities (like AC-identities) of a rewriting system separated in the set E from the rules R. This gives rise to a new relation $\rightarrow_{R/E}$, which is defined on equivalence classes of terms ($[s]_{\approx_E}$ is the class of all terms identical modulo E):

$$[s]_{\approx_E} \rightarrow_{R/E} [t]_{\approx_E} \Leftrightarrow \exists s', t' : s \approx_E s' \rightarrow_R t' \approx_E t.$$

In the context of rewriting modulo an equational theory E (or simply E-rewriting), each rewrite step involves E-matching, i.e., matching modulo $\approx_E$. Also, the critical pair computation involves E-unification. Two terms s and t are joinable modulo E, written $s \downarrow_E t$, if $s \xrightarrow{*} s' \approx_E t' \xleftarrow{*} t$.

AC-matching and AC-unification are NP-complete in general: the number of substitutions (unifiers) for any two terms can be exponential in the size of the terms, see, e.g., [103]. The LGQ$^{\rightarrow}$ rewrite rules of this chapter ensure a polynomial upper bound to the AC-matching, because they restrict severly the matchings of their variables. Section 4.5 details on this issue.

4.3 Rewrite Rules preserving LGQ Equivalence

This section introduces equivalence-preserving (rewrite) rules of reverse and forward formulas. These rules are used later in Section 4.4 to rewrite LGQ formulas into equivalent forward LGQ formulas by repeatedly contracting the formulas until a normal form is reached.

4.3.1 Rules adding single-join DAG-Structure

This section considers a simple yet powerful equivalence-preserving rule of reverse binary atoms and forward formulas. The lhs and rhs of this rule are also expressible in XPath syntax, as considered in our previous work [128]. Based on this rule, Section 4.4 shows how any LGQ formula can be rewritten into an equivalent LGQ forward formula, where

each reverse atom in the initial formula induces a multi-sink variable in the rewritten LGQ formula.

Lemma 4.3.1 (Equivalence-preserving rule adding single-join DAG-Structure). *Applying the rewrite rule*

$$\alpha(\underline{x}, \underline{y}) \rightarrow \alpha^{-1}(\underline{y}, \underline{x}) \wedge \mathsf{child}^{+}(z, \underline{y}) \wedge \mathsf{root}(z) \tag{4.1}$$

to an LGQ formula e, which contains a reverse α-atom, yields an LGQ formula t equivalent to s, where z is a fresh LGQ variable in t.

Proof. For an instance $l \rightarrow r$ of the above rule under a substitution $\sigma = \{\underline{x} \mapsto x, \underline{y} \mapsto y\}$, we show that (1) $l \equiv r$, and (2) $s \equiv s[r/l]$.

The first part of the proof follows from the observation that all nodes in the tree are descendants of the root. Then,

$$\begin{aligned}\mathcal{LF}[\![\alpha(x,y)]\!](\beta) &= \{t \mid t \in \beta, \alpha(t(x), t(y))\}\\ &= \{t \mid t \in \beta, \alpha^{-1}(t(y), t(x)), \mathsf{child}^{+}(root(t(x)), t(y))\}\\ &= \{t \mid t \in \beta, \alpha^{-1}(t(y), t(x)), \mathsf{child}^{+}(z, t(y)), z = root(t(x))\}\\ &= \mathcal{LF}[\![\alpha^{-1}(y,x) \wedge \mathsf{child}^{+}(z,y) \wedge \mathsf{root}(z)]\!](\beta).\end{aligned}$$

The second part of the proof follows from Proposition 3.3.1, with the condition that the subformula of s obtained by removing l or r does not contain variables appearing only in r, respectively l, and not in the other one. Indeed, the only new variable appearing in r is the fresh variable z. ☐

Remark 4.3.1. The lhs and rhs of Rule (4.1) can be also expressed using XPath extended with the identity-based equality `==` . Let P be a rule variable standing for an XPath relative formula, N and M nodetest holders (rule variables), a_n a forward axis, a_m a reverse axis, and b_m the symmetrical axis to a_m. Cf. [128],

$$\begin{aligned}/P/a_n\!::N/a_m\!::M &\rightarrow /\mathsf{descendant}::M[b_m::N \texttt{ == } /P/a_n::N]\\ P[a_m::M] &\rightarrow P[\mathsf{descendant}::M/b_m::\mathsf{node()} \texttt{ == } \mathsf{self}::\mathsf{node()}]\end{aligned}$$

Arguably, the above two equivalences in XPath are harder to grasp than Rule (4.1) expressed in $\text{LGQ}^{\rightarrow}$: In XPath, a location step, made out of an axis and a nodetest, is an atomic construct, and filters are enclosed by square brackets. Therefore, both cases of reverse steps inside and outside filters have to be considered in XPath. In LGQ, however, the formulas corresponding to XPath filters are not explicitly marked, and the nodetest predicates are not necessary for the rule and therefore not carried over. ☐

Example 4.3.1. Consider the journal archive example of Section 2.2 and the tree instance of Figure 2.1. The LGQ tree query

$$Q(v_3) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}(v_0, v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{par}(v_2, v_3) \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2)$$

selects the parent node of an editor node that is child of a journal node, which is in its turn a child node of the root. For the given tree instance, Q_1 selects the journal node.

According to Rule (4.1) and Proposition 3.3.1, Q is equivalent to

$$FQ(v_3) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}(v_0, v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{child}(v_3, v_2) \wedge \mathsf{child}^+(v_3', v_3) \wedge \mathsf{root}(v_3') \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2).$$

For Q there is an equivalent XPath query

/child::journal/child::editor/parent::node().

For FQ there is only an equivalent XPath query with equality based on node-identity:

/descendant::node()[child::editor == /child::journal/child::editor]

□

There is an order $>^{rev}_{type}$ between an LGQ formula s and the formula t obtained by applying Rule (4.1) to s. Recall from Section 3.7 that the order $>^{rev}_{type}$ is derived from the multiset order $>_{mul}$ by $s >^{rev}_{type} t \Leftrightarrow type^{rev}(s) >_{mul} type^{rev}(t)$.

Proposition 4.3.1 ($>^{rev}_{type}$-Decrease). *An application of Rule* (4.1) *to an LGQ formula s containing a redex of that rule yields an LGQ formula t that has a smaller type factor than s: $s >^{rev}_{type} t$.*

Proof. Let $\sigma = \{\underline{x} \mapsto x, \underline{y} \mapsto y\}$. We consider there are n reverse binary atoms in s. The reverse type factor for s is

$$type^{rev}(s) = \{i_1, \ldots, i_n\}.$$

Let i_k be the encoding of the existence of that reverse α-atom in $type^{rev}(s)$ $(\exists k : 1 \leq k \leq n)$.

Recall that for two multisets $A, B \in \mathrm{M(N)}$, the strict order $>_{mul}$ is defined by

$$A >_{mul} B \Leftrightarrow \exists C, D \in \mathrm{M(N)} : \emptyset \neq X \subseteq A, B = (A - C) \cup D, \forall d \in D : \exists c \in C : c > d.$$

As ensured by Rule (4.1), the reverse α-atom is removed. Hence,

$$type^{rev}(t) = (type^{rev}(s) - C) \cup D, C = \{i_k\}, D = \emptyset \Rightarrow type^{rev}(s) >_{mul} type^{rev}(t) \Rightarrow s >^{rev}_{type} t.$$

□

4.3.2 Rules preserving Tree-Structure

This section gives equivalence-preserving rules for paths of a forward binary atom followed by a reverse binary atom, by systematically exploiting each possible combination of forward binary atoms with fstChild, child, child$^+$, nextSibl, and nextSibl$^+$ predicates, and reverse binary atoms with par, par$^+$, prevSibl, and prevSibl$^+$ predicates. All in all, there are 20 such rules. Note that the combinations of the forward self-atom and the reverse atoms are already covered in built-in identities of rewriting systems, cf. Section 4.2, and therefore are not needed anymore in the rules. The rest of reverse and forward atoms can be safely left out of discussion, as explained further in Lemma 4.3.2.

These rules can be formulated also in XPath syntax, as considered in our previous work [128]. Based on these rules, Section 4.4 shows how any LGQ formula can be rewritten into an equivalent forward LGQ formula with no more multi-sink variables than in the initial formula, thus the equivalent forward formula does not have additional graph structure.

Lemma 4.3.2 (foll, prec, child*, nextSibl*, par*, prevSibl* Elimination). *Consider $\alpha \in \{\mathsf{child}, \mathsf{par}, \mathsf{nextSibl}, \mathsf{prevSibl}\}$, and a, b fresh variables. Then, the application of the each of following rewrite rules to an LGQ formula yields an equivalent LGQ formula.*

$$\mathsf{foll}(\underline{x}, \underline{y}) \rightarrow \mathsf{par}^*(\underline{x}, a) \wedge \mathsf{nextSibl}^+(a, b) \wedge \mathsf{child}^*(b, \underline{y}) \quad (4.2)$$

$$\mathsf{prec}(\underline{x}, \underline{y}) \rightarrow \mathsf{par}^*(\underline{x}, a) \wedge \mathsf{prevSibl}^+(a, b) \wedge \mathsf{child}^*(b, \underline{y}) \quad (4.3)$$

$$\alpha^*(\underline{x}, \underline{y}) \rightarrow (\alpha^+(\underline{x}, \underline{y}) \vee \mathsf{self}(\underline{x}, \underline{y})). \quad (4.4)$$

Proof. The first two rules follow directly from the definitions of the predicates foll and prec, and the last rule from the definition of reflexive transitive closure of binary predicates. ☐

Lemma 4.3.3 (Equivalence-preserving rules preserving tree-structure). *Applying each of the rewrite rules of Figure 4.4 to an LGQ formula s, which contains a path of the form $\alpha_1(x, y) \wedge \alpha_2(y, z)$ with α_1 a forward predicate and α_2 a reverse predicate, yields an equivalent LGQ formula t.*

Proof. The proofs for all rules are given in Appendix. ☐

Remark 4.3.2. The lhs and rhs of Rules (4.5) through (4.24) involving predicates that have corresponding XPath axes can be also expressed using XPath. Let N and M be nodetest holders (rule variables). Cf. [128], the Rule (4.7) can then be expressed as

descendant::N/parent::M → descendant-or-self::N[child::M]
descendant::N[parent::M] → descendant-or-self::N/child::M.

Note there are two rules necessary in XPath to express Rule (4.7), for the case of reverse steps inside and outside filters. Both rules are similar and the only difference consists in the explicit syntactical marking with square brackets of XPath filters. ☐

$$\mathsf{fstChild}(\underline{x}, \underline{y}) \land \mathsf{par}(\underline{y}, \underline{z}) \to \mathsf{self}(\underline{x}, \underline{z}) \land \mathsf{fstChild}(\underline{z}, \underline{y}) \tag{4.5}$$
$$\mathsf{child}(\underline{x}, \underline{y}) \land \mathsf{par}(\underline{y}, \underline{z}) \to \mathsf{self}(\underline{x}, \underline{z}) \land \mathsf{child}(\underline{z}, \underline{y}) \tag{4.6}$$
$$\mathsf{child}^{+}(\underline{x}, \underline{y}) \land \mathsf{par}(\underline{y}, \underline{z}) \to \mathsf{child}^{*}(\underline{x}, \underline{z}) \land \mathsf{child}(\underline{z}, \underline{y}) \tag{4.7}$$
$$\mathsf{nextSibl}(\underline{x}, \underline{y}) \land \mathsf{par}(\underline{y}, \underline{z}) \to \mathsf{nextSibl}(\underline{x}, \underline{y}) \land \mathsf{par}(\underline{x}, \underline{z}) \tag{4.8}$$
$$\mathsf{nextSibl}^{+}(\underline{x}, \underline{y}) \land \mathsf{par}(\underline{y}, \underline{z}) \to \mathsf{nextSibl}^{+}(\underline{x}, \underline{y}) \land \mathsf{par}(\underline{x}, \underline{z}) \tag{4.9}$$

$$\mathsf{fstChild}(\underline{x}, \underline{y}) \land \mathsf{par}^{+}(\underline{y}, \underline{z}) \to (\mathsf{fstChild}(\underline{x}, \underline{y}) \land \mathsf{par}^{+}(\underline{x}, \underline{z}) \lor \mathsf{fstChild}(\underline{x}, \underline{y}) \land \mathsf{self}(\underline{x}, \underline{z})) \tag{4.10}$$
$$\mathsf{child}(\underline{x}, \underline{y}) \land \mathsf{par}^{+}(\underline{y}, \underline{z}) \to (\mathsf{child}(\underline{x}, \underline{y}) \land \mathsf{par}^{+}(\underline{x}, \underline{z}) \lor \mathsf{child}(\underline{x}, \underline{y}) \land \mathsf{self}(\underline{x}, \underline{z})) \tag{4.11}$$
$$\mathsf{child}^{+}(\underline{x}, \underline{y}) \land \mathsf{par}^{+}(\underline{y}, \underline{z}) \to (\mathsf{child}^{+}(\underline{x}, \underline{y}) \land \mathsf{par}^{+}(\underline{x}, \underline{z}) \lor \mathsf{child}^{*}(\underline{x}, \underline{z}) \land \mathsf{child}^{+}(\underline{z}, \underline{y})) \tag{4.12}$$
$$\mathsf{nextSibl}(\underline{x}, \underline{y}) \land \mathsf{par}^{+}(\underline{y}, \underline{z}) \to \mathsf{nextSibl}(\underline{x}, \underline{y}) \land \mathsf{par}^{+}(\underline{x}, \underline{z}) \tag{4.13}$$
$$\mathsf{nextSibl}^{+}(\underline{x}, \underline{y}) \land \mathsf{par}^{+}(\underline{y}, \underline{z}) \to \mathsf{nextSibl}^{+}(\underline{x}, \underline{y}) \land \mathsf{par}^{+}(\underline{x}, \underline{z}) \tag{4.14}$$

$$\mathsf{fstChild}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}(\underline{y}, \underline{z}) \to \bot \tag{4.15}$$
$$\mathsf{child}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}(\underline{y}, \underline{z}) \to \mathsf{child}(\underline{x}, \underline{z}) \land \mathsf{nextSibl}(\underline{z}, \underline{y}) \tag{4.16}$$
$$\mathsf{child}^{+}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}(\underline{y}, \underline{z}) \to \mathsf{child}^{+}(\underline{x}, \underline{z}) \land \mathsf{nextSibl}(\underline{z}, \underline{y}) \tag{4.17}$$
$$\mathsf{nextSibl}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}(\underline{y}, \underline{z}) \to \mathsf{self}(\underline{x}, \underline{z}) \land \mathsf{nextSibl}(\underline{z}, \underline{y}) \tag{4.18}$$
$$\mathsf{nextSibl}^{+}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}(\underline{y}, \underline{z}) \to \mathsf{nextSibl}^{*}(\underline{x}, \underline{z}) \land \mathsf{nextSibl}(\underline{z}, \underline{y}) \tag{4.19}$$

$$\mathsf{fstChild}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}^{+}(\underline{y}, \underline{z}) \to \bot \tag{4.20}$$
$$\mathsf{child}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}^{+}(\underline{y}, \underline{z}) \to \mathsf{child}(\underline{x}, \underline{z}) \land \mathsf{nextSibl}^{+}(\underline{z}, \underline{y}) \tag{4.21}$$
$$\mathsf{child}^{+}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}^{+}(\underline{y}, \underline{z}) \to \mathsf{child}^{+}(\underline{x}, \underline{z}) \land \mathsf{nextSibl}^{+}(\underline{z}, \underline{y}) \tag{4.22}$$
$$\mathsf{nextSibl}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}^{+}(\underline{y}, \underline{z}) \to (\mathsf{nextSibl}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}^{+}(\underline{x}, \underline{z}) \lor \mathsf{nextSibl}(\underline{x}, \underline{y}) \land \mathsf{self}(\underline{x}, \underline{z})) \tag{4.23}$$
$$\mathsf{nextSibl}^{+}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}^{+}(\underline{y}, \underline{z}) \to (\mathsf{nextSibl}^{+}(\underline{x}, \underline{y}) \land \mathsf{prevSibl}^{+}(\underline{x}, \underline{z}) \lor \mathsf{nextSibl}^{*}(\underline{x}, \underline{z}) \land \mathsf{nextSibl}^{+}(\underline{z}, \underline{y})) \tag{4.24}$$

Figure 4.4: Equivalence-preserving rules for paths of forward and reverse atoms

As depicted in Figure 4.4, the interactions of forward (α_1) and reverse (α_2) atoms of some rules behave similarly. In order to characterize them more compactly, we define six *interaction* classes. The classification of the rules depends on the predicate classes involved in those rules, as shown in Figure 4.5. Characteristics of predicate classes common to both atoms are factored out in the name of the interaction class, e.g., the interaction class $\mathsf{H/V(F,R)^+}$ stands for $\mathsf{(HF^+,HR^+)}\cup\mathsf{(VF^+,VR^+)}$, which contains rules where both forward and reverse atoms have transitive closure predicates that are either vertical or horizontal.

(f, r)	$f(\underline{x}, \underline{y}) \wedge r(\underline{y}, \underline{z}) \rightarrow$	Rules
$(\{\mathsf{fstChild}\}, \mathsf{HR}^?)$	$\perp$	(4.15),(4.20)
$(\{\mathsf{child}, \mathsf{child}^+\}, \mathsf{HR}^?)$	$f(\underline{x}, \underline{z}) \wedge r^{-1}(\underline{z}, \underline{y})$	(4.16),(4.17),(4.21),(4.22)
$(\mathsf{HF}, \mathsf{VR})^?$	$f(\underline{x}, \underline{y}) \wedge r(\underline{x}, \underline{z})$	(4.8),(4.9),(4.13),(4.14)
$\mathsf{H/V(F,R^+)}$	$(f(\underline{x}, \underline{y}) \wedge r(\underline{x}, \underline{z}) \vee$ $f(\underline{x}, \underline{y}) \wedge \mathsf{self}(\underline{x}, \underline{z}))$	(4.10),(4.11),(4.23)
$\mathsf{H/V(F^?,R)}$	$f'(\underline{x}, \underline{z}) \wedge r^{-1}(\underline{z}, \underline{y})$	(4.5),(4.6),(4.7),(4.18),(4.19)
$\mathsf{H/V(F,R)^+}$	$(f(\underline{x}, \underline{y}) \wedge r(\underline{x}, \underline{z}) \vee$ $f'(\underline{x}, \underline{z}) \wedge r^{-1}(\underline{z}, \underline{y}))$	(4.12),(4.24)

$(f, f') \in \{(\mathsf{child}, \mathsf{self}), (\mathsf{fstChild}, \mathsf{self}), (\mathsf{child}^+, \mathsf{child}^*), (\mathsf{nextSibl}, \mathsf{self}), (\mathsf{nextSibl}^+, \mathsf{nextSibl}^*)\}$

Figure 4.5: Characterization of atom interactions of rules from Figure 4.4

Example 4.3.2 *(*par*).* Consider the journal archive example of Section 2.2 and the tree instance of Figure 2.1. The LGQ tree formula

$$Q_1(v_3) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}(v_0, v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{par}(v_2, v_3) \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2)$$

selects the parent node of an editor node that is child of a journal node, which is in its turn a child node of the root. For the given tree, Q_1 selects the journal node.

According to Rule (4.6) and Proposition 3.3.1, Q_1 is equivalent to

$$FQ_1(v_3) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}(v_0, v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{self}(v_1, v_3) \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2)$$

or more compact, by replacing all occurences of v_1 by v_3

$$FQ'_1(v_3) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}(v_0, v_3) \wedge \mathsf{child}(v_3, v_2) \wedge \mathsf{journal}(v_3) \wedge \mathsf{editor}(v_2).$$

Note there are equivalent XPath queries for the above LGQ trees.

Consider now the same tree instance and the LGQ DAG formula

$$Q_2(v_3) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{child}^+(v_0, v_3) \wedge \mathsf{nextSibl}(v_1, v_2) \wedge \mathsf{par}(v_2, v_3) \wedge \mathsf{name}(v_1) \wedge \mathsf{name}(v_2) \wedge \mathsf{authors}(v_3)$$

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 given tree, Q_2 selects the authors node.

According to Rule (4.8) and Proposition 3.3.1, Q_2 is equivalent to

$$FQ_2(v_3) \leftarrow \mathsf{root}(v_0) \land \mathsf{child}^+(v_0, v_1) \land \mathsf{child}^+(v_0, v_3) \land \mathsf{nextSibl}(v_1, v_2) \land \mathsf{par}(v_1, v_3) \land \mathsf{name}(v_1) \land \mathsf{name}(v_2) \land \mathsf{authors}(v_3)$$

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 (4.7) can be applied now and we get

$$FQ_2'(v_3) \leftarrow \mathsf{root}(v_0) \land \mathsf{child}^*(v_0, v_3) \land \mathsf{child}^+(v_0, v_3) \land \mathsf{nextSibl}(v_1, v_2) \land \mathsf{child}(v_3, v_1) \land \mathsf{name}(v_1) \land \mathsf{name}(v_2) \land \mathsf{authors}(v_3)$$

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_3 hold at the same time the relation child* and a subset of it child$^+$, FQ_2' can be further compacted to

$$FQ_2''(v_3) \leftarrow \mathsf{root}(v_0) \land \mathsf{child}^+(v_0, v_3) \land \mathsf{nextSibl}(v_1, v_2) \land \mathsf{child}(v_3, v_1) \land \mathsf{name}(v_1) \land \mathsf{name}(v_2) \land \mathsf{authors}(v_3)$$

Note that FQ_2'' is an LGQ path and has a corresponding XPath query, whereas its equivalent Q_2 is an LGQ DAG and has no corresponding XPath query.

Such repeated redex detections and contractions constitute the basis of a rewriting system for LGQ formulas, as proposed next in Section 4.4.

□

Example 4.3.3 *(*par$^+$*).* Consider the journal archive example of Section 2.2 and the tree instance of Figure 2.1. The LGQ path formula

$$Q_3(v_2) \leftarrow \mathsf{root}(v_0) \land \mathsf{child}^+(v_0, v_1) \land \mathsf{nextSibl}(v_1, v_2) \land \mathsf{par}^+(v_2, v_3) \land \mathsf{name}(v_2) \land \mathsf{name}(v_1)$$

selects the ancestors of name nodes that follow name sibling nodes descendants of the root. For the given tree, Q_3 selects the nodes authors, journal, and the root.

According to Rule (4.13) and Proposition 3.3.1, Q_3 is equivalent to

$$FQ_3(v_2) \leftarrow \mathsf{root}(v_0) \land \mathsf{child}^+(v_0, v_1) \land \mathsf{nextSibl}(v_1, v_2) \land \mathsf{par}^+(v_1, v_3) \land \mathsf{name}(v_2) \land \mathsf{name}(v_1)$$

because an ancestor v_3 of a sibling node v_2 of a node v_1 is also an ancestor of that node v_1. According to Rule (4.12) and Proposition 3.3.1, FQ_3 is equivalent to

$$FQ_3'(v_2) \leftarrow \mathsf{root}(v_0) \land \mathsf{nextSibl}(v_1, v_2) \land \mathsf{name}(v_2) \land \mathsf{name}(v_1) \land (\mathsf{child}^+(v_0, v_1) \land \mathsf{par}^+(v_0, v_3) \lor \mathsf{child}^*(v_0, v_3) \land \mathsf{child}^+(v_3, v_1))$$

or more compact (considering that $\mathsf{root}(v_0) \land \mathsf{par}^+(v_0, v_3) \rightarrow \bot$)

$$FQ_3''(v_2) \leftarrow \mathsf{root}(v_0) \land \mathsf{nextSibl}(v_1, v_2) \land \mathsf{child}^*(v_0, v_3) \land \mathsf{child}^+(v_3, v_1) \land \mathsf{name}(v_2) \land \mathsf{name}(v_1).$$

Note there are equivalent XPath queries for the LGQ paths Q_3, FQ_3, and FQ_3'', and also for the LGQ tree FQ_3'.

□

Example 4.3.4 *(*prevSibl *and* prevSibl$^+$*).* Consider the journal archive example of Section 2.2 and the tree instance of Figure 2.1 and the LGQ tree

$$Q_4(v_3) \leftarrow \mathsf{root}(v_0) \land \mathsf{child}^+(v_0,v_1) \land \mathsf{prevSibl}(v_1,v_2) \land \mathsf{prevSibl}^+(v_1,v_3) \land \mathsf{price}(v_1) \land \mathsf{authors}(v_2)$$

that selects the nodes that precede price sibling nodes that are immediately preceded by an authors sibling node. For the given tree, Q_4 selects the nodes authors, editor and title.

According to Rule (4.17) and Proposition 3.3.1, Q_4 is equivalent to

$$FQ_4(v_3) \leftarrow \mathsf{root}(v_0) \land \mathsf{child}^+(v_0,v_2) \land \mathsf{nextSibl}(v_2,v_1) \land \mathsf{prevSibl}^+(v_1,v_3) \land \mathsf{price}(v_1) \land \mathsf{authors}(v_2)$$

because a preceding sibling node v_2 of a descendant node v_1 from another node v_0 is a descendant node of v_0 that has a following sibling v_1. According to Rule (4.23) and Proposition 3.3.1, FQ_4 is equivalent to

$$\begin{aligned} FQ'_4(v_3) \leftarrow\ & \mathsf{root}(v_0) \land \mathsf{child}^+(v_0,v_2) \land \mathsf{price}(v_1) \land \mathsf{authors}(v_2) \\ & \land (\mathsf{nextSibl}(v_2,v_1) \land \mathsf{prevSibl}^+(v_2,v_3) \lor \mathsf{nextSibl}(v_2,v_1) \land \mathsf{self}(v_2,v_3)) \end{aligned}$$

because nodes v_3, which precede siblings v_2 that have immediate next siblings v_1, are either the siblings v_1 or precede them. □

Properties of Rules (4.5) **through** (4.24)

The applications of Rules (4.5) through (4.24) (1) preserve the variables from the initial formula, (2) do not transform 1-sink variables into multi-sink variables, and (3) ensure an order between LGQ formulas and their contractions. The second property is very useful, because the applications of such rules can never transform a tree formula into a DAG formula, or into a graph with cycles. The third property guarantees that LGQ formulas can not be rewritten endlessly, thus the rewriting terminates.

Proposition 4.3.2 (Variable, variable type and connections preservation). *The application of each rule of Lemma 4.3.3 to an LGQ formula s does not introduce fresh variables, it preserves the sink-arity of variables, and also the connections of non-sink variables.*

no fresh variables are introduced

$$\mathrm{Vars}(s) \supseteq \mathrm{Vars}(t)$$

non-sink variables remain non-sink

$$\forall x,y,z \in \mathrm{Vars}(s):\ x \not\rightsquigarrow_s y \Leftrightarrow z \not\rightsquigarrow_t y$$

connections of non-sink variables are preserved

$$\forall x,y,z \in \mathrm{Vars}(s):\ z \not\rightsquigarrow_s x \rightsquigarrow_s y \Leftrightarrow x \rightsquigarrow_t y$$

no multi-sink variables remain no multi-sink

$$\forall y,x_1,x_2,x'_1,x'_2 \in \mathrm{Vars}(s): x_1 \neq x_2, x'_1 \neq x'_2, x_1 \not\rightsquigarrow_s y, x_2 \not\rightsquigarrow_s y \Leftrightarrow x'_1 \not\rightsquigarrow_s y, x'_2 \not\rightsquigarrow_s y.$$

Proof. This can be easily seen by inspecting all interaction classes of Figure 4.5. □

The application of each rule of Lemma 4.3.3 ensures an order between the LGQ formulas containing redexes of that rule and their contractions. This order is built up from simpler orders on LGQ formulas, as defined next.

Definition 4.3.1 **($>^{rev}_{type\times pos}$).** *Given the strict order $>_{mul}$ on the multisets $\{type^{rev}(e) \mid e \in LGQ\}$ and on $\{pos^{rev}(e) \mid e \in LGQ\}$, the lexicographic product $>^{rev}_{type\times pos}$ of $>_{mul}$ with itself on $LGQ \times LGQ$ is defined by*

$$s >^{rev}_{type\times pos} t \Leftrightarrow type^{rev}(s) >_{mul} type^{rev}(t) \text{ or } type^{rev}(s) = type^{rev}(t), pos^{rev}(s) >_{mul} pos^{rev}(t).$$

Because $>_{mul}$ is strict order, so is $>^{rev}_{type\times pos}$, cf. Section 4.2.

Proposition 4.3.3 **($>^{rev}_{type\times pos}$-Decrease).** *An application of any rule of Lemma 4.3.3 to an LGQ formula s containing a redex of that rule yields an LGQ formula t that has a smaller reverse factor than s: $s >^{rev}_{type\times pos} t$.*

Proof. Let $\sigma = \{\underline{x} \mapsto x, \underline{y} \mapsto y, \underline{z} \mapsto z\}$, and $l \to r$ an instance of a rule of Lemma 4.3.3 under the substitution σ.

The ordering property can be shown by inspecting all six interaction classes of Figure 4.5. For all classes, the interaction is specified within disjuncts, so we can safely consider only one disjunct in s that contains l, and the other disjuncts are not changed.

Let x be the non-sink variable in both l and r. We consider there are n reverse binary atoms in s such that $root \leadsto_s x \overset{p}{\leadsto}_s v$ where $root$ is a non-sink variable, p is a connection sequence that ends with a reverse predicate, and v is a variable. The multiset of these lengths $|p|$ is denoted by $\{p_1, \ldots, p_n\}$, and a subset $\{p_1, \ldots, p_m\}$ $(m \leq n)$ of them are the lengths of connections from non-sink variables via the variable z. The rest of position-sets of other reverse binary atoms is denoted by $Rest_1$. The types of reverse predicates appearing in the sequences p are encoded in the multiset $\{i_1, \ldots, i_n\}$, and $Rest_2$ is the multiset of the types of the rest of reverse predicates existent in s. Then, the reverse factors are

$$pos^{rev}(s) = \{p_1, \ldots, p_n\} \cup Rest_1, type^{rev}(s) = \{i_1, \ldots, i_n\} \cup Rest_2.$$

Let i_k be the encoding of the predicate r from l in $type^{rev}(s)$ $(\exists k : 1 \leq k \leq n)$.

Recall that for two multisets $A, B \in \mathrm{M}(\mathbb{N})$, the strict order $>_{mul}$ is defined by

$$A >_{mul} B \Leftrightarrow \exists C, D \in \mathrm{M}(\mathbb{N}) : \emptyset \neq C \subseteq A, B = (A - C) \cup D, \forall d \in D : \exists c \in C : c > d.$$

Classes ($\{\mathsf{fstChild}, \mathsf{child}, \mathsf{child}^+\}$,$\mathsf{HR}^?$), $\mathsf{H/V}$($\mathsf{F}^?$,R). The reverse predicate r is removed.

$$type^{rev}(t) = (type^{rev}(s) - C) \cup D, C = \{i_k\}, D = \emptyset \Rightarrow type^{rev}(s) >_{mul} type^{rev}(t)$$
$$\Rightarrow s >^{rev}_{type\times pos} t.$$

Class $(\mathsf{HF,VR})^?$. The lengths of the m connections of non-sink variables to reverse predicates via the variable z are decreased by one, the connections via y do not change (y has the same connection to x), and x remains non-sink. Then,

$$\begin{aligned}
&type^{rev}(t) = type^{rev}(s), pos^{rev}(t) = (pos^{rev}(s) - C) \cup D,\\
&C = \{p_j \mid 1 \leq j \leq m\}, D = \{p_j - 1 \mid 1 \leq j \leq m\}\\
&\Rightarrow type^{rev}(s) = type^{rev}(t), pos^{rev}(s) >_{mul} pos^{rev}(t) \Rightarrow s >^{rev}_{type \times pos} t.
\end{aligned}$$

Classes $\mathsf{H/V(F,R^+)}$, $\mathsf{H/V(F,R)^+}$. In this case, $r = r_1 \vee r_2$. The lengths of the m connections of non-sink variables to reverse predicates via the variable z are decreased by one, the connections via y did not change. Note that the number of connections via x is doubled. However, in each created disjunct the position factor is decreased, as in the previous case. Moreover, in the second disjunct, there is one reverse predicate less. Then, it can be checked similarly to previous cases that

$$\begin{aligned}
&type^{rev}(s) = type^{rev}(e \wedge r_1), pos^{rev}(s) >_{mul} pos^{rev}(e \wedge r_1) \Rightarrow s >^{rev}_{type \times pos} e \wedge r_1\\
&type^{rev}(s) >_{mul} type^{rev}(e \wedge r_2) \Rightarrow s >^{rev}_{type \times pos} e \wedge r_2.
\end{aligned}$$

The order $>^{rev}_{type \times pos}$ holds also between $norm(s)$ and $norm(t)$, because

$$\begin{aligned}
type^{rev}(s) >_{mul} type^{rev}(s) \Rightarrow type^{rev}(norm(s)) >_{mul} type^{rev}(norm(t))\\
pos^{rev}(s) >_{mul} pos^{rev}(s) \Rightarrow pos^{rev}(norm(s)) >_{mul} pos^{rev}(norm(t)).
\end{aligned}$$

Both implications hold because the rule applications do not change the reverse factors $Rest_1$ and $Rest_2$, which refer also to all reverse predicates that appear once in s and t and are not reachable from x, y, or z, and appear also in several disjuncts of s and t after normalization. The reverse factors of all other reverse atoms are already considered for $s >^{rev}_{type \times pos} t$. □

Lemma 4.3.3 gives some rules (i.e., (4.10), (4.11), (4.12), (4.23), and (4.24)) where each rhs has more atoms than its lhs: $|lhs| > |rhs|$. This growing is not ad-hoc, and in fact, for a given lhs one can not give a rule with a size-smaller rhs that preserves the properties given in Propositions 4.3.2 and 4.3.3. This means that each rule of Lemma 4.3.3 is size-minimal. There exists, of course, other rules than those of Lemma 4.3.3, where rhs has the size of its lhs or less, but these rules do not preserve all the aforementioned properties. For example, an adaptation of Rule 4.1 so as to syntactically match the lhs of rules from Lemma 4.3.3 is: $f(\underline{x}, \underline{y}) \wedge r(\underline{y}, \underline{z}) \rightarrow f(\underline{x}, \underline{y}) \wedge r^{-1}(\underline{z}, \underline{y})$. In this case, the reverse predicate r from the lhs is replaced by its forward one r^{-1} in the rhs, but instances of $\underline{y}$ become multi-sink variables.

Theorem 4.3.1 (Size-minimality of rules under property set). *Rules* (4.5) *through* (4.24) *are size-minimal under the set of properties of Propositions 4.3.2 and 4.3.3. Furthermore, any other property-preserving rule is an extension of one Rule* (4.5) *through* (4.24) *with redundant formulas.*

Proof. The redexes of Rules (4.10), (4.11), (4.12), (4.23), and (4.24) with interactions of type $\mathsf{H/V(F,R^+)}$ and $\mathsf{H/V(F,R)^+}$, are the only ones that have the rhs size-bigger than the lhs. More specifically, the rhs has double the amount of binary atoms of the lhs. For the other rules, the instances of the lhs and rhs have the same size or less, and this size is minimal, because all three variables that appear in rhs can be interconnected with at least two binary atoms.

We consider an instance of the Rule (4.12) under the substitution $s = \{\underline{x} \mapsto x, \underline{y} \mapsto y, \underline{z} \mapsto z\}$ with the lhs l and the rhs r (the case of (4.24) is dual, and the others similar)

$$\mathsf{child}^+(x,y) \wedge \mathsf{par}^+(y,z) \equiv \mathsf{child}^+(x,y) \wedge \mathsf{par}^+(x,z) \vee \mathsf{child}^*(x,z) \wedge \mathsf{child}^+(z,y)$$

Note that the size of $l = s(lhs)$ ($r = s(rhs)$) is the size of lhs (rhs), because the substitution s instantiates here LGQ$^\rightarrow$ variables to LGQ variables.

We conduct a proof by contradiction, i.e., we assume there exists a right-hand side r' with fewer binary atoms than r.

The left-hand side l of the rule instance is a disjunct of only vertical formulas, thus the nodes matched by all three variables are along a path such that the node $s(y)$ has as ancestors nodes $s(x)$ and $s(z)$ in any order. Hence, there can be two possibilities to arrange the matched nodes along the path (from root to leaf):

$$(s(x), s(z), s(y)) \text{ and } (s(z), s(x), s(y))$$

r' preserves all three variables, hence it has at least two binary atoms. The binary predicates on the images of variables must be only vertical also in r', i.e., fstChild, child, par, their transitive and reflexive transitive closures.

We argue next that only closure formulas can be used in r'. Indeed, the difference of tree levels of nodes matched by all three variables varies from one (in case of fstChild and child) and the maximum depth of the tree instance. This depth is not known beforehand, and therefore also the number of child (and also fstChild and par) predicates necessary to relate the nodes $s(x)$, $s(z)$, and $s(y)$.

Now, r' could be $\mathsf{child}^+(x,z) \wedge \mathsf{child}^+(z,y)$, which preserves the properties, but $\mathcal{LF}[\![r']\!] \subseteq \mathcal{LF}[\![l]\!]$, thus it is not sufficient. r' can not be $\mathsf{child}^+(z,x) \wedge \mathsf{child}^+(x,y)$ because it does not preserve the non-sink type of x. It can be seen that any other combination of two atoms with vertical closure predicates does not suffice, because r' is not equivalent to l, and even more because some properties are eventually invalidated.

Therefore, r' has size bigger than l.

Adding a third vertical closure atom to r' implies that either a DAG, a tree, or a path is created. For the DAG case, either each variable appears as source and sink (and then x is not anymore non-sink), or one variable becomes multi-sink (which invalidates a property). For the tree case, the properties are satisfied, but all three variables must not necessarily match along a path (contradicts the semantics). For the path case, the new (third) added atom is a self-atom, which makes its addition useless. A disjunction of disjuncts can not be created with three binary atoms, because each disjunct must contain all three variables, hence minimum two binary atoms.

Therefore, r' has at least the size of r, which concludes the first part of the proof. We show now that r' must include rhs.

Adding a fourth binary atom with a vertical closure predicate to r' derives all cases of the previous step (which are all unsatisfactory) and the case of a disjunction of two disjuncts, each with two binary atoms with vertical closure predicates: $r' = r_1 \vee r_2$. Each predicate encodes one of the two possible cases to have the nodes matched by the variables aligned along a path. For the first case (x, z, y), only $\alpha_1(x, z) \wedge \alpha_2(z, y)$ (with $\alpha_1, \alpha_2 \in \{\mathsf{child}^+, \mathsf{child}^*\}$) preserves the properties. For the second case (z, x, y), only $\alpha_3(x, z) \wedge \alpha_4(x, y)$ (with $\alpha_3 \in \{\mathsf{par}^+, \mathsf{par}^*\}$ and $\alpha_4 \in \{\mathsf{child}^+, \mathsf{child}^*\}$) preserves the properties. However, the following predicates hold (see l): $\mathsf{child}^+(x, y)$ and $\mathsf{child}^+(z, y)$. The only remained possibilities for r' are:

$$\begin{array}{ll}
Case\ 1: & \mathsf{child}^+(x,z) \wedge \mathsf{child}^+(z,y) \vee \mathsf{par}^+(x,z) \wedge \mathsf{child}^+(x,y) \\
Case\ 2: & \mathsf{child}^*(x,z) \wedge \mathsf{child}^+(z,y) \vee \mathsf{par}^*(x,z) \wedge \mathsf{child}^+(x,y) \\
Case\ 3: & \mathsf{child}^*(x,z) \wedge \mathsf{child}^+(z,y) \vee \mathsf{par}^+(x,z) \wedge \mathsf{child}^+(x,y) \\
Case\ 4: & \mathsf{child}^+(x,z) \wedge \mathsf{child}^+(z,y) \vee \mathsf{par}^*(x,z) \wedge \mathsf{child}^+(x,y)
\end{array}$$

The third case expresses exactly r. The first case does not cover the possibility of $x = z$ and is excluded. The second case covers the aforementioned possibility in both disjuncts, hence redundantly, and therefore extends r. The fourth r' is equivalent to r, but the possibility $x = z$ appears with the reverse atom $\mathsf{par}^*(x, z)$ (rather than with the forward atom). In this case, the property $>^{rev}_{type \times pos}$-decrease is violated:

$$type^{rev}(r') >_{mul} type^{rev}(l) \Rightarrow r' >^{rev}_{type \times pos} l.$$

r' can consist of r and new atoms that keep it still equivalent to l, e.g., by adding an already existing formula. Some of these extensions are subject to duplicate elimination and navigation compaction, as formalized in Lemma 4.3.6. ☐

4.3.3 Rules removing DAG-Structure

This section considers an equivalence-preserving rule of simple forward DAG formulas made out of two binary atoms having the same sink variable, and path formulas created by replacing one of the two binary atoms by its reverse. Based on this rule and other rules of this section, Section 4.4 shows how any LGQ formula can be rewritten to an equivalent forward LGQ forest formula.

Lemma 4.3.4 (Rule removing DAG-Structure). *Applying the rewrite rule*

$$fwd_1(\underline{x}, \underline{y}) \wedge fwd_2(\underline{z}, \underline{y}) \rightarrow fwd_1(\underline{x}, \underline{y}) \wedge fwd_2^{-1}(\underline{y}, \underline{z}). \quad (4.25)$$

to an LGQ formula s yields an equivalent LGQ formula t.

Proof. For an instance $l \rightarrow r$ of the above rule under a substitution $\sigma = \{\underline{x} \mapsto x, \underline{y} \mapsto y\}$, we show that (1) $l \equiv r$, and (2) $s \equiv s[r/l]$.

The first part of the proof follows from the observation that $\alpha(x,y) \equiv \alpha^{-1}(y,x)$, for any LGQ binary predicate α. Then,

$$\mathcal{LF}[\![fwd_1(x,y) \wedge fwd_2(z,y)]\!](\beta) = \mathcal{LF}[\![fwd_1(x,y) \wedge fwd_2^{-1}(y,z)]\!](\beta).$$

The second part of the proof follows from Proposition 3.3.1, with the condition that the subformulas of s and t obtained by removing l, respectively r, do not contain variables appearing only in r, respectively l, and not in the other one. Indeed, both l and r have the same variables. □

Remark 4.3.3. The rhs of Rule (4.25) 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 variables that are instances of $\underline{z}$. □

Example 4.3.5. Consider the journal archive example of Section 2.2 and the tree instance of Figure 2.1. The LGQ DAG formula

$$Q_5(v_3) \ \leftarrow \ \mathsf{root}(v_0) \wedge \mathsf{child}(v_0,v_1) \wedge \mathsf{child}(v_1,v_2) \wedge \mathsf{child}(v_3,v_2) \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2)$$

that selects the parent node of an editor node that is child of a journal node, which is in its turn a child node of the root. For the given tree, Q_5 selects the journal node.

According to Rule (4.1) and Proposition 3.3.1, Q_5 is equivalent to

$$FQ_5(v_3) \ \leftarrow \ \mathsf{root}(v_0) \wedge \mathsf{child}(v_0,v_1) \wedge \mathsf{child}(v_1,v_2) \wedge \mathsf{par}(v_2,v_3) \wedge \mathsf{journal}(v_1) \wedge \mathsf{editor}(v_2).$$

For FQ_5 there is an equivalent XPath query, but for Q_5 there is only an equivalent XPath query with equality based on node-identity (`==`).

□

The application of Rule (4.25) ensures that the LGQ formulas containing instances of the lhs of that rule have a greater DAG factor than the result of such a rule application.

Proposition 4.3.4 ($>^{dag}_{type}$-Decrease). *An application of Rule* (4.25) *to an LGQ formula s containing a redex of that rule yields an LGQ formula t that has a smaller type factor than s: $s >^{dag}_{type} t$.*

Proof. Let $\sigma = \{\underline{x} \mapsto x, \underline{y} \mapsto y, \underline{z} \mapsto z\}$. We consider the DAG type factor for s

$$type^{dag}(s) = \{i_1, \ldots, i_n\}, \forall 1 \leq j \leq n : i_j > 1.$$

Let i_k ($\exists k : 1 \leq k \leq n$) be the forward sink-arity of y in s (i.e., the number of forward binary atoms that have y as sink and that appear in a disjunct of s).

As ensured by Rule (4.25), the variables x and z have the same forward sink-arities in t and s. Also, the forward sink-arity i_k of y is decreased by one in t. Thus, if y is forward

2-sink in s (i.e., $i_k = 2$), then it is not anymore forward multi-sink in t, otherwise y remains forward multi-sink, but with decreased forward sink-arity.

Recall that for two multisets $A, B \in \mathrm{M}(\mathbb{N})$, the strict order $>_{mul}$ is defined by

$$A >_{mul} B \Leftrightarrow \exists C, D \in \mathrm{M}(\mathbb{N}) : \emptyset \neq C \subseteq A \wedge B = (A - C) \cup D \wedge \forall d \in D : \exists c \in C : c > d.$$

Then, we get

$$type^{dag}(t) = (type^{dag}(s) - C) \cup D \wedge C = \{i_k\}, D = \begin{cases} \emptyset & , i_k = 2 \\ i_k - 1 & , \text{otherwise} \end{cases}$$
$$\Rightarrow type^{dag}(s) >_{mul} type^{dag}(t) \Rightarrow s >^{dag}_{type} t.$$

□

4.3.4 Rules for LGQ Normalization

This section describes basic rules for bringing LGQ formulas in disjunctive normal form, where additionally all useless parentheses are dropped.

Lemma 4.3.5 (Rules for DNF Normalization). *The application of any of the following rules to an LGQ formula s yields an equivalent LGQ formula t.*

$$X \wedge (Y \vee Z) \to X \wedge Y \vee X \wedge Z \quad (4.26)$$
$$X \vee (Y \vee Z) \to X \vee Y \vee Z \quad (4.27)$$
$$(Y \wedge Z) \to Y \wedge Z. \quad (4.28)$$

Proof. The first rule is due to the distributivity of $\wedge$ over $\vee$, the second rule is due to the associativity of $\vee$, and the third rule is due to the precedence of $\wedge$ over $\vee$. □

The following proposition states that the rules of Lemma 4.3.5 ensure an order, denoted $>_{dnf}$, between LGQ formulas s and t, where t is a contraction of s. The order $>_{dnf}$ is derived from the order on multisets of natural numbers representing the amount of parentheses that nest each atom in LGQ formulas.

Proposition 4.3.5 ($>_{dnf}$-Decrease). *The application of any rule of Lemma 4.3.5 to an LGQ formula s containing a redex of that rule yields a LGQ formula t with the number of parentheses that nest each atom less than for s:* $s >_{dnf} t$.

Proof. Consider $\phi_{()}$ a function that computes the multiset of numbers representing the amount of parentheses that nest each atom in a given LGQ formula. Consider $\phi_{()}(s) = \{i_1, \cdots, i_m\}$, where $\{i_j, \cdots, i_k\} \subseteq \phi_{()}(s)$ is the multiset of the numbers of parentheses that nest each atom in Y and Z. By inspecting the rules of Lemma 4.3.5, it follows

$$\phi_{()}(t) = (\phi_{()}(s) - \{i_j, \cdots, i_k\}) \cup \{i_j - 1, \cdots, i_k - 1\} \Leftrightarrow \phi_{()}(t) <_{mul} \phi_{()}(s) \Leftrightarrow s >_{dnf} t.$$

□

4.3.5 Rules for LGQ Simplification

LGQ formulas can be unsatisfiable or can contain redundancies. An unsatisfiable formula is, e.g., $\mathsf{child}(x, x)$, whereas a formula with redundancies is $\mathsf{child}(x, y) \wedge \mathsf{child}^+(x, y)$. The former formula is unsatisfiable because no node is the child of itself. The latter formula states that, for a substitution s consistent with that formula and a tree, both predicates child and child^+ hold on the nodes $s(x)$ and $s(y)$. Because the predicate child^+ is the transitive closure of child, it is clear that $\mathsf{child}^+(s(x), s(y))$ holds if $\mathsf{child}(s(x), s(y))$ holds. In such cases, it would be desirable to rewrite the formula to its simpler equivalent $\mathsf{child}(s(x), s(y))$.

Such redundancies may not be so obvious. Rewriting formulas with redundancies using the rules presented in this chapter can, however, discover and eliminate such redundancies, by reducing complex cases to trivial ones, as given below in Lemma 4.3.6. Example 4.4.2 shows in the next section such cases. Towards the goal of rewriting arbitrary LGQ graphs into forward LGQ forests, the elimination of some redundancies is a must, in order to ensure there are no multi-sink variables. Note that in the above formula with redundancies, y is a multi-sink variable, thus a forest LGQ formula can not have such variables.

This section introduces simplification rules that help in the process of rewriting by removing redundancies and detect unsatisfiability.

Lemma 4.3.6 (General Rules). *Consider two nodetests $\mathsf{nodetest}_1$, $\mathsf{nodetest}_2$, such that for any node n test$(n, \mathsf{nodetest}_1) \neq$ test$(n, \mathsf{nodetest}_2)$, and the LGQ predicates $r \in \mathsf{R} \cup \mathsf{R}^+$, $f \in \mathsf{F} \cup \mathsf{F}^+$, $vh \in \mathsf{V} \cup \mathsf{V}^+ \cup \mathsf{H} \cup \mathsf{H}^+$. Then, the application of any of the following rules to an LGQ formula s containing a redex of that rule yields an equivalent LGQ formula t.*

(Un)satisfiability Detection

$$vh(\underline{x}, \underline{x}) \to \bot \tag{4.29}$$

$$\mathsf{nodetest}_1(\underline{x}) \wedge \mathsf{nodetest}_2(\underline{x}) \to \bot \tag{4.30}$$

$$\mathsf{self}(\underline{x}, \underline{x}) \to \top \tag{4.31}$$

$$\mathsf{root}(\underline{x}) \wedge r(\underline{x}, \underline{y}) \to \bot \tag{4.32}$$

$$\mathsf{root}(\underline{x}) \wedge f(\underline{y}, \underline{x}) \to \bot \tag{4.33}$$

(Un)satisfiability Propagation

$$X \wedge \bot \to \bot \tag{4.34}$$

$$X \vee \bot \to X \tag{4.35}$$

$$X \wedge \top \to X \tag{4.36}$$

$$X \vee \top \to \top \tag{4.37}$$

Duplicate elimination

$$X \wedge X \to X \tag{4.38}$$

$$X \vee X \to X. \tag{4.39}$$

Remark 4.3.4. Several other rules for navigation compaction and (un)satisfiability detection can be derived using the already existing rules:

$$\alpha_1(\underline{x},\underline{y}) \wedge \alpha_2(\underline{x},\underline{y}) \rightarrow \alpha_1(\underline{x},\underline{y}) \tag{4.40}$$

$$\mathit{refl}(\underline{x},\underline{x}) \rightarrow \top \tag{4.41}$$

$$v(\underline{x},\underline{y}) \wedge h(\underline{x},\underline{y}) \rightarrow \bot \tag{4.42}$$

$$v(\underline{x},\underline{y}) \wedge h(\underline{y},\underline{x}) \rightarrow \bot \tag{4.43}$$

$$vh(\underline{x},\underline{y}) \wedge vh(\underline{y},\underline{x}) \rightarrow \bot \tag{4.44}$$

where $(\alpha_1, \alpha_2) \in \{(\mathsf{self}, \mathsf{child}^*), (\mathsf{self}, \mathsf{nextSibl}^*), (\mathsf{child}, \mathsf{child}^+), (\mathsf{child}, \mathsf{child}^*), (\mathsf{child}^+, \mathsf{child}^*), (\mathsf{nextSibl}, \mathsf{nextSibl}^+), (\mathsf{nextSibl}, \mathsf{nextSibl}^*), (\mathsf{nextSibl}^+, \mathsf{nextSibl}^*)\}$, $\mathit{refl} \in \mathsf{F}^* \cup \mathsf{R}^*$, $v \in \mathsf{V} \cup \mathsf{V}^+$, $h \in \mathsf{H} \cup \mathsf{H}^+$, and $vh \in \mathsf{V} \cup \mathsf{V}^+ \cup \mathsf{H} \cup \mathsf{H}^+$.

The first rule states that if the predicates α_1 and α_2 are applied to the same variables and α_1 is more specific than α_2, then their conjunction can be simplified to the α_1-atom. The other rules are self-explanatory. We show next how the first three rules can be derived from the existing ones.

$$\mathsf{child}(x,y) \wedge \mathsf{child}^+(x,y) \overset{(4.25)}{\rightarrow} \mathsf{child}(x,y) \wedge \mathsf{par}^+(y,x)$$
$$\overset{(4.11)}{\rightarrow} \mathsf{child}(x,y) \wedge \mathsf{par}^+(x,x) \vee \mathsf{child}(x,y) \wedge \mathsf{self}(x,x) \overset{(4.29)}{\rightarrow} \mathsf{child}(x,y) \wedge \bot \vee \mathsf{child}(x,y) \wedge \mathsf{self}(x,x)$$
$$\overset{(4.34)}{\rightarrow} \bot \vee \mathsf{child}(x,y) \wedge \mathsf{self}(x,x) \overset{(4.35)}{\rightarrow} \mathsf{child}(x,y) \wedge \mathsf{self}(x,x) \overset{(4.31)}{\rightarrow} \mathsf{child}(x,y) \wedge \top \overset{(4.36)}{\rightarrow} \mathsf{child}(x,y).$$

The second rule can be derived as (consider $\mathit{refl} = \alpha^*$, $\alpha \in \mathsf{R} \cup \mathsf{F}$)

$$\mathit{refl}(\underline{x},\underline{x}) \overset{(4.4)}{\rightarrow} \mathsf{self}(\underline{x},\underline{x}) \vee \alpha^+(\underline{x},\underline{x}) \overset{(4.31)}{\rightarrow} \top \vee \alpha^+(\underline{x},\underline{x}) \overset{(4.37)}{\rightarrow} \top.$$

The third rule can be derived as

$$v(\underline{x},\underline{y}) \wedge h(\underline{x},\underline{y}) \overset{(4.25)}{\rightarrow} v(\underline{x},\underline{y}) \wedge h^{-1}(\underline{y},\underline{x}) \rightarrow$$
$$\begin{cases} \bot & , \text{v} = \mathsf{fstChild} \\ v(\underline{x},\underline{x}) \wedge h(\underline{x},\underline{y}) \overset{(4.29)}{\rightarrow} \bot \wedge h(\underline{x},\underline{y}) \overset{(4.36)}{\rightarrow} \bot & , \text{v} \neq \mathsf{fstChild}. \end{cases}$$

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

$$\alpha_1(\underline{x},\underline{y}) \vee \alpha_2(\underline{x},\underline{y}) \rightarrow \alpha_2(\underline{x},\underline{y}) \tag{4.45}$$

□

The applications of the rules of Lemma 4.3.6 ensures that the LGQ formulas containing redexes of that rule have a greater size than their contractions. This ordering property can be specified using the strict order $>_{size}$ on formulas derived from the order $>$ on natural numbers representing the size of formulas: $s >_{size} t \Leftrightarrow |s| > |t|$.

Proposition 4.3.6 ($>_{size}$-Decrease). *The application of any rule of Lemma 4.3.5 to an LGQ formula s containing a redex of that rule yields a LGQ formula t that has a smaller size than the size of s, i.e., $s >_{size} t$.*

Proof. It can be easily seen by inspecting all rules of Lemma 4.3.6. Recall from Section 3.7 that the size of each atom is given by its arity, the size of each boolean connective is one, and the size of a formula is the sum of the sizes of its constituent atoms and connectives. □

4.4 Three Approaches to Rewrite LGQ to Forward LGQ Forests

Using the rewrite rules defined in Section 4.3, we can rewrite LGQ formulas representing the bodies of LGQ rules into forward LGQ formulas. These rewrite rules are distributed non-disjunctively in three sets that define three (term) rewriting systems:

- TRS_1 is the set containing Rule (4.1),
- TRS_2 is the set of Rules (4.4) through (4.24) and (4.26) through (4.39),
- TRS_3 includes TRS_2 and Rule (4.25).

Recall from Section 4.2 that all three rewriting systems contain also the AC-identities expressing the associativity and commutativity properties of $\wedge$, $\vee$, and self, and therefore they use AC-rewriting.

The rewrite relation $\rightarrow$ can be accompanied by an index specifying its corresponding rewriting system: e.g., $\rightarrow_1$ for TRS_1. However, if it is understood from the context, we spare the explicit writing of this index and avoid cluttering. For the same reason, $s \downarrow_E t$ is simply written as $s \downarrow t$ without explicitly mentioning the set of AC identities, which are always the same.

The properties of all three rewriting systems can be summarized as follows:

- What can the systems rewrite?
  TRS_1, TRS_2, and TRS_3 are sound and complete for LGQ formulas, i.e., each of them rewrites any LGQ formula into an equivalent forward LGQ formula.
- What is the relation between the type of input and of rewritten LGQ formulas?
  - TRS_1 rewrites LGQ single-join DAGs into forward LGQ single-join DAGs, and LGQ graphs into forward LGQ graphs;
  - TRS_2 rewrites LGQ forests into forward LGQ forests, LGQ single-join DAGs into LGQ single-join DAGs, and LGQ graphs into forward LGQ graphs;

  - TRS_3 rewrites LGQ graphs into forward LGQ forests; moreover, if the input formula contains only closure predicates, respectively non-closure predicates, then its equivalent rewriting contains also only closure predicates, respectively non-closure predicates.

- Do the systems terminate?
  TRS_1, TRS_2, and TRS_3 terminate and all of them employ terminating orders derived from multiset orders.

- Are the systems confluent, i.e., yield they the same rewritten forward LGQ formula regardless of the order of rules applications?

  - TRS_1 and TRS_3 are confluent for any input LGQ formula,
  - TRS_2 is confluent only for LGQ forests.

Among the enumerated properties, perhaps the most interesting one is that TRS_3 yields forward LGQ forests for input LGQ paths, forests, single-join DAGs, and (even cyclic) graphs. The intuition behind this result is that an LGQ formula satisfiable on tree data reflects the tree structure of the data, fact that renders forward LGQ forest formulas as sufficient to express structural constraints among nodes in a tree as general LGQ formulas do. As a corollary, it follows that forward XPath forest queries (or queries with filters and unions), which are another syntax for forward LGQ forests, are sufficient to express general LGQ queries. However, this nice property comes at the expense of using forward LGQ forests of size (possibly) exponential in the size of the equivalent general LGQ query. This observation on the LGQ and XPath expressiveness is used later in Chapter 5, where an efficient evaluation of LGQ forests is sufficient to cover the evaluation of arbitrary LGQ formulas. Also, a direct evaluation of arbitrary LGQ queries does not get around the exponential complexity, as shown independently by [74].

Another salient result is that LGQ forests (thus XPath queries) can be rewritten into forward LGQ single-join DAGs of size linear in the input forests. Because the evaluation of forward LGQ single-join DAGs has polynomial complexities (see later Chapter 5), it follows that one effective and efficient solution for the evaluation of XPath queries, particularly in a context where the XPath reverse axes are not desirable, is to first rewrite them into forward XPath queries, and then to evaluate the latter.

4.4.1 Rewriting Examples

This section considers two rewriting examples of one LGQ tree and one LGQ graph formulas into forward LGQ forest formulas, as illustrated in Figures 4.6 and 4.7. The thick edges in the digraph representations of formulas represent the predicates that are considered next in the rewriting process. Each thick (rewrite) arrow between the digraph representations of formulas is accompanied by the reference to the rewrite rule to apply next.

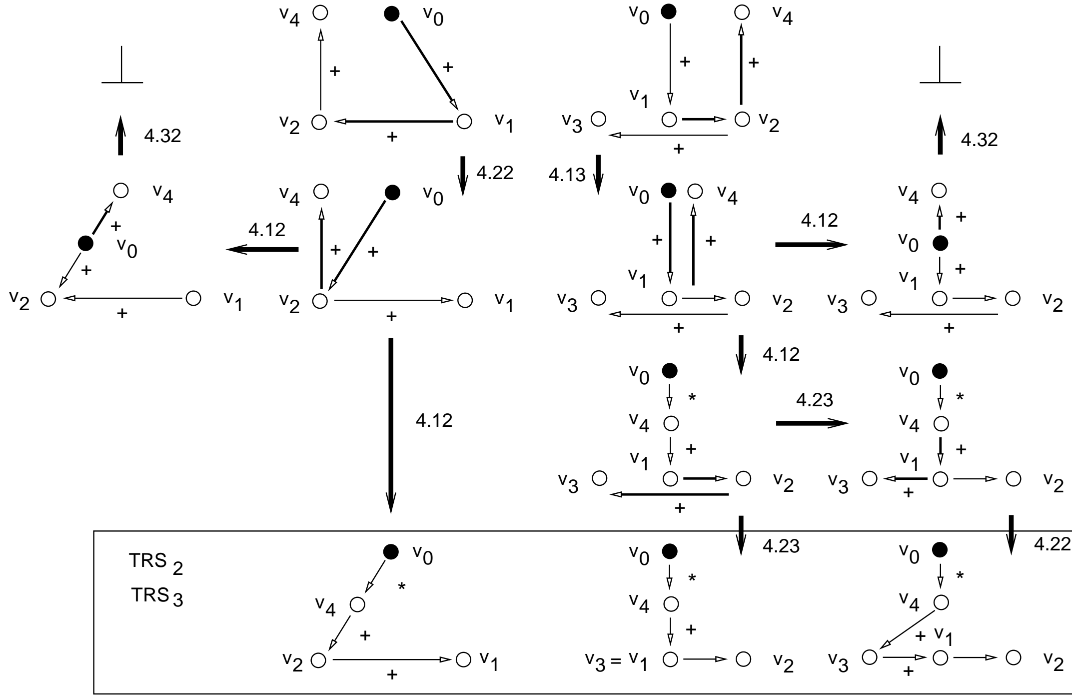


Figure 4.6: Rewriting of the LGQ tree formula of Example 4.4.1

Example 4.4.1. Consider the LGQ tree formula

$$\mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge (\mathsf{prevSibl}^+(v_1, v_2) \vee \mathsf{nextSibl}(v_1, v_2) \wedge \mathsf{prevSibl}^+(v_2, v_3)) \wedge \mathsf{par}^+(v_2, v_4)$$

Figure 4.6 shows how this LGQ tree can be rewritten into an equivalent forward formula, which is a forest of paths, by using the rewrite rules of TRS_2 (which are also of TRS_3).

The forward LGQ formula equivalent to e, represented graphically in the lower box, is

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

Note that it can be simplified by factoring out the first two atoms of each conjunct.

The initial LGQ tree formula can be rewritten using the rewrite rules of TRS_1 into the following single-join DAG formula (the reverse binary atoms are simply turned into their corresponding forward binary atoms with their sources reachable from a fresh non-sink variable):

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

□

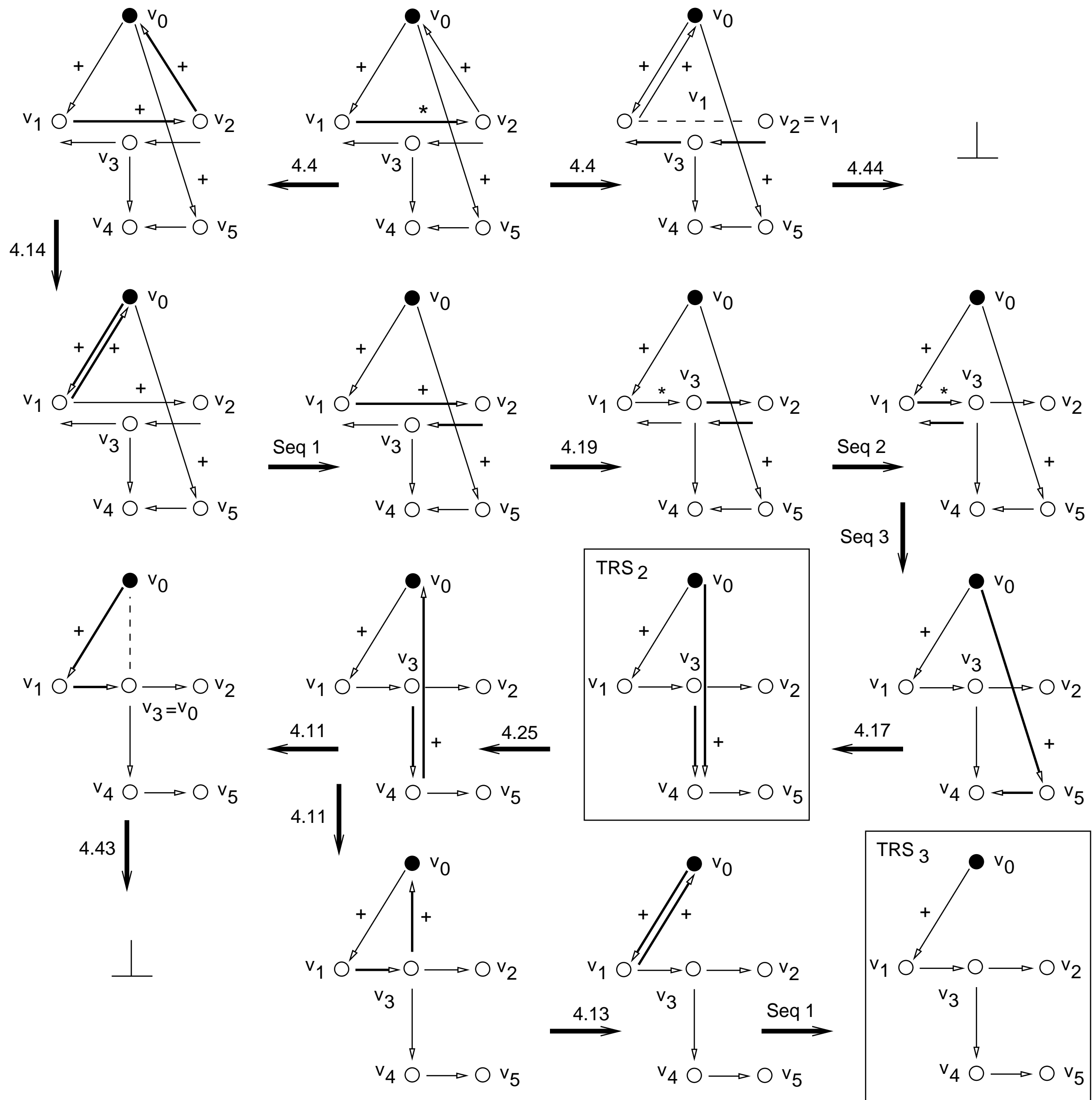


Figure 4.7: Rewriting of the LGQ graph formula of Example 4.4.2

Example 4.4.2. Consider the LGQ graph formula e

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

Figure 4.7 shows how e can be rewritten successively into equivalent forward formulas fe and fe'. Using the rewrite rules of TRS_2, we obtain fe (see box with label TRS_2 in Figure 4.7)

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

The formula fe is forward, but still a DAG. Using the additional rewrite rule of TRS_2, we obtain the formula fe' (see box with label TRS_3 in Figure 4.7)

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

that is forward and a forest (in this case even a tree). It is worth noting also that the formula fe' is variable-preserving minimal, i.e., the amount of binary atoms in fe' is exactly the number of its variables minus one. Also, the (non-trivial) redundancies of e, mainly derived from the repeated up-down and left-right navigations in the tree instance, 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 following compacted rules:

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	$\mathsf{nextSibl}^*(x, y) \wedge \mathsf{prevSibl}(y, x) \rightarrow \mathsf{nextSibl}(x, y).$

Such compacted rules were not added to the set of simplification identities of Lemma 4.3.6, because they can be derived from already existing rules, as discussed also in Remark 4.3.4. We show next how *Seq* 1 is obtained.

$$\begin{aligned}
\mathsf{child}^+(x, y) \wedge \mathsf{par}^+(y, x) &\overset{(4.12)}{\rightarrow} \mathsf{child}^+(x, y) \wedge \mathsf{par}^+(x, x) \vee \mathsf{child}^*(x, x) \wedge \mathsf{child}^+(x, y) \\
&\overset{(4.29)}{\rightarrow} \mathsf{child}^+(x, y) \wedge \bot \vee \mathsf{child}^*(x, x) \wedge \mathsf{child}^+(x, y) \\
&\overset{(4.34)}{\rightarrow} \bot \vee \mathsf{child}^*(x, x) \wedge \mathsf{child}^+(x, y) \\
&\overset{(4.35)}{\rightarrow} \mathsf{child}^*(x, x) \wedge \mathsf{child}^+(x, y) \\
&\overset{(4.41)}{\rightarrow} \top \wedge \mathsf{child}^+(x, y) \\
&\overset{(4.36)}{\rightarrow} \mathsf{child}^+(x, y).
\end{aligned}$$

The initial LGQ tree formula can be rewritten using the rewrite rules of TRS_1 into the following LGQ graph formula (the reverse binary atoms are simply turned into their corresponding forward binary atoms with their source reachable from a new non-sink variable, see lines 2,3, 4, and 6 below):

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

□

4.4.2 Soundness and Completeness

This section shows that all three rewriting systems TRS_i ($1 \leq i \leq 3$) are sound and complete for LGQ formulas, i.e., each of them rewrites any LGQ formula to an equivalent forward LGQ formula. Furthermore, it is shown how the structure of the rewritten LGQ formula relates to the structure of the input LGQ formula.

Theorem 4.4.1 (Soundness and Completeness of TRS_i). *All TRS_i are sound and complete for LGQ formulas:*

- (Soundness) *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*

$$\forall s, t \in LGQ : s \xrightarrow{*} t \Rightarrow s \equiv t$$
$$\forall s, t \in LGQ : s \rightarrow^{!} t \Rightarrow t \in LGQ[\mathsf{F}^?].$$

- (Completeness) *for any equivalent LGQ formulas s and forward t, TRS_i rewrites s to a normal form forward LGQ formula t' that is equivalent to t*

$$\forall s \in LGQ : \forall t \in LGQ[\mathsf{F}^?] : s \equiv t \Rightarrow \exists t' \in LGQ[\mathsf{F}^?] : s \rightarrow^{!} t', t \equiv t'.$$

Proof. For each instance $l \rightarrow r$ of the rules of Lemmas 4.3.1 through 4.3.6 that define all three rewriting systems TRS_i, it holds that $l \equiv r$, and $s \equiv t = s[r/l]$, i.e., any formula s and its contraction t are equivalent. Thus, s derives in one step equivalent LGQ formulas t: $s \rightarrow t \Rightarrow r \equiv t$. It follows directly by complete induction that $s \xrightarrow{*} t \Rightarrow s \equiv t$.

We show next for each TRS_i that if t is irreducible (i.e., normal form), then t is a forward LGQ formula. Recall that a derived formula t is irreducible if no subformula of it is an instance of the lhs of a rule.

TRS_1 consists in Rule (4.1) that rewrites any LGQ reverse binary atom to a forward LGQ equivalent formula. Hence, only a formula t without reverse binary atoms, i.e., forward, is irreducible.

TRS_2 consists in Rules (4.4) through (4.24) and (4.26) through (4.39). There are three cases concerning the type of binary atoms in s.

(A) If s is already a forward LGQ formula, then some simplification rules of Lemma 4.3.3 may apply, that yield an irreducible equivalent forward formula t, which is either $\bot$ or $\top$ formulas, or a forward formula, because no reverse binary atom appears on rhs but not on lhs of a rule.

(B) If s has only reverse binary atoms, then there must be connections from non-sink variables to each reverse binary atom, and for each non-sink variable v there is a $\mathsf{root}(v)$ unary predicate (recall that we consider only connected and absolute LGQ formulas). Applying repeatedly Rule (4.32) for unsatisfiability detection and Rules (4.34) and (4.35) for unsatisfiability propagation, the normal form is obtained as $t = \bot$.

(C) If s has reverse and forward binary atoms, then, along a connection sequence in s, there are either (i) forward predicates appearing *before* reverse predicates, (ii) or no forward predicate appears before reverse predicates. The latter case is treated as no forward binary atoms appear in s (see case B). In the former case, there are in s disjuncts of one forward and one reverse binary atom such that the sink of the former is the source of the later. Such disjuncts are rewritten, according to Lemma 4.3.3, either to (1) paths of two forward binary atoms, or to (2) trees where one branch is a forward, the other a reverse binary atom, or to (3) forests of trees as in (2) and paths as in (1).

As ensured by Theorem 4.4.2, all rewriting systems terminate, in particular also TRS_2. In cases (2) and (3), the connections to some reverse binary atoms have shorter sequences, but there can be more such connections. Also, some rules of Lemma 4.3.5 for bringing derivable terms into DNF may apply. Next, either case (A), or (B), or (C) applies.

Note that the rules of Lemma 4.3.6, without (4.32) and (4.33), are just simplification rules based on navigation compaction and unsatisfiability detection and propagation. Thus, such rules can be left out without jeopardizing the reachability of an equivalent forward normal form t.

TRS_3 extends TRS_2 with Rule (4.25) that rewrites conjunctions of two forward binary atoms with the same sink to a path of one forward and one reverse binary atom. Therefore, a forward formula that is normal form for TRS_2 is not a normal form for TRS_3, if it contains multi-sink variables. We consider the following cases where s contains a disjunct of two forward formulas, both having a variable y as sink:

(A) One formula is $\mathsf{root}(y)$ and the other is $fwd(x, y)$, or one formula is vertical and the other is horizontal, both having also the same source. Then, the whole disjunct is rewritten to $\bot$ and the multi-sink variable is eliminated, cf. Rules (4.29) through (4.33), and (4.34) through (4.37).

(B) Both formulas do not correspond to the above case. Then, one formula, say

$fwd(x, y)$, is rewritten to its reverse $fwd^{-1}(y, x)$. The obtained formula s' containing the reverse binary atom is then subject to rewriting using TRS_2, which is embedded in TRS_3. Most notably, the obtained normal form t' of TRS_2 does not contain additional multi-sink variables, as ensured by Proposition 4.3.2. The procedure can continue until all variables have sink-arity at most one. The normal form t for TRS_3 is then a forward formula, but also without multi-sink variables. □

We state next the relations between the structure of a LGQ formula and of its equivalent forward LGQ formula, as obtained by rewriting the former using TRS_i.

Proposition 4.4.1 (Yield of TRS_1). *TRS_1 rewrites any LGQ single-join DAG formula into a forward LGQ single-join DAG formula, and any LGQ graph formula into a forward LGQ graph formula.*

Proof. TRS_1 consists in Rule (4.1) with instances $rev(x, y) \equiv rev^{-1}(y, x) \wedge \mathsf{child}^+(y', y) \wedge \mathsf{root}(y')$. This rule ensures that the (multi-)sink y of the reverse atom in lhs remains (multi-)sink also in t, with one connection directly from the fresh non-sink variable y'. Also, if x is a 1-sink in s, then it becomes 2-sink in t.

Let s and t be the input, respectively the output, formulas. It follows that

(A) if s is a forest formula (or one of its subcases tree and path), i.e., it has no multi-sink variables, then t is a single-join DAG formula with as many 2-sink variables (like x) as reverse binary atoms in s.

(B) if s is a single-join DAG formula, then t is a single-join DAG formula with at least as many multi-sink variables as there are in s, because y keeps its sink-arity, and further x can become 2-sink (if it is not multi-sink in s).

(C) if s is a graph formula, then t is a graph formula, because Rule (4.1) does not remove cycles.

□

Proposition 4.4.2 (Yield of TRS_2). *TRS_2 rewrites any LGQ forest formula into a forward LGQ forest formula, any LGQ single-join DAG formula into a LGQ single-join DAG formula, and any LGQ graph formula into a forward LGQ graph formula.*

Proof. TRS_2 consists in Rules (4.4) through (4.24) and (4.26) through (4.39). An important property of these rules is that for a formula s containing a redex of any rule, its equivalent contraction t is a forest formula, cf. Proposition 4.3.2, i.e., the sink/non-sink variables from s remain sink/non-sink in t. It follows that

(A) if s is a forest formula (or one of its subcases tree and path), 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 formula, then t is a single-join DAG formula with the same multi-sink variables as there are in s.

(C) if s is a graph formula, then t is in general a graph formula, because multi-sinks and cycles are not necessarily removed via rewriting with TRS_2.

□

Proposition 4.4.3 (Yield of TRS_3). *TRS_3 rewrites any LGQ graph formula into a forward LGQ forest formula. Moreover, if the input formula contains only closure predicates, respectively non-closure predicates, then its equivalent rewriting contains also only closure predicates, respectively non-closure predicates.*

Proof. TRS_3 extends TRS_2 with Rule (4.25). As shown in proof of Theorem 4.4.1, the normal form obtained using TRS_3 does not contain multi-sink variables, hence it is a forest (or one of its simpler cases of trees, paths, $\perp$, or $\top$).

The application of any rule of TRS_3 yields for redexes without closure predicates always contractions without closure predicates. Using complete induction over the (finite) number of rule applications, it follows that also the (forward) normal forms do not have closure predicates. The case of redexes containing only closure predicates is similar. □

The same rewriting property of fragment closedness is not ensured by TRS_1 for input formulas without closure predicates, because TRS_1 yields rewritings containing as many child^* predicates as reverse predicates in the input formulas. Also, TRS_2 has this property up to the forest restriction or rewritings equivalent to graph formulas.

From Proposition 4.4.3 it follows also that the LGQ fragments containing formulas only with closure predicates, respectively without closure predicates, are as expressive as their forward forest subfragments.

Proposition 4.4.4. $\mathsf{LGQ[F{\cup}R]} = \mathsf{LGQ[F]\ Forests}$ *and* $\mathsf{LGQ[F^{+}{\cup}R^{+}]} = \mathsf{LGQ[F^{+}]\ Forests}$.

Proof. The right-hand sides of both equations are included in their left-hand sides

$$\mathsf{LGQ[F \cup R]} = \mathsf{LGQ[F]} \cup \mathsf{LGQ[R]} \supseteq \mathsf{LGQ[F]} \supseteq \mathsf{LGQ[F]\ Forests}$$
$$\mathsf{LGQ[F^{+} \cup R^{+}]} = \mathsf{LGQ[F^{+}]} \cup \mathsf{LGQ[R^{+}]} \supseteq \mathsf{LGQ[F^{+}]} \supseteq \mathsf{LGQ[F^{+}]\ Forests}.$$

Proposition 4.4.3 ensures that any LGQ formula can be rewritten using TRS_3 into a forward forest LGQ formula, both formulas either containing closure predicates or not:

$$\mathsf{LGQ[F \cup R]} \subseteq \mathsf{LGQ[F]\ Forests} \qquad \mathsf{LGQ[F^{+} \cup R^{+}]} \subseteq \mathsf{LGQ[F^{+}]} \supseteq \mathsf{LGQ[F^{+}]\ Forests}.$$

□

We conclude this inspection on the expressivity of LGQ with a remark on the ability of TRS_3 to detect and remove redundancies of input formulas. For any graph formula, its equivalent rewriting contains tree formulas as disjuncts, which are known to be variable-preserving minimal, cf. Proposition 3.5.1. This minimization property of TRS_3 reaches its apogee when rewriting graph formulas without closure predicates. In such a case, the forward normal form obtained for each input disjunct is a tree formula, thus rewriting graph formulas to variable-preserving minimal forest formulas, thus with size independent of the size of the input graph formulas and only dependent on the maximum number of the variables appearing in their disjuncts.

4.4.3 Termination

This section shows that all three rewriting systems TRS_i ($1 \leq i \leq 3$) terminate. For this, we employ the strict, well-founded, orders $>^{rev}_{type}$, $>^{rev}_{pos}$, $>^{dag}_{type}$, $>_{dnf}$, $>_{size}$, and lexicographic products of them, all defined on LGQ formulas.

Theorem 4.4.2 (Termination property of TRS$_{1,2,3}$). *All three rewrite systems, i.e., TRS$_1$, TRS$_2$, and TRS$_3$ terminate.*

Proof. The rewriting systems TRS_i are terminating if, for all LGQ formulas s and t, $s \rightarrow t$ implies $s >_i t$, with $>_i$ terminating (well-founded) orders. The formula t is one-step derivable from s.

As shown below, these orders $>_i$ are defined using the strict and terminating orders $>^{rev}_{type}$, $>^{rev}_{pos}$, $>^{dag}_{type}$, $>_{dnf}$, $>_{size}$, and their lexicographic products. The latter are terminating because they are embeddings into the strict and terminating order $>_{mul}$ on multisets over natural numbers, and $>$ on natural numbers. More precisely, the orders $>_i$ are:

TRS$_1$. $>_1$ is $>^{rev}_{type}$.
Proposition 4.3.1 ensures that the terminating order $>^{rev}_{type}$ holds between s and t.

TRS$_2$. $>_2$ is $>^{rev}_{type} \times >^{rev}_{pos} \times >_{dnf} \times >_{size}$.
For applications of rules of Lemmas 4.3.3, the terminating order $>^{rev}_{type} \times >^{rev}_{pos}$ is ensured by Proposition 4.3.3 between $norm(s)$ and $norm(t)$.

For applications of rules of Lemma 4.3.5, the terminating order $>_{dnf}$ between s and t is ensured by Proposition 4.3.5, and also such rule applications preserve the order $>^{rev}_{type} \times >^{rev}_{pos}$ between formulas $norm(s)$ and $norm(t)$, i.e., such rule applications do not change the reverse factors of $norm(s)$ and $norm(t)$.

For applications of rules of Lemma 4.3.6 the terminating order $>_{size}$ is ensured by Proposition 4.3.6, and at the same time such rule applications preserve the orders $>^{rev}_{type} \times >^{rev}_{pos}$ between $norm(s)$ and $norm(t)$, and $>_{dnf}$ between s and t.

Then, the lexicographic product of the later and the former (in this product order) is also terminating: $>^{rev}_{type} \times >^{rev}_{pos} \times >_{dnf} \times >_{size}$.

TRS$_3$. $>_3$ is $>^{dag}_{type} \times >^{rev}_{type} \times >^{rev}_{pos} \times >_{dnf} \times >_{size}$.
TRS$_3$ has a single more rule in addition to TRS$_2$, namely Rule (4.25) and for its applications the terminating order $>^{dag}_{type}$ between s and t is ensured by Proposition 4.3.4. At the same time, all rules of TRS$_2$ preserve the terminating order $>^{dag}_{type}$ between s and t. Then, the lexicographic product of the former and the later (in this product order) is terminating.

□

4.4.4 Confluence

Because all three rewriting systems terminate, cf. Theorem 4.4.2, showing confluence can be boiled down to showing local confluence [122], a much easier task. The following theorem states the local confluence properties of each of our three rewriting systems.

Theorem 4.4.3 (Local confluence of TRS$_1$). *The term rewriting systems TRS$_1$ and TRS$_3$ are locally confluent for any input LGQ formulas, whereas TRS$_2$ is locally confluent for input LGQ forests, and not confluent for input LGQ DAGs and graphs.*

Proof. The proof is given in the Appendix. □

4.5 Complexity Analysis

Discussion on the complexity of AC-matching for LGQ$^{\rightarrow}$ rules

AC-matching (and AC-unification) is NP-complete in general: the number of substitutions (unifiers) for any two terms is finitary, but it can be exponential in the size of the terms, see, e.g., [103]. In the particular case of TRS$_{1,2,3}$, we show next that AC-matching is polynomial. The intuition for this result is that the rewrite rules restrict severly the matchings of their contained variables.

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

1. a single binary LGQ$^{\rightarrow}$ atom, where its variables range over LGQ variables,
2. an LGQ$^{\rightarrow}$ path made out of two atoms with different function symbols, where additionally all variables range over LGQ variables,
3. an LGQ$^{\rightarrow}$ formula containing only two or three variables ranging over LGQ formulas.

In the first case, the AC-matching problem is reducible to syntactic matching, which is linear in the size of both participating terms. In the second case, the AC-matching problem is reducible to syntactic matching of the variables from each of the two atoms, followed by checking whether the variable appearing in both atoms matched the same constant. This procedure takes at most quadratic time in the term to match. In the third case, the LGQ$^{\rightarrow}$ variables can match any subterms of the LGQ formula. The number of all possible combinations of matchings of these variables is exponential in their number, where the basis is the size of the term to match. Because the number of the LGQ$^{\rightarrow}$ variables is bound by a constant (less than or equal three), 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 powerful and elegant LGQ representation of formulas and rules could have been traded for more compact representations. For example, the rewritings could be done on the digraph representations of LGQ formulas, and applications of our rules can be performed in linear time as, e.g., matchings of paths of length two in graphs. The quadratic time

of the second case would be needed then only once for the construction of the digraph representation of the LGQ formula to rewrite.

The complexity factor of the AC-matching algorithm used by our rewriting systems do not appear in the following complexity results.

Complexity of LGQ formula rewriting using $\mathbf{TRS}_{1,2,3}$

We conduct the complexity study of rewriting LGQ formulas using $\text{TRS}_{1,2,3}$ with the following declared objectives:

- the time and space complexity for rewriting LGQ formulas using each TRS_i,
- the size of the rewritten LGQ forward formula,
- LGQ fragments for which some TRS_i have the above complexities better.

For an input LGQ formula s, the following parameters can influence the above complexities of its rewriting:

- its size $|s|$, and the size of its DNF normalization $|norm(s)|$ where additionally with reflexive transitive closure formulas rewritten into disjunctions of self and transitive closure atoms (see Rule (4.4)),
- the reverse factors $type^{rev}(s)$, $type^{rev}(norm(s))$ and $pos^{rev}(s)$, and the DAG factor $type^{dag}(norm(s))$,
- the variable connection relation $\overset{p}{\leadsto}_s$ with connection sequence p.

The results regarding the above complexities can be summarized as follows:

- TRS_1 rewrites an LGQ formula s into an equivalent forward LGQ formula in linear time and logarithmic space, and the size of t is linear in the size of s, more precisely $|t| = |s| + 2 \times |type^{rev}(s)|$.
- In general, TRS_2 needs time and space, and generates equivalent forward LGQ formula t with a number of disjuncts exponential in the number of reverse binary atoms in the normalized formula $norm(s)$, i.e., in $|type^{rev}(norm(s))|$; also, the size of each disjunct of t is linear in $|s|$. As ensured by Theorem 4.3.1, the exponentiality behaviour of rewriting using TRS_2 can not be avoided, and the output of TRS_2 is optimal.
- TRS_3 adds to the exponential factor from the complexity results of TRS_2 the sum of forward sink-arities of variables in the normalized formula $norm(s)$. However, in contrast to TRS_2, each disjunct in t is a tree (and hence is variable-preserving minimal) and its size is then linear in the maximum number of variables in a disjunct of s, which is notably independent on the size of s, and can be much smaller than the number of binary atoms of that disjunct.

- For LGQ formulas, where each connection sequence has neither vertical closure reverse predicates after vertical forward predicates, nor horizontal closure predicates immediately after horizontal reverse predicates, TRS_2 needs time linear and space logarithmic in the normalized size $|norm(s)|$ of s, and TRS_2 generates rewritten formula t of size at most the normalized size $|norm(s)|$. The same complexities are achieved also by TRS_3 if additionally there is no connection sequence with vertical closure forward predicates, having as sink a variable with a forward sink-arity greater than one, after vertical forward predicates.

- Finally, an alternative technique is described for finding the upper bound on the number of tree formulas in t by means of orders on the variables of s.

In the rest of this section, the aforementioned claims are proven.

Theorem 4.5.1 (Complexities for TRS_1). *TRS_1 rewrites any LGQ formula s into an equivalent forward LGQ formula t in linear time and logarithmic space, and $|t| = |s| + 2 \times |type^{rev}(s)|$.*

Proof. TRS_1 consists of Rule (4.1) that rewrites each reverse binary atom into two forward binary atoms and a unary formula ($root$). The size $|type^{rev}(s)|$ of the reverse type factor gives the number of reverse binary atoms in s. The size of t can be then obtained trivially as $|t| = |s|+2\times|type^{rev}(s)|$. Note that it is not necessary to normalize s, for a rewrite works locally on a reverse binary atom. TRS_1 traverses the entire formula s and needs to store just a pointer to the current binary atom. Therefore, TRS_1 needs only extra logarithmic space. □

The complexities for TRS_2 depend highly on the kind of binary atoms existent in the formula s to rewrite, and on their connections. In order to analyze such complexities, we conduct a study on the form of a connection sequence p from a non-sink variable. Recall from Definition 3.7.1 that a connection from a variable a to a variable b with connection sequence p in an formula s exists, written $a \overset{p}{\rightsquigarrow}_s b$, if the binary atom $p(a, b)$ exists in e, or if there are variable connections $a \overset{p_1}{\rightsquigarrow}_s v \overset{p_2}{\rightsquigarrow}_s b$ with $p = p_1.p_2$. In the following, we consider p having m reverse predicates (bounded by $type^{rev}(s)$), each of them having n_i forward predicates appearing before them in p ($1 \leq i \leq m$). We consider also these reverse predicates ascendingly ordered by their position in the connection sequence p, thus the reverse predicate with a greater index appears in p before a reverse predicate with a smaller index. Then, the number of forward predicates n_i appearing before the reverse predicate i is greater than or equal to the number of forward predicates n_{i+1} appearing before the reverse predicate $i+1$. This means also that the number of (forward and reverse) predicates appearing before the reverse binary atom i is n_i+m-i, and this number belongs to the reverse position-set of i, hence to the reverse position factor $pos^{rev}(s)$ of s.

The number of disjuncts in t obtained by rewriting one disjunct in s using TRS_2 depends on the type of interactions between reverse and forward binary atoms in s, as given in Figure 4.5, and it is computed by a family of functions $\{\phi_i \mid 1 \leq i \leq m\}$ for each class

of interactions between forward and reverse binary atoms. A function $\phi_i(n_i, n_{i-1}, \ldots, n_1)$ $(1 \leq i \leq m)$ has i parameters, i to 1, where parameter j represents the number of forward predicates n_j appearing before the reverse predicate j in p, and simulates the rewrite rule applications for different interaction classes: the time complexity of rewriting s using TRS_2 is then the number of computation steps required to compute ϕ_m, and the value computed by ϕ_m is the number of disjuncts in t obtained by rewriting one disjunct in s. Note that the average size of each such disjunct in t is bounded by $|s|$.

A family of functions $\{\phi_i \mid 1 \leq i \leq m\}$ is defined next for each class of interactions between forward and reverse binary atoms. Note that these functions simulate a rewriting sequence where rules are applied in such an order so that the first reverse binary atom is always considered first. That is, we consider always the first interaction to be found in the connection sequence. There are, of course, other possible simulations. In fact, for every possible rewriting sequence one can define another family of functions simulating the rewriting of a given formula. All such families have to compute the same value, if the term rewriting system is confluent. However, the number of their computation steps may differ.

Definition 4.5.1. *For classes* $(\mathsf{VF},\mathsf{HR})^?$, $\mathsf{H/V}(\mathsf{F}^?,\mathsf{R})$ *the functions* ϕ_i *are defined by*

$$\phi_i(n_i, \ldots, n_1) = \begin{cases} \phi_{i-1}(n_{i-1}, \ldots, n_1) & , i > 1 \\ 0 & , i = 1 \text{ and for class } (\{\mathsf{fstChild}\},\mathsf{HR})^? \\ 1 & , i = 1 \text{ and for the other classes.} \end{cases}$$

This definition can be read also in terms of applications of rules corresponding to interactions between binary atoms vertical forward and horizontal reverse i (respectively forward and non-closure reverse i, both either vertical or horizontal): the effect of an interaction of such a forward and reverse binary atoms is that the reverse binary atom is removed $(n_i = 0)$.

Definition 4.5.2. *For the class* $(\mathsf{HF},\mathsf{VR})^?$*, the functions* ϕ_i *are defined 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 above definition reads in terms of rule applications as follows: the effect of an interaction of such binary atoms, i.e., forward and reverse i, is that the number of forward binary atoms is reduced by 1 for i and for all reverse binary atoms j that follow it in the connection sequence (i.e., $j < i$). When there is no forward binary atom for i $(n_i = 0)$, i.e., the reverse formula has been removed, the interaction of forward formulas and reverse binary atom $i - 1$ is considered.

Definition 4.5.3. *For the class* H/V(F,R$^+$) *the functions* ϕ_i *are defined by*

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

The above definition can be read in terms of rule applications as follows: the effect of an interaction of such binary atoms, i.e., forward and reverse i, is that two disjuncts are created. The first disjunct has still the reverse binary atom i, but, like in Definition 4.5.2, the number of forward binary atoms is reduced by 1 for i and for all reverse binary atoms j that follow it in the connection sequence (i.e., $j < i$). The second disjunct does not have the reverse binary atom i ($n_i = 0$), and, like in like in Definition 4.5.1, the number of forward binary atoms is reduced by 1 for all reverse binary atoms j that follow it in the connection sequence (i.e., $j < i$).

Definition 4.5.4. *For the class* H/V(F,R)$^+$ *the functions* ϕ_i *are defined by*

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

The first branch encodes the creation of two disjuncts. The first disjunct still contains the reverse binary atom i and the number of forward binary atoms is reduced by 1 for all reverse binary atoms $j \leq i$ in the connection sequence like in Definition 4.5.2. The second disjunct does not contain the reverse binary atom i.

In Definitions 4.5.3 and 4.5.4, for the case when forward and reverse predicates are both horizontal, the functions hold only if the connection sequence has n_i forward predicates appearing before the closure reverse predicate i. Otherwise, the simpler functions of Definition 4.5.1 hold.

Proposition 4.5.1. *The families of functions from Definitions 4.5.1 through 4.5.4 for the interaction classes of Figure 4.5 have the most number of computation steps among all such families of functions.*

Proof. (Sketch) Other possible rewriting sequences for the interaction classes considered in Definitions 4.5.1 and 4.5.2 have the same number of computation steps as the rewriting sequences described in these definitions. Consider, e.g., that we start rewriting the j reverse binary atom. For the former interaction class, it means that the reverse formula i is removed after exactly one rewrite step. For the latter class, it means that j is pushed towards a non-sink variable only if $n_i > n_{i+1}$. Otherwise, another reverse binary atom $k \neq i$ with $n_k > n_{k+1}$ is pushed. It is easy to see that independently on the chose of j and k, the same number of rewrite steps are necessary.

We discuss next the simulations for the interaction classes considered in Definitions 4.5.3 and 4.5.4. Each application of rules for the first reverse binary atom i (e.g., a vertical closure) having n_i forward binary atoms (e.g., vertical closure) appearing before it, can create two disjuncts and all binary atoms j appearing after i in a connection sequence ($1 \leq j < i$) would have now to be rewritten in both disjuncts (e.g., interactions of closures either vertical or horizontal). However, this doubling of the number of disjuncts must not necessarily imply the doubling of the number of rewritings for the binary atoms j appearing *before* i in a connection sequence ($1 \leq j < i$), if the whole formula s is not normalized before rewriting these reverse binary atoms, but only after all reverse formulas are removed. □

Theorem 4.5.2 (Complexities for TRS$_2$). *TRS_2 rewrites an LGQ formula s into an LGQ formula t in time T and space S, and with the size $|t|$ of t in $O(a^b \times |s|)$, where $a = max(pos^{rev}(norm(s)))$ and $b = type^{rev}(norm(s))$.*

Moreover, TRS_2 rewrites s into t in time linear and extra space logarithmic in $|norm(s)|$, and generates rewritten formula t of size at most $|norm(s)|$, if each connection sequence in s contains neither

- *vertical closure reverse predicates after vertical forward predicates, nor*
- *horizontal closure reverse predicates immediately after horizontal reverse predicates.*

Proof. An inspection of all rules of TRS$_2$ show that these rules do not increase the number of base formulas in disjuncts in t, but (possibly) the number of disjuncts in t. Therefore, the average size of each disjunct in $norm(s)$ remains the same also for t, and is bounded by $|s|$.

We use the the families of functions $\{\phi_i \mid 1 \leq i \leq m\}$ defined previously for each interaction class. We compute in each case the function ϕ_m and the number of computation steps that corresponds to rewriting m reverse binary atoms. After that, we show how this result can be extended to the rewriting of all reverse binary atoms.

Recall that $b = |type^{rev}(norm(s))|$ is the number of reverse binary atoms in $norm(s)$ and $a = \max(pos^{rev}(norm(s))$ is the maximum connection length from a reverse binary atom to a non-sink variable in $norm(s)$. Clearly, the number of forward predicates n_i, which appear before the reverse predicates along a connection sequence, is smaller than a.

1. Classes $(\mathsf{VF},\mathsf{HR})^?$, $\mathsf{H/V}(\mathsf{F}^?,\mathsf{R})$. The computation of ϕ_m requires m steps and the number of disjuncts in t obtained by rewriting one disjunct in s is 0 or 1:

$$\phi_m(n_m, \ldots, n_1) = \begin{cases} 0 & \text{, for class } (\{\mathsf{fstChild}\},\mathsf{HR}^?) \\ 1 & \text{, otherwise.} \end{cases}$$

The entire formula $norm(s)$ is traversed once and only one pointer to the current binary atom is needed. Therefore, only extra logarithmic space in $|norm(s)|$ is needed.

2. Class $(\mathsf{HF},\mathsf{VR})^?$. The number of disjuncts in t obtained by rewriting one disjunct in s is 1:

$$\phi_m(n_m, \ldots, n_1) = 1$$

The number of steps required for the computation of ϕ_m is

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

As for the first case, only extra logarithmic space in $|norm(s)|$ is needed.

For the interaction of horizontal forward and horizontal reverse of the next two cases, the complexities hold only if before a horizontal reverse i its forward horizontal predicates n_i, or other horizontal reverse predicates appear immediately before it. Otherwise, if there is a vertical predicate inbetween, the better complexities of case one hold.

3. Class $\mathsf{H/V(F,R^+)}$. The number of disjuncts in t obtained by rewriting one disjunct in s, as also the number of computation steps, is exponential in m:

$$\phi_m(n_m, \ldots, n_1) = \sum_{i_m=1}^{n_m} (\phi_{m-1}(n_{m-1} - i_m, \ldots, n_1 - i_m)) = \sum_{i_m=1}^{n_m} \sum_{i_{m-1}=1}^{n_{m-1}-i_m} \ldots \sum_{i_2=1}^{n_2-i_3} \phi_1(n_1 - i_2).$$

4. Class $\mathsf{H/V(F,R)^+}$. The number of disjuncts in t obtained by rewriting one disjunct in s, as also the number of computation steps, is exponential in m:

$$\phi_m(n_m, \ldots, n_1) = \sum_{i_m=1}^{n_m} (\phi_{m-1}(n_{m-1}, \ldots, n_1)) = \sum_{i_m=1}^{n_m} \ldots \sum_{i_2=1}^{n_2} (n_1) = \prod_{i=1}^{m} (n_i).$$

The behaviour of combinations of any of these interaction classes follows the more complex class (with respect to the number of computation steps and of disjuncts).

The above considerations are just for one connection sequence that might not contain all reverse predicates, i.e., m might be smaller than b. We show next that the above results are extensible from m to b.

If $m < b$, then there are other reverse predicates along another connection sequence than p, containing, say, m' reverse predicates with n_i $(m + 1 \leq i \leq m' + m)$ forward predicates before them. This new connection sequence is to be considered in each of the already generated disjunct. Let us consider the fourth case above. The other cases can be treated similarly. In this case, each disjunct is replaced by other $\prod_{i=m+1}^{m+m'} (n_i)$ disjuncts. The total number of disjuncts is now $\prod_{i=1}^{m+m'} (n_i) = (\prod_{i=1}^{m} (n_i))(\prod_{i=m+1}^{m+m'} (n_i))$, and after treating all connection sequences containing reverse predicates, we can conclude for the fourth case to be $\prod_{i=1}^{b} (n_i)$. This result can be approximated to $\prod_{i=1}^{b} (a) = a^b$.

□

TRS_3 adds to the complexities of TRS_2 the overhead of transforming variables with a forward sink-arity greater than one into one-sink variables, cf. Section 4.4.2 and Rule (4.25). Recall that each application of Rule (4.25) preserves the size of the rewritten formula. For an formula s, there are $|type^{dag}(norm(s))|$ such variables j, and the number of forward predicates on the longest connection path until variable j denoted by n_j. We denote by $fan\text{-}in$ the sum of forward sink-arities of all multi-sink variables in s:

$$fan\text{-}in = \sum_{j \in type^{dag}(norm(s))} (j).$$

$fan\text{-}in$ can be also seen as the number of reverse binary atoms introduced by the applications of Rule (4.25), and this adds to the exponential complexity factor of rewriting using TRS_2.

Also, TRS_3 rewrites any LGQ formula to a forward LGQ forest formula, where each constituent tree is in fact variable-preserving minimal, cf. Proposition 3.5.1 and is not unsatisfiable with respect to the unsatisfiability detection rules of Lemma 4.3.6.

Theorem 4.5.3 (Complexities for TRS_3). *TRS_3 rewrites an LGQ formula s into an LGQ formula t in time T and space S in $O(a^c \times |s|)$, and with the size $|t|$ of t in $O(a^c \times n)$, where $a = max(pos^{rev}(norm(s)))$ and $c = |type^{rev}(norm(s))| + fan\text{-}in$, and n is the maximum number of variables in a disjunct of $norm(s)$.*

Moreover, TRS_3 rewrites s into t in time linear and extra space logarithmic in $|norm(s)|$, and generates rewritten formula t of size at most $|norm(s)|$, if each connection sequence in s contains neither

- *vertical closure reverse predicates after vertical forward predicates, nor*
- *horizontal closure predicates immediately after horizontal reverse predicates, nor*
- *vertical closure forward predicates, having as sink a variable with a forward sink-arity greater than one, after vertical forward predicates.*

Proof. Recall that TRS_3 contains TRS_2, which has time and space complexity and generates t with a number of disjuncts in $O(a^b \times |s|)$ in general, and has time linear, extra space logarithmic, and $|t|$ linear in $|norm(s)|$ for particular cases of s without connection sequences with vertical closure reverse predicates after vertical forward predicates, or horizontal closure predicates immediately after horizontal reverse predicates. The latter complexities apply also for TRS_3, if the application of the extra rule (4.25), which rewrites forward binary atoms having as sink a variable with a forward sink-arity greater than one, does not generate sequences of the above kind, thus any connection sequence in s must not contain also vertical closure forward predicates, having as sink a variable with a forward sink-arity greater than one, after vertical forward predicates.

TRS_3 rewrites LGQ formulas to forward LGQ forest formulas, cf. Theorem 4.4.3, where each constituent disjunct is a tree, thus variable-preserving minimal, cf. Proposition 3.5.1. This means that each disjunct in t has as many binary atoms as the maximum number of variables n in a disjunct in $norm(s)$, which can be much smaller than the size of that disjunct, which is on its turn bounded in $|s|$. Also, the number of nodetest-formulas is constant per each variable.

Because TRS_3 is confluent, the order of rules applications does not matter for $|t|$. Let us consider that Rule (4.25) is applied until no other applications are possible. Then, $\sum_{j \in type^{dag}(norm(s))} (j-1) < fan\text{-}in$ new reverse predicates are introduced in addition to the existing ones $type^{rev}(s)$. The case distinction from the proof of Theorem 4.5.2 applies (see that proof for details), by replacing $type^{rev}(norm(s))$ with $type^{rev}(norm(s)) + fan\text{-}in$. □

Alternative technique for finding the upper bound on the number of trees in t

Consider the orders $<_v$ and $<_h$ on LGQ variables of an LGQ formula s, defined by ($vr \in \mathsf{VR}^?$, $vf \in \mathsf{VF}^?$, $hr \in \mathsf{HR}^?$, $hf \in \mathsf{HF}^?$):

$$vr(x,y) \subseteq e \Leftrightarrow y <_v x, \qquad vf(x,y) \subseteq e \Leftrightarrow x <_v y$$
$$hr(x,y) \subseteq e \Leftrightarrow y <_h x, \qquad hf(x,y) \subseteq e \Leftrightarrow x <_h y.$$

Intuitively, $<_v$ and $<_h$ are partial orders on LGQ variables that appear in vertical, respectively horizontal binary atoms, and $\{x,y\} \in <_c$ ($c \in \{v,h\}$) if for each LGQ substitution consistent the input formula and tree, the image of x appears in document order before the image of y.

Total orders can be obtained from $<_v$ and $<_h$ by creating all possible permutations of LGQ variables consistent with $<_v$ and $<_h$. Then, the combination of one possible total order obtained from $<_v$ together with one obtained from $<_h$ defines a possible disjunct of t, by translating back the pairs of these orders in $\wedge$-connected binary atoms (as shown above). These total orders can be derived by repeatedly decomposing $<_v$ and $<_h$ and eliminating the unorderedness ($\alpha \subseteq \{<_v, <_h\}$):

$$\alpha = \alpha' \cup \{(x,y),(z,y)\} \Rightarrow (\{\alpha' \cup \{(x,z),(z,y)\}, \alpha' \cup \{(z,x),(x,y)\}\}). \qquad (4.46)$$

Note that in this way each disjunct in t does not contain any two vertical/horizontal binary atoms having the same variable as sink. The disjunct becomes a tree, if two additional conditions are satisfied: there are no multi-sink variables and no cycles.

For the former condition, that disjunct must not contain any two binary atoms having the same variable as sink, i.e., there is no pair in the corresponding total orders, say α_v and α_h, derived from $<_v$ and $<_h$ respectively, having the same variable appearing on the second position. This condition is ensured by (cf. the interaction class $(\mathsf{VF},\mathsf{HR})^?$)

$$\alpha_v = \alpha'_v \cup \{(x,y)\}, \alpha_h = \alpha'_h \cup \{(z,y)\} \Rightarrow \alpha_v = \alpha'_v \cup \{(x,z)\}, \alpha_h = \alpha'_h \cup \{(z,y)\}.$$

The latter condition can be simply checked by computing the transitive closures α_v^* and α_h^* of the total orders α_v and α_h. These closures are recursively defined using the following straightforward equivalences ($\forall x, y \in \mathrm{Vars}(s)$):

$$x\alpha_v^* y \Leftrightarrow x\alpha_v y \text{ or } \exists y \in \mathrm{Vars}(e) : x\alpha_v y \alpha_v^* z$$
$$x\alpha_h^* y \Leftrightarrow x\alpha_h y \text{ or } \exists y \in \mathrm{Vars}(e) : x\alpha_h y \alpha_h^* z.$$

The following propagation rule can be further derived from the semantics of LGQ predicates

$$x\alpha_v^* y \alpha_h^* z \alpha_v^* w \Rightarrow x\alpha_v^* w,$$

because the descendants w of siblings z of nodes y that have ancestors x, have also as ancestors the nodes x. Detecting unsatisfiability can be simply done by checking whether $\alpha_c^*(c \in \{v,h\})$ contains at least a pair of a variable with itself ($x\alpha_c^* x$), or that a same variable pair appears in both α_v^* and α_h^* ($(x,y) \in \alpha_v^*, (x,y) \in \alpha_h^*$).

The number of total orders derived from $<_v$ or $<_h$ is clearly exponential in the number of constituent pairs from both of them having the same variable appearing on the second position (see (4.46)), i.e., it is exactly the number of reverse binary atoms plus the sum of the forward sink-arities of variables in s. This number is the same exponential factor of Theorem 4.5.3. Furthermore, the number of combinations of each total order derived from $<_v$ and of each total order derived from $<_h$, which is the product of the number of total orders for each $<_v$ and $<_h$, gives an upper bound for the number of trees in t.

4.6 Related Work

The ERRA problem is an expressiveness problem and this chapter gives a positive solution to it. The existence of such a solution ensures that one can safely consider the forward fragment of XML query languages for tasks like evaluation, containment, etc., because the reverse language fragment can be simply expressible within the forward fragment, though with its inherent complexity overhead. This is why the ERRA problem is of high importance for XML query languages, and the existence of the solution presented in this chapter is used in various contexts:

- query evaluation against XML streams; the rules of Lemma 4.3.3 are used by [126, 127, 84, 138, 106, 131, 129, 124, 125]. Rule (4.1) is used by [21] (at an algebraic level, not expressed syntactically).
- static inference of query properties like duplicate freeness and result ordering [85, 112].
- complexity results for XPath query evaluation; [140] proposes algorithms for evaluation of XPath without closure axes, and notes that for this XPath fragment the rewriting using TRS_2 is LOGSPACE.
- query evaluation against XML data using relational databases; [79, 80] propose efficient spatial data indexes and point out that identities of Lemma 4.3.3 can be used to optimize query evaluation in such contexts by pruning index regions.
- expressivity of XPath; Some of the rules of Lemmas 4.3.3 and 4.3.4 are recently used by [74] to show also that the language of conjunctive queries over some of the LGQ built-in predicates is as expressive as XPath.

This section describes next some recent results on the fields of XPath query containment and equivalence, minimization, and rewriting. Note that XPath (and also LGQ) queries are essentially specialized conjunctive queries on a tree-structured domain. Containment of relational queries, thus also their equivalence and minimization that are based upon, is known to be NP-complete [38].

The work found in the area of query containment and equivalence, rewriting, and minimization, can be classified in two categories: model-based and syntax-oriented approaches. The former category relies on a modeling of queries as tree patterns or various kind (tree,

two-way) of automata. The problems are then reduced to tests at the level of these models. The latter category applies syntactic operations, like rewriting. Arguably, syntax-oriented approaches come with several advantages, like remaining at the level of the query language, thus capturing the exact semantics and properties of the queries and delivering the result queries directly. Also, the encoding of such problems at the level of automata suffers when translating back the obtained solutions from automata to queries.

Query Containment and Equivalence

The problem of query containment is to check whether the answers of one query are contained in the answer of a second query for all databases. Equivalence can be seen then as two-way containment. This problem received significant attention in the context of XML, e.g., [33, 147, 56, 148, 115, 149, 121, 144, 116]. The motivation underlying such robust body of work relies in practical issues like query optimization, e.g., [38] or answering queries using views (see below). Query containment is also the first step in addressing more involved problems like query minimization, rewriting, and answering queries using views.

[65] studies the containment of the union-free and negation-free fragment of the StruQL query language for querying semistructured data seen as graph. LGQ and StruQL are incomparable, because LGQ has relations that are not expressible in StruQL (the horizontal and reverse relations), and StruQL allows closures on paths. [65] shows that for this StruQL fragment containment is decidable, and gives semantic and syntactic criteria for checking containment. The semantic criteria is based on canonical databases for a given query, i.e., databases on which the query answer is not empty and the removal of one database node renders the answer empty. Although there are infinitely many canonical databases for a given query, [65] shows that for checking containment it suffices to use just a finite number of them, which depends on the considered queries. Because the complexity of this approach is triple exponential, [65] develops a more efficient syntactical criteria with only exponential complexity.

Strongly related to results of this chapter, [65] shows also that, for a further restriction of StruQL to simple regular path expressions (that correspond in fact to LGQ[$\{\mathsf{child}, \mathsf{child}^+\}$] path queries), the intersection $R \cap R'$ of two such expressions R and R' is expressible as a union of expressions $R_1 \cup \ldots \cup R_k$, where the size of each R_i is at most the size of both R and R', and the number k of union terms can be exponential in the size of R and R'. This result, although stated only for simple path expressions, mirrors (almost perfectly) three important properties of rewriting arbitrary LGQ formulas, as investigated in this chapter: (1) DAG formulas (as obtained via intersection) can be always rewritten to equivalent forest formulas, (2) which can have sizes exponential in the sizes of the DAG formulas, and (3) contain only tree formulas, whose sizes are bound in the size of the DAG formulas. Note that this chapter sharpens the third property by ensuring the size of the resulted tree formulas dependent only on the number of variables and not of the predicates of the DAG formulas. The approach of [65] to these results is also different from ours: there, each simple regular path expression is compiled into an NFA, and their intersection is equivalent to the product automaton. All possible paths from an initial to a final state

define then possible regular path expressions contained in their intersection, and there are exponentially many such paths.

[115, 116] are follow-up works of [65] with declared focus on the XPath fragment of child and child^+ axes, wildcards, and filters. In particular, it is shown that for the aforementioned XPath fragment, the containment problem is coNP-complete. Earlier research shows that for the XPath fragment without any of the constructs (1) child^+ axis [148], (2) wildcards [9], (3) filters, the containment problem is PTIME. For checking containment, [115, 116] propose an efficient (PTIME), sound algorithm that is also complete in some practical cases (no filters), and a sound and complete algorithm (EXPTIME) that is efficient (PTIME) in particular cases of interest (the number of child^+ axes, or of wildcards, or of filters is bounded). The techniques of [115, 116] for checking containment are similar to the ones of [65], namely canonical models (similar to canonical databases), and pattern homomorphism between two queries.

[33] considers the containment problem for conjunctive regular path queries with inverse (CRPQI). In contrast to queries of the StruQL fragment considered in [65], binary relations created via composition (concatenation, Kleene-*, union) of child relations admit inverses. The technique of [33] for checking non-containment is based on checking non-emptiness of a two-way finite automaton constructed from the two queries. [33] gives the EXPSPACE upper bound for CRPQI containment shown also by [65] for CRPQ (i.e., without the inverse operator). This upper bound is furthermore shown to be also a lower bound, even for CRPQ. An interesting open issue is the problem of finding an equivalent forward CRPQ (forest) query to any CRPQI query. We conjecture that a (non-trivial) extension of the results of this chapter in the direction of coping with Kleene-* composition of path expressions would provide a solution to the problem.

[144] proposes a technique for checking containment of XPath queries based on an inference and rewriting system that allows asserting and proving containment properties by using judgments.

In the presence of schemas like DTDs or of strictly more powerful regular tree grammars, the XPath query containment problem proves to be harder [146, 147, 56, 148, 121, 149]. For DTDs and simple XPath integrity constraints the problem is undecidable [56]. For the XPath fragment with filters (no closures or wildcards), for which the standard containment problem is PTIME, the query containment problem is coNP-complete [148, 121]. Query containment under DTDs is decidable (EXPTIME) for the XPath fragment containing child^+ axis, filters and wildcards [149]. The technique for checking containment is based in [149] on the transformation of queries into regular tree grammars, and the use of known decidability and closure results for regular tree grammars. However, XPath query containment in presence of disjunctions, variable bindings, equality testing, and DTD constraints is undecidable [121].

Query Minimization

The query minimization problem is to find for a given query an equivalent one that has the smallest size among all its equivalents. Minimization is one important path to query

optimization, because a decrease in the query size affects positively the query evaluation. Note that the minimization problem is at least as hard as the containment problem, on which it is based. Therefore, complexity lower bounds of the latter apply also to the former.

Results of this chapter have direct relevance to the state of the art of the query minimization problem. More specifically, the rewriting approach presented in this chapter has also some minimization properties. First, the rules of Lemma 4.3.6 eliminate simple syntactic redundancies. Second, and more important, non-obvious semantic redundancies are detected and eliminated. TRS_3 yields forward forest formulas, where each tree is variable-preserving minimal, even if the input formula does not have this minimality. Recall that variable-preserving minimality means that the rewritten formula contains as many binary atoms as variables in the tree minus one, thus no redundant binary atoms appear in any disjunct of the rewritten forward formula. Also, there are no redundancies among disjuncts. Only some rules of TRS_2 create disjunctions, but each disjunct is not (semantically) contained in any other, and the number of these disjunctions is minimal under certain properties, cf. Theorem 4.3.1. The variable-preserving minimality, as investigated in this chapter, is complementary to the more involved minimality objective of [148, 9, 135, 64], where variables can be also removed.

There are several efforts towards XPath query minimization [148, 9, 135, 64]. In the absence of DTDs, simple XPath expressions built up from child axis, wildcard, and filters, have a unique minimal equivalent expression, which can be found in polynomial time [148]. These results carry over even if child^+ axis, but no wildcard, is allowed [9]. For XPath queries with child and child^+ axes, filters, and wildcards, the minimization problem is NP-hard [64], and its variant of finding an equivalent query with size less than a given threshold is coNP-complete (based on results for containment [115]).

In the presence of child/parent and sibling constraints derived from DTDs, the simple XPath expressions do not necessarily have a unique minimal equivalent [148]. For simpler constraints, like required child, required descendant, and required co-occurrences, the equivalent minimal query remains however unique [9]. Latter, [135] gives more efficient variants of the polynomial algorithms of [9].

All presented approaches have at their core the observation that a minimum size query equivalent to a query having child and child^+ axes, filters, and wildcards, can be found among the subpatterns of the latter (thus the minimal query is done by pruning redundant subqueries until no subquery can be removed while preserving equivalence). Also, for the queries of [148, 9, 135], a containment between two such queries p and q can be reduced to finding a homomorphism from q to p, which can be done in polynomial time. For queries having child and child^+ axes, filters, and wildcards, this property does not necessarily hold [115, 116], and the minimization problem becomes NP-hard [64].

View-based Query Processing

There are two (similar) approaches to view-based query processing [35]: view-based query rewriting and answering.

In the view-based query rewriting approach, we are given a query and a set of views,

and the goal is to reformulate the query into an expression, the rewriting, that refers only to the views, and provides the answer to the query. Typically, the rewriting is formulated in the same language used for the query and the views, but in the alphabet of the view names, rather than in the alphabet of the database. Thus, query processing is divided here in two steps, where the first step re-expresses the query in terms of a given query language over the alphabet of the view names, and the second step evaluates the rewriting over the view extensions.

The ERRA problem can be seen as a specialization of the view-based query rewriting, where the views are derived from the rewrite rules of this chapter. There are still some important differences between the two problems: in the ERRA problem, (1) the rewriting must be equivalent to the initial query, (2) the reformulation must not be necessarily done using only the views (only query parts with reverse atoms should be rewritten), and (3) there can be more than one reformulation step, for there can be views that map formulas to equivalent formulas that still contain reverse atoms. The approach to query rewriting taken in this chapter it is different from [130] (and others described in [83]) where an algorithm for rewriting regular path queries using techniques like containment mappings and chase is proposed. In our case, the exact rewriting procedure (that may correspond to a so-called evaluation strategy of rewriting systems) is of no immediate importance, because our focus is more on properties like uniqueness of normal forms, which we show to be preserved by any of the rewriting strategies (because of the confluence property). Proprietary rewriting algorithms are, however, tuned to obtain efficiency for particular classes of queries to be rewritten.

In the view-based query answering approach, besides the query and the view definitions, we are also given the extensions of the views. The goal is to compute the answers that are implied by these extensions. Thus, we do not pose any limit to query processing, and the only goal is to compute the answer to the query by exploiting all possible information, in particular the view extensions.

The complexities of view-based rewriting and answering problems for regular path queries are studied in [31, 32]. The rewriting problem is 2EXPTIME, and the answering problem is coNP-complete in the size of the view extensions. These results are further extended to regular path queries with inverse [34].

View-based query processing has important application domains [83], e.g., query optimization, database design, data integration, data warehouse, and semantic data caching.

In the context of query optimization, computing a query using previously materialized views can speed up query processing because part of the computation necessary for the query may have already been done while computing the views.

In the context of database design, view definitions provide a mechanism for supporting the independence of the physical and the logical view of the data, thus enabling the modifications of the storage schema of the data (i.e., the physical view) without changing its logical schema; the storage schema can then be seen as a set of views over the logical schema.

In the context of data integration, a uniform query interface (a mediated schema) is provided to a multitude of autonomous data sources (that are semantically mapped to

relations from the mediated schema via source descriptions). The difference of the data integration domain to the query optimization and database design domains consists in their different focuses: in the data integration domain, the number of views (i.e., sources) tend to be much larger, the sources do not contain the complete extensions of the views, and the rewriting can be (maximally-)contained in the initial query, not necessarily equivalent [123].

In the context of data warehouses, it is needed to choose a set of views to materialize in the warehouse.

In the context of semantic data caching in client-server systems, the data cached at the client is modeled semantically as a set of views, rather than at the physical level as a set of data pages or tuples. Later, deciding which data needs to be sent from the server to the client in order to answer a given query requires to analyze which parts of the query can be answered by the cached views.

Chapter 5

Evaluation of Forward LGQ Forest Queries against XML Streams

This chapter introduces the problem of streamed and progressive evaluation of forward LGQ forest queries against XML streams (SPEX) and describes a solution for it [127, 124, 125]. Recall that the LGQ fragment of forest queries is equivalent to XPath, cf. Chapter 3. Moreover, Chapter 4 shows that the fragment of forward LGQ forests is equivalent to full LGQ. Therefore, it is important to stress that the evaluation strategy of forward LGQ forest queries, as considered in this chapter, applies to XPath queries and to unrestricted LGQ graph queries, too.

After stating the SPEX problem and positioning it briefly in the context of query evaluation and tree pattern matching, Section 5.2 introduces a strategy for the SPEX evaluation against unbounded XML streams by means of so-called stream processing functions. For each LGQ predicate, there is a corresponding stream processing function that computes, for a given set of source nodes, the sink nodes that stand in that predicate with any of the source nodes. The composition of LGQ atoms into LGQ formulas and queries is reflected in the sequential and parallel composition of the corresponding functions. Section 5.3 gives an efficient realization of the proposed evaluation strategy by means of networks of communicating deterministic pushdown transducers. A transducer network is a directed acyclic graph where nodes are transducers and the communication between transducers is directed by the graph edges. Two minimization problems for transducer networks are discussed in Section 5.4: the problem of finding the minimal network equivalent to a given network, and the problem of minimal stream routing within a given network. For the latter problem, an effective solution is given that improves considerably the processing time of transducer networks. Section 5.5 investigates the complexity upper bounds for the evaluation of queries from eight fragments of forward LGQ forests. All cases enjoy polynomial complexities in both the size of the query and of the input stream, and some of them have complexities independent of the stream size. Correlating these results with the complexity results of LGQ query rewriting from Chapter 4, it is shown further that a large fragment of LGQ graph queries has polynomial complexity for the evaluation. These theoretical complexities are also confirmed by extensive experimental evaluations in Section 5.6. Finally,

Section 5.7 is devoted to related work in the field of XPath query evaluation.

5.1 Problem Description

The *streamed and progressive evaluation of forward LGQ forest queries against XML streams (SPEX)* problem is: given a forward LGQ forward query and a (possibly unbounded) well-formed XML stream, compute and deliver as soon as possible the exact answers to the query in a single pass over the stream, provided no knowledge about the stream is used.

There are three salient aspects of the SPEX problem: (1) the kind of queries to evaluate, (2) the streamed and (3) the progressive aspects of the evaluation. In the sequel it is argued that the first aspect of the problem, i.e., the evaluation of forward LGQ queries, has similarities with standard tree matching problems, though it is different. Also, it is shown that the latter two aspects of the problem make an important difference to the general problem of evaluating forest LGQ queries against in-memory XML data.

1. The evaluation of forward LGQ queries is defined on ordered unranked trees with labels on nodes, not directly on XML streams that are serializations of such trees. Also, query answers are defined as sets of nodes from trees, cf. Chapter 3. In order to accommodate LGQ (and also XPath) evaluation to XML streams, we still refer to nodes and trees implicitly conveyed in the XML streams, while considering the mapping of nodes to stream (or well-formed XML document) fragments, as detailed in Section 3.1. Note that because the trees are not really materialized, the LGQ predicates on nodes in trees have to be rediscovered at processing time. Indeed, for the LGQ evaluation against XML streams we use the indispensable parent/child and sibling binary predicates encoded in the XML streams by some a-priori fixed orders of opening and closing tags of fragments corresponding to nodes. Query answers are then well-formed fragments of XML streams that correspond to nodes in the tree conveyed in the stream.

2. The *streamed* aspect of the evaluation resides in the sequential access to the messages of the XML stream, which corresponds to the (depth-first, left-to-right) preorder traversal of the tree conveyed in the XML stream. This order is also called *document order* [46], because it corresponds to the order of the opening and closing tags of nodes in an XML document, hence also to the order of tags in an XML stream.

3. The *progressive* aspect of the evaluation resides in the incremental delivering of the query answers as soon as possible. This is motivated mainly by processing in dynamic environments and by memory issues. The former issue is evident when the query answers are input for other processes in pipeline processing, thus immediate delivering of answers improves the overall performance. The query evaluation against unbounded XML streams should deliver answers incrementally, because there is no expected evaluation end. Also, collecting all answers to be delivered at the evaluation end can require unbounded memory.

Recall that LGQ queries have intuitive graphical representations, called digraph rep-

resentations in Section 3.4. The digraphs of LGQ forest queries are unordered trees with binary predicates on edges. The relation between two data nodes mapped by two directly connected query nodes in a query digraph can be besides parent/child, also ancestor/descendant, preceding/following, or preceding-sibling/following-sibling, as corresponding to the LGQ predicates.

There is a striking similarity between the evaluation problem of LGQ forest queries and two variations of the popular *tree matching* problem introduced by [87]. Despite of their similarity, these problems are still different, mainly with clear implications on the algorithmic design and the evaluation complexity of their solutions.

The tree matching problem [87] consists in matching a data tree with a set of tree patterns (the queries). [87] shows that the tree patterns can be preprocessed into a structure of exponential size, which factors out all common subpatterns, such that every data tree can subsequently be matched bottom-up in linear time. The best algorithm to date is $O(nlog^3m)$ [49]. This technique cannot be applied to the evaluation of LGQ forest queries because (1) all patterns of [87] are ordered and represent only LGQ forest queries with par/child predicates, and (2) the data tree is traversed bottom-up, condition that contradicts the constraint of a streamed query evaluation.

A more similar problem is introduced in [97] as the *unordered tree inclusion* problem: given the pattern and the data tree, can the pattern be obtained from the data tree by node deletions? This problem is different from the LGQ forest query evaluation because (1) such patterns corresponds to LGQ forest queries only with $\mathsf{par}^+/\mathsf{child}^+$ predicates, and (2) two nodes from the pattern can not be mapped to the same node in the data tree. The latter difference makes the unordered tree inclusion NP-complete [97], whereas the evaluation of the LGQ forest queries remains polynomial.

The standard approach for XPath evaluation is given by [70]. Although this approach meets good complexity results, it does not meet the streamed and progressive aspects of the SPEX problem: the XML document has to be stored in memory a priori to query evaluation, and the answers are delivered only at the very end of the evaluation process.

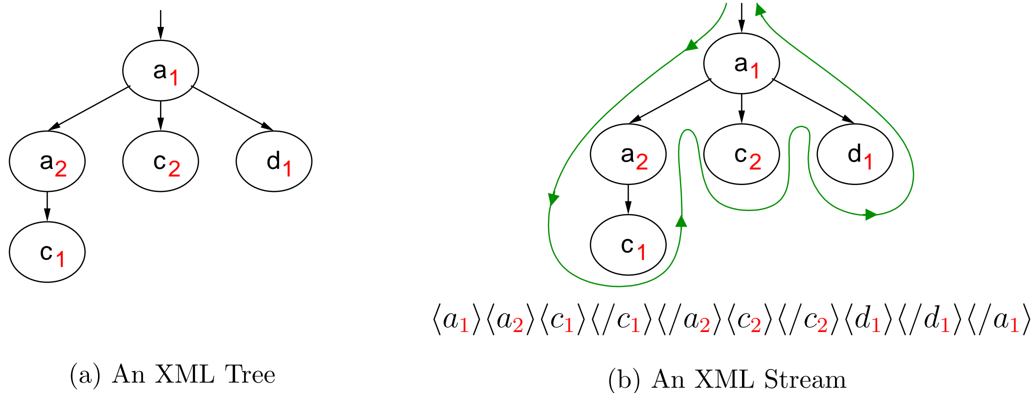
XML Streams versus In-memory XML Data

This section highlights some of the challenges of the query evaluation against XML data streams by comparing it to a query evaluation approach for in-memory XML data, as used, e.g., by [70, 78]. The salient features of both approaches are stressed also by an illustrating example.

In the following, we distinguish between two scenarios: the evaluation against the in-memory XML tree of Figure 5.1(a), and against the XML stream of Figure 5.1(b) corresponding to the (depth-first left-to-right) preorder traversal of that XML tree. Note that the label indices do not belong to nodes and are only used to ensure a clearer node identification.

As query example, let us consider the LGQ tree query

$$Q(v_3) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{a}(v_1) \wedge \mathsf{child}(v_1, v_2) \wedge \mathsf{d}(v_2) \wedge \mathsf{child}^+(v_1, v_3) \wedge \mathsf{c}(v_3).$$



(a) An XML Tree

(b) An XML Stream

The evaluation of this query on a tree yields all c-nodes descendants of a-nodes that have at least a d-child. For the tree of Figure 5.1(a), the answer is the set of both c-nodes. In a stream context, the answer is the serialization of these two c-nodes in the document order.

In-memory XML data. We sketch now a standard XPath evaluation approach, as used by efficient XPath evaluators for in-memory XML data currently available [70, 78], but failed by popular XPath evaluators [11, 47, 113, 61]. It consists in the stepwise (decoupled) evaluation of the query, where the evaluation of each predicate is done in one processing step with respect to an input set of source nodes and yields an output set of nodes that are then input for the evaluation of other predicates. For the above query, its evaluation can be accomplished by (1) computing all a-nodes that stand in the predicate child^+ with the root node, (2) then by computing all d-nodes that stand in the predicate child with the previously selected a-nodes, (3) by collecting only the a-nodes selected in the first step that have at least a d-child, and finally (4) by computing the set of all c-nodes that stand in the predicate child^+ with any a-node selected in the previous step.

For the tree of Figure 5.1(a), the first step computes the set $\{a_1, a_2\}$, the second step computes $\{d_1\}$, the third step computes $\{a_2\}$, and the fourth step computes $\{c_1,c_2\}$.

There are (at least) two important characteristics of this evaluation strategy:

1. the evaluation is query-driven with random access to data nodes. Each predicate is evaluated once for good, and the evaluation of several predicates is not intertwined. Also, the *same* node can be visited several times, because several intermediary result sets can have common nodes. Moreover, the evaluation of the same predicate can even require to visit the same node several times.

2. An intermediary result set can not exceed the amount of all nodes in the tree and is only meaningful for the evaluation of the next predicate(s). to ensure the former constraint, the intermediary result sets are subject to duplicate removal operations.

XML data streams. Figure 5.1(b) depicts the previous tree together with the XML stream corresponding to its (depth-first, left-to-right) preorder traversal, as highlighted by

the green curve. Recall that the XML stream is the XML document in unparsed format, and the correspondence between the tree and the stream is simply obtained with the preorder traversal of the tree as follows: on entering a node, its opening tag is appended to the stream, on exiting that node, its closing tag is appended to the stream. Recall that the order of opening and closing tags in an XML stream is the "document order" that corresponds also to the order of tags in an XML document.

The requirement of the in-memory evaluation strategy to visit the *same* nodes at different times violates one of the main goals of the query evaluation against XML streams, namely to use a single pass over the input XML stream, i.e., one (depth-first left-to-right) preorder traversal of the conveyed tree. The novel strategy considered here is to evaluate all predicates of a query simultaneously, while considering also their inherent dependencies regarding their source and sink nodes.

For the evaluation of the same query against that XML stream, the tags are processed stepwise in the order imposed by their appearance in the XML stream. We consider also that each predicate is implemented by some some sort of automaton that is instructed to find incrementally the tags of all sink nodes that stand in that predicate with some given source nodes. For example, an automaton for the predicate child^+ finds the opening tags of all sink nodes in the stream that are descendants of any given source node. In order to evaluate an entire query, the independent automata for the predicates constituting the query communicate with each other as indicated by the source and sink variables of the atoms having those predicates: if the source variable of an α-atom is the sink variable of an α'-atom, then the automaton for α' informs the automaton for α about its findings. Such automata can be also composed. We detail next how our query can be evaluated by such a machinery made out of three automata, say $\alpha \cdot \eta$ for the compositions of the binary predicate α followed by the unary predicate η. More precisely, we consider the automata $\mathsf{child}^+\cdot\mathsf{a}$, $\mathsf{child}\cdot\mathsf{d}$, and $\mathsf{child}^+\cdot\mathsf{c}$.

On encountering the opening tag $\langle a_1\rangle$, the automaton $\mathsf{child}^+\cdot\mathsf{a}$ matches and communicates this information to its immediate next automata $\mathsf{child}\cdot\mathsf{d}$ and $\mathsf{child}^+\cdot\mathsf{c}$. These next automata try to match now opening tags of d-nodes children of a_1, and c-nodes descendants of a_1 respectively. The same procedure happens for the next opening tag $\langle a_2\rangle$. On encountering $\langle c_1\rangle$, the automaton $\mathsf{child}^+\cdot\mathsf{c}$ matches for a_1 and a_2 and communicates that c_1 is a *potential* answer. Such potential answers should be buffered, together with the stream fragments between their opening and closing tags, until the decision on their appurtenance to the result is met. On encountering $\langle /a_2\rangle$, it is known that no d-node child of a_2 was found, thus the c_1-node is not anymore a potential answer due to the constraints of the a_2-node. However, the c_1-node remains a potential answer due to the (not yet satisfied) constraints of the a_1-node. On receiving $\langle c_2\rangle$, the automaton $\mathsf{child}^+\cdot\mathsf{c}$ matches for a_1 and communicates the beginning of a new potential answer c_2. On $\langle d_1\rangle$, the automaton $\mathsf{child}\cdot\mathsf{d}$ matches for a_1, and both c_1 and c_2 become answers and are immediately output. The rest of the stream does not bring any new potential answers and its processing is skipped here.

It is worth noting some challenges of query evaluation against XML streams, for these challenges shed the light on the important characteristics of the evaluation approach pro-

posed here:

1. For coping with the query evaluation in a stream context, Chapter 4 introduces rules for rewriting LGQ queries to forward LGQ queries. These rewrite rules ensure that no reverse predicates occur in the query to be evaluated. The motivation for such rewrites lies in the expensive evaluation in a stream context of queries with reverse predicates that can require to maintain a history of the already processed stream. Because the evaluation of a forward LGQ query from a source node always yields a set of sink nodes located in the tree after it in document order. i.e., later in the stream conveying the tree, it is possible to evaluate forward LGQ queries while traversing the stream only once. The evaluation strategy described in the next section is based on this vital observation and is indeed able to evaluate forward LGQ queries using a single pass over the input XML stream.

2. The forward LGQ binary predicates on nodes of a tree are not a priori computed, rather they have to be rediscovered during processing of a stream conveying that tree. The rediscovery of such structural predicates in a stream can be naturally done using stacks to remember the depths of various nodes in the stream. This way, e.g., the children of a node n can be discovered by searching for nodes n' with (1) the opening and closing tags appearing enclosed by the opening and closing tags of n and with (2) the depth being the depth of n plus 1.

3. The result of the evaluation of a predicate q is communicated by the automata implementing q to the automaton(a) implementing the immediate next predicate(s) in the query. Such a communication must ensure that the immediate next automata get as soon as possible and progressively input they have to work on. The approach proposed here realizes the communication along the stream by annotating the nodes that are input for the immediate next automata.

   In order to better control the amount of memory and kind of operations used for query evaluation, a natural choice for the implementation of the forward LGQ predicates is offered by low-level finite state automata that need a stack (see the first characteristic above) and need to communicate (see the second characteristic above). A formal model that meets all these requirements is the pushdown transducer, i.e., an automaton with stack and output. Section 5.3 gives indeed an implementation of all forward LGQ predicates by means of deterministic pushdown transducers.

4. The evaluation of forward LGQ queries against XML streams can require theoretically to buffer stream fragments (as exemplified above), and in worst case the entire stream needs to be buffered, though practical cases point to buffers of size linearly dependent on the stream depth, and independent of the stream size. In contrast, the exemplified evaluation approach for in-memory XML data [70, 78] requires always to store the entire data in memory and needs also memory linear in the size of the input data for intermediary results.

5.2 Specification

This section presents an evaluation strategy for the SPEX problem. This strategy is efficiently realized in the next section using networks of communicating deterministic pushdown transducers.

For the evaluation of forward LGQ forest queries, we consider the computation of their constituent predicates restricted to the following task: given a set of source nodes, compute the set of sink nodes that stand in that predicate with any of the source nodes. Such limited accesses to the forward LGQ predicates can be specified using functions with sets of nodes as domains and co-domains. If we consider such computations of all predicates p_i of a given LGQ query specified by functions f_i, then the whole query can be evaluated by the application of appropriate compositions of those functions f_i. For example, the computation of two predicates p_1 and p_2, where the source nodes of p_2 are the sink nodes of p_1, can be specified naturally by the sequential composition of their corresponding functions $f_1 \cdot f_2$. The computation of the same predicates, where the source nodes of both of them are the same, can be specified naturally by the parallel composition of their corresponding functions $f_1 +\!\!+ f_2$. Generalizing, for any forward LGQ query, its computation can be specified using such sequential and parallel compositions of functions.

Note that such an evaluation is independent on the immediate implications of storing or streaming of the input XML data. As explained in the previous section, if the data is stored, then an efficient evaluation strategy would evaluate each predicate at a time, whereas if the data is streamed, that approach is not possible and all predicates should be evaluated at the same time. For the processing with functions, the former case would correspond to an innermost evaluation order, whereas the latter case to an outermost-like evaluation order: for $f_1 \cdot f_2$, using the innermost evaluation order, we evaluated completely f_1 and then f_2, whereas using the outermost-like evaluation order, we evaluate f_1 incrementally as much as needed for the evaluation of f_2. In other words, the kind of evaluation we perform is reflected by its order, and our same evaluation strategy can be applied in both cases. However, from now on, we detail on our evaluation strategy arguing only from the side of the streaming case.

Two important ingredients are used for specifying the present evaluation strategy: stream annotations and functions specifying the evaluation of forward LGQ predicates against XML streams, called here *stream processing functions*. Stream annotations are important for marking source and sink nodes of predicates in the XML stream. The stream processing functions use the annotations to differentiate the source nodes from the others in their input stream, and the sink nodes from the others in their output stream. Applied on an XML stream with specially marked source nodes, a stream processing function for a predicate α moves the annotations of each source node to all sink nodes that stand in the predicate α with that source node. In this way, the nodes in the stream remain unchanged and only their annotations may change. The sequential and parallel compositions of such functions, which specify analogous compositions of atoms in LGQ queries, may propagate annotations of initial source nodes to final sink nodes, which constitute the result of the evaluation of LGQ queries.

In the following, stream messages and stream processing functions and their compositions are introduced, followed by the step-by-step specification of LGQ query evaluation via compositions of stream processing functions. Section 5.3 shows how stream processing functions can be efficiently implemented using deterministic pushdown transducers.

5.2.1 Stream Messages

Streams are depth-first, left-to-right, preorder serializations of trees, thus they are made of well-formed XML documents representing such serializations. A message in such a stream is an opening or a closing tag. Additionally, the streams considered here contain annotations that appear immediately after opening tags. During the evaluation of a query, annotations are used to mark in the stream nodes selected by its subqueries.

An annotation is expressed using a finite (possibly empty) list of natural numbers in ascending order, e.g., [1,2]. There are two special annotations: the empty annotation, noted [ ], corresponding to the empty list, and the full annotation, noted [0], corresponding to the list containing all annotations. There are three operations defined for annotations: union $\sqcup$, intersect $\sqcap$, and inclusion $\sqsubseteq$, the semantics of which resemble that of the well-known set operations $\cup$, $\cap$, and $\subseteq$. For example, the operation $c \sqcup s$ denotes the union of annotations c and s with duplicate removal, like in [1,2]$\sqcup$[2,3]=[1,2,3]. Any annotation contains the empty annotation and is contained in the full annotation.

Although a node is not a stream message, we may often speak in the following about streams made up of nodes, rather than of tags. This more abstract view upon a stream is motivated by conciseness, clarity, and also by the vocabulary congruence of processing streams with functions, as detailed later in this section, and computing answers with the LGQ semantics introduced in Chapter 3. In this respect, it should be clear that the wordings (1) "all children of a node n are annotated in the output stream with the annotation of n from the input stream" and (2) "all opening tags of children of a node n are immediately followed in the result stream by the annotation that immediately follows the opening tag of n in the input stream" are equivalent.

Let E be the set of all opening and closing tags, A the set of all possible annotations, and $M = E \cup A$ the set of all stream messages, i.e., the set of annotations and tags. A stream s over a set of messages M is a (finite and possibly unbounded) sequence of messages: $s \in M^* = \bigcup_{n \geq 0} M^n$. We write a stream containing the messages m_1 and m_2 in this order as $m_1 m_2$ (while reading from left to right, the stream comes from right to left).

Relations on streams. Each message in a stream has an identity given by its position in the stream: to denote a message m in a stream, we may alternatively write (m, i) to explicitly state that the message m appears at position $i \in \mathbb{N}$ in that stream. Using positions in streams, one can differentiate two distinct messages with the same content. However, if not explicitly needed, the position of a message in a stream may be skipped.

The document order $\ll$ on nodes in trees, i.e., the depth-first, left-to-right preorder, is applicable also to nodes in streams. For a given stream s, the order $\ll_s$ is the total

order among the messages of the stream as given by the natural order of their positions: $(m, i) \ll_s (m', i')$ if $i < i'$.

The relation $\lessdot$ is the membership relation between messages and streams: $m \lessdot s$ means that the message m is in the stream s. The relation @ is defined only for nodes in streams and denotes the annotation of nodes: $@(n)$ is the annotation of the node n. Because the same node can have different annotations in different streams, we may write $@_s(n)$ to explicitly state that the node n has the annotation $@_s(n)$ in the stream s.

5.2.2 Stream Processing Functions

We consider here a class of functions, called stream processing functions, that take as input x streams and return y streams of type M^*, which additionally are preorder serializations of the *same* tree:

$$f : (M^*)^x \to (M^*)^y.$$

The stream processing functions are node-preserving and node-monotone. Node-preserving means that the nodes from the input streams belong also to the output streams. Node-monotone means that the order of nodes in the input streams is preserved also in the output streams. Note that both properties are ensured if all input and output streams are preorder serializations of the same tree. The only changes done by such functions refer to the annotations of nodes. Stream processing functions are defined in the following sections by specifying only the differences between the input and the output streams. It is implicitly assumed that besides the specified changes, the other messages are simply copied from the input to the output streams.

Function compositions. We use here two kinds of function composition: the sequential composition $\cdot$ and the parallel composition $+\!\!+$:

$$(f \cdot g)(x) = g(f(x))$$

$$(f +\!\!+ g)(x) = (f(x), g(x))$$

We consider that the sequential composition $(\cdot)$ binds stronger than the parallel composition ($+\!\!+$).

Note that these compositions are analogous to the composition of LGQ formulas: path formulas are constructed by the sequential composition of atoms, and tree and forest formulas are constructed by the parallel composition of atoms.

A peculiarity of the evaluation order of sequential compositions of stream processing functions is that the component functions are evaluated stepwise, such that each stream message output by the first function becomes the input to the next function before the first function processes the next message in the input stream. This way, the intermediary streams need not be stored.

Base functions. For processing M^* streams, three base functions are defined: the *merge* function $\oplus$, the *filter* function $|$, and the *annotation-merge* function Θ_c.

The merge function $\oplus : M^* \times M^* \rightarrow M^*$ intertwines two input streams such that in the result stream one message from the first stream is followed by one message from the second stream. The closing tags from the streams (if there are any) are simply added to the result stream without requiring a counterpart message from the other stream. If a stream is shorter than the other, then the remainder of the bigger stream is simply appended to the result stream.

We may write the $\oplus$-function as an infix operator, i.e., $s_1 \oplus s_2$ for the $\oplus$-application on the streams s_1 and s_2. A straightforward usage of the $\oplus$-function is to annotate a stream containing only messages corresponding to nodes (i.e., tags and strings) with annotations from another stream. The $\oplus$-function can be used also as a more general stream constructor: the concatenation of a message m with a stream s can be expressed by $\oplus$-merging the stream containing the message m and the stream s: $m \oplus s$.

The symbol-filter function $|$ takes a stream and a set of messages and returns the fragment of the input stream, where the messages not occurring in the input message set are filtered out: $|: M^* \times \Sigma \rightarrow M^*$. For a stream s and a message set Σ, we write the $|$-function as an infix operator $s|_\Sigma$ for the application of $|$ on s and Σ.

$$|: M^* \times \Sigma \rightarrow M^*, s|_\Sigma = \langle x \mid x \lessdot s \land x \in \Sigma \rangle.$$

It is easy to observe that for a stream $s \in M^*$, the $\oplus$-merging of the two substreams $s|_E$ and $s|_A$ of a stream s yields back the original stream s:

$$s = s|_E \oplus s|_A.$$

The annotation-merge function $\Theta_c : M^* \times M^* \rightarrow M^*$ for a boolean connective c is a node-preserving and node-monotone function that takes two streams with the same nodes and returns one stream where a node appears only once. An annotation a appears in the output stream if and only if a appears in *both* input streams (for $c = \land$), or in *at least one* input stream (for $c = \lor$). More specifically, the annotation a appears in the output stream as soon as it is encountered in the input stream(s): a appears in the output stream at a position i, if and only if a appears (1) in one input stream at position i and in the other input stream at a position lower than or equal to i (for $c = \land$), or (2) in at least one input stream at the position i (for $c = \lor$). The function Θ_c is defined using the following equivalences, where the function is used in infix form:

$$(n,i) \lessdot s_1|_E \land (n,i) \lessdot s_2|_E \Leftrightarrow (n,i) \lessdot (s_1 \;\Theta_c\; s_2)|_E.$$
$$a \sqsubseteq a_1 \land a \sqsubseteq a_2 \land (a_1,i_1) \lessdot s_1|_A \land (a_2,i_2) \lessdot s_2|_A \Leftrightarrow a \sqsubseteq a' \land (a', \mathsf{max}(i_1,i_2)) \lessdot (s_1 \;\Theta_\land\; s_2)|_A.$$
$$(a,i) \lessdot s_1|_A \lor (a,i) \lessdot s_2|_A \Leftrightarrow a \sqsubseteq a' \land (a',i) \lessdot (s_1 \;\Theta_\lor\; s_2)|_A.$$

The variables appearing only in the left side of equivalences are universally quantified, whereas the remaining ones are existentially quantified. The first equivalence ensures that in the output stream, the node from both input streams appear only once. The last two

equivalences ensure that annotations are computed in the output stream according to the textual definition of the Θ_c function. A simplified equivalence concerning the annotations is inferred from the last two equivalences:

$$\underset{1\leq j\leq 2}{C}(a \sqsubseteq a_j \wedge a_j \lessdot s_j|_A) \Leftrightarrow a \sqsubseteq a' \wedge a' \lessdot (s_1\ \Theta_c\ s_2)|_A.$$

For k input streams, the previous equivalences become:

$$\bigwedge_{1\leq j\leq k} ((n,i) \lessdot s_j|_E) \Leftrightarrow (n,i) \lessdot (s_1\ \Theta_\vee\ \ldots\ \Theta_\vee\ s_k)|_E. \tag{5.1}$$

$$\bigwedge_{1\leq j\leq k} (a \sqsubseteq a_j \wedge (a_j, i_j) \lessdot s_j|_A) \Leftrightarrow a \sqsubseteq a' \wedge (a', \underset{1\leq j\leq k}{\mathsf{MAX}}(i_j)) \lessdot (s_1\ \Theta_\wedge\ \ldots\ \Theta_\wedge\ s_k)|_A. \tag{5.2}$$

$$\bigvee_{1\leq j\leq k} ((a,i) \lessdot s_j|_A) \Leftrightarrow a \sqsubseteq a' \wedge (a',i) \lessdot (s_1\ \Theta_\vee\ \ldots\ \Theta_\vee\ s_k)|_A. \tag{5.3}$$

$$\underset{1\leq j\leq k}{C}(a \sqsubseteq a_j \wedge a_j \lessdot s_j|_A) \Leftrightarrow a \sqsubseteq a' \wedge a' \lessdot (s_1\ \Theta_c\ \ldots\ \Theta_c\ s_k)|_A. \tag{5.4}$$

5.2.3 From LGQ to Stream Processing Functions

This section gives the translation scheme of LGQ forest queries into stream processing functions and shows how the LGQ semantics defined in Chapter 3 and the evaluation of such functions are related.

Translation Scheme of Forward LGQ Forest Queries to Function Graphs

The translation scheme has three distinct phases, as detailed below.

Pre-translation Phase. In this phase, we simplify the LGQ forest queries. First, the query to be translated is brought in disjunctive normal form. Second, each atom $\mathsf{self}(x,y)$ appearing in a disjunct is removed and the variable y is replaced by x in that disjunct (however, if y is the head variable, then y replaces x). Third, we add a new unary predicate, *head*, to the head variable in each disjunct. The semantics of this novel predicate is the same as for a wildcard nodetest predicate. Thus, it does not change the semantics of the query, and serves solely the purpose of a simplified translation phase, as detailed next.

Translation Phase. The translation of the body formula of a forward LGQ forest query is given in Figure 5.1 by the translation function $\mathcal{F}$ defined using pattern matching on the structure of LGQ forest formulas.

The result of a formula translation is a stream processing function representing sequential and parallel compositions of (1) the functions α_f for LGQ predicates α, (2) the function $\boxed{head}$ for the novel head predicate *head*, (3) the functions $\overrightarrow{scope}$ and $\overleftarrow{scope}$ for dealing with the treeness of queries, (4) the input and output functions *in* and *out*, and (5) the identity function *Id*. Two functions are composed in sequence, if they are induced by two formulas

$$
\begin{aligned}
&\mathcal{F} : \text{Formula} \times \text{Variable} \rightarrow M^* \rightarrow M^* \\
&\qquad\qquad \mathcal{F}[\![L \vee R]\!](x) = (\mathcal{F}[\![L]\!](x) +\!\!+ \mathcal{F}[\![R]\!](x)) \cdot \vee_f \\
&\quad \mathcal{F}[\![\mathsf{root}(y) \wedge R]\!](x) = in \cdot \mathcal{F}[\![R]\!](y) \cdot out \\
&\quad \mathcal{F}[\![\alpha(x,y) \wedge R]\!](x) = \overrightarrow{scope}_x \cdot (\alpha_f \cdot \mathcal{F}[\![R]\!](y) +\!\!+ \mathcal{F}[\![R]\!](x)) \cdot \wedge_f \cdot \overleftarrow{scope}_x \\
&\qquad \mathcal{F}[\![\eta(x) \wedge R]\!](x) = \begin{cases} \eta_f \cdot \mathcal{F}[\![R]\!](x) & , \eta \neq head \\ \overrightarrow{scope}_x \cdot (\boxed{head} +\!\!+ \mathcal{F}[\![R]\!](x)) \cdot \overleftarrow{scope}_x & , \eta = head \end{cases} \\
&\qquad\quad \mathcal{F}[\![\alpha(x,y)]\!](x) = \alpha_f \\
&\qquad\qquad \mathcal{F}[\![\eta(x)]\!](x) = \eta_f \\
&\quad \mathcal{F}[\![\alpha(y,z) \wedge R]\!](x) = \mathcal{F}[\![\eta(y) \wedge R]\!](x) = \mathcal{F}[\![R]\!](x) \\
&\qquad\quad \mathcal{F}[\![\alpha(y,z)]\!](x) = \mathcal{F}[\![\eta(y) \wedge R]\!](x) = Id.
\end{aligned}
$$

Figure 5.1: Translation Scheme of LGQ Forest Formulas to Function Graphs

such that the sink variable of the first formula is the source variable of the second one. Two functions are composed in parallel, if they are induced by two formulas that have the same source variable. The next sections of this chapter detail all these functions.

The *graph* of a stream processing function obtained by translating a query is constructed similarly to the digraph representation of that query. In a function graph, the nodes are labeled with the component functions and a directed edge exists between two nodes if there is a sequential composition of the functions labeling these nodes, or of the function labeling the first node and a parallel composition of some other component functions and the function labeling the second node. The source nodes of a function graph are labeled by the function *in*, its sinks by the function *out*, and the inner nodes are labeled with functions for predicates and with the functions $\overrightarrow{scope}$ and $\overleftarrow{scope}$.

Note that the translations of two queries, which are the same modulo the commutativity of the $\wedge$ connective, yield two non-isomorphic function graphs, though their evaluation results are the same. The next and last translation phase simplifies such function graphs such that they become isomorphic (and even smaller).

Remark 5.2.1. The LGQ negation and the single-join DAGs are not considered in this chapter. The evaluation approach presented here can, however, be extended to treat them too. A publicly available query evaluator (`http://spex.sourceforge.net`) based on the results of this chapter supports both aforementioned extensions. □

Post-translation Phase. The outcome of the translation phase can be simplified in several directions, while still preserving its semantics. The simplification can be achieved by the term rewriting system defined below. Although not shown here, it can be checked that the system is terminating and confluent modulo the associativity and commutativity

of $+\!\!+$. The variables X, Y, and Z stand for arbitrary (compositions of) functions, x stands for LGQ variables.

$$\overrightarrow{scope}_x \cdot (X +\!\!+ Id) \cdot \wedge_f \cdot \overleftarrow{scope}_x \to X \tag{5.5}$$

$$\overrightarrow{scope}_x \cdot (X +\!\!+ \eta_f) \cdot \wedge_f \cdot \overleftarrow{scope}_x \to \eta_f \cdot X \tag{5.6}$$

$$\overrightarrow{scope}_x \cdot (\overrightarrow{scope}_x \cdot (X +\!\!+ Y) \cdot \wedge_f \cdot \overleftarrow{scope}_x +\!\!+ Z) \cdot \wedge_f \cdot \overleftarrow{scope}_x \tag{5.7}$$
$$\to \overrightarrow{scope}_x \cdot (X +\!\!+ Y +\!\!+ Z) \cdot \wedge_f \cdot \overleftarrow{scope}_x$$

$$((X +\!\!+ Y) \cdot \vee_f +\!\!+ Z) \cdot \vee_f \to (X +\!\!+ Y +\!\!+ Z) \cdot \vee_f \tag{5.8}$$

$$(in \cdot X \cdot out +\!\!+ in \cdot Y \cdot out) \cdot \vee_f \to in \cdot (X +\!\!+ Y) \cdot \vee_f \cdot out \tag{5.9}$$

$$X \cdot Y +\!\!+ X \cdot Z \to X \cdot (Y +\!\!+ Z) \tag{5.10}$$

$$X +\!\!+ X \cdot Z \to X \cdot Z \tag{5.11}$$

Rule (5.5) eliminate the identity function *Id*. Rules (5.6), (5.7), and (5.8), relax the rankedness of function graphs, i.e., every node in a function graph can have now more than two outgoing edges. The parallel compositions of functions for unary predicates and other functions are transformed into sequential compositions of the latter and the former. Rule (5.9) factors out the functions *in* and *out*. Further simplifications with (5.10) and (5.11) factor out common prefixes of subgraphs with the same source.

Example 5.2.1. Consider the LGQ tree query

$$Q(v_3) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge \mathsf{a}(v_1) \wedge \mathsf{child}^+(v_1, v_2) \wedge \mathsf{d}(v_2) \wedge \mathsf{child}^+(v_1, v_3) \wedge \mathsf{c}(v_3)$$

that selects all c-nodes descendants of a-nodes that have at least a d-descendant. Figure 5.2.3 shows two simplified versions of the function graph for the body of Q after adding the $\boxed{head}$ predicate. In contrast to the first version from Figure 5.2(a), the second version is the normal form, i.e., it is not anymore reducible using the rewrite rules of the post-translation phase. ☐

It is easy to see that the above translation scheme creates function graphs linear in the size of the input query.

Proposition 5.2.1 (Linearity of the Translation Scheme). *For a given forward LGQ forest query, the translation scheme of Figure 5.1 creates a function graph linear in the size of that query.*

Proof. It results from the simple observation that the translation of each query construct induces a constant amount of functions in the function graph. ☐

Equivalence of LGQ Semantics and Evaluation of Stream Processing Functions

The LGQ semantics is given in Section 3.3 using the functions $\mathcal{LQ}$ for LGQ queries and $\mathcal{LF}$ for LGQ formulas. For a given set of substitutions of query variables to the nodes in the tree conveyed by the XML stream, these functions keep only those substitutions

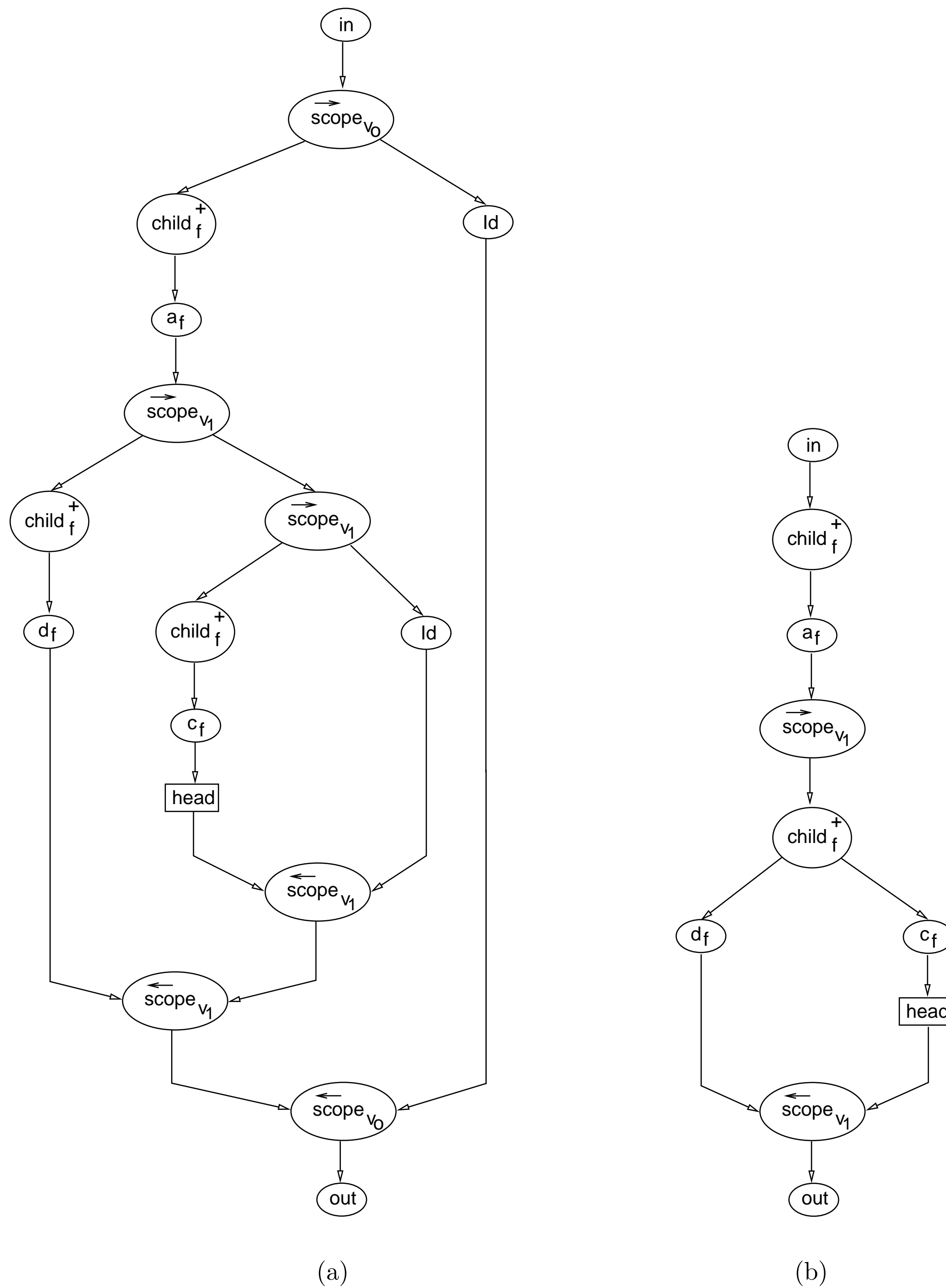


Figure 5.2: Two equivalent function graphs for the query of Example 5.2.1

that are consistent with the query and the tree, i.e., such that for any substitution the predicates on the variables holds also on their images.

For a query $Q(v) \leftarrow f$ and an XML stream s, the link between the semantics of Q and the answers computed by the stream processing function $\mathcal{F}[\![f \wedge head(v)]\!]$ is given by the function τ, which maps trees to XML streams.

Definition 5.2.1 (Tree-to-Stream mapping). *The function* τ : Tree $\rightarrow M^*$ *maps a tree to its depth-first, left-to-right, preorder traversal that yields an XML stream.*

For the query Q and a given tree T, the mapping between the LGQ semantics of Q $\mathcal{LQ}_T[\![Q(v) \leftarrow f]\!]$ and the stream processing function $\mathcal{F}[\![f \wedge head(v)]\!]$ follows by

$$\mathcal{F}[\![f \wedge head(v)]\!](\tau(T)) = \mathcal{LQ}_T[\![Q(v) \leftarrow f]\!]. \tag{5.12}$$

The next sections gives stream processing functions that specify the evaluation of LGQ formulas of increased complexity: atoms, paths, and trees. Then, the computation of answers using these stream processing functions is detailed.

5.2.4 Evaluation of Atoms

The evaluation of the α-atoms, where α is an LGQ predicate, is reduced here to the following problem: given a set of source nodes from a tree T, compute the set of sink nodes from T that stand in α with any of the source nodes. The problem is approached here by computing the sink nodes *simultaneously* for all source nodes (a *set-at-a-time* approach). Note that such an approach differs from, e.g., [11], where for each source node the set of sink nodes standing in a predicate with that source node is computed independently (a *node-at-a-time* approach). The subtle difference between the two approaches has a tremendous effect regarding both their efficiency and applicability in a stream environment. The node-at-a-time approach can compute for several source nodes non-disjunct sets of sink nodes that are then merged into a single set. Thus, some nodes can be visited several times. An example of non-disjunct sets of nodes computed from several source nodes is the set of descendant nodes of source nodes that stand themselves in a child^+ predicate: the set of sink nodes computed for a source node contains then the set of sink nodes computed for any of its descendants. The set-at-a-time approach computes simultaneously the set of sink nodes for all source nodes, thus avoiding the duplicate removal in the final merging phase of non-disjunct sets, and also to visit nodes several times.

For a uniform treatment of unary and binary atoms, we consider in the following binary variants of the unary atoms. In this sense, the binary variant $\eta(v_1, v_1)$ of the unary predicate $\eta(v_1)$ consists in the pairs of all nodes that are also in that unary predicate.

For each LGQ predicate α consisting of pairs of source and sink nodes from a tree T, we define the stream processing function $\alpha_f : M^* \rightarrow M^*$ with its input and output streams serializations of T, where the annotation of each source node in the input stream is non-empty and is included in the annotations of the sink nodes that stand in α with that source node.

Definition 5.2.2. *The node-preserving and node-monotone stream processing function $\alpha_f : M^* \to M^*$ for an LGQ predicate α computes for each sink node n' a new annotation that is the union of the annotations of all source nodes n that stand in α with n':*

$$s' = \alpha_f(s), \forall n' \lessdot s'|_E : @_{s'}(n') = \bigsqcup(@_s(n) \mid n \lessdot s|_E, \alpha(n, n')).$$

The annotations are the only messages that are changed in the output stream. The computation of new annotations expressed in the above equation meets the textual definition of the stream processing functions for LGQ predicates. The node n' gets the annotation $\bigsqcup(@_s(n) \mid n \lessdot s|_E, \alpha(n, n'))$ that is the union of annotations $@_s(n)$ of all nodes n in the stream s such that n stands in α with n'. The annotation a is empty either if $\alpha(n, n')$ does not hold for any n, or the annotation of n is empty.

The union of annotations is necessary because sink nodes n' can stand in a (transitive or reflexive transitive closure) predicate with several source nodes n. For example, a sink node can be the descendant of several source nodes. However, a sink node n' can stand in a non-closure predicate with (at most) one node n. For example, a sink node can be the child of (at most) one source node.

Example 5.2.2. Figure 5.2.4 shows a tree with annotated nodes, as conveyed in an input XML stream, and the reannotations of these nodes as generated by the application of functions (1) child_f, (2) $\mathsf{child}^+{}_f$, (3) $\mathsf{nextSibl}^+{}_f$, and (4) foll_f for processing the input stream.

The input stream contains two a-nodes annotated with [1] and [2], and three b-nodes annotated with [3], [4], and respectively with [ ]. Recall that an annotation for a node follows immediately the opening tag of that node.

1. The function child_f moves the annotation of each source node to its children.

2. The annotation of each node in the output stream of the function $\mathsf{child}^+{}_f$ is the union of annotations of all its ancestors. For example, the annotation of the first b-node becomes the union [1,2] of the annotations [1] and [2] of both a-nodes.

3. The function $\mathsf{nextSibl}^+{}_f$ annotates each node with the union of annotations of all sibling nodes that precede it. For example, the last b-node is annotated with the annotation [2] of its preceding sibling a-node.

The function foll_f annotates each node with the union of annotations of all nodes that precede it. For example, the last b-node is annotated with the union [2,3,4] of annotations all other b-nodes ([3] and [4]) and of the second a-node ([2]). □

As expressed by Definition 5.2.2 and exemplified by Figure 5.2.4, the annotation of any node n from the input stream is included in the annotations of all nodes n' in the output stream of a function α_f, if n stands in α with n'. Based on this observation, the following propositions give two important properties of annotations created by such functions.

Proposition 5.2.2 (Node reachability). *If a node n stands in predicate α with a node n', then the annotation $@_s(n)$ of n in the stream s is contained in the annotation $@_{\alpha_f(s)}(n')$ of n' in the stream $\alpha_f(s)$:*

$$\alpha(n, n') \Rightarrow @_s(n) \sqsubseteq @_{\alpha_f(s)}(n').$$

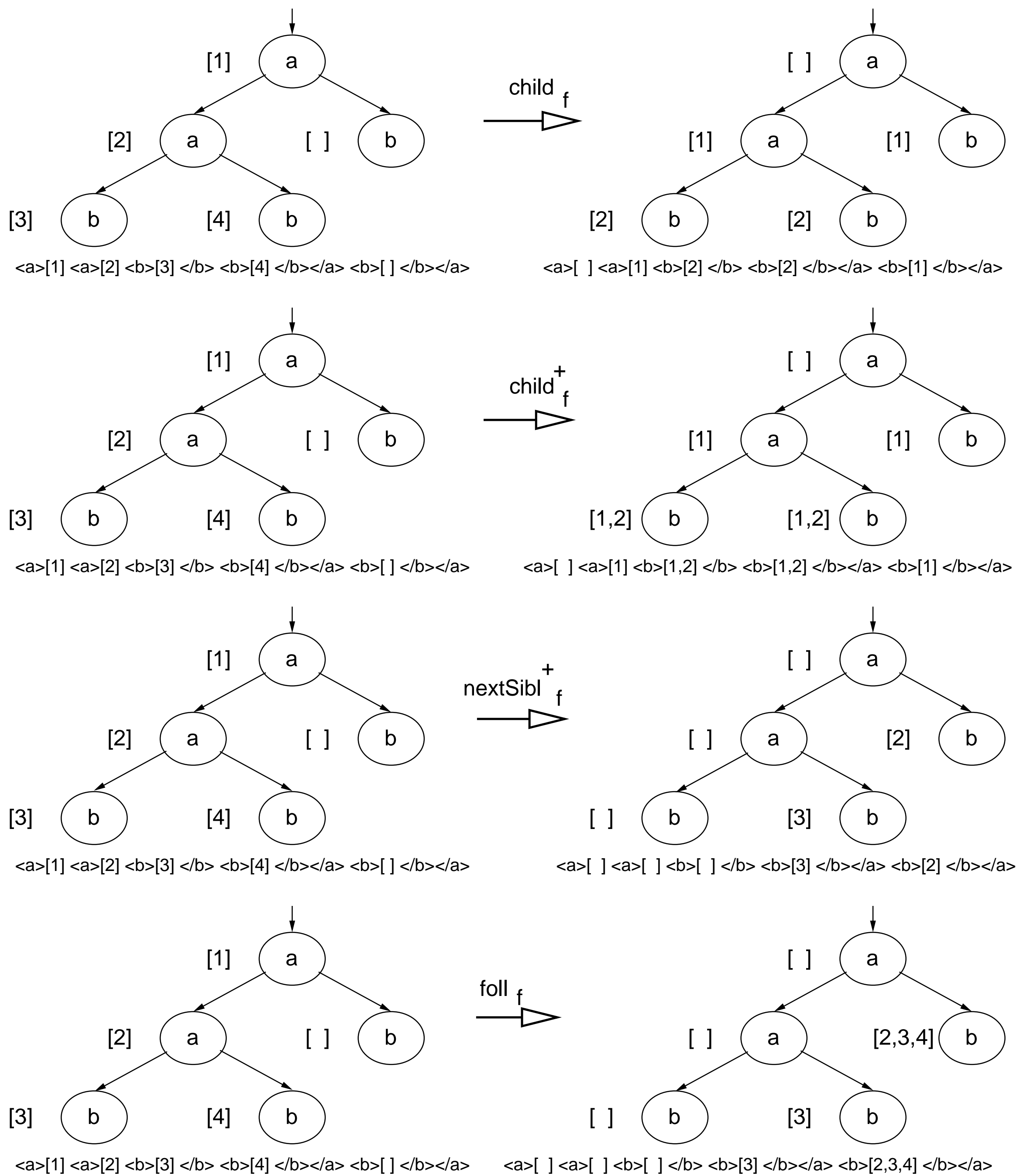


Figure 5.3: Processing with child_f, $\mathsf{child}^+{}_f$, $\mathsf{nextSibl}^+{}_f$, and foll_f

The above implication says also that if n has a non-empty annotation in s, then n' has also a non-empty annotation in the stream $\alpha_f(s)$:

$$\alpha(n, n'), @_s(n) \neq [\,] \Rightarrow @_{\alpha_f(s)}(n') \neq [\,].$$

The implication of Proposition 5.2.2 does not hold in both directions because several nodes can have the same annotations, as stated next. Thus, the annotations can not be used as node identifiers, for they need not be unique.

Proposition 5.2.3 (Annotation ambiguity). *The output stream of a stream processing function for an LGQ predicate can contain an annotation several times, or the intersection of two annotations in the output stream can be non-empty.*

5.2.5 Evaluation of Path Formulas

The translation of a path formula, which is a conjunction of atoms, yields a sequential composition of stream processing functions, which are translations of the component atoms.

Definition 5.2.3. *The node-preserving and node-monotone stream processing function $p_f : M^* \rightarrow M^*$ for an LGQ path formula $p = \alpha^1(v_0, v_1) \wedge \ldots \wedge \alpha^k(v_{k-1}, v_k)$ is the sequential composition of functions α^i_f for the predicates α^i ($1 \leq i \leq k$):*

$$p_f = \alpha^1_f \cdot \ldots \cdot \alpha^k_f.$$

Recall that the evaluation order of the stream processing function p_f imposes that all component functions are evaluated stepwise, such that each stream message output by the first function becomes the input to the next function before the first function processes the next message in the input stream. This way, the intermediary streams need not be stored.

Example 5.2.3. Figure 5.2.5 shows a tree with annotated nodes (top-left), as conveyed in an XML stream, and the reannotation of this tree (bottom-right), as generated by the stream processing function $p_f = \mathsf{child}_f \cdot \mathsf{nextSibl^+}_f \cdot \mathsf{b}_f \cdot \mathsf{foll}_f \cdot \mathsf{self}_f \cdot \mathsf{d}_f$ for evaluating the path formula $p(v_1, v_5) = \mathsf{child}(v_1, v_2) \wedge \mathsf{nextSibl^+}(v_2, v_3) \wedge \mathsf{b}(v_3, v_3) \wedge \mathsf{foll}(v_3, v_4) \wedge \mathsf{self}(v_4, v_5) \wedge \mathsf{d}(v_5, v_5)$. The intermediary results of the component functions child_f, $\mathsf{nextSibl^+}_f \cdot \mathsf{b}_f$, foll_f, and $\mathsf{self}_f \cdot \mathsf{d}_f$ are also shown, albeit they are not materialized during processing. The input stream contains two a-nodes annotated with [1] and [2], three b-nodes annotated with [3], [4], and [5], and one d-node that has an empty annotation. The function p_f computes a stream where the annotation of each node n moves to each d-node that follows b-next siblings of children of n. For our tree, the path p contains only the pair of the first a-node and the d-node, thus the latter node gets the annotation of the former in the output stream. The other nodes get the empty annotation.

1. The function child_f moves the annotation of each node to its children.

2. The annotation of each b-node in the output stream of the function $\mathsf{nextSibl^+}_f \cdot \mathsf{b}_f$ is the union of annotations of all its preceding sibling nodes. For example, the annotation of the second b-node becomes the annotation [2] of the first b-node, and the annotation of the first b-node becomes the empty annotation [ ], for there are no preceding siblings of it.

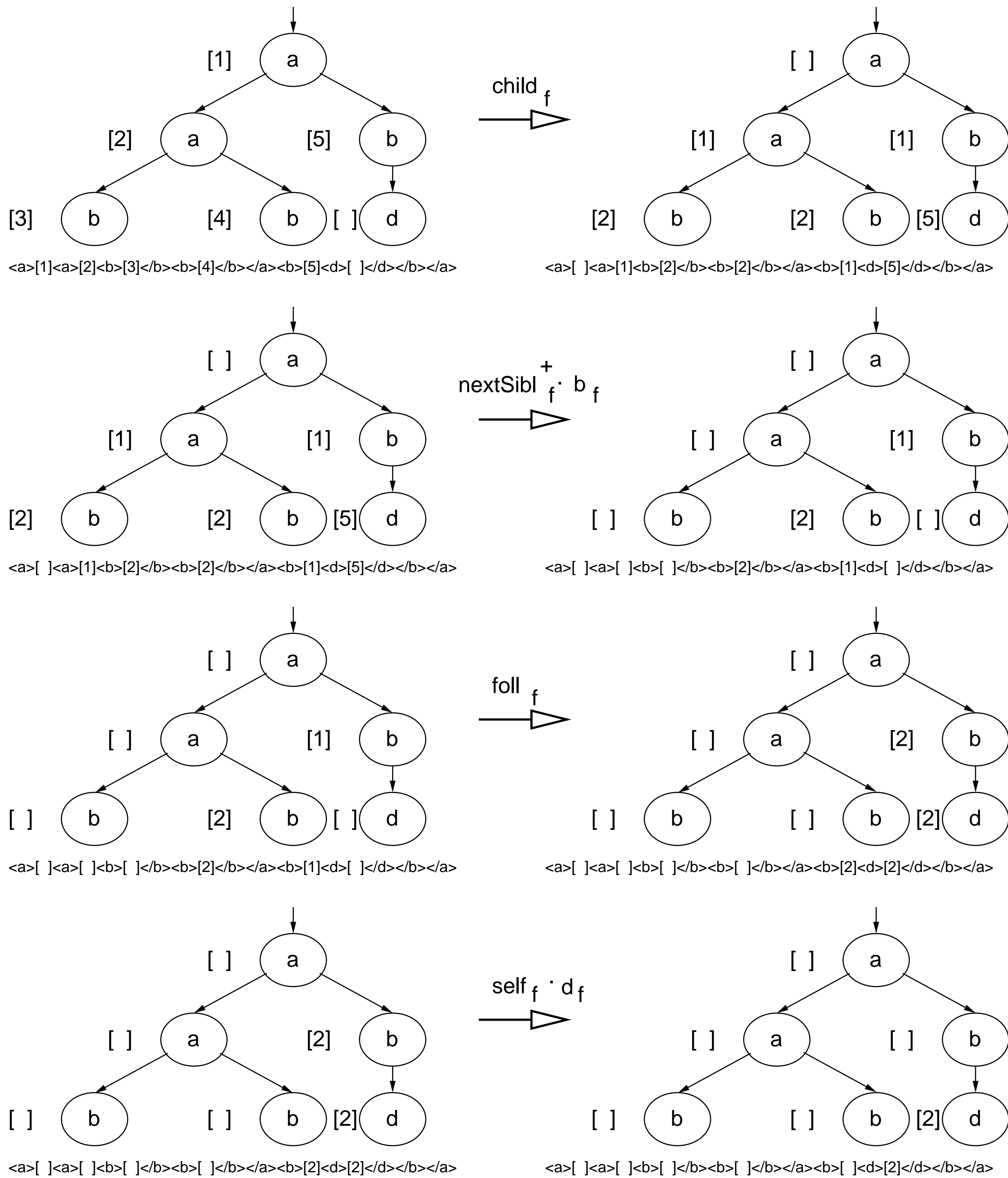


Figure 5.4: Processing with $\mathsf{child}_f \cdot \mathsf{nextSibl}^+{}_f \cdot \mathsf{b}_f \cdot \mathsf{foll}_f \cdot \mathsf{self}_f \cdot \mathsf{d}_f$

3. The function foll_f annotates each node with the union of annotations of all nodes that precede it. For example, the last b-node is annotated with the union [2] of annotations of all other b-nodes ([ ] and [2]) and of the second a-node ([ ]).

4. The function $\mathsf{self}_f \cdot \mathsf{d}_f$ replaces all annotations of nodes with other label than d with the empty annotation. Thus, the only d-node in the XML stream keeps its non-empty annotation. □

The processing of streams with functions for path formulas p inherit the properties of processing streams with functions for constituent predicates:

- the nodes from the input stream are copied in the output stream, and
- the annotation of each source node is moved in the output stream to each sink node that stands in p with that source node.

Also, the properties of annotations from streams created by functions for predicates hold also for annotations from streams created by functions for path formulas p:

- the annotation of each sink node in the output stream contains the annotations of all source nodes that stand in p with that sink node (node reachability), and
- an annotation can appear several times in the output stream (annotation ambiguity).

These results can be easily derived from the analogous Propositions 5.2.2 and 5.2.3 by using complete induction over the number of predicates in the path.

5.2.6 Evaluation of Tree Formulas

Tree formulas are structurally more complex than path formulas in that they allow multi-source variables, thus several subformulas having the same source variable. The evaluation of such subformulas is based on the parallel composition of their corresponding stream processing functions, because these functions have to process the same stream containing source nodes that are bindings for their common source variable. This consideration rises several new challenges. First, the output streams of these functions must be brought together into a single aggregated stream that is the serialization of the same tree as the input stream, and contains some of the annotations that appear in the output streams. Second, in order to find the sink nodes standing in a given tree formula with a source node, the relation between the source nodes from the input stream and the sink nodes from the aggregated stream should be uniquely established with help of the annotations propagated from the input to the aggregated stream.

The first challenge is partially solved by the evaluation order of stream processing functions: because one message is processed by all functions at a time in the order dictated by their compositions, the aggregation of the output streams resumes to delivering further the same message when read from all output streams. Additionally, the annotation of each message in the aggregated stream depends on the annotations already read in all the output streams.

The second challenge can not be solved immediately and needs significant extensions of the current evaluation strategy. Its non-triviality stems from the fact that several source nodes can have the same annotation in the input stream, thus it is not immediately clear which sink nodes stand in a given tree formula with a source node. As detailed later, one possibility is (1) to reannotate uniquely the source nodes from the input streams that are to be processed by several functions in parallel, and (2) to remember the mappings between the original and the new annotations.

Both challenges are addressed in detail next.

Stream Aggregation

The translation scheme of Section 5.2.3 translates boolean connectives $c \in \{\wedge, \vee\}$ of LGQ formulas to corresponding stream processing functions c_f.

Definition 5.2.4 (Connective Functions). *For k input streams that are annotated serializations of the same tree T, the node-monotone stream processing functions $\wedge_f, \vee_f : (M^*)^k \rightarrow M^*$ compute output streams that are also serializations of T and where an annotation marks a node only if it appears before that node in all input streams ($\wedge_f$), respectively in at least one of the input streams ($\vee_f$):*

$$c_f(s_1, \ldots, s_k) = s_1 \ \Theta_c \ \ldots \ \Theta_c \ s_k, \ c \in \{\wedge, \vee\}.$$

The above equation of c_f matches its textual counterpart, because the annotation-merge function Θ_c, defined in Section 5.2.2, computes the aforementioned aggregation of streams. Note also that, according to the definition of the parallel composition $+\!\!+$ of functions, the application of c_f on k streams is the application of c_f on the parallel composition of all streams s_1 to s_k: $c_f(s_1, \ldots, s_k) = c_f(s_1 +\!\!+ \ldots +\!\!+ s_k)$.

Annotation Mappings

Recall that the annotations of nodes in an arbitrary stream are not necessarily unique, thus they are not identifiers for the nodes they accompany, as also stated by Proposition 5.2.3. This fact makes it difficult to detect which sink nodes from the output streams of several functions composed in parallel stand in a tree formula, specified by these functions, with the same source node from an input stream. In order to overcome this difficulty, we (1) reannotate uniquely the source nodes from the input stream, and (2) remember the mappings between the original and the new annotations. By using unique annotations for all source nodes in the input stream, the detection in the output streams of sink nodes, which stand in the tree formula with a source node, is reducible to testing whether the unique annotation of that source node is contained in all annotations of these sink nodes, cf. Proposition 5.2.2. The annotation mappings are necessary because the original annotations of the source nodes encode dependencies of these nodes to other nodes, as computed by functions corresponding to other subformulas.

In order to evaluate functions composed in parallel, we proceed then as follows. In the first phase, the source nodes from the input stream are reannotated uniquely and the

mappings between the original and the fresh annotations are stored within the stream. In the second phase, the functions composed in parallel process the same new input stream and deliver their output streams. In the third phase, the output streams are aggregated. In the fourth and last phase, the fresh annotations from the aggregated stream are mapped back to their original counterparts. In this way, the old dependencies of the source nodes conveyed by the original annotations are kept while safely evaluating the parallel composition of functions. Thus, the fresh annotations are used exclusively for the evaluation of parallel compositions and are dropped afterwards.

There is a general resemblance of this evaluation strategy to the implementation of function calls in abstract machines for programming languages: the initial phase declares new variables within the scope of the called function, the next two phases use these variables to compute and carry the results of the function call, and the last phase maps these results to values of variables from the upper scope, where the function is called.

The phases are realized as follows. The first phase is done by the so-called scope-begin functions $\overrightarrow{scope}$, which create annotation scopes, the second phase is done by the functions composed in parallel, the third phase is done by the connective functions $\wedge_f$ and $\vee_f$ defined above, and the last phase is done by the so-called scope-end functions $\overleftarrow{scope}$, which close annotation scopes.

We detail now on how the mappings are created and used. For each multi-source variable $i \in \text{Vars}(f)$ of a formula f, we define the in-mapping function $\stackrel{i}{\rightarrow}$ and the out-mapping function $\stackrel{i}{\leftarrow}$ that map non-empty source annotations a to sink annotations b. The annotation mappings are written $a \stackrel{i}{\rightarrow} b$ for in-mappings and $b \stackrel{i}{\leftarrow} a$ for out-mappings. Because $\stackrel{i}{\rightarrow}$ and $\stackrel{i}{\leftarrow}$ are functions, they can not map the same source annotation to different sink annotations. For in-mappings, the annotations a and the fresh annotations b are non-empty, whereas for out-mappings the annotations b can be also empty.

Such annotation mappings are created only for multi-source variables, because only in this case the annotations in the input stream need to be unique. Annotations of a multi-source variable t can stand in an in-mapping relation with annotations of another multi-source variable s that leads to t in f $s \leadsto_f t$ and there is no multi-source variable u with $s \leadsto_f u \leadsto_f t$. Between the full annotation created by the *in* function and the annotations of any other multi-source variables can hold the transitive closure in-mapping relation $\stackrel{i}{\rightarrow}^+$. For a multi-source variable i, the relation $\stackrel{i}{\rightarrow}^+$ is the set of pairs of the full annotation $[0]$ and the (non-empty) annotations a_i, if

- there is no multi-source variable leading to i, or
- there is a multi-source variable j with $a_j \subseteq a'_j \stackrel{i}{\rightarrow} a_i$ and $[0] \stackrel{j}{\rightarrow}^+, a_j$.

More compact, the transitive closure in-mapping relation is defined by the equivalence

$$[0] \stackrel{i}{\rightarrow}^+ a_i \Leftrightarrow [0] \stackrel{i}{\rightarrow} a_i \text{ or } [0] \stackrel{j}{\rightarrow}^+ a_j \sqsubseteq a'_j \stackrel{i}{\rightarrow} a_i.$$

If $[0] \stackrel{i}{\rightarrow}^{+} a_i$ holds, we say that a_i is reachable from [0]. This means also there is at least one annotation in-mapping for each multi-source variable connected to i that maps annotations reachable from [0] and from which a_i is reachable.

Analogously, the transitive closure out-mapping relation $\stackrel{i}{\leftarrow}^{+}$ is the set of all pairs of (non-empty) annotations a_i and [0] such that [0] can be reached from a_i using (at least) one annotation out-mapping of each multi-source variable leading to i:

$$[0] \stackrel{i}{\leftarrow}^{+} a_i \Leftrightarrow [0] \stackrel{i}{\leftarrow} a_i \text{ or } [0] \stackrel{j}{\leftarrow}^{+} a_j \sqsubseteq a'_j \stackrel{i}{\leftarrow} a_i.$$

An annotation mapping is expressed in the stream as a new message type. The set of stream messages M is now extended to contain also the set of annotation mappings $A^{\leftrightarrow}$:

$$A^{\leftrightarrow} = \bigcup_{i \in \text{Vars}} (\stackrel{i}{\rightarrow} \cup \stackrel{i}{\leftarrow}) = \{a\ X\ b \mid a \in A, b \in A, X \in \{\stackrel{i}{\rightarrow}, \stackrel{i}{\leftarrow}\}, i \in \text{Vars}(f), f \in LGQ\}.$$

An annotation mapping $a\ X\ b$ follows in the stream the fresh annotation b (for $X = \stackrel{i}{\rightarrow}$) or the annotation a (for $X = \stackrel{i}{\leftarrow}$), hence also the node having that annotation. The number of annotation mappings that can accompany a node is bounded in double the number of multi-source variables, because a node can have at most one annotation in-mapping (respectively at most one annotation out-mapping) for each such variable. The annotation mappings of a node in a stream can be accessed using the function $\mu : M^* \times E \rightarrow \mathcal{P}(A^{\leftrightarrow})$ that returns for a given node the set of all its annotation mappings in a given stream.

Annotation Scopes

An annotation scope delimits the lifetime of a fresh annotation, similar to the scope delimiting the lifetime of variables declared locally to procedures in programming languages. The lifetime of a fresh annotation spans over the stream fragment delimited by the source node having that annotation in the input stream and the last of its sink nodes in the aggregated stream. Recall that the sink nodes come always after the source nodes, because the stream processing functions, which compute the output streams to be aggregated, specify forward LGQ formulas. In the following, we consider first that the end of such a stream fragment coincides with the end of the whole stream. Then, it is shown that depending on the type (sdown, pdown, or rdown) of the LGQ formula specified by the stream processing functions, the lifetime of a fresh annotation can be considerably shortened. It is, of course, of advantage to fix at compile-time the maximum lifetime of an annotation. In this way, annotations that are not further needed can be discarded during processing, and not only at the very end.

An annotation scope is opened and closed by two complementary stream processing functions scope-begin $\overrightarrow{scope}$ and scope-end $\overleftarrow{scope}$. When opening it, in-mappings of original and fresh annotations are created, and original annotation are replaced by fresh annotations. The closing of an annotation scope consists in mapping back the fresh annotations to the original ones, using out-mappings.

Definition 5.2.5 (Scope-Begin). *Consider a multi-source variable i and a stream s. The node-preserving and node-monotone function $\overrightarrow{scope}_i : M^* \rightarrow M^*$ (1) creates a fresh annotation $@_{s'}(n) = \mathsf{new}(@_s(n))$ for each non-empty annotation $@_s(n)$, and (2) adds the in-mapping message $@_s(n) \xrightarrow{i} @_{s'}(n)$ after a_2 to the output stream:*

$$s' = \overrightarrow{scope}_i(s), \forall n \lessdot s'|_E, @_s(n) \neq [\,] : @_{s'}(n) = \mathsf{new}(@_s(n)),$$
$$\mu_{s'}(n) = \mu_s(n) \cup \{(@_s(n) \xrightarrow{i} @_{s'}(n))\}.$$

The new function creates a fresh annotation for each received non-empty annotation. The above definition describes only the stream changes done by $\overrightarrow{scope}$. The other messages are copied unchanged from the input to the output stream.

Definition 5.2.6 (Scope-End). *Consider a multi-source variable i and a stream s. The node-preserving and node-monotone function $\overleftarrow{scope}_i : M^* \rightarrow M^*$ (1) creates for each non-empty annotation $@_s(n)$ the union $@_{s'}(n)$ of all annotations that are mapped in s to parts of $@_s(n)$, and (2) adds the out-mapping message $@_s(n) \xleftarrow{i} @_{s'}(n)$ after $@_{s'}(n)$ to the output stream:*

$$s' = \overleftarrow{scope}_i(s), \forall n \lessdot s'|_E : @_{s'}(n) = \bigsqcup(a_1 \mid a_2 \sqsubseteq @_s(n), (a_1 \xrightarrow{i} a_2) \lessdot s),$$
$$\mu_{s'}(n) = \mu_s(n) \cup \{(@_{s'}(n) \xleftarrow{i} @_s(n))\}.$$

As for $\overrightarrow{scope}$, the above definition describes only the stream changes done by $\overleftarrow{scope}$. The other messages are copied unchanged from the input to the output stream.

Example 5.2.4. Figure 5.5 shows the evaluation of the function

$$\overrightarrow{scope} \cdot ((\mathsf{child^+}_f \cdot \mathsf{a}_f \mathbin{+\!\!+} \mathsf{child^+}_f \cdot \mathsf{c}_f) \cdot \wedge_f \mathbin{+\!\!+} \mathsf{nextSibl}_f \cdot \mathsf{b}_f) \cdot \vee_f \cdot \overleftarrow{scope}$$

that specifies the LGQ formula

$$(\mathsf{child^+}(v, v_2) \wedge \mathsf{a}(v_2) \wedge \mathsf{child^+}(v, v_3) \wedge \mathsf{c}(v_3)) \vee \mathsf{nextSibl}(v, v_4) \wedge \mathsf{b}(v_4).$$

For avoiding cluttering in the figure, we intentionally omitted the index v of $\overrightarrow{scope}$ and $\overleftarrow{scope}$. The figure shows the input stream and the tree conveyed within, together with selected streams representing the output of the component functions $\overrightarrow{scope}$, $\vee_f$, and $\overleftarrow{scope}$. The result of processing can be interpreted as follows: only the source nodes with non-empty annotations contained in annotations appearing in the output stream stand in that formula with some other nodes. In particular, these source nodes are the first b-node and the first a-node, because their annotation [2] appears in the output stream. ☐

Proposition 5.2.4. *Consider the stream processing functions scope-begin $\overrightarrow{scope}_i$, scope-end $\overleftarrow{scope}_i$ for a multi-source variable i, and the node-preserving and node-monotone f that does not filter out annotation mappings for i. Consider also the streams s_1, $s_2 = \overrightarrow{scope}_i(s_1)$, $s_3 = f(s_2)$, and $s_4 = \overleftarrow{scope}_i(s_3)$. Then, the following implication holds:*

$$\forall n, n' \lessdot s_1|_E : @_{s_2}(n) \sqsubseteq @_{s_3}(n') \Rightarrow @_{s_1}(n) \sqsubseteq @_{s_4}(n').$$

Moreover, if s_1 has only unique annotations, then the implication holds in both directions.

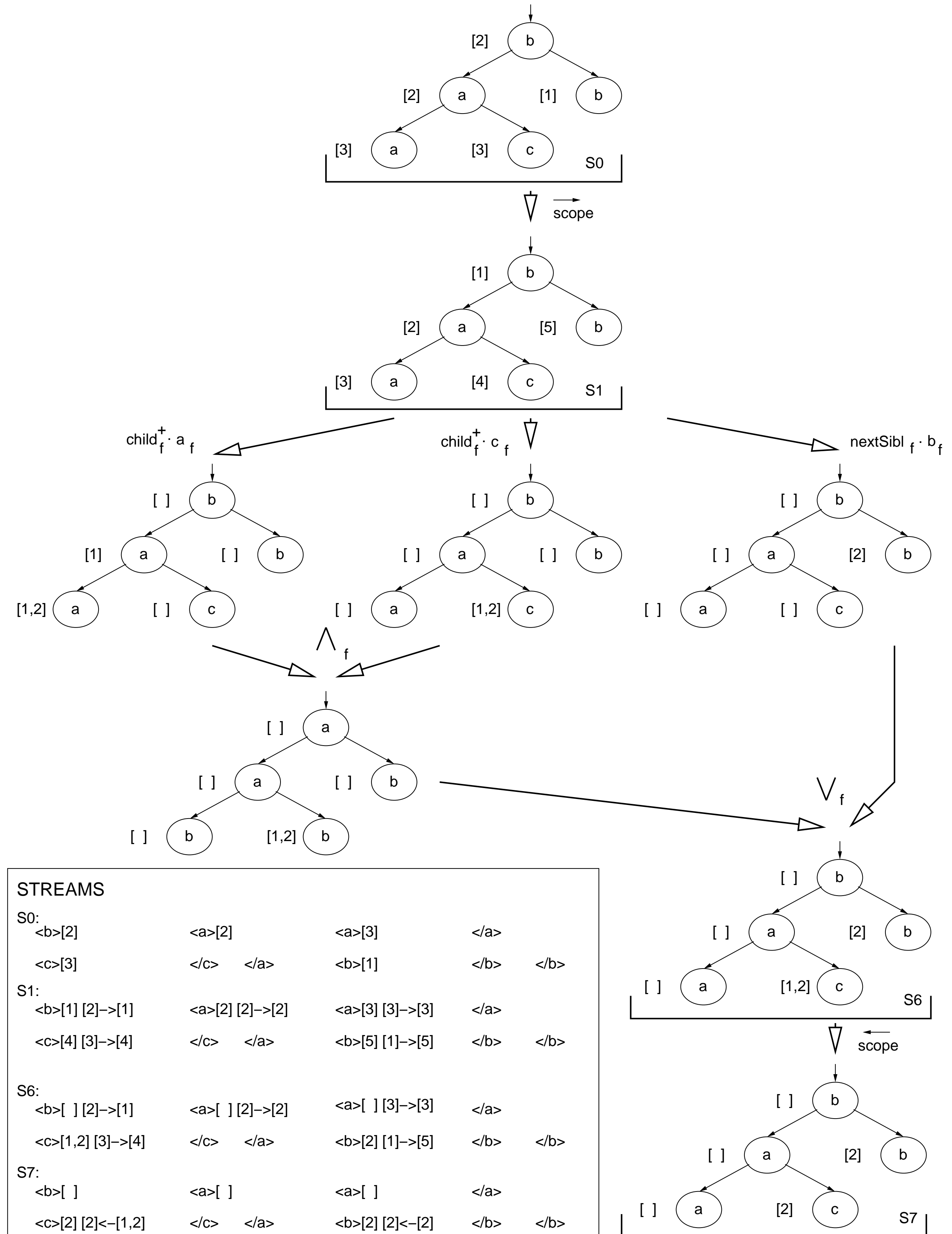


Figure 5.5: Processing with $\overrightarrow{scope}\cdot((\mathsf{child}^{+}{}_f\cdot\mathsf{a}_f +\!\!+ \mathsf{child}^{+}{}_f\cdot\mathsf{c}_f)\cdot\wedge_f +\!\!+ \mathsf{nextSibl}_f\cdot\mathsf{b}_f)\cdot\vee_f\cdot\overleftarrow{scope}$

Proof. The functions $\overrightarrow{scope}_i$ and $\overleftarrow{scope}_i$ preserve the non-emptiness of annotations, cf. Definitions 5.2.5 and 5.2.6. Hence, $@_{s_j}(n) \neq [\,] \Leftrightarrow @_{s_{j+1}}(n) \neq [\,]$, for $j \in \{1, 3\}$.

The case where $@_{s_1}(n) = @_{s_2}(n) = [\,]$ holds immediately, for $[\,] \sqsubseteq @_s(n')$ holds for every node n' in every stream s. It remains to prove the implication for the non-trivial case $@_{s_1}(n) \neq [\,]$.

$$\begin{aligned}
@_{s_2}(n) \sqsubseteq @_{s_3}(n') \Leftrightarrow (@_{s_1}(n) \xrightarrow{i} @_{s_2}(n)) \lessdot s_2,\ @_{s_2}(n) \sqsubseteq @_{s_3}(n'),\\
@_{s_4}(n') = \bigsqcup(a_1 \mid a_2 \sqsubseteq @_{s_3}(n'),\ (a_1 \xrightarrow{i} a_2) \lessdot s_4)\\
\rightarrow @_{s_1}(n) \sqsubseteq @_{s_4}(n').
\end{aligned}$$

The last implication is due to the observation that a_2 can be $@_{s_2}(n)$. □

Reducing the Annotation Scopes

The lifetime of a fresh annotation spans over the stream fragment delimited by the source node having that annotation in the input stream and the last of its sink nodes in the output stream. The position of those sink nodes relative to the source nodes is, however, highly dependent on the kind of formulas specified by the functions producing the output stream. We address next this issue for the three types of forward formulas introduced in Section 3.6: source-down (sdown), parent-down (pdown), and root-down (rdown).

Sdown formulas contain only sdown path formulas that relate any source node to some of its descendants. Thus, the lifetime of a fresh annotation marking a source node is limited to the stream fragment enclosed by the start and end tags of that source node.

Pdown formulas contain sdown and pdown path formulas that relate any source node to some of its followings that are also descendants of its parent node. Thus, the lifetime of a fresh annotation marking a source node is limited to the stream fragment enclosed by the start tag of that source node and the end tag of its parent node.

Rdown formulas contain sdown, pdown, or rdown path formulas that relate any source node to some of its followings. Thus, the lifetime of a fresh annotation marking a source node is limited to the stream fragment enclosed by the start tag of the context node and the end of the stream.

Using this information on the lifetime of annotations, some annotations can be discarded as soon as their scope is exhausted. The important implications of the limitation of annotation lifetime are

1. the reusability of annotations, and

2. the limitation of the number of fresh conditions alive at a time.

The effect of reusing a dismissed fresh annotation can be simply seen as a redefinition of the in-mapping and out-mapping functions for that annotation. In fact, the functions are not changed, but rather they are defined to consider only the last (in- or out-) annotation mapping of a given annotation.

The limitation of the number of fresh annotations alive at a time becomes the bound for the domains of the in/out-mapping functions serialized in the stream, as stated later by Proposition 5.2.5.

Summing up, the stream fragment sufficient to evaluate a formula of type x from a source node n starts with the opening tag of n and ends with (1) the closing tag of n (for $x = sdown$), (2) the closing tag of the parent of n (for $x = pdown$), or (3) the last closing tag of the stream (for $x = rdown$). We define the function end^x that returns the last tag of such a stream fragment depending on the predicate type x and source node n. Also, we define the function new^x that creates new annotations for each encountered non-empty annotation in a given stream. In contrast to the function *new* that creates always unique annotations, the function new^x reuses annotations according to x. Thus, when the lifetime of an annotation is ended, the same annotation can be reused.

We distinguish between three types of scope-begin functions, depending on the type of formulas specified by the functions processing the stream created by such scope-begin functions: the sdown $\overrightarrow{scope}^{\,sdown}$, the pdown $\overrightarrow{scope}^{\,pdown}$, and the rdown $\overrightarrow{scope}^{\,rdown}$ scope-begin functions. Note that the rdown scope-begin function is more general than the other two, and its definition corresponds to Definition 5.2.5 of the basic scope-begin function $\overrightarrow{scope}$.

Definition 5.2.7 (sdown, pdown, and rdown scope-begin). *Let us consider a multi-source variable i being the path source of formulas of type $x \in \{cdown, pdown, rdown\}$ only, and a stream s. The node-preserving and node-monotone function $\overrightarrow{scope}_i^{\,x} : M^* \to M^*$ (1) replaces each non-empty annotation a_1 with a fresh annotation a_2, and adds to the output stream (2) the in-mapping annotation $a_1 \xrightarrow{i} a_2$ after a_2, and (3) the out-mapping annotation $[\,] \xleftarrow{i} a_2$ at the end of the lifetime of a_2. The stream changes done by $\overrightarrow{scope}$ are described as follows (the other messages are copied unchanged from the input to the output stream):*

$$s' = \overrightarrow{scope}_i^{\,x}(s), \forall n \lessdot s'|_E : @_{s'}(n) = \mathsf{new}^x(@_s(n)), \mu_{s'}(n) = \mu_s(n) \cup \{(@_s(n) \xrightarrow{i} @_{s'}(n))\},$$
$$\mu_{s'}(\mathsf{end}^x(n)) = \mu_s(\mathsf{end}^x(n)) \cup \{([\,] \xleftarrow{i} @_{s'}(n))\}.$$

Example 5.2.5. Figure 5.6 gives an input stream together with the tree it conveys, and the reannotated stream created by the scope-begin functions $\overrightarrow{scope}^{\,sdown}$, $\overrightarrow{scope}^{\,pdown}$, and $\overrightarrow{scope}^{\,rdown}$ for the input stream. Note that the number of fresh annotations alive at a time differs for all three cases. This number is bounded either (1) in the tree depth, or (2) in the sum of the maximum tree depth and breadth, or (3) in the tree size. ☐

The following proposition states the size of in-mapping relations for a predicate contained in one of the previously defined classes.

Proposition 5.2.5. *Consider the evaluation of LGQ formulas f, which are of type $x \in \{sdown, pdown, rdown\}$ and have as source a multi-source variable i, on a stream conveying*

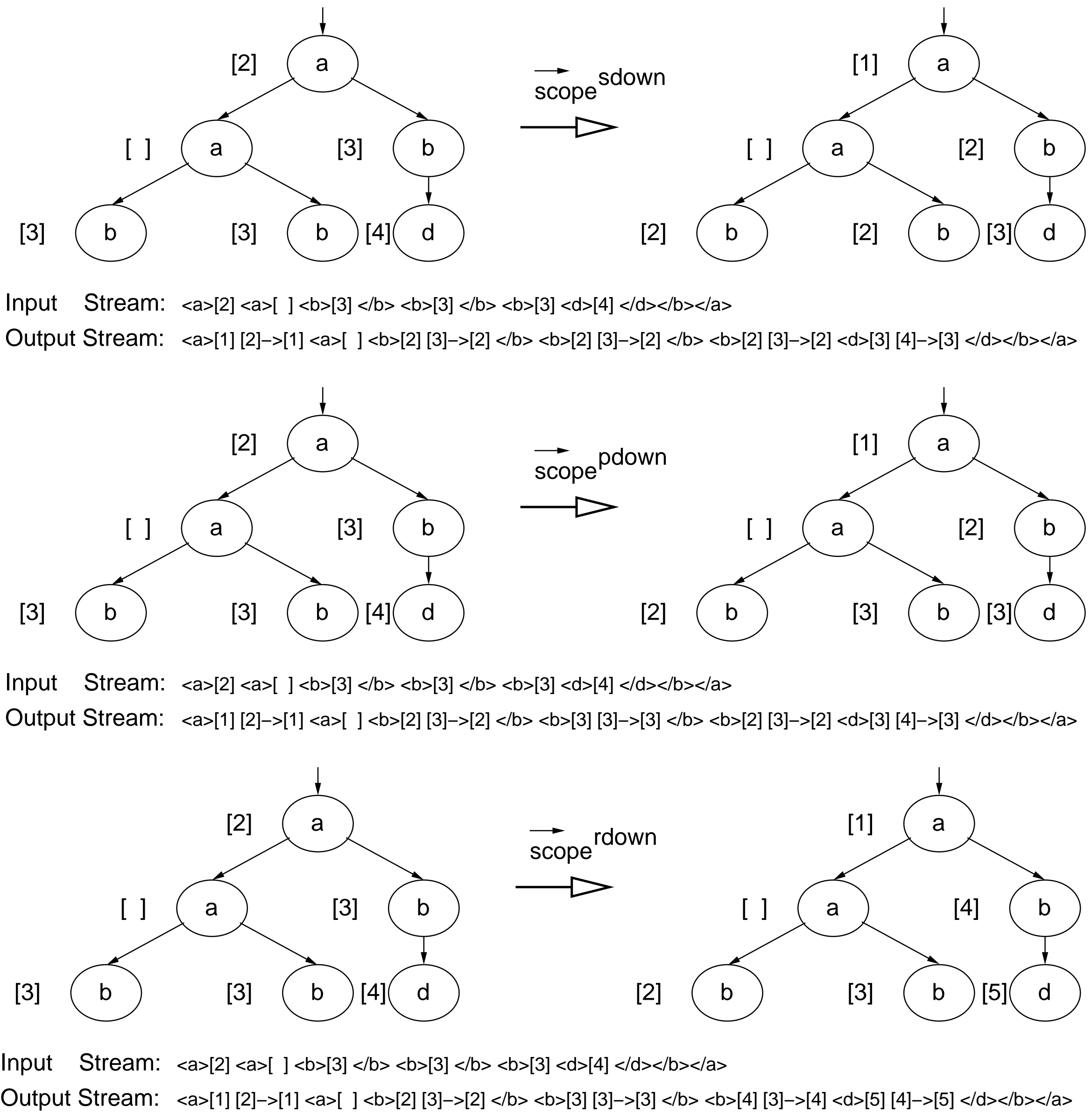


Figure 5.6: Example with sdown, pdown, and rdown scope-begin functions

a tree with depth d, breadth b, and size s. The maximum size $|\stackrel{i}{\rightarrow}|$ of the in-mapping relation $\stackrel{i}{\rightarrow}$ required for the evaluation of f is

$$|\stackrel{i}{\rightarrow}| = \begin{cases} d & \text{if } x = sdown, \\ d+b & \text{if } x = pdown, \\ s & \text{if } x = rdown. \end{cases}$$

Proof. Recall that after the opening tag of each source node, there is a non-empty annotation for which a scope-begin function $\overrightarrow{scope}^{\,x}$ creates a fresh annotation and an in-mapping annotation.

Case $x = sdown$. After the closing tag of each source node, the lifetime of the annotation created at the corresponding opening tag is ended. There can be at most d opening tags before encountering one of their closing tags, thus at most d new annotations alive.

Case $x = pdown$. After the closing tag of the parent of each source node, the lifetime of the annotation created at the corresponding opening tag is ended. There can be at most $d + b$ opening tags before encountering the closing tag corresponding to the parent of the node with the last opening tag. The bound $d + b$ is ensured by the maximum number of nested opening tags until the closing tag corresponding to the last opening tag is encountered (d) and the number of opening tags of sibling nodes that can follow (b).

Case $x = rdown$. After the last closing tag of the stream, the lifetime of the annotation created at the corresponding opening tag is ended. There can be at most s opening tags after a context node and before the last closing tag of the stream and possibly a non-empty annotation after each opening tag. ☐

5.2.7 Answer Computation

The answers computed by a stream processing function for a given LGQ query are among the nodes marked by the $\boxed{head}$ function with non-empty annotations that are either full annotations, or stand in the transitive closure out-mapping relation with the full annotation. We discuss next the functions in and out, then the computation of answers for the case of path queries and for the more complex case of tree queries.

Annotation schemes for the input stream

The nodes from an input XML stream are annotated initially by the stream processing function in, such that the first node (i.e., the root) gets a full annotation, and the other nodes get empty annotations. This annotation scheme corresponds to the evaluation of absolute queries, i.e., queries that are evaluated from the root node. A query is then absolute or relative depending on the position and amount of full annotations in the input stream. For example, an interesting scheme is obtained by marking each node from the input stream with a full annotation. This annotation scheme enforces the computation of the set of all nodes reachable via a query from any node from the stream, and corresponds to the evaluation of a special case of relative queries.

Creation of potential answers and detection of answers

The potential answers are the nodes specifically marked by the function $\boxed{head}$ with a non-empty annotation. Recall from Section 5.2.3 that the is function $\boxed{head}$ induced in a stream processing function by a unary predicate on the head variable of the corresponding query.

The last function in a sequential compositions of functions specifying the evaluation of a query is the function out. Given a stream containing all nodes from the initial stream, possibly with out-mapping annotations as results of annotation scopes, together with the annotations done by the $\boxed{head}$ function, the out function detects which nodes marked by the $\boxed{head}$ function are answers, and which not.

There are two distinct cases to consider for the detection of answers: (1) for path queries, and (2) for forest and tree queries.

1. Path queries are translated into sequential compositions of functions corresponding to the component predicates. Because the head predicate is on a non-source variable, the $\boxed{head}$ function is the last-but-one in sequence, immediately before the out function. There are no annotation scopes, and the only annotation used during the processing is the full annotation. The nodes marked by the $\boxed{head}$ function with the full annotation reach immediately the out function and are the query answers.

2. Tree queries are translated into sequential and parallel compositions of functions corresponding to the component predicates. The stream s processed in this case by the $\boxed{head}$ function contains annotations from a certain annotation scope i, and the nodes marked by this function with non-empty annotations a_i are query answers only if the following condition is satisfied.

The annotation a_i of an answer node must stand in the transitive closure out-mapping relation with the full annotation [0]. This holds only if a_i stands in the out-mapping relation either with the full annotation, or with an annotation a_j from another scope j and a part a'_j of it stands in the transitive closure out-mapping relation with the full annotation.

$$[0] \stackrel{i}{\leftarrow}{}^{+} a_i \Leftrightarrow [0] \stackrel{i}{\leftarrow} a_i \text{ or } [0] \stackrel{j}{\leftarrow}{}^{+} a_j \sqsubseteq a'_j \stackrel{i}{\leftarrow} a_i.$$

This condition ensures that all annotations a'_j and the annotation a_i are in the aggregated streams of their corresponding scopes j and i. This means also that the annotation a'_j is collected from all (at least one) of the output streams of the functions specifying tree formulas.

If the lifetime of annotations is reduced, cf. Section 5.2.6, then the same annotation can be used several times and the above characterization of answers does not hold in general. Instead, the condition must be strengthen such that the annotations a'_j and a_i are alive. This means that, in the input stream of the out function, the nearest out-mapping of each such annotation must not be to the empty annotation: neither $[\,] \stackrel{j}{\leftarrow} a'_j$ nor $[\,] \stackrel{i}{\leftarrow} a_i$.

Because (1) the input stream to the out function can not be stored, and (2) when receiving a particular potential answer, the out function may need to explore out-mappings in the stream's history or future, a reasonable implementation of the out function must

output the encountered answers as soon as possible, and buffer only the potential answers until the decision on their appurtenance to the result is met. In particular, on each received out-mapping, the out function must check whether it is relevant for the buffered potential answers. Note also that the out-mappings appear in the stream as soon as their source annotations are encountered in the aggregated streams, and these source annotations are propagated optimal in the output streams to aggregate due to the definition of the stream processing functions for LGQ predicates.

5.3 Implementation

For the implementation of the evaluation strategy described in Section 5.2, we chose deterministic pushdown transducers, i.e., automata with pushdown store and output tape, due to several reasons. First, the computation of structural relations between nodes in trees conveyed in XML streams, like the forward LGQ predicates specify, is done naturally using pushdown automata. The pushdown stores of such automata are used to remember the depths of various nodes in the stream, and they suffice to compute relations defined using the tree depth. Second, the output tape of such automata is useful for establishing communication with other automata. Complex stream processing functions specifying LGQ formulas are realized by networks of communicating automata, where their connections reflect the (parallel and sequential) compositions of functions for the LGQ predicates making up the formulas.

After introducing the necessary preliminaries on pushdown transducers, this section gives the transducers for forward LGQ predicates and for other stream processing functions used for the evaluation of LGQ forest formulas.

5.3.1 SPEX Transducers and Transducer Networks

Pushdown transducers are automata with pushdown store and output tape. More formally, a pushdown transducer [82] is an eight-tuple $\langle Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F\rangle$ that satisfies the following conditions:

- Q is a finite set of states.
- Σ, Γ and Δ are alphabets. Σ is the input alphabet, and its elements are called input symbols. Γ is the pushdown alphabet, and its elements are called pushdown symbols. Δ is the output alphabet, the elements of which are called output symbols.
- δ is a relation from $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})$ to $2^Q \times \Gamma^* \times (\Delta \cup \{\varepsilon\})$. δ is called the transition table, the elements of which are called transition rules.
- q_0 is an element in Q, called the initial state.
- Z_0 is an element in Γ, called the bottom pushdown symbol.

- F is a subset of Q. The states in the subset F are called the accepting, or final states.

Deterministic pushdown transducers allow at most one transition from any of its states. In this case, the transition relation becomes a function from $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})$ to $Q \times \Gamma^* \times (\Delta \cup \{\varepsilon\})$.

One of the main usages of pushdown transducers is to support recursion. In fact, recursive finite-domain programs are characterized by pushdown transducers [82]. Processing an XML stream with pushdown transducers corresponds to a depth-first, left-to-right, preorder traversal of the (implicit) tree conveyed by the XML stream, and uses also an implicit form of recursion, in order to discover which closing tag corresponds to which opening tag. Exploiting the affinity between depth-first search and stack management, the transducers use their stacks for tracking the node depth in such trees. This way, the forward LGQ predicates can be evaluated in a single pass, which corresponds to a run of pushdown transducers on the XML stream.

SPEX Transducers

We use in the following a simplified class of deterministic pushdown transducers, which we call SPEX transducers.

Definition 5.3.1 (SPEX Transducer). *A SPEX transducer is a single-state deterministic pushdown transducer, where the input and output alphabets are the set of all opening and closing tags and annotations M, the stack alphabet is the set of all annotations A, the bottom pushdown symbol Z_0 is the empty annotation [ ], and the transition function δ is canonically extended to the configuration-based transition function $\vdash: M \times A^* \to A^* \times M^*$.*

Note that for defining a SPEX transducer, it is only necessary to give its transition function δ.

Example 5.3.1. Consider the SPEX transducer defined by the following transitions

$$
\begin{array}{ll}
1. & ([c]\;, \quad \gamma) \vdash ([c] \mid \gamma, \quad \varepsilon) \\
2. & (\langle \eta \rangle\;, [s] \mid \gamma) \vdash ([s] \mid \gamma, \langle \eta \rangle [s]) \\
3. & (\langle / \eta \rangle, [s] \mid \gamma) \vdash (\quad \gamma, \quad \langle / \eta \rangle)
\end{array}
$$

On receiving an input symbol $[c]$, which is an annotation, the first transition pushes that symbol onto the stack and does not output anything. Note that the stack content $[c] \mid \gamma$ is $|$-separated in its top $[c]$ and the rest γ, and no output is simulated by writing on the output tape the empty symbol ε.

On receiving an input symbol $\langle \eta \rangle$, which is an opening tag, and with the annotation $[s]$ as the top of the stack, the second transition keeps the same stack configuration and outputs first the input symbol $\langle a \rangle$ *followed* by the top of the stack $[s]$.

On receiving an input symbol $\langle / \eta \rangle$, which is a closing tag, and with the annotation $[s]$ as the top of the stack, the third transition outputs the input symbol and pops the top annotation off the stack.

In effect, this SPEX transducer moves the annotations of nodes to their children. □

SPEX Transducer Networks

The sequential and parallel compositions of stream processing functions are implemented by sequential and parallel compositions of pushdown transducers, where the meaning of transducer compositions is the same as for functions: the output of one transducer is the input for the immediate next ones (for sequential composition), respectively several transducers have the same input (for parallel composition). For such compositions are oriented (from one transducer to another), the implementation for a function specifying an arbitrary LGQ formula is done using networks of transducers, or directed acyclic graphs where each node is a pushdown transducer and each edge between two transducers inforces that the input tape of the sink transducer is the output tape of the source transducer. In this respect, a transducer network is isomorphic to the function graph of the stream processing function it implements.

5.3.2 Transducers for Forward LGQ Predicates

A transducer for a forward LGQ binary predicate α : Node $\times$ Node $\rightarrow$ Boolean, or simpler an α-transducer, implements the function f_α : Set(Node) $\rightarrow$ Set(Node) that computes, for a given tree T and a set of source nodes n in T, the set of all sink nodes m in T that stand in the predicate α with n, i.e., $\alpha(n, m)$ holds. More precisely, instead of processing directly the set of source nodes, a transducer processes the stream conveying all nodes in T, where those source nodes n are marked with non-empty annotations. The yield of the transducer is the stream conveying all nodes in T, where only the sink nodes have non-empty annotations, and a sink node m is annotated precisely with the union of annotations of all source nodes n that stand in the predicate α with m.

For accomplishing this task, an α-transducer uses its stack to store the annotations of the source nodes until their corresponding sink nodes are encountered in the coming stream. The key issue on designing such transducers stems in satisfying the constraint that when a sink node is encountered on the stream, the annotations of its corresponding source nodes are on the top of the stack, thus justifying the use of such an access-restricted memory store. In this way, the sink nodes can be easily marked with the annotations of their corresponding source nodes. This section shows that, indeed, there are deterministic pushdown transducers that implement the functions f_α of the forward LGQ predicates α, and this fact makes our choice for pushdown transducers natural. Section 5.5 shows that a relaxation of the stackwise access to the store of each transducer brings a better space complexity, at the expense of a more complicated store management.

Configuration-based transitions defining α-transducers are given in the following, and a processing example with them is given later in this section. Initially, an empty annotation [ ] is pushed onto the stack of each transducer. Note that the α-transducers differ only in the first transition, which is a compaction of several simpler transitions that do only one stack operation.

We define next the transducers for the forward binary predicates: child, fstChild, and nextSibl, then for the transitive closure predicates child^+ and nextSibl^+, and then for the

reflexive transitive closures child^* and $\mathsf{nextSibl}^*$. For the implementation of the nodetest predicates, we use finite transducers (i.e., without the pushdown store). Finally, we give some thoughts on the capabilities of such transducers to implement more sophisticated predicates, and even simple compositions of predicates.

The child-transducer moves the annotations of nodes to their children. The transitions of this transducer read as follows: (1) if an annotation $[c]$ is received, then $[c]$ is pushed onto the stack and nothing is output; (2) if an opening tag $\langle\eta\rangle$ is received, then it is output followed by the annotation from the top of the stack; (3) if a closing tag is received, then it is output and the top annotation is popped off the stack.

$$\begin{array}{ll} 1. & ([c]\ ,\ \gamma) \vdash ([c] \mid \gamma,\ \varepsilon) \\ 2. & (\langle\eta\rangle\ ,\ [s] \mid \gamma) \vdash ([s] \mid \gamma,\ \langle\eta\rangle[s]) \\ 3. & (\langle/\eta\rangle,\ [s] \mid \gamma) \vdash (\gamma,\ \langle/\eta\rangle) \end{array}$$

Recall that the annotation of a node n follows its opening tag. When receiving a node n annotated with $[c]$, $[c]$ is pushed onto the stack. The following two cases can then appear:

(1) the closing tag of n is received, and $[c]$ is popped off the stack. This corresponds to the case when there are no other child nodes of n left in the incoming stream.

(2) the opening tag of a child node m of n is received, and it is output followed by $[c]$. Thus, the node m is annotated correctly with $[c]$, which was the annotation of n.

In the second case, a new annotation, say $[c']$, is received afterwards, pushed onto the stack, and used to annotate children p of m. Only when the closing tag of p is received, $[c']$ is popped and $[c]$ becomes again the top of the stack. At this time, siblings of m can be received and annotated with $[c]$ (the above cases 2), or the closing tag of n is received (the above case 1).

The fstChild-transducer moves the annotations of nodes to their first children. This transducer is a simplification of the child-transducer, by restricting a stored annotation $[s]$ of a node n to mark at most one node. This node is necessarily the first child of n, as ensured by the left-to-right traversal of the children existent in the stream. This restriction can be realized by replacing $[s]$ with the empty annotation as soon as a child of n and its annotation, say $[c]$, is received. Below, we give the first transition modified accordingly. The other transitions are as for the child-transducer.

$$1.\ \ ([c],\ [s] \mid \gamma) \vdash ([c] \mid [\,] \mid \gamma,\ \varepsilon)$$

The nextSibl-transducer moves the annotations of nodes to their immediate next sibling, if any. The transitions of this transducer are the same as for the child-transducer, except for the first one, which is given below. In the first transition, this transducer replaces the top of the stack $[s]$ with the received annotation $[c]$ of the source node n and pushes an empty annotation $[\,]$ onto the stack. The annotation $[\,]$ is then used to annotate children of n. When the closing tag of n is received, the annotation $[\,]$ is popped and its next sibling node m can be annotated with the top annotation $[c]$. The other next siblings can not be annotated with $[c]$, because $[c]$ is replaced by the annotation of m, say $[c']$, and now the immediate next sibling of m can be annotated with $[c']$.

$$1.\ ([c],\ [s] \mid \gamma) \vdash ([\,] \mid [c] \mid \gamma,\ \varepsilon)$$

Remark 5.3.1. Note that for the base predicates $\alpha \in \{\mathsf{fstChild}, \mathsf{child}, \mathsf{nextSibl}\}$ and any sink node m, there exists at most one source node n such that $\alpha(n, m)$ holds. Therefore, an α-transducer does not need to compute unions of annotations of several source nodes n for annotating sink nodes. We see next that the transducers for closure predicates α^+ and α^* have to compute such unions, because there can be several nodes n for which $\alpha^+(n, m)$, respectively $\alpha^*(n, m)$, holds. □

The child^+-transducer moves the annotations of nodes to their descendants. The transitions of this transducer are the same as for the child-transducer, except for the first one, which is given below. In the first transition, this transducer pushes onto the stack the received annotation $[c]$ *together* with the top annotation $[s]$: $[c]\sqcup[s]$. The difference to the child-transducer is that also the annotations $[s]$ of the ancestors n_a of n are used to annotate children m of n, for the nodes m are also descendants of the nodes n_a.

$$1.\ ([c],\ [s] \mid \gamma) \vdash ([c]\sqcup[s] \mid [s] \mid \gamma,\ \varepsilon)$$

When receiving a node n annotated with $[c]$, $[c]$ is pushed onto the stack together with the current top $[s]$: $[c]\sqcup[s]$. The following two cases can then appear:

(1) the closing tag of n is received, and $[c]\sqcup[s]$ is popped off the stack. This corresponds to the case when there are no other descendants of n left in the incoming stream.

(2) the opening tag of a child m of n is received, and it is output followed by $[c]\sqcup[s]$. Thus, the children of n, which are also descendants of n, are annotated correctly.

In the second case, a new annotation, say $[c']$, is received afterwards, the annotation $[c']\sqcup[c]\sqcup[s]$ is pushed onto the stack and used to annotate children p of m. Thus, the annotation $[c]$ is also used to annotate children p of m (n''), hence descendants of n. Only when the closing tag of p is received, $[c']\sqcup[c]\sqcup[s]$ is popped and $[c]\sqcup[s]$ becomes again the top of the stack. At this time, siblings of m can be received and annotated with $[c]\sqcup[s]$ (the above case 2), or the closing tag of n is received (the above case 1).

The $\mathsf{nextSibl}^+$-transducer moves the annotations of source nodes to their next siblings. The transitions of this transducer are the same as for the child-transducer, except for the first one, which is given below. In the first transition, this transducer adds to the top of the stack $[s]$ the received annotation $[c]$ of the source node n and pushes an empty annotation $[\,]$. The annotation $[\,]$ is then used to annotate children of n. When the closing tag of n is received, the annotation $[\,]$ is popped and its next sibling nodes m can be annotated with the top annotation $[c]$. Because the old top of the stack $[s]$ is kept together with the newly received annotation $[c]$, then annotations of preceding siblings of n are also used to annotate the following siblings of n.

$$1.\ ([c],\ [s] \mid \gamma) \vdash ([\,] \mid [c]\sqcup[s] \mid \gamma,\ \varepsilon)$$

The transducers for the reflexive transitive closures are simple variations of the ones for transitive closures defined above. We explain them shortly below.

The child*-transducer moves the annotations of each node n to its descendants and to the node n itself. This transducer is defined below similar to the child$^+$-transducer, with the difference that a node n keeps its own annotation, say $[c]$, together with the annotations of its ancestors, say $[s]$.

$$
\begin{array}{lllll}
1. & ([c] & , [s] \mid \gamma) \vdash ([c]\sqcup[s] \mid [s] \mid \gamma, & [c]\sqcup[s]) \\
2. & (\langle\eta\rangle & , \quad \gamma) \vdash (\quad \gamma, & \langle\eta\rangle) \\
3. & (\langle/\eta\rangle & , [s] \mid \gamma) \vdash (\quad \gamma, & \langle/\eta\rangle)
\end{array}
$$

The nextSibl*-transducer moves the annotations of each node n to its next siblings and to the node n itself.

$$
\begin{array}{lllll}
1. & ([c] & , [s] \mid \gamma) \vdash ([\,] \mid [c]\sqcup[s] \mid \gamma, & [c]\sqcup[s]) \\
2. & (\langle\eta\rangle & , \quad \gamma) \vdash (\quad \gamma, & \langle\eta\rangle) \\
3. & (\langle/\eta\rangle & , [s] \mid \gamma) \vdash (\quad \gamma, & \langle/\eta\rangle)
\end{array}
$$

Transducers for LGQ Nodetest Predicates

A finite transducer for an LGQ nodetest predicate nodetest : Node $\times$ NodeTest $\rightarrow$ Node, or simply a nodetest-transducer, implements the function $f_{nodetest}$: Set(Node) $\times$ NodeTest $\rightarrow$ Set(Node) that computes, for a given tree T and a set of source nodes in T, the subset of it consisting only in the nodes with that nodetest. As for α-transducers, the nodetest-transducers processes the stream conveying all nodes in T and outputs the same stream where only annotations can be changed. A transducer for a nodetest η replaces the annotations of nodes without that nodetest with the empty annotation. The transitions of an η nodetest-transducer are given next. For simplification, each transition can consider two input symbols at once. Also, the nodetest $\neg\eta$ stands for any nodetest but η. Because we consider finitely many nodetests, this compact representation for any nodetest but η holds.

$$
\begin{array}{ll}
1. & (\langle\eta\rangle[c]) \vdash (\langle\eta\rangle[c]) \\
2. & (\langle\neg\eta\rangle[c]) \vdash (\langle\eta\rangle[\,]) \\
3. & (\langle/\eta\rangle) \vdash (\langle/\eta\rangle) \\
4. & (\langle/\neg\eta\rangle) \vdash (\langle/\neg\eta\rangle)
\end{array}
$$

Variations of Transducers for LGQ Predicates

From the transitions of the α-transducers, it can be observed that the relation between the binary predicates α and their corresponding α-transducers is determined by how the annotation of each node n is stored onto the transducer's stack. These relations can be resumed as follows ($[c]$ is the annotation of n currently read and $[s]$ is the current top of the stack):

1. $[c]$ is output as soon as it is read. Then, $[c]$ is used to mark also n.

2. $[c]$ is pushed in the new top. Then, $[c]$ is used to mark also the children of n.

3. $[c]$ is pushed in the old top. Then, $[c]$ is used to mark also the next sibling of n.

4. $[s]$ is pushed in the new top. Then, $[s]$ is used to mark also the descendants of n.

5. $[s]$ is pushed in the old top. Then, $[s]$ is used to mark also the next siblings of n.

By mixing the above behaviours 1 to 5, one can get the transducers implementing the desired built-in predicates. For example, combining behaviours 1 and any other ensures the reflexivity of the implemented predicate. Combining behaviours 4 and 2, or 5 and 3, ensures the transitivity of the implemented predicate. And combining 1 and 2 and 4, or 1 and 3 and 5, ensures both the transitivity and reflexivity of the implemented predicate.

There are, of course, other possible combinations. For example, the combination of behaviours 2 to 5 gives the implementation of the complex predicate $\mathsf{child}^+\text{-}\mathsf{or}\text{-}\mathsf{nextSibl}^+ = \mathsf{child}^+ \cup \mathsf{nextSibl}^+$. These combinations are reflected in the following changed transition:

$$1.\ ([c],\ [s] \mid \gamma) \vdash ([c]\sqcup[s] \mid [c]\sqcup[s] \mid \gamma,\ \varepsilon)$$

More non-trivial predicates can be supported by changing also the other transitions of the child-transducer. We exemplify this with the **foll-transducer** defined below. In the first transition, it replaces the old top annotation $[s]$ with the new annotation $[c]$ and then pushes also the old top $[s]$. Because the nodes following a node n are all nodes reachable in the further stream after closing the node n, the annotation $[c]$ becomes part of the top of the stack and used to annotate incoming nodes as soon as the node n is closed (transition 3). In contrast to the α-transducers previously defined, once an annotation becomes part of the stack, it remains there, because the following sibling nodes of the ancestor nodes of n follow also n.

$$\begin{array}{lllll}
1. & ([c]\ , & [s] \mid \gamma) \vdash ([s] \mid [c] \mid \gamma, & \varepsilon) \\
2. & (\langle\eta\rangle\ , & [s] \mid \gamma) \vdash (\quad [s] \mid \gamma, & \langle\eta\rangle[s]) \\
3. & (\langle/\eta\rangle, [c] \mid & [s] \mid \gamma) \vdash ([c]\sqcup[s] \mid \gamma, & \langle/\eta\rangle)
\end{array}$$

Although pushdown transducers are not closed under composition, the composition of pushdown and finite transducers is possible and even beneficial. In this sense, one can create transducers implementing composition of binary and nodetest predicates. We give below the transitions of a transducer for the composition of the child binary predicate and the a nodetest-predicate defining, for a set of nodes, the set of their children with nodetest a. Note that such compositions are general and natural. The generality of such compositions ensures that they can be applied on any binary and unary predicate. Their naturality is ensured by the usage in the practical XML query language XPath of atomic constructs called location steps made out of a binary and a unary predicate, like in child::a. By convenience, we name the transducer, implementing such a combination of a binary predicate α and a nodetest predicate η, the $\alpha{::}\eta$-transducer.

1. $([c] \quad , \quad \gamma) \vdash ([c] \mid \gamma, \quad \varepsilon)$
2. $(\langle \mathsf{a} \rangle \quad , [s] \mid \gamma) \vdash ([s] \mid \gamma, \quad \langle \mathsf{a} \rangle [s])$
3. $(\langle \neg \mathsf{a} \rangle \ , \quad \gamma) \vdash (\quad \gamma, \langle \neg \mathsf{a} \rangle [\ ])$
4. $(\langle / \mathsf{a} \rangle \ , [s] \mid \gamma) \vdash (\quad \gamma, \quad \langle / \mathsf{a} \rangle)$
5. $(\langle / \neg \mathsf{a} \rangle, [s] \mid \gamma) \vdash (\quad \gamma, \ \langle / \neg \mathsf{a} \rangle)$

5.3.3 Processing Example with Transducers for LGQ Predicates

We show next how the child::b-transducer processes incrementally the stream

$$\langle a \rangle [1] \ \langle a \rangle [2] \ \langle b \rangle [3] \ \langle /b \rangle \langle /a \rangle \langle b \rangle [\ ] \ \langle /b \rangle \langle /a \rangle$$

containing two a-nodes and two b-nodes.

Recall that the stack is initialized with an empty annotation [ ]. The stack configuration changes only on receiving annotations and closing tags. On receiving opening tags matching its nodetest, the transducer outputs that opening tag followed by the top of its stack.

$\langle \mathsf{a} \rangle$ It outputs the tag, followed by its (initial) top annotation [ ]. Thus, the first a-node does not have in the input stream a parent with a non-empty annotation.

The stack configuration remains [ ].

[1] It pushes [1] onto the stack, This way, it is instructed to mark all b-children of the first a-node with [1].

The stack configuration becomes [1]|[ ] (the top is at the left).

$\langle \mathsf{a} \rangle$ It outputs the tag, followed by [ ]. Although the top annotation is [1], this output is correct, because the received node does not have a b-nodetest.

The stack configuration remains [1]|[ ].

[2] It pushes [2] onto the stack. This way, it is instructed to mark all b-children of the second a-node with [2].

The stack configuration becomes [2]|[1]|[ ].

$\langle \mathsf{b} \rangle$ It outputs the tag, followed by the top annotation [2]. This output is correct, because the received node does have a b-nodetest and is a child of the second a-node.

The stack configuration remains [2]|[1]|[ ].

[3] It pushes [3] onto the stack. This way, it is instructed to mark all b-children of the first b-node with [3].

The stack configuration becomes [3]|[2]|[1]|[ ].

$\langle / \mathsf{b} \rangle$ It pops the top [3] off the stack, meaning that there are no children of the first b-node left in the stream. This is correct, because the first b-node does not have children at all.

The stack configuration becomes [2]|[1]|[ ].

$\langle /\mathsf{a}\rangle$ It pops the top [2] off the stack, meaning that there are no children of the second a-node left in the stream.

The stack configuration becomes [1]|[ ].

$\langle \mathsf{b}\rangle$ It outputs the tag, followed by the top annotation [1]. This output is correct, because the received node does have a b-nodetest and is a child of the first a-node.

The stack configuration remains [1]|[ ].

[ ] It pushes [ ] onto the stack. This way, it is instructed to mark all b-children of the second b-node with [ ]. Because the other children are also marked with [ ], we can conclude that the transducer will mark all children of the second b-node with [ ].

The stack configuration becomes [ ]|[1]|[ ].

$\langle /\mathsf{b}\rangle$ It pops the top [ ] off the stack, meaning that there are no children of the second b-node left in the stream.

The stack configuration becomes [1]|[ ].

$\langle /\mathsf{a}\rangle$ It pops the top [1] off the stack, meaning that there are no children of the first a-node left in the stream.

The stack configuration becomes [ ] and the processing is finished.

The output streams produced by the transducers child$^+$::b, nextSibl$^+$::b, and foll::b when processing the same input stream are shown below:

input	$\langle a\rangle$	[1]	$\langle a\rangle$	[2]	$\langle b\rangle$	[3]	$\langle /b\rangle$	$\langle /a\rangle$	$\langle b\rangle$	[ ]	$\langle /b\rangle$	$\langle /a\rangle$
child$^+$::b	$\langle a\rangle$	[ ]	$\langle a\rangle$	[ ]	$\langle b\rangle$	[1,2]	$\langle /b\rangle$	$\langle /a\rangle$	$\langle b\rangle$	[1]	$\langle /b\rangle$	$\langle /a\rangle$
nextSibl$^+$::b	$\langle a\rangle$	[ ]	$\langle a\rangle$	[ ]	$\langle b\rangle$	[ ]	$\langle /b\rangle$	$\langle /a\rangle$	$\langle b\rangle$	[2]	$\langle /b\rangle$	$\langle /a\rangle$
foll::b	$\langle a\rangle$	[ ]	$\langle a\rangle$	[ ]	$\langle b\rangle$	[ ]	$\langle /b\rangle$	$\langle /a\rangle$	$\langle b\rangle$	[2,3]	$\langle /b\rangle$	$\langle /a\rangle$

5.3.4 Transducers for Other Stream Processing Functions

The evaluation strategy of this chapter uses also rather complex stream processing functions, e.g., for dealing with aggregations of several streams, annotation scopes, and management of potential answers, and SPEX transducers are not expressive enough to implement all of them. Therefore, we discuss here the implementations of some of these functions, like of the scope functions $\overrightarrow{scope}_i$ and $\overleftarrow{scope}_i$ and of the connective functions $\wedge_f$ and $\vee_f$, by means of SPEX transducers with straightforward extensions.

Recall from Definition 5.2.7 that the node-preserving and node-monotone functions $\overrightarrow{scope}_i{}^x$ (for a multi-source variable i) replaces each annotation $[c]$ with a fresh annotation $[s+1]$, which is a singleton list, and adds to the output stream the in-mapping annotation $[c] \xrightarrow{i} [s+1]$ after $[c]$, and the out-mapping annotation $[\,] \xleftarrow{i} [s+1]$ at the end of the lifetime of $[s+1]$. The fresh annotation $[s+1]$ is generated using the top annotation $[s]$ from the stack. We give next the relevant transition rules of the transducer for the function

$\overrightarrow{scope_i}^{\,sdown}$. The transitions for the other message types consist in simply copying the messages from the input to the output stream.

$$\begin{array}{lllll}
1. & ([c]\ \ , [s] \mid \gamma) \vdash & ([s+1] \mid [s] \mid & \gamma, & [s+1]([c] \xrightarrow{i} [s+1])) \\
2. & (\langle\eta\rangle\ , \quad \gamma) \vdash & (& \gamma, & \langle\eta\rangle) \\
3. & (\langle/\eta\rangle, [s] \mid \gamma) \vdash & (& \gamma, & \langle/\eta\rangle([\,] \xleftarrow{i} [s]))
\end{array}$$

Note that the lifetime of the annotation $[s{+}1]$ ends as soon as the tree depth, where that fresh annotation is created, is reached again. The transducers for the other scope-begin types, i.e., $\overrightarrow{scope_i}^{\,pdown}$ and $\overrightarrow{scope_i}^{\,rdown}$, are defined similarly, with the only difference that the lifetime of $[s+1]$ ends as soon as (1) the tree depth smaller by one than the tree depth, where that fresh annotation is created, is reached again (for $\overrightarrow{scope_i}^{\,pdown}$), and (2) the end of the stream is reached (for $\overrightarrow{scope_i}^{\,rdown}$).

The node-preserving and node-monotone function $\overleftarrow{scope_i}$ replaces each non-empty annotation encountered in the input stream with the union of all annotations that are mapped to parts of the former annotation. It also adds the out-mapping message of the former to the latter annotation to the output stream after the latter annotation. Its implementation is done by a transducer that uses a random accessible store (i.e., a Turing machine) for keeping the in-mappings encountered in the input stream.

The connective functions $\wedge_f$ and $\vee_f$ are responsible for aggregating several streams into a single stream being the serialization of the same tree as the input streams, cf. Definition 5.2.4. Their definitions are based on the annotation-merge function Θ_c defined in Section 5.2.2, which ensures that an annotation a appears in the output stream at a given position p if and only if a appears in *all* input streams at positions previous to p (for $c = \wedge$), or in *at least one* input stream at a position previous to p (for $c = \vee$).

The implementation of $\Theta_\wedge$ is given below by a modified SPEX transducer without stack, but with an array, whose size is given by the number n of streams to aggregate. Also, this transducer has an input tape for each of its input streams. The transitions for messages of other types, like in-mappings, are not shown, because such messages are simply copied to the output.

$$\begin{array}{llll}
1. & (([c_1],\ldots,[c_n])\ , ([s_1],\ldots,[s_n])) \vdash & (([s_1]\sqcup[c_1],\ldots,[s_n]\sqcup[c_n]), & \prod_{i=1}^{n}([s_i]\sqcup[c_i])) \\
2. & ((\langle\eta\rangle,\ldots,\langle\eta\rangle)\ \ , ([s_1],\ldots,[s_n])) \vdash & (\quad ([s_1],\ldots,[s_n]), & \langle\eta\rangle) \\
3. & ((\langle/\eta\rangle,\ldots,\langle/\eta\rangle), ([s_1],\ldots,[s_n])) \vdash & (\quad ([s_1],\ldots,[s_n]), & \langle/\eta\rangle)
\end{array}$$

The transducer for $\Theta_\vee$ differs from that of $\Theta_\wedge$ in the treatment of annotations. The former one copies all annotations from the input streams to the output stream.

5.4 Minimization Problems for SPEX Transducer Networks

When dealing with networks of transducers, there are at least two minimization problems to address: the problem of finding the minimal network equivalent to a given network, and

the problem of minimal stream routing within a given network.

An equivalent minimal network is a network that produces the same output as the initial network for a given input and has less transducers that the initial network. Such a network could be obtained, e.g., by

1. composing several pushdown transducers into a single pushdown transducer,

2. reducing the network to an equivalent fragment of it, and

3. finding a completely other network equivalent to the initial one.

The first possibility is excluded, for *pushdown* transducers are in general not closed under composition [42]. The last two possibilities can be partially lifted at the level of LGQ as a query reformulation and minimization problem: for a given query, find an equivalent minimal query. This problem is partially addressed in Chapter 4 and it is not further addressed here. Recall from Section 5.2.3 that the translation of LGQ formulas into stream processing functions has a simplification phase that can dramatically reduce the size of function graphs, and thus of the corresponding transducer networks. In that case, the simplifications are not possible at the level of LGQ.

The minimal stream routing problem within a network is: given a transducer network and an arbitrary input stream, instruct the transducers to send further only stream fragments that can be of interest to the successor transducers. This problem is (partially) addressed next.

The stream processing functions used in this chapter are node-preserving, i.e., all nodes from the input stream appear in the output stream. Consequently, the transducers implementing them are also node-preserving. This property ensures an easier and uniform treatment of transducers, although at the cost of routing within the corresponding networks also stream fragments that are not relevant for the computation of query answers. Consider, e.g., an XML stream containing information about articles possibly followed only at the very end of the stream by information about books, and a query asking for authors of books with given prices and publishers. For this query, our evaluation strategy creates a network, whose number of transducers is linear in the number of component LGQ predicates and of multi-source variables. The transducers in the network process the stream incrementally, and each transducer sends further the stream to its successive transducers. In case the transducer instructed to find books-nodes, say the books-transducer, encounters such a node, then it sends that node further to its successors, with an additional non-empty annotation. In case it encounters other nodes, e.g., article-nodes, then it still sends it further, but with an additional empty annotation. Either way, all nodes from the stream reach all transducers from the network.

We consider here two routing strategies to restrict the stream fragments sent between transducers.

1. Recall that all transducers succeeding the books-transducer look always for nodes in the stream fragment that follows the books-nodes. Thus, the query evaluation is not altered,

if the books-transducer sends further only the stream fragment starting with the first books-node and ending together with the stream, and the other transducers do the same for the nodes they are instructed to find relative to nodes found by their previous transducers. The transducers would process then a much smaller fragment of the input stream. Although this observation does not change the worst-case complexity of our evaluation strategy, it proves very competitive in practical cases.

2. The minimization of the stream routed between transducers does not stop here. Let us consider again the previous example, and assume that the transducers receiving (directly or indirectly) stream from the books-transducer look for nodes to be found only inside the stream fragments corresponding to books-nodes (like their descendants, or siblings of their descendants). Then, the books-transducer can safely send further only such stream fragments corresponding to books-nodes.

Such information on the interest of transducers can be inferred from both the query to evaluate and the characteristics of the stream (e.g., its grammar). For the SPEX setting considered here, i.e., no a-priori knowledge of the incoming stream is provided, only the first inference case is reasonable. We are confident that exploring the second case is rewarding too, but we delegate it to future research for now.

Both aforementioned approaches to minimal stream routing, called here phase_1 and respectively phase_2 routing, can be easily supported by the evaluation strategy presented in this chapter. Their use deviates a bit from our previous explanations in that the already existing transducers are not drastically changed, but rather new transducers, called structural filters, are placed correspondingly at compile-time in the network.

The improvement achieved by using structural filters depends tremendously on the selectivity of the query evaluated by the transducer network. In the previous example, the selectivity is rather high, because the books-transducer, positioned near the top of the network, finds books-nodes at the very end of the stream. In such cases, the gain is fully rewarding. However, in cases where the query is not selective, the effort to run the additional routers can be reflected in worse timing of the evaluation. Section 5.6 shows that in practice such routers bring the evaluation time up to several times better than of the original network.

In a stream context, the selectivity of the (continuous) query can change over time, due to changes in the input stream. Therefore, an interesting question, which is not addressed here, is to add or remove the routers at run-time, depending on the changes in the query selectivity.

A final remark before defining the router transducers. Due to the fragmentation they operate on (well-formed) XML streams, such routers output stream fragments that are not necessarily well-formed. In particular, the routers can send closing tags without their accompanying opening tags. The needed changes to the existing transducers are minimal: the stacks of the SPEX transducers have a non-removable bottom-symbol (which is interpreted as the empty annotation) which may not be removed.

Phase$_1$ Routing

After each transducer for a forward LGQ predicate, we add to the network a phase$_1$ router transducer which sends further the stream fragment starting with the first opening tag followed by a non-empty annotation. For a more compact definition, we may read two input symbols at once[1]. The transition rules read as follows. If no non-empty annotation has been already received (stated by the empty annotation as the only entry on the stack), then no message is let through. As soon as the stack consists in a non-empty annotation, all subsequent messages are let through. Finally, in case the received node has a non-empty annotation ($[s] \neq [\,]$), then it is sent through and the annotation becomes the stack content ($__$ stands for any annotation).

$$
\begin{array}{ll}
1. & (\langle\eta\rangle[\,]\,,\ [\,]) \vdash ([\,],\quad \varepsilon) \\
2. & (\langle/\eta\rangle\ \ ,\ [\,]) \vdash ([\,],\quad \varepsilon) \\
3. & (\langle\eta\rangle[\,]\,,\ [s]) \vdash ([s],\ \langle\eta\rangle[\,]) \\
4. & (\langle/\eta\rangle\ \ ,\ [s]) \vdash ([s],\ \langle/\eta\rangle) \\
5. & (\langle\eta\rangle[s],\ __) \vdash ([s],\ \langle\eta\rangle[s])
\end{array}
$$

Phase$_2$ Routing

After each transducer for a forward LGQ predicate, we add to the network a so-called phase$_2$ router transducer which sends further only stream fragments that can be relevant to the other transducers down the network. We can distinguish here the cases of (sub)networks evaluating sdown, pdown, and rdown formulas. The first case corresponds to our previous example, because all transducers under the phase$_2$ router transducer look for nodes to be found only inside the stream fragments corresponding to nodes matched by the transducer positioned above that router. The second case restricts the routed stream to the fragments between the node having a non-empty annotation and the closing tag of its parent. A phase$_2$ router transducer for the third case is the same as for phase$_1$ and as defined above, because it restricts the routed stream to the fragment between the first node having a non-empty annotation and the end of the stream.

We give next the phase$_2$ router transducer for the sdown case. In contrast to the phase$_1$ router, the phase$_2$ router uses its stack to remember the smallest depth of a received node with a non-empty annotation. Therefore, only if the stack consists in an empty annotation, then the opening and closing tags of nodes with empty annotations are not let through.

$$
\begin{array}{ll}
1. & (\langle\eta\rangle[\,]\,,\ \ [\,]) \vdash (\ [\,],\quad \varepsilon) \\
2. & (\langle/\eta\rangle\ \ ,\ \ [\,]) \vdash (\ [\,],\quad \varepsilon) \\
3. & (\langle\eta\rangle[c],\ \ __) \vdash (\ [c] \mid __,\ \langle\eta\rangle[c]) \\
4. & (\langle\eta\rangle[\,]\,,\ __ \mid \gamma) \vdash ([\,] \mid __ \mid \gamma,\ \langle\eta\rangle[\,]) \\
5. & (\langle/\eta\rangle\ \ ,\ __ \mid \gamma) \vdash (\ \gamma,\ \ \langle/\eta\rangle)
\end{array}
$$

[1]This relaxation does not make the phase$_1$ router more expressive than SPEX transducers.

5.5 Complexity Analysis

This section gives upper bounds for the time and space complexities of the LGQ query evaluation using the evaluation strategy developed in this chapter. After analyzing the complexities for forward LGQ forests, which are explicitly targeted in this chapter, the complexities for LGQ graphs are also discussed, as derivable from both the complexities of rewriting of graphs into forward forests and the complexities of evaluating the latter. Several other ideas on improving the space complexity are also presented, though not thoroughly investigated.

The evaluation of forward LGQ forest queries proposed in this chapter has polynomial combined complexity (i.e., in the size of the data and of the query) near the lower bound [71] for *in-memory* evaluation of Core XPath, a strict fragment of forward LGQ forests. Although in general it is considered that the query is fixed, in a stream context there are good reasons to take also the query size into account, especially when dealing with sets of (millions of) queries to be evaluated at the same time, as encountered in publish-subscribe systems [76].

The presentation of the complexity results for forward LGQ forests is guided by the following thread. First, a discussion on the size of annotations and of transducer stacks is conducted. Then, the time and space complexities are investigated for eight classes of forward LGQ forests. Inside these fragments, some subfragments that enjoy even better complexities are defined further, though their syntactical characterization gets complex.

In the remainder we consider that the LGQ query has size q, the XML stream (conveying trees) has maximal depth d, maximal breadth b, and size s.

Discussion on the size of an annotation

An annotation is represented as a list of natural numbers in ascending order. We discuss the memory requirements to store such a list, where the greatest number allowed n depends linearly on the stream parameters d, b, or s.

Case 1. The list is an empty list (corresponding to an empty annotation) or a singleton list containing the number 0 (corresponding to a full annotation) or 1. In this case, the list can be represented using constant space $O(1)$.

Case 2. The list is either (a) a singleton list, e.g., [3], or (b) a continuous list of (successive) numbers, e.g., [2,3,4], where the numbers in the list are less than the greatest number allowed n. Such a continuous list can be represented as an interval where the upper delimiter is less that n. Note that a number less than n can be represented using $\log_2 n$ bits. In this case, the list can be represented using $O(\log n)$ bits.

Case 3. The list is an uncontinuous (i.e., with holes) list of numbers, e.g., [2,4], where the numbers in the list are less than the greatest number allowed n. In this case, the list can be represented as a bitset with at most n positions, thus with size $O(n)$.

The stacks of all pushdown transducers defined in Section 5.3.1 contain only annotations, as ensured by their definitions. Recall from Proposition 5.2.5 that the amount of

annotations stored onto the stacks of $\overrightarrow{scope}^{\,x}$ transducers ($x \in \{sdown, pdown, rdown\}$) is d for $x = sdown$, $d + b$ for $x = pdown$, and s for $x = rdown$. The following proposition gives the bound on the maximum number of annotations existent at a time on a stack of transducers for binary predicates.

Proposition 5.5.1. *The stack of a transducer for an LGQ binary predicate has at most d entries, where each entry is an annotation.*

Proof. The stack of each such transducer changes as follows: for each annotation following an opening tag the transducer pushes an annotation onto the stack (it may be the received annotation, the empty annotation or another computed annotation) and for each closing tag an annotation is popped from the stack. A stack can have at most d annotations (entries), for there can be at most d opening tags encountered in the stream before one of their closing tags is received. □

Because each stack entry is an annotation, the size of a transducer stack depends on the size of the annotation, as discussed in the previous paragraph. For each of the above cases of different annotation sizes, different stack sizes are defined that are d times bigger than the annotation size.

Discussion on the size of the buffer for potential answers

The memory needed for processing LGQ queries on streams with SPEX is given by the memory used for transducer stacks and also by the memory used for buffering stream fragments when needed.

The evaluation of a tree query can require extra memory for buffering potential answers. As pointed out in Section 5.2.7, if, for a particular substitution consistent with that tree query and the stream, the image of the head variable (the head image) in encountered in the stream *before* the image of another variable, then that head image becomes a potential answer and has to be buffered until either all variables get an image (in which case the head image becomes an answer), or it is known that they can not get images (in which case the head image is dropped).

In worst case, the entire stream is a potential answer that depends on a variable substitution that happens only at the end of the entire stream. In this pathological case, the entire stream is buffered.

It is worth noting that this buffering of potential answers is a constant aspect of the SPEX problem itself, and thus independent of the method described here.

Well-ordered Queries have buffer-free Evaluation

We noticed there is a forward LGQ fragment containing queries, for which all substitutions consistent with them and any tree ensure that the head image appears after the images of the other variables. Thus, the evaluation of such queries does not need buffering. We call the queries with this fortunate order of variable images *well-ordered queries*, and their class LGQ^{woq}.

Forward forest queries from LGQ^{woq} admit an easy graphical characterization: for all their possible digraph representations, all nodes above the node corresponding to the head variable are on the path to the latter, which additionally has no outgoing edges. LGQ^{woq} contains also all queries that admit an equivalent forward forest query with the above graphical characterization. For example, in Figure 4.3, the first two and the last two digraphs represent well-ordered queries, whereas the middle digraph not. Clearly, LGQ^{woq} includes the LGQ fragment of forward paths, because the head variable of forward path queries is non-source (the first condition), and there is no upper node that does not lead to the node corresponding to the head variable (the second condition).

Although not addressed here, it is interesting to study LGQ fragments that become LGQ^{woq} only in the presence of particular classes of streams, as defined by grammars.

Combined Complexities for Eight Forward LGQ Forest Fragments

The space and time combined complexities for the evaluation of queries from eight forward LGQ forest fragments are given below. The rationale behind choosing these fragments is given by the various sizes of annotations created during query evaluation and by the lack or need to buffer stream fragments. The syntactical characterization of these fragments is given in Table 5.1 and their combined complexities in Table 5.2. All these fragments contain nodetests and all LGQ boolean connectives. The differences between them consist in the types of permissible tree formulas (sdown, pdown, rdown, cf. Section 3.6) and of the LGQ built-in predicates. Recall that an sdown/pdown/rdown tree formula has multi-source variables being also sources of sdown/pdown/rdown paths subformulas. An sdown paths contains only forward vertical atoms. A pdown path contains only forward vertical and horizontal predicates atoms, and starts with a horizontal atom. An rdown path contains foll-atoms.

From any of these eight fragments, a subfragment constructed by removing query constructs listed in Table 5.1 lies in the same complexity class as the fragment from which it is derived, if this subfragment is not already listed in the table separately.

Fragment	sdown/pdown/rdown tree formulas	F	$F^{+} \cup F^{*} \cup \{foll\}$
LGQ_1	none	+	+
LGQ_2	sdown	+	−
LGQ_3	sdown	−	+
LGQ_4	sdown, pdown	−	+
LGQ_5	sdown, pdown, rdown	−	+
LGQ_6	sdown	+	+
LGQ_7	sdown, pdown	+	+
LGQ_8	sdown, pdown, rdown	+	+

Table 5.1: Syntactical Characterization of considered LGQ Fragments

Fragment	Annotation Size	Space Complexity S_i	Time Complexity T_i
LGQ_1	$O(1)$	$O(q \times d)$	$O(q \times s)$
LGQ_2	$O(1)$	$O(q \times d + s)$	$O(q \times s)$
LGQ_3	$O(\log(d))$	$O(q \times d \times \log(d) + s)$	$O(q \times \log(d) \times s)$
LGQ_4	$O(\log(d+b))$	$O(q \times d \times \log(d+b) + s)$	$O(q \times \log(d+b) \times s)$
LGQ_5	$O(\log(s))$	$O(q \times d \times \log(s) + s)$	$O(q \times \log(s) \times s)$
LGQ_6	$O(d)$	$O(q \times d^2 + s)$	$O(q \times d \times s)$
LGQ_7	$O(d+b)$	$O(q \times d \times (d+b) + s)$	$O(q \times (d+b) \times s)$
LGQ_8	$O(s)$	$O(q \times d \times s)$	$O(q \times s^2)$

Table 5.2: Combined Complexity of considered LGQ Fragments

Theorem 5.5.1 (Complexity of Forward LGQ Query Evaluation). *For the LGQ_i fragments defined in Table 5.1, the space S_i and time T_i combined complexities for the evaluating queries of these fragments are the ones given in Table 5.2 ($1 \leq i \leq 8$).*

Discussion. For all eight LGQ fragments the following three properties hold. First, the size of a transducer network for an LGQ query is linear in the size of the query. This property holds (1) due to the linear size of the stream processing function in the corresponding query, as ensured by Proposition 5.2.1 for the translation scheme of Section 5.2.3, and (2) due to the one-to-one mapping of stream processing functions to transducers. Second, each node in the stream has an annotation, the size of which influences both the time and the space complexities of query evaluation. Third, a transducer stack can store at most d annotations, as ensured by Proposition 5.5.1.

The processing with a transducer network requires then time linear in the size of the query q and space linear in the query size q and in the depth d. The time complexity T_i and space complexity S_i depend also on the size of annotations created during processing, as highlighted in Table 5.2. The remainder investigates the size of an annotation that differentiates the complexities of the defined LGQ fragments.

LGQ_1 evaluation needs annotations of constant size. No sdown/pdown/rdown formulas in the query means no multi-source variables, thus no tree queries. This means there are no $\overrightarrow{scope}^{\,x}$ transducers in the network corresponding to the query, and no buffering is needed. Then, the only non-empty annotations on the transducer stacks are the full annotations produced by the in transducer. This corresponds to Case (1) of annotations of constant size. The unions of annotations done by transducers for closure predicates yield always empty or full annotations of constant size.

All remained fragments LGQ_i ($2 \leq i \leq 8$) allow tree queries. The previous discussion on the size of the buffer for potential answers points out that the evaluation of tree queries can require a buffer of maximum size s.

The translation scheme of Section 5.2.3 for a given query adds to the corresponding stream processing function a $\overrightarrow{scope}^{\,x}$ function for each multi-source variable in the query. The corresponding scope transducer creates fresh annotations that are at most d for $x =$

sdown, at most $d + b$ for $x = pdown$, and at most s for $x = rdown$, cf. Proposition 5.2.5. All annotations created by a scope transducer are singleton lists containing one number spanning from 1 to the maximum amount of fresh annotations. The above bounds hold also for the size of the in-mapping and out-mapping relations, cf. the same proposition.

LGQ_2 evaluation needs annotations of constant size. No closure predicates in the query means no unions of annotations done by transducers. Therefore, the annotations have constant size.

The next six fragments LGQ_i $(3 \leq i \leq 8)$ allow queries with closure predicates to a varying degree. Based on this degree, various $\overrightarrow{scope}^{\ x}$ transducers can appear in the transducer network for a given query. The type of such transducers determines the amount of annotations existent at a time on their stacks and circulated downstream the network, as given above. Therefore, depending on these cases, the unions of annotations can have size $O(d)$, $O(d + b)$, or $O(s)$.

Among the next six fragments LGQ_i, the next three $(3 \leq i \leq 5)$ allow only queries with closure predicates, hence *all* transducers for such predicates compute unions. The result of such a union is always a continuous list of numbers and it can be represented as an interval, where the biggest number is bounded by d, $d + b$, or s respectively. This corresponds to Case (2.2) of annotations of size bounded by $\log(d)$, $\log(d + b)$, or $\log(s)$.

The last three fragments LGQ_i $(6 \leq i \leq 8)$ allow queries with non-closure predicates and also to a varying degree closure predicates. The result of annotation unions can be an uncontinuous list of numbers, and it can be represented as a bitset with at most d, $d + b$, or s positions. Apart of the size of annotations, these last three fragments are symmetrical to the previous three ones.

Summarizing the results, we get:

1. The queries contain sdown, but not pdown/rdown formulas. Then, their corresponding networks can contain $\overrightarrow{scope}^{\ sdown}$ transducers that store at most d annotations at a time. If the queries contain only closure predicates, then the unions of annotations can always be represented as intervals, otherwise as bitsets. The latter case corresponds to LGQ_3 and the annotations have size at most $\log(d)$. The former case corresponds to LGQ_6 and the annotations have size at most d.

2. The queries contain sdown and pdown, but not rdown formulas. Then, their corresponding networks can contain $\overrightarrow{scope}^{\ pdown}$ transducers that store at most $d+b$ annotations at a time. If the queries contain only closure predicates, then the unions of annotations can always be represented as intervals, otherwise as bitsets. The latter case corresponds to LGQ_4 and the annotations have size at most $\log(d + b)$. The former case corresponds to LGQ_7 and the annotations have size at most $d + b$.

3. The queries contain sdown, pdown, and rdown formulas. Then, their corresponding networks can contain $\overrightarrow{scope}^{\ rdown}$ transducers that store at most s annotations at a time. If the queries contain only closure predicates, then the unions of annotations can always be represented as intervals, otherwise as bitsets. The latter case corresponds to LGQ_5 and the annotations have size at most $\log(s)$. The former case corresponds to LGQ_8 and the annotations have size at most s. □

Remark 5.5.1. Recall from Chapter 4 that the language of forward LGQ forests keeps its expressiveness, even if the foll-predicate is removed, because foll-atoms can be rewritten into formulas without foll, but with reverse atoms. Such formulas can be rewritten into (possibly) exponentially bigger equivalent forward formulas.

On the other hand, Theorem 5.5.1 states that the time complexity can be quadratic, and the space complexity linear, in the stream size for LGQ_8 containing the foll-predicate.

The tradeoff between the complexities of the latter and the former cases can be easier motivated by various application scenarios. For the evaluation of rather complex queries against a stream of small (but many) independent XML documents [7], the latter approach makes sense, whereas for the evaluation of simpler queries against a stream of large (possibly unbounded) XML documents, the former approach is more appropriated. □

Combined Complexities for Graph Queries

In general, graph queries are rewritten into forward forests with size exponential in the size of the graph queries. Therefore, the evaluation strategy introduced in this chapter would require exponential complexity in the size of the initial query for evaluating them. However, as shown by [74], the evaluation of graph queries is exponential in general. Our work refinds, thus, this result of [74]. As explained below, it goes even beyond and identifies a large LGQ fragment of graph queries, whose evaluation has polynomial upper bounds. This makes our evaluation strategy optimal and complete for graph queries, and, although exponential in general, it is polynomial in particular cases.

These complexity results are derivable from both the complexities of evaluating forward forests, as detailed in this chapter, and the complexities of rewriting graph queries into forward forests, as detailed in Chapter 4. In particular, the rewriting of a graph query s yields a forward forest query, whose size is linear in the size of s, if each connection sequence in s contains neither (i) vertical closure reverse predicates after vertical forward predicates, nor (ii) horizontal closure predicates immediately after horizontal reverse predicates, and nor (iii) vertical closure forward predicates, having as sink a variable with a forward sink-arity greater than one, after vertical forward predicates (cf. Theorem 4.5.3).

We conjecture that an even larger LGQ fragment of graph queries can be evaluated with polynomial combined complexities. By relaxing the condition on the forwardness of the rewritten forest queries, one can expect that more graph queries can be rewritten into polynomially-sized forest queries. This result, co-related with the existence of polynomial (main-memory) evaluation strategies for forest queries (with forward and reverse predicates), e.g., [71], makes the evaluation of these graph queries polynomial.

Improving Space Complexity

When considering transducer networks for the evaluation of LGQ_6 and LGQ_7 queries, there is an interesting monotonicity relation between the annotations existent at any time on the stack of transducers for predicates: any two annotations representing consecutive stack entries have the property that the one near the top is either (1) empty, or (2) the same

as the other one, or (3) represents a list containing numbers that are all greater than the numbers of the other annotation, or (4) the two annotations have a common sublist, and the previous third case applies to the rest of both annotations.

We define the monotonicity binary relation corresponding to the cases 2,3, and 4 above, by extending the order $\leq$ from numbers to lists of numbers, i.e., to annotations:

$$[a] \leq [b] \Leftrightarrow \forall i \lessdot [a], \exists j \lessdot [b] : i \leq j.$$

We give without proof the following proposition that summarizes our observation.

Proposition 5.5.2. *Consider the following configuration of the stack of a transducer for a horizontal or vertical forward LGQ predicate during processing an XML stream:* $[c_n] \mid \ldots \mid [c_1]$ *where* $[c_1]$ *is the stack bottom. Then, either* $[c_i] = [\,]$ *or* $[c_{i-1}] \leq [c_i]$*, where* $1 < i \leq n$.

Such a property is very useful because it exhibits the possibility to store the annotations more efficiently than using a stack. The gain lies in avoiding to store redundant annotations.

Consider that instead of the rigorous access policies of stacks, we allow occasionally ourselves to access entries below their top. We also consider a new symbol, called marker. The basic stack operations can be, then, implemented as follows:

push. We follow the aforementioned four cases. In the first and third cases, we push the marker and then the received empty annotation. In the other two cases, we push only the difference between the received annotation and the top annotation. It should be clear that our stack does not contain overlapping annotations as entries, if they are non-empty.

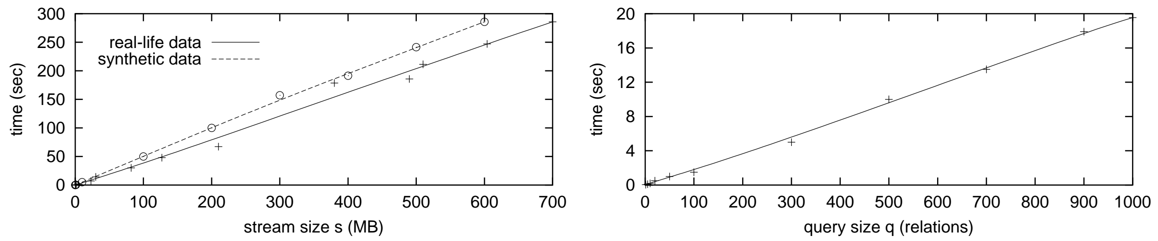
pop. We pop the top entry, as for normal stacks. If after popping the new top becomes a marker, then we pop it too.

top. We collect all annotations starting with the top and ending when the bottom of the stack is reached, or when a marker is found. The union of the collected annotations represents the top annotation.

5.6 Experimental Results

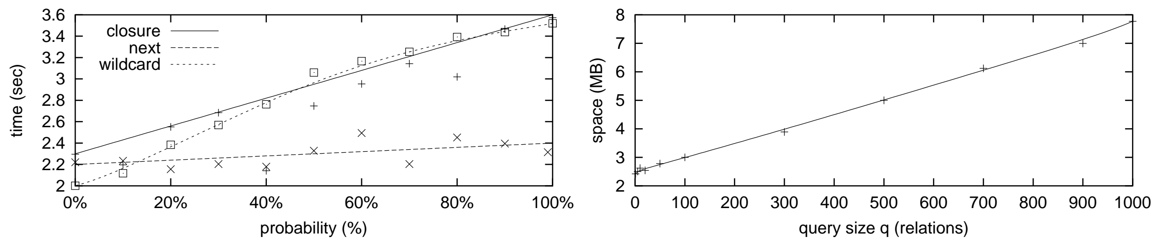
The theoretical complexity results of Section 5.5 are verified by an extensive experimental evaluation conducted on a prototype implementation of our SPEX evaluation in Java (Sun JRE 1.5) on a Pentium 1.5 GHz with 500 MB under Linux 2.4.

XML Streams. The effect of varying the stream size s on the evaluation time is considered for two XML stream sets. The first set [114] provides real-life XML streams, ranging in size from 21 to 21 million nodes and in depth from 3 to 36. The second set provides synthetic XML streams with a slightly more complex structure that allows more precise variations in the workload parameters. The synthetic data is generated from information about the currently running processes on computer networks and allows the specification of both the size and the maximum depth of the generated data (see also Chapter 6).



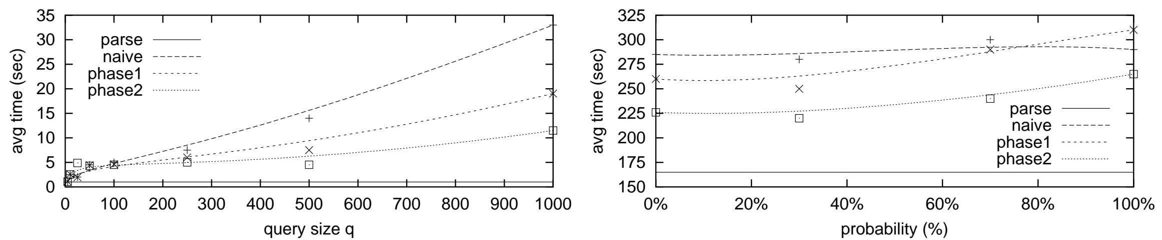
(a) Varying stream size s ($q = 10, 3 \leq d \leq 32$) (b) Varying query size q ($s = 244$ kB, $d = 32$)

Figure 5.7: Scalability ($p_* = p_+ = p_{\mathsf{nextSibl}} = p_\curlywedge = 0.5$)



(a) Effect of p_*, p_+, and p_{nextSibl} (b) Effect of varying query size q

Figure 5.8: If not varied, $s = 244$ kB, $d = 32$, $q = 10$, $p_* = p_+ = p_{\mathsf{nextSibl}} = p_\curlywedge = 0.5$



(a) Varying query size q ($s = 450$ kB) (b) Effect of p_+ ($s = 700$ MB, $q = 10$)

Figure 5.9: Effect of filters. If not varied, $p_* = p_+ = p_{\mathsf{nextSibl}} = 0.5$

Queries. Only LGQ queries that are "grammar-aware" are considered, i.e., that express structures compatible with the grammar of the XML streams under consideration. Their generation has been tuned with the query size q and several probabilities: p_{nextSibl} and p_+ for next-sibling, respectively closure predicates, $p_\curlywedge$ for a multi-source variable, and p_* for the probability of a wildcard nodetest. For example, a path query has $p_\curlywedge = 0$. For each parameter setting, 10–50 queries have been tested, totaling about 2000 queries.

Scalability. Scalability results are presented for stream and query size. In both cases, the depth is bounded in a rather small constant ($d \leq 36$) and its influence on processing time showed to be considerably smaller than of the stream and query size. Figure 5.7 emphasizes the theoretical results: Query processing time increases linearly with the stream size as well as with the query size. The effect is visible in both the real-life and the synthetic data set, with a slightly higher increase for the synthetic data due to its more complex structure.

Varying the query characteristics. Figure 5.8(a) shows an increase of the evaluation time by a factor of less than 2 when p_* and p_+ increase from 0 to 100%. It also suggests that the evaluation times for nextSibl and child are comparable. Further experiments have shown that the evaluation of forward tree and DAG queries is slightly more expensive than the evaluation of simple path queries.

The **memory usage** is almost constant over the full range of the previous tests. Cf. Figure 5.8(b), an increase of the query size q from 1 to 1000 leads to an increase from 2 to 8 MB of the memory for the network and for its processing. The memory use is measured by inspecting the properties of the Java virtual machine (e.g., using `Runtime.totalMemory()` and `Runtime.freeMemory()`).

Stream routing minimization. All previous tests show experimental results for the "naive" version of our evaluation strategy, i.e., that version without the phase_1 and phase_2 routers described in Section 5.4. Figure 5.9 shows how these routers affect the evaluation time. The phase_2 router improves the evaluation time up to 3 times for our tests using queries, whose sizes range from 5 to 1000, cf. Figure 5.9(a). The same figure shows also that, for small XML streams, our evaluation strategy is in average five times slower than the mere parsing of the XML stream[2], if phase_2 is used, 10 times slower if phase_1 is used, and 15 times slower for the naive version (i.e., without routers). Using the phase_2 routers, an increase in the query size q tends to have little to constant influence on the evaluation time. This result is explained by the fact that an increase in the query size leads often to an increase in its selectivity, which sustain afterall the rationale for the usage of routers (see Section 5.4 for more). The same rationale applies for the results of Figure 5.9(b), where the increase of the closure probability (p_+) makes the queries less selective. This leads to a less effective gain achieved by using the routers.

[2]We used the Crimson SAX parser available at `http://xml.apache.org/crimson/`.

5.7 Related Work

Since the XPath standard was proposed as a W3C Recommendation [46] and used by other W3C Recommendations like XSLT [45], XQuery [23], XML-Schema [59], and XPointer [54], significant research and application interest for the XPath language was growing constantly. Some of the research questions related to properties of XPath, like query containment, rewriting, or minimality are detailed in the related work section of Chapter 4. We detail here on the query evaluation problem.

The problem of XPath query evaluation against XML data (may it be stored in main memory or streamed) is one of the most basic and widespread database query problem in the context of XML. In the following, we position its SPEX variant (i.e., against XML streams) versus the query evaluation against tuple (relational) streams. Then, we state shortly the theoretical complexity of XPath evaluation as found in the literature, and give a succinct overview on existing XPath query evaluation techniques by firstly describing some significant work in the field of main-memory query evaluation against XML data, and later surveying approaches in the field of query evaluation against XML streams.

Discussion on the XPath evaluation: XML streams versus tuple streams

Besides the apparent discrepancy concerning modeling aspects between relational and XML data, flat relations (i.e., sets of tuples of rather constant size) can be, of course, used to describe hierarchies (tree data), as XML does. Augmented also by the flourishing research on the topic of querying tuple streams, e.g., [15, 18, 17, 77, 117, 1, 39, 36, 40], the idea of querying *tuple streams conveying XML data* can be appealing at a first glance. However, just a short look would reveal a salient unnaturalness of this approach: to benefit from the hierarchical structure conveyed in the stream, expensive structural joins like parent/child and preceding-sibling/following-sibling have to be computed. Same work in the field of querying tuple streams identified computations of joins as very expensive, for they can require unbounded memory (provided no knowledge about the incoming stream is at hand). In this sense, [12] proves space lower bounds for the evaluation of continuous select-project-join queries over tuple streams, and gives efficient join algorithms for specific cases, e.g., for joins on numerical values. As a general approach to cope with join computation that can require unbounded memory in a tuple stream (as well as in an XML stream) environment, relevant work [36] proposes join computation under memory or time constraints, called *windows*.

In a sense, XML streams can be seen as views upon tuple streams where particular joins (i.e., parent/child and preceding-sibling/following-sibling) are already conveniently materialized for easying further processing. Convenient join materialization means in the context of XML that, along an XML stream, it is easy to encounter the pairs of nodes participating to these joins by simply using a stack to keep track of nodes depth (as done also by our SPEX evaluator). For the parent/child join, (1) a child of a (parent) node is placed in the XML stream between the opening and closing tags of the parent node, and has a depth with one unit higher than the depth of the parent node. For the preceding-

sibling/following-sibling join, (1) the siblings have the same parent node and depth, and one appears after the other in the stream's sequence.

It would be interesting to research, however, on how other kind of joins, e.g., ID-IDREF in XML data enabling graph structures, can be conveniently materialized in a stream of XML. Note that X-scan [94], a automaton-based query operator for XML filtering, computes naively such ID-IDREF joins along an XML stream by simply storing all nodes that might be part of the join and testing, after encountering each new node in the stream, whether this new node is in the join with some already stored nodes.

Discussion on key issues of an efficient XPath evaluation

Recall that the evaluation of an XPath query yields a set of nodes (hence duplicate free) that it further sorted in document order. This can be achieved, e.g., by sorting the list and pruning the duplicates at the end of the query evaluation, or by sorting and pruning the duplicates after the evaluation of each XPath step. The former approach is used by popular XSLT/XQuery processors like [11, 47, 61] and can lead to an exponential blowup of the intermediate results in the size of the query. The latter approach makes the sorting operation a major bottleneck. To partially overcome this, [85] detects and removes unnecessary sorting operations of intermediary results. However, the duplicate elimination operation after the evaluation of each step jeopardizes any attempt to progressive (pipelined) processing that, by avoiding to build intermediate results, is one of the major reasons why query evaluation in relational databases is highly efficient.

The key issue of an efficient XPath evaluation consists in avoiding the creation of duplicates at any time during processing, as failed by, e.g., [11, 47, 113, 61, 7], and successfully considered by, e.g., [70, 78, 84, 86], and also by SPEX.

Combined Complexity of XPath Evaluation is P-complete [71]

Recent work [71, 140, 19] discusses the complexity lower bounds of XPath evaluation.

Polynomial upper bounds for combined complexity (i.e., in the size of the data and the query) of XPath evaluation are given, e.g., by [70], and also by the SPEX evaluator of this chapter.

[71] shows further that the XPath evaluation problem is P-hard by reduction from the monotone boolean circuit value problem, which is P-complete. The combined complexity of several restricted fragments of XPath is further detected: Core XPath is also P-complete, positive (i.e., without negation) Core XPath is LOGCFL-complete, the fragment of path queries (corresponding to LGQ_1) is NL-complete (nondeterministic logarithmic space)[3]. [74] shows recently that the evaluation of LGQ-like graph queries is NP-complete. Recall that Section 5.5 refinds this result and shows further there is a large fragment of LGQ graph queries, whose evaluation has polynomial complexities.

[3]Recall that NL $\subseteq$ LOGCFL $\subseteq$ P.

5.7.1 Query Evaluation against stored XML Data

The main characteristics of query evaluation against stored XML data reside in the random access to the data. This enables several passes over the data, the creation of various indexes, or the compression of the data before processing it, as discussed next.

A. Plain XML data stored in main memory

The issue of efficient XPath evaluation against in-memory XML data received attention recently, when popular XSLT (i.e., XPath-based) processing tools, like Xalan [11], XT [47], and Internet Explorer [113], proved to be highly inefficient on ever-growing XML documents that shifted from simple Web pages to large XML repositories [120, 60, 28, 114]. Experimental evaluations of the above mentioned XSLT processors, as well as XQuery processors, e.g., Galax [61], as performed by [70, 106], show that such processors break for rather small XML documents, e.g., around 33 MB for Galax and 75 MB for Xalan. This fact is exacerbated by expensive main-memory representations of XML documents: DOM-like main-memory structures for XML documents tend to be four-five times larger than their original XML documents [91].

Xalan [11], XT [47], and IE6 [113]

As pointed out also by [70], popular tools like Xalan, XT, and IE, are in fact inefficient even for small XML documents and large queries. At a critical inspection of their code, the XPath evaluation strategy of these tools resembles a straightforward node-at-a-time implementation of the XPath denotational semantics given, e.g., in [128], and as explained next. For a given query, these processors evaluate the first query construct from the root node, get as a result a bag of nodes, then proceed to the evaluation of the next query construct from each node from the previously computed node bag and so on. It is clear that the evaluation of each query construct from a node may result in a set of nodes of size linear in the size of the XML document (e.g., the number of descendants of a node is linear in the XML document size). In this way, the recursive evaluation of the constructs of an XPath query ends up consuming time exponential in the size of the query in the worst case, even for very simple path queries. Consider the evaluation of the XPath query /descendant::a/.../descendant::a against an XML document containing only nested a-nodes. The evaluation of the first step yields the bag of all a nodes in the XML document, the evaluation of the second descendant::a step yields for each a node the set of all its descendants (which has size linear in the document size). The evaluation of the third step is done now from a number of nodes quadratic in the document size and yields a bag of nodes cubic in the document size. Although these intermediary node bags have size exponential in the size of the initial XML document, the amount of *distinct* nodes they contain does not (and can not) exceed the size of the initial XML document.

Context-Value Table Principle [70, 72, 73]

An efficient implementation of full XPath is provided in [70, 72] and improved in [73]. A simplified version of it is used to define the semantics of XPath in [69] and in Chapter 3.

[70] defines a formal bottom-up semantics of full XPath, which leads to a bottom-up main-memory XPath processing algorithm that runs in the worst case in low-degree polynomial time in terms of the data and of the query size. By a bottom-up algorithm is meant a method of processing XPath while traversing the parse tree of the query from its leaves up to its root. The evaluation strategy is based on a *context-value table* principle: given an expression e that occurs in the input query, the context-value table of e specifies all valid combinations of contexts and values, such that e evaluates to a value in a given context. Such a table for expression e is obtained by first computing the context-value tables of the direct subexpressions of e in the parse tree and subsequently combining them into the context-value table for e. Given that the size of each of the context-value tables has a polynomial bound and each of the combination steps can be effected in polynomial time, query evaluation in total under this principle has also a polynomial time bound. A general mechanism for translating the bottom-up algorithm into a top-down one is further discussed, motivated by the computation of fewer useless intermediate results of the latter algorithm. [70] identifies also an XPath fragment, called *Core XPath*, that enjoys linear-time combined complexity. Core XPath contains all XPath axes, nodetests, and the composition operators / and [ ] for constructing paths and filters, and it is a proper subset of an XPath fragment equivalent to LGQ.

Similar Tree Pattern Matching Problems

As pointed out also in Section 5.1, there are similarities of the XPath query evaluation problem with variations of tree matching problems [87, 97], where as well queries and as data can be considered trees and the query evaluation can be reduced to computing the matching of the query tree into the data tree. However, the query trees considered in [87, 97] can be expressed in XPath using only the restricted amount of XPath child and following-sibling axes [87] and descendant axes [97], and it is not trivial to extend these algorithms to cover all axes of XPath.

B. Compressed XML data stored in main memory

In order to deal with large amounts of XML data that can not be kept entirely in main memory, recent research work split into two main directions: querying compressed XML data stored in main memory, and querying streams conveying unmaterialized XML data. Research work for the latter approach is considered further in Section 5.7.2, whereas some work [66, 30, 13] for the former is shortly presented here.

XML data compression is effective because of the high redundancy of self-describing XML documents. Approaches like [66, 30, 13] that compress XML data and evaluate queries in the compressed domain provide a twofold advantage, by avoiding (i) to store and (ii) to query redundant data. [30] separates the text from the skeleton (structure) of

an XML data instance and compresses the skeleton based on sharing of common subtrees into directed acyclic graphs with multiple edges. This compression method can lead to an exponential reduction in the instance size. The compressed, uncompressed, and partially decompressed instances of the same XML document are all equivalent under a bisimulation relation that preserves the structure and the order of the initial XML document. [30] gives also evaluation techniques for XPath axes and set operations on compressed instances and shows that the evaluation of XPath queries only with reverse axes is linear in the size of the query and of the compressed instance (they navigate the instance upwards and do no decompression), whereas for queries with forward axes can be exponential in the query size (they navigate the instance downward and unfold it), making forward axes undesirable. Note that this contrasts to our stream context where forward axes are preferred to reverse axes, because the evaluation of the latter would require to keep a history of the already seen stream.

[66] supplements [30] by further showing that the method of [30] is PSPACE-complete and for positive (i.e., without negation) Core XPath is NP-complete, though on uncompressed trees it is PTIME-complete [71].

XQuec [13] focuses on the compression of the values found in an XML document, motivated by the fact that for a rich corpus of (real and synthetic) XML datasets the measures of [13] speculate a value percentage of up to 80% of the whole document. Each value is compressed individually using an order-preserving textual compression algorithm adequated to that value type, thus enabling to evaluate, in the compressed domain, inequality comparisons. Because XQuec compresses only the textual content of an XML document, its query evaluation technique is rather orthogonal to our SPEX evaluation strategy (which has a big deal on the evaluation of structural queries). In fact, a mixed approach using our evaluation strategy for querying the structure and XQuec methods for querying the text of XML data instances (provided one can a-priori compress the textual components in the input stream) would be surely beneficial.

C. XML data stored in relational databases

XPath Accelerator [80, 79, 78] and Friends [86]

[80, 79, 78] propose an index structure for XML trees, the XPath accelerator, that can completely live inside a relational database system and supports all XPath axes. This index is based on the *pre/post* encoding schemes of the nodes in the XML tree, where by pre/post encoding scheme is meant the association of each node to its preorder/postorder rank as computed in a preorder/postorder traversal of the XML tree. Based on this pre/post encoding, the selection of some nodes from other nodes can be easily specified using relationships between their pre/post values. E.g., the descendant nodes of a node n are those nodes n' that have a preorder value greater than the one of n, and its postorder value lower than the one of n. [80] shows how this index can benefit from the rewrite rules of Chapter 4 by rewriting queries to equivalent queries containing axes further optimizable.

[86] adapts straightforwardly the evaluation of XPath axes based on the relational

storage of the input XML document and the pre/post encoding scheme of [78] to main-memory DOM [145] structures. In this way, [86] improves upon [70], where XPath axes are evaluated linearly in the size of the XML document, whereas in [86] axes are evaluated linearly in the size of the intermediate results which are often much smaller than the entire document.

5.7.2 Query Evaluation against XML Data Streams

Recall that the problem of query evaluation against XML streams bears some of the challenges of the problem of query evaluation against stored data (like efficiency) and further faces new challenges imposed by the sequential one-time access to data.

In the following, we distinguish between the query matching problem, i.e., given a query and a stream, check whether the query selects a non-empty set of nodes from the stream, and the query answering problem, i.e., given a query and the stream, deliver the set of nodes selected by the query from the stream.

A. Query Matching

In the context of publish-subscribe or event notification systems, the XML stream needs to be filtered by a large number of queries. In contrast to the approach of this chapter, filtering engines like [7, 37, 14, 76] assume the stream partitioned into comparatively small XML documents (in the range of hundreds to thousands of elements per XML document), and it is deemed sufficient to determine whether some queries match an XML document, rather than answering the queries. Such XPath queries are often called boolean queries, because the result of their evaluation is a yes/no answer, rather than a set of nodes.

Discussion on DFAs versus PDAs for processing XML streams

Finite automata (FAs) [88] are a natural and effective way to process simple queries like XPath paths. Several works [7, 37, 14, 76] use modified deterministic or non-deterministic FAs to process path queries. Steps of a path are mapped to states of such a non-deterministic machine. A transition from an active state is fired when an element is found in the XML document that matches the transition. If an accepting state is reached, then the document is said to satisfy the query.

There are several interesting issues to mention about the evaluation of XML queries using modified deterministic finite automata (DFAs). All these issues are related to the unboundness of the XML stream characteristics like its depth, or alphabet size, and to the tree-like data conveyed therein. First, the number of states in such a DFA depends on the query **and** on the data stream, and in order to give an upper bound for the number of states, one needs to make various upper bound assumptions on the stream characteristics. Second, such modified DFAs have a computational model significantly different from that of the standard automaton that borrows its name. The computation of states at runtime in such modified DFAs resembles the computation of stack configurations in pushdown automata

(PDAs). In fact, dealing with an unbounded number of states in a DFA resembles even closer the unboundness of a stack size in PDAs. Third, a stack of size proportional to the maximum depth of the trees conveyed in XML streams is necessary for even the most basic sequential navigation and parsing tasks of XML streams. Such tasks require to keep track of the depth of nodes in the tree while traversing it depth-first. Therefore, all DFA-based approaches [7, 37, 14, 76] use, in addition to other data structures, also a stack.

As a proof of concept for the above first and second observation, consider the following scenario for the evaluation of forward paths using sequential compositions of pushdown transducers, as done by our SPEX evaluator described in this chapter, under the assumption that the XML stream depth is bounded. Each SPEX transducer has a stack bounded in the depth of the tree conveyed in the input XML stream. For a given upper bound on this depth, each SPEX transducer can be encoded as a finite transducer, where the stack configurations become states. The number of states is then exponential in the depth of the stream, but finite. Then, the states are computed also at run-time, hence lazily, as stack configurations are computed by SPEX transducers. Note that because stack configurations already encode the node depths, also the third observation is considered. Furthermore, a sequence of finite transducers can be reduced to a single finite transducer, because finite transducers are closed under composition [42]. Such a scenario is reasonable when it is a priori known that such a bound for the stream depth exists, e.g., as inferred from a stream grammar.

Several approaches to the query matching problem [7, 37, 14, 76, 75] are detailed in the following. All of them compile queries into some sort of finite automata that (1) are extended with structures varying from stacks [76] to tries [37] and hash tables [7], and (2) compute the states of the automaton at run-time. These evaluation techniques are designed to process large amounts (i.e., millions) of boolean queries. They identify and eliminate also common subquery prefixes in the structure navigation [57, 37] and also in the content comparison part [81], and are based on the premises that in publish-subscribe systems significant commonality among user interests represented as a set of queries will exist. Such structure and content commonalities among large amounts of XPath queries are also discovered using various cost-based heuristics and used in a SPEX extension, called M-SPEX [67].

X-scan [94], XMLTK [14, 76, 75], and XPush [81]

In the context of integration of large heterogeneous XML data where the data is streamed from remote sources across a network and the query results are seldomly reused for subsequent queries, [94] identifies the on-the-fly evaluation of regular path expressions against XML data as an efficient alternative to the evaluation of joins on locally stored relational tables containing XML data. [94] proposes a novel operator, called X-scan, used in the Tukwila integration system [95], to compute bindings for query variables while XML data arrives across a network, and to deliver incrementally these bindings to other operators of the Tukwila system. The central mechanism underlying the operation of X-scan is a set

of deterministic state machines created for the regular path expressions to be processed. X-scan proceeds as follows: the XML data gets parsed and stored locally as an XML graph, a structural index is built to facilitate fast graph traversal, and the state machines perform a depth-first search over the structural index. When a machine reaches an end state, then the associated regular path matched and a binding is found. The potential problem with the compilation of regular paths into deterministic finite state machines (FSMs), is that the states of each FSM have to be constructed at compile-time, although not all of them might be used at run-time, and their number can be exponential in the size of the regular path.

XMLTK [14, 76, 75] considers the problem of answering a large number of boolean queries (XPath paths with child and descendant axes) against a same XML stream using DFAs. The salient contribution resides in the theoretical study on the number of states in the DFA constructed eagerly, as in X-Scan [94], and lazily, as used in text processing. The lazy computation of the states means that the states are expanded at run-time and only those states are created that are necessary to process the given XML data instance. [76] shows that the number of states of an eager DFA can grow exponentially in the number of XPath queries, and even in the number of wildcards for a single query. For the lazy DFA, [76] proves an upper bound on its number of states that is independent on the number and shape of XPath expressions, and only depends exponentially in the characteristics of the stream grammar. However, if no grammar is available, there is no upper bound guarantee on the amount of memory used. A query matcher based on lazy DFAs validates experimentally the theoretical claims by obtaining a constant throughput independent on the number of queries. In order to guarantee hard upper bounds on the amount of space used, [76] proposes to combine its lazy DFA approach with slower, but more robust alternative evaluation methods like [37].

The idea of computing lazily the automaton states is further used for the XPush machine [81], a modified deterministic pushdown automaton. In order to overcome the relatively high cost of computing states at run-time, [81] proposes a training of the XPush machine before running it on the actual data, training that precomputes some states and transition entries. In addition to [76], XPush eliminates at compile-time common query prefixes as well in the structure navigation part as also in the filter comparison part. The theoretical and empirical analysis of [81] show that the number of states in the lazy XPush machine is about the same order of magnitude as the total number of atomic filters in the query set, much less than the worst case exponential number.

XFilter [7], YFilter [58, 57], and XTrie [37]

XFilter [7] compiles XPath boolean queries with child and descendant axes and filters into a set of finite state machines (FSMs), with each machine responsible for the matching of some query step. Each FSM has extra information regarding, e.g., the identifier of the query containing its corresponding step, the position of its step in the query, and the continuously updated information on the depth level in the XML document where it is supposed to match (information that can be simulated in fact with a stack). The collection of FSMs

of all queries are indexed using a hash table on the nodetests of their corresponding steps. The hash table is used at processing time to keep track of the FSMs that are supposed to match next. When a state machine for a last step in a query has matched, then the whole query, with the identifier carried by the FSM, has matched. Because it keeps track of all instances of partially matched queries, XFilter has an exponential complexity in the size of the query.

[37] proposes a novel index structure, termed XTrie, that is based on decomposing queries viewed as tree patterns into collections of substrings (i.e., sequences of nodetests) and indexes them using a trie. XTrie is more space-efficient than XFilter since the space cost of XTrie is dominated by the number of substrings in each tree pattern, while the space cost of XFilter is dominated by the number of nodetests in each tree pattern (i.e., steps). Also, by indexing on substrings instead of single nodetests, the substring-table entries in XTrie are also probed less often that the hash table entries in XFilter. Furthermore, XTrie ignores partial matchings of queries that are redundant, in contrast to XFilter [7].

YFilter [57, 58] translates a set of boolean queries into a single query where common prefixes are identified and eliminated, and compiles the resulted query into an NFA. Each state of the NFA is associated with (possibly) many queries and when an accepting state is reached at processing time, then the associated queries are satisfied by the input document. Also here, a run-time stack is used in addition to track the active and previously processed states.

B. Query Answering

XSM [104], XSQ [132, 131], and $\chi\alpha o\varsigma$ [20, 21]

An XSM (XML Stream Machine) [104] is a finite state machine, augmented with random-access (input and output) buffers, that processes XML streams with non-recursive structure definition on the fly. Various XQuery [23] primitive expressions e, i.e., filters with joins, the descendant axis, and static element constructors, are translated into XSMs M_e. An XQuery expression is then reduced to an XSM network where the buffer of M_e is an input buffer for $M_{e'}$, if e is a subexpression of e'. Then, an entire XSM network is composed into a single optimized XSM that is finally compiled into a C program. Each buffer is used to store stream fragments depending on the query to evaluate and on the input stream, and has associated a set of read and write pointers.

The extension of XSM approach to handle (1) streams with recursive structure definition and (2) all XPath axes, as both are considered by SPEX, is not further addressed. Real XML data used for information interchange between applications has in general recursive structure definition. A survey [44] of 60 real datasets found 35 to be recursive, from which the ones for data interchange are all recursive. We conjecture that the adaptation of XSMs to process XML streams with recursive structure definition would require an additional stack for each XSM, thus upgrading XSMs to pushdown transducers with random-access buffers. The composition of XSMs into a single XSMs becomes then more complicated, considering at a first glance there is no standard method to compose push-

down transducers into a single one (in fact, standard pushdown transducers are not closed under composition [42]).

XSQ [131] compiles restricted XPath queries (only child and descendant axes, unnested filters with at most one such axis) into an exponential number of pushdown transducers augmented with queues that are gathered into a hierarchical deterministic pushdown transducer. Concerning the worst-case time complexity, XSQ can perform an exponential number of operations per stream message, even for non-recursive streams.

$\chi\alpha o\varsigma$ [20, 21] is an algorithm for evaluating XPath queries with child and descendant axes and their symmetrical reverse axes parent and ancestor. A query is compiled into a DAG structure where nodes are XPath nodetests and edges are XPath axes. The reverse axes are rewritten using rewrite rules similar to the ones of Section 4.3.1, and as also used in previous work of the present author [125]. The evaluation of such a DAG query is based on the incremental construction of a matching-structure consisting of mappings of query nodes from the DAG query to nodes from the tree conveyed in the input stream. This evaluation approach is similar to the tree pattern evaluation algorithm of [116], though the latter constructs the matching-structure bottom-up in the data tree, whereas the former constructs the structure top-down, as imposed by the stream sequence, i.e., depth-first left-to-right preorder traversal of the data tree. All answers of the query are accumulated in this matching-structure, and they are delivered at the very end of the stream (thus no progressive processing is performed). An answer is determined uniquely by exactly one matching of each query node, and all these matchings are accumulated also until the end of the processing. SPEX does also construct such a matching-structure, which is updated constantly on the arrival of new stream messages and distributed on the stacks of transducers, but it contains only sufficient information to determine the next answers, and previous matchings that are not anymore needed for possible new answers are dropped.

XSAG [98] and FluXQuery [100, 99]

XML Stream Attribute Grammars (XSAG) [98] represent a query language for XML streams that allows data transformation. In this formalism, queries are expressed as extended regular tree grammars [102] that (1) are annotated with attribution functions that describe the output to be produced from the input stream, and (2) have productions with right-hand sides being strongly one-unambiguous regular expressions, i.e., expressions for which the parse tree of any word can be unambiguously constructed incrementally with just one symbol lookahead. XSQG queries are processed in linear time with memory consumption bounded in the depth of the stream. Note that our SPEX evaluator has similar time and space complexities for two important LGQ fragments (1) LGQ_1, and (2) LGQ^{woq} with the buffer-free evaluation.

The difference between [98] and our SPEX evaluator takes two important directions. First, the usage of XSAGs is based on the premise that the grammar of the XML stream is known a priori, and no loose specification of the data to be found is allowed (e.g., by means of closure predicates like child^+). Second, as also shown in [149], there is an interesting connection between XPath queries that are always evaluated on some XML

streams (documents) to a non-empty set, and the (regular tree) grammar that defines the class of those XML streams. Simple path queries (thus queries from LGQ_1) can be translated to grammars, whose number of productions is exponential in the size of the query. The intuition resides in the intrinsic difficulty to translate closure predicates to standard grammar formalisms. Structural constraints, as specified by grammars, can be, however, translated linearly into LGQ forest queries containing only horizontal and non-closure vertical predicates.

FluX [100, 99] is an extension of the XQuery language [23] that supports event-based query processing and the conscious handling of memory buffers. [100] defines also safe Flux queries that are never executed before the data items referred to have been fully read from the stream and may be assumed available in main memory buffers. This safety is ensured by the order constraints between selected data items, as provided by grammars. Note that, similar to the notion of query safety, this chapter proposes also the more general notion of query well-orderedness (for queries, whose evaluation does not require buffers) that does not necessarily require grammar information.

C. General-purpose Processing of XML Streams

There are nowadays various SAX-based APIs [110] for processing XML streams. To model such APIs, [133] defines a type and effect system for a programming language λ_{str} with operations that read (conditional destructively and non-destructively) sequentially messages from an XML input stream and write messages to output streams. The benefit of a type and effect system is the static analysis of programs in order to ensure, e.g., that the programs read and write words in which opening and closing tags match. The basis for such a system are visibly pushdown expressions (VPEs) that are used as effects, and correspond to the class of newly discovered visibly pushdown languages [8], which are a proper subset of deterministic context-free languages closed under concatenation, union, intersection, complementation, and Kleene-*. VPEs can be seen as the stream counterparts of regular expression types [89], a notation for regular tree languages [50] used as types for the XDuce programming language [90] that manipulates XML documents as trees.

5.7.3 Hybrid Approaches

By a hybrid evaluation technique is meant here a combination of techniques for main memory XML data and for streams, e.g., [106]. Such approaches are motivated by the constant size increase of real XML documents, e.g., [114, 120], that can not be processed anymore in main memory, and are based on the rather strong assumption that several passes over the input XML document are possible. The general strategy of these approaches is (1) to filter out from the original document fragments that are irrelevant to the query at hand, and (2) to evaluate the query on the (presumably much smaller) filtered XML document. Note that although this method may be temporarily a sufficient solution to process larger XML documents ([106] reports the processing of XML documents several

times larger than the original ones in average), it can still be proven inefficient as soon as XML documents get even larger.

In the context of the Galax XQuery engine, [106] proposes a static inference algorithm that identifies at compile-time the XPath simple paths (only with child and descendant axes) that are required to evaluate a given XQuery query. [106] gives also an algorithm for the simultaneous evaluation of a set of simple paths against the streamed XML document, in the spirit of XFilter [7] presented previously. As XFilter, the algorithm considers a limited fragment of XPath and can perform exponentially in the depth of the XML document.

Chapter 6

Applications

We describe here two real-world applications that have been implemented using the SPEX evaluator described in this work.

6.1 Monitoring Computer Processes

The first application [25] has a twofold goal. First, it monitors parameters of processes running on UNIX computers. Second, it demonstrates the features of our SPEX evaluator:

1. the processing of XML streams with recursive structure definition and unbounded size as gathered from the information about UNIX processes, and
2. the detection of specific patterns in such richly structured XML streams based on the evaluation of rather complicated XPath queries.

This application uses also a novel, sophisticated visualization of its run-time system, called SPEX Viewer, that makes possible to visualize

1. the rewriting of XPath queries into equivalent queries without reverse axes,
2. the networks of pushdown transducers generated from such queries,
3. the incremental processing of XML streams with transducer networks under various optimization settings, and
4. the progressive generation of answers.

Unbounded XML streams. The parameters of processes running on UNIX computers are constantly gathered as a continuous XML stream from the output of the `ps -elfH` command. The information about a process is represented as an XML element process containing child elements for various properties of a process, such as memory and time used, current priority and state, and child processes. Thus, the process hierarchy is represented by parent-child relations between process-elements.

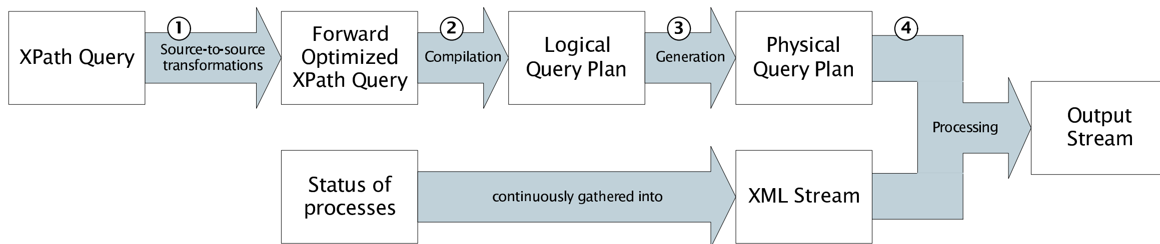


Figure 6.1: Processing Steps of the SPEX processor

The XML stream generated in this manner is unbounded in size and depth, because (1) new process information wrapped in XML is repeatedly sent in the stream and (2) the process hierarchy can contain arbitrarily nested processes. Note that, in practice, many UNIX versions allow at most 512 processes running at a time on one machine, thus limiting the process hierarchy depth of one computer in the monitored system to 512. However, in computer networks, processes running on one computer can lunch subprocesses on other computers, thus the process hierarchy can surpass the barrier of 512.

XPath queries. By means of XPath queries, the monitoring application allows the user to specify what process information conveyed in the XML stream is to be watched and reported back. One can, e.g., monitor suspended processes with CPU and memory expensive subprocesses. More specifically, these can be processes with a certain low priority (e.g., below 10) that are currently stopped and are ancestors of at least one process in the process hierarchy. Furthermore, this other process must use more than 500 MB main memory or be already running for more than 24 hours. The corresponding XPath query is given below

```
/descendant::process[child::time > 24 or child::memory > 500]/ancestor::process
                                  [child::priority < 10 and child::state = 'stopped']
```

SPEX can evaluate also queries with simple aggregations that are not introduced in Chapter 3, but make sense in real-world scenarios. For example, monitoring queries can select processes that together with their subprocesses use a certain amount of memory or that have more than a given number of subprocesses. Note that rather complex and possibly nested queries can be expressed in XPath and processed with SPEX. Query nestings reflect process nestings expressed in the XML stream. The combination of the XML encoding of process information used here and the SPEX evaluator turns out to be a natural, declarative, and effective solution for monitoring parameters of processes.

How the monitoring system works? Querying XML streams with SPEX consists in four steps, as shown in Figure 6.1. First, the input XPath query is rewritten into a forward XPath query, as detailed in Chapter 4. The forward query is compiled into a logical query plan that abstracts out details of the concrete XPath syntax. This is the

function graph of the query. Then, a physical query plan is generated by extending the logical query plan with operators for determination and collection of answers. This is the SPEX transducer network of that function graph. In the last step, the XML stream, which in the chosen application scenario consists in information about the status of processes, is processed continuously with the physical query plan, and the output stream conveying the answers to the original query is generated progressively.

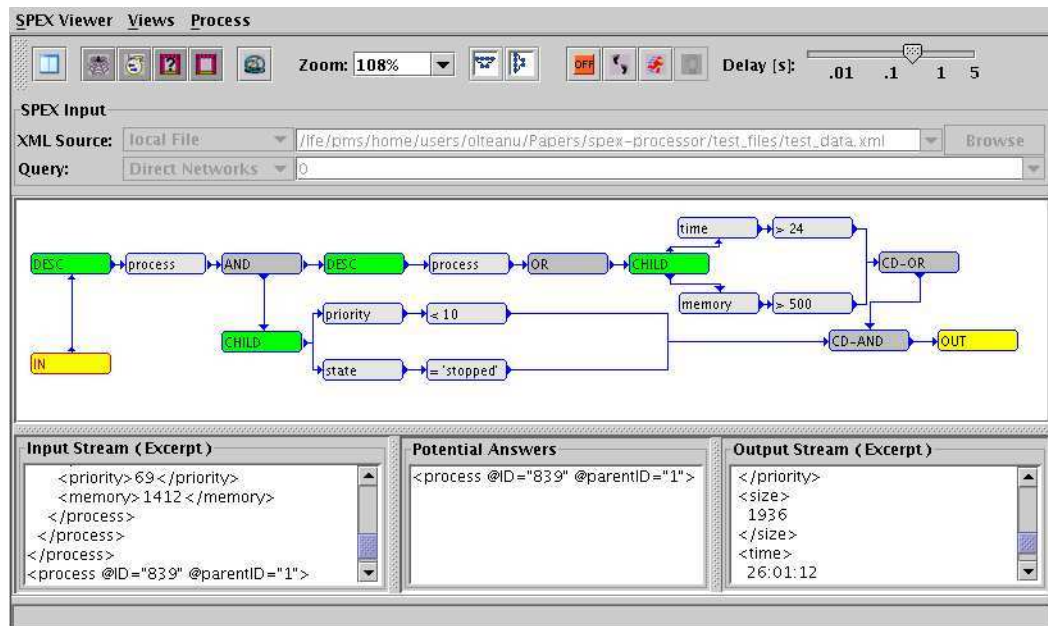


Figure 6.2: SPEX Viewer illustrates how SPEX processes XML streams

How the monitoring system is demonstrated? The system is demonstrated using the SPEX Viewer, that visualizes how our SPEX evaluator processes XML streams. The salient features of the SPEX Viewer consist in illustrating the four steps of the SPEX processor, in particular showing (1) the logical and physical query plans, (2) the stepwise processing of XML streams with physical query plans together with the progressive generation of answers, and (3) the windows over the most recent messages from the input XML stream and the most recent answers.

A vector-based graph rendering engine has been designed and implemented that fits the needs of demonstrating SPEX. Since query plans and SPEX transducer networks may be quite large, reversible visualization actions like moving, hiding parts, and zooming are offered. As transducer stacks change during query processing in content and size, automatic on-line graph reshaping is provided. Figure 6.2 shows a rendering of the physical query plan for an XPath query in the middle area of the visualization tool. The lower area shows (from left to right) windows over the most recent fragment of the input XML stream, over the current potential answers, and over the most recent query answers.

For a detailed insight into the XML stream processing, three processing modi are provided that can be switched at any time during processing: the step-by-step, running, and pause modi. In the *step-by-step mode*, the content of each transducer stack and the message passing between transducers can be inspected for each incoming stream message. In the *running mode*, the input stream is processed message after message with a speed chosen by the user (cf. the delay slider on the topright of Figure 6.2). The *pause mode* is used to interrupt the processing for a detailed inspection of transducers in the network. While in the pause mode, processing can be resumed by selecting either the step-by-step or the running mode. Breakpoints can be specified to alert when a given XML tag reaches given transducers, or when given transducers have particular stack configurations.

6.2 Streamed Traffic and Travel Information

The second application is currently under development within a practical course "Streamed Traffic and Travel Information"[1] offered at the Institute for Computer Science, University of Munich, in the winter term 2004/2005 and co-supervised by the present author. It is a monitoring system for traffic and travel information, such as announcements of traffic congestions or reports on weather conditions. The traffic information is captured from RDS/TMC radio signals [93], first converted into sequences of bits and later into XML streams that are to be watched for data patterns specific or relevant to a given region/time. The main components of the monitoring system are briefly described below.

1. The RDS-information acquisition component consists of (i) dedicated hardware for capturing in real-time RDS/TMC radio signals and converting them into digital information, and of (ii) software for decoding and converting the gathered information into XML streams. Such XML streams contain information about events like location, category, duration, and direction. The location of an event is represented in the XML stream by an identifier together with its country-dependent administrative hierarchy that enables to specify on which fragment of which street in which city, county, state, etc. a given event happens. The event category describes the kind of events dependent also on the location, e.g,, traffic congestion (can happen anywhere) or bull or tomato fights (can happen on the streets of Spain).

2. The XML stream monitoring component is based on the evaluation of XPath queries against XML streams conveying traffic news using our SPEX evaluator. At this step, only the XML stream fragments corresponding to special events relevant to a given location are selected and converted into SVG [63] document fragments (which are also XML-based), whose visualizations are relevant for the corresponding events.

3. The SVG output of the second component updates an already existing SVG-based map of a given location. For example, our application used an SVG-based map of the city Munich.

[1] `http://www.pms.ifi.lmu.de/lehre/praktikum/traveltraffic/04ws05/`

Chapter 7

Conclusion

The work presented in this thesis is devoted to the problem of XPath query evaluation against XML streams. For this problem, it identifies its characteristics and proposes an effective solution. The salient aspects of the proposed solution, e.g., one-pass, progressive, and automata-based, are evolving in key goals for the trend of XPath query evaluation techniques that follows it, thus making it representative.

The problem of XPath query evaluation against XML data (may it be stored in main memory or streamed) is one of the most basic and widespread database query problem in the context of XML. Since the XPath standard was proposed as a W3C Recommendation [46] and used by other W3C Recommendations like XSLT [45], XQuery [23], XML-Schema [59], and XPointer [54], the research and application interest for the XPath language was growing constantly.

Data streams are preferable to data stored in memory in contexts where the data is too large or volatile, or a standard approach to data processing based on data storing is too time or space consuming. In many applications, XML streams are more appropriate than tuple streams, for XML data is tree-like, its size and nesting depth can be unbounded and its structure can have recursive definition. Because of all these characteristics, the problem of query evaluation against XML data streams poses interesting research challenges.

For approaching the problem, this work takes two complementary directions.

First, it identifies that forward queries can be evaluated in a single traversal of the input XML stream. This fact is of importance, because XML streams can be unbounded, and several passes are not affordable. The other queries can be accommodated also to one-pass evaluation by rewriting them into equivalent forward ones. In this respect, this work proposes three rewriting systems that rewrite any query from two query languages considered, i.e., XPath and an abstraction of it, called LGQ, into an equivalent forward query. Our rewriting techniques show the tradeoff between the structural simplicity of the equivalent forward queries and their size. For example, this work gives an exponential (lower and upper) bound for the rewriting of graph queries (expressible directly only in LGQ) into equivalent forward forest queries (expressible in both XPath and LGQ). Also, a linear upper bound is given for the rewriting of forest queries into equivalent forward single-join DAG queries, which are more complex than forest queries. Using the rewriting

systems, this work investigates also several other properties of LGQ (and also XPath), e.g., the expressivity of some of its fragments, the query minimization, and even the query evaluation.

Second, a streamed and progressive evaluation strategy of forward forest queries against XML streams is proposed. The streamed aspect of the evaluation resides in the sequential (as opposed to random) access to the messages of the XML stream. A progressive evaluation delivers incrementally the query answers as soon as possible. The proposed evaluation strategy compiles queries in networks of deterministic pushdown transducers that process XML streams with polynomial time and space complexities in both the stream and the query sizes.

The results of this work took various dissemination directions. Our results on XPath query rewriting are used, e.g., for other XPath query evaluators against XML streams [84, 138, 106, 131, 129, 21] or for optimization of XPath query evaluation in relational databases [79, 80]. For practitioners, implementations of the rewriting and evaluation algorithms are publicly available at `http://spex.sourceforge.net`, and are used in applications for monitoring highway traffic events or processes running on UNIX computers.

Recently, new research relevant to the problem of querying XML streams and based directly upon this work has been co-investigated by the present author. Its general threads converge towards dealing with contexts where the number of queries to be evaluated simultaneously is large [67], or the memory available for query evaluation has given bounds [139]. There, cost-based heuristics are deployed to find out efficient query plans for sets of queries, and respectively to find out which potential answers stored in memory can be discarded when free memory is needed. Such work is not detailed here, but it constitutes a natural continuation of the research investigations in the area of querying XML streams started by this work.

Appendix A

Proofs

Proof of Lemma 3.8.1

I. *XPath* $\subseteq$ *LGQ Forests*. We prove that for any XPath query p and tree T, its answer is the answer delivered by the LGQ query q representing the encoding of p.

Let us consider $(r, f) = \overrightarrow{\mathcal{XL}}[\![p]\!](v)$, for any XPath query p. Also, consider $\beta = \mathit{subst}(\text{Vars}(f), T)$ the set of all possible substitutions mapping variables in f to nodes in T. Thus, $\forall v \in \text{Vars}(f) : \pi_v(\beta) = \text{Nodes}(T)$.

This proof has two parts. First, we show that the set of substitutions from β that are consistent to f and T and that are further restricted to the variables v and r is the set of pairs of source and answer nodes as computed by the semantics function $\mathcal{XQ}$ on p and T:

$$\pi_{v,r}(\mathcal{LF}_T[\![f]\!](\beta)) = \mathcal{XQ}_T[\![p]\!]. \tag{A.1}$$

Second, we show that the answer to p is the set of images of r under the consistent substitutions from β.

We prove Equation (A.1) using induction on the structure of XPath queries.

Base Case. $p = \alpha{::}\eta$. Then, $f = \alpha'(v, v_1) \wedge \eta(v_1)$ with $\alpha' = \mathit{pred}^{-1}(\alpha)$.

$$\begin{aligned}
\mathcal{XQ}[\![\alpha{::}\eta]\!] &= \{(x,y) \mid \alpha'(x,y), \mathit{test}(y,\eta)\} \stackrel{*}{=} \{(s(v), s(v_1)) \mid s \in \beta, \alpha'(s(v), s(v_1)), \mathit{test}(s(v_1), \eta)\}\\
&= \{(s(v), s(v_1)) \mid s \in \mathcal{LF}_T[\![\alpha'(v, v_1)]\!](\beta), s \in \mathcal{LF}_T[\![\eta(v_1)]\!]\}\\
&= \pi_{v,v_1}(\mathcal{LF}_T[\![\alpha'(v, v_1) \wedge \eta(v_1)]\!](\beta)).
\end{aligned}$$

The equality $*$ holds for $x = s(v)$ and $y = s(v_1)$, where s is a substitution consistent with f and T.

Induction Hypothesis. Equation (A.1) holds for the XPath queries p_1 and p_2

$$\pi_{v,v_1}(\mathcal{LF}_T[\![f_1]\!](\beta)) = \mathcal{XQ}_T[\![p_1]\!], \text{ where } (v_1, f_1) = \overrightarrow{\mathcal{XL}}[\![p_1]\!](v) \tag{A.2}$$

$$\pi_{v',v_2}(\mathcal{LF}_T[\![f_2]\!](\beta)) = \mathcal{XQ}_T[\![p_2]\!], \text{ where } (v_2, f_2) = \overrightarrow{\mathcal{XL}}[\![p_2]\!](v') \tag{A.3}$$

In case p_1 is in a filter, Equation (A.2) becomes (the same holds also for p_2)

$$\pi_v(\mathcal{LF}_T[\![f_1]\!](\beta)) = \mathcal{XF}_T[\![p_1]\!],\ \text{where } (v_1, f_1) = \overrightarrow{\mathcal{XL}}[\![p_1]\!](v).$$

Induction Steps. We show next that Equation (A.1) holds also for the XPath queries $/p_1$, p_1/p_2, $p_1 \mid p_2$, $p_1 - p_2$, $p_1[p_2]$, p_1 or p_2, p_1 and p_2, and $not(p_1)$.

1. $p = /p_1$. Then, $(v_1, \mathsf{root}(v_0) \wedge f_1) = \overrightarrow{\mathcal{XL}}[\![p]\!](v_0)$.

$$\begin{aligned}\mathcal{XQ}_T[\![/p_1]\!] &= \text{Nodes}(T) \times \{y \mid (x,y) \in \mathcal{XQ}_T[\![p_1]\!], test(x, \mathsf{root})\}\\ &= \text{Nodes}(T) \times \{y \mid (x,y) \in \pi_{v_0,v_1}(\mathcal{LF}_T[\![f_1]\!](\beta)), test(x, \mathsf{root})\}\\ &= \text{Nodes}(T) \times \{s(v_1) \mid s \in \mathcal{LF}_T[\![f_1]\!](\beta), test(s(v_0), \mathsf{root})\}\\ &= \text{Nodes}(T) \times \pi_{v_1}(\mathcal{LF}_T[\![f_1 \wedge \mathsf{root}(v_0)]\!](\beta)).\end{aligned}$$

This means that the pairs of any node and the nodes computed by $\pi_{v_1}(\mathcal{LF}_T[\![f_1 \wedge \mathsf{root}(v_0)]\!](\beta))$ is in the result.

2. $p = p_1/p_2$. Then, $(v_2, f_1 \wedge f_2) = \overrightarrow{\mathcal{XL}}[\![p_1/p_2]\!](v)$ and $v' = v_1$ in Equation (A.3).

$$\begin{aligned}\mathcal{XQ}_T[\![p_1/p_2]\!] &= \{(x,z) \mid (x,y) \in \mathcal{XQ}_T[\![p_1]\!], (y,z) \in \mathcal{XQ}_T[\![p_2]\!]\}\\ &= \{(x,z) \mid (x,y) \in \pi_{v,v_1}(\mathcal{LF}_T[\![f_1]\!](\beta)), (y,z) \in \pi_{v_1,v_2}(\mathcal{LF}_T[\![f_2]\!](\beta))\}\\ &= \{(s(v), s(v_2)) \mid s \in \mathcal{LF}_T[\![f_1]\!](\beta), s \in \mathcal{LF}_T[\![f_2]\!](\beta)\} = \pi_{v,v_2}(\mathcal{LF}_T[\![f_1 \wedge f_2]\!](\beta)).\end{aligned}$$

3. $p = p_1 \mid p_2$. Then, $(v_1, f_1 \vee f_2) = \overrightarrow{\mathcal{XL}}[\![p_1 \mid p_2]\!](v)$ and $v' = v$, $v_2 = v_1$ in Equation (A.3).

$$\begin{aligned}\mathcal{XQ}[\![p_1 \mid p_2]\!] &= \mathcal{XQ}[\![p_1]\!] \cup \mathcal{XQ}[\![p_2]\!] = \pi_{v,v_1}(\mathcal{LF}_T[\![f_1]\!](\beta)) \cup \pi_{v,v_1}(\mathcal{LF}_T[\![f_2]\!](\beta))\\ &= \pi_{v,v_1}(\mathcal{LF}_T[\![f_1 \vee f_2]\!](\beta)).\end{aligned}$$

4. $p = p_1[p_2]$. Then, $(v_1, f_1 \wedge f_2) = \overrightarrow{\mathcal{XL}}[\![p_1[p_2]]\!](v)$ and $v' = v_1$ in Equation (A.3) and p_2 is in a filter.

$$\begin{aligned}\mathcal{XQ}_T[\![p_1[p_2]]\!] &= \{(x,y) \mid (x,y) \in \mathcal{XQ}_T[\![p_1]\!], y \in \mathcal{XF}_T[\![p_2]\!]\}\\ &= \{(x,y) \mid (x,y) \in \pi_{v,v_1}(\mathcal{LF}_T[\![f_1]\!](\beta)), y \in \{x_0 \mid \exists x : (x_0, x) \in \mathcal{XF}_T[\![p_2]\!]\}\}\\ &= \{(s(v), s(v_1)) \mid s \in \mathcal{LF}_T[\![f_1]\!](\beta), s(v_1) \in \mathcal{LF}_T[\![f_2]\!](\beta)\}\\ &= \pi_{v,v_1}(\mathcal{LF}_T[\![f_1 \wedge f_2]\!](\beta)).\end{aligned}$$

5. $p = p_1$ or p_2. Then, $(v, f_1 \vee f_2) = \overrightarrow{\mathcal{XL}}[\![p_1 \text{ or } p_2]\!](v)$ and $v' = v$ in Equation (A.3), and p, p_1, and p_2 are in filters.

$$\begin{aligned}\mathcal{XF}_T[\![p_1 \text{ or } p_2]\!] &= \mathcal{XF}_T[\![p_1]\!] \cup \mathcal{XF}_T[\![p_2]\!] = \pi_v(\mathcal{LF}_T[\![f_1]\!](\beta)) \cup \pi_v(\mathcal{LF}_T[\![f_2]\!](\beta))\\ &= \pi_v(\mathcal{LF}_T[\![f_1 \vee f_2]\!](\beta)).\end{aligned}$$

6. $p = p_1$ and p_2. Then, $(v, f_1 \wedge f_2) = \overrightarrow{\mathcal{XL}}[\![p_1 \text{ and } p_2]\!](v)$ and $v' = v$ in Equation (A.3), and p, p_1, and p_2 are in filters.

$$\begin{aligned}\mathcal{XF}_T[\![p_1 \text{ and } p_2]\!] &= \mathcal{XF}_T[\![p_1]\!] \cap \mathcal{XF}_T[\![p_2]\!] = \pi_v(\mathcal{LF}_T[\![f_1]\!](\beta)) \cap \pi_v(\mathcal{LF}_T[\![f_2]\!](\beta))\\ &= \pi_v(\mathcal{LF}_T[\![f_1 \wedge f_2]\!](\beta)).\end{aligned}$$

7. $p = p_1 - p_2$. Then, $(v_1, f_1 \wedge \neg Q(v_1)) = \overrightarrow{\mathcal{XL}}[\![p_1 - p_2]\!](v)$, where $Q(v_1) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v_1) \wedge f_2$. Also, $v' = v$, $v_2 = v_1$ in Equation (A.3).

$$\begin{aligned}&\pi_{v,v_1}(\mathcal{LF}_T[\![f_1 \wedge \neg Q(v_1)]\!](\beta)) = \pi_{v,v_1}(\mathcal{LF}_T[\![f_1]\!](\beta)) \cap \pi_{v,v_1}(\mathcal{LF}_T[\![\neg Q(v_1)]\!](\beta))\\ &= \pi_{v,v_1}(\mathcal{LF}_T[\![f_1]\!](\beta) \cap (\beta - \mathcal{LF}_T[\![Q(v_1)]\!](\beta))) = \pi_{v,v_1}(\mathcal{LF}_T[\![f_1]\!](\beta) - \mathcal{LF}_T[\![Q(v_1)]\!](\beta))\\ &= \mathcal{XQ}_T[\![p_1]\!] - \pi_{v,v_1}(\{s \mid s \in \beta, s(v_1) \in \mathcal{LQ}_T[\![clause(Q)]\!]\})\\ &\stackrel{+}{=} \mathcal{XQ}_T[\![p_1]\!] - \pi_{v,v_1}(\{s \mid s \in \beta, s(v_1) \in \pi_{v_1}(\mathcal{LF}_T[\![\mathsf{root}(v_0) \wedge \mathsf{child}^*(v_0, v) \wedge f_2]\!](\beta'))\})\\ &= \mathcal{XQ}_T[\![p_1]\!] - \pi_{v,v_1}(\{s \mid s \in \beta, s(v_1) \in \pi_{v_1}(\mathcal{LF}_T[\![\mathsf{root}(v_0) \wedge \mathsf{child}^*(v_0, v)]\!](\beta') \cap \mathcal{LF}_T[\![f_2]\!](\beta'))\})\\ &= \mathcal{XQ}_T[\![p_1]\!] - \pi_{v,v_1}(\{s \mid s \in \beta, s(v_1) \in (\text{Nodes}(T) \cap \pi_{v_1}(\mathcal{LF}_T[\![f_2]\!](\beta')))\})\\ &= \mathcal{XQ}_T[\![p_1]\!] - \pi_{v,v_1}(\{s \mid s \in \beta, s(v_1) \in \pi_{v_1}(\mathcal{LF}_T[\![f_2]\!](\beta'))\})\\ &\stackrel{*}{=} \mathcal{XQ}_T[\![p_1]\!] - \pi_{v,v_1}(\{s \mid s \in \beta, s(v_1) \in \pi_{v_1}(\mathcal{LF}_T[\![f_2]\!](\beta))\})\\ &= \mathcal{XQ}_T[\![p_1]\!] - \pi_{v,v_1}(\{s \mid s \in \mathcal{LF}_T[\![f_2]\!](\beta)\})\\ &= \mathcal{XQ}_T[\![p_1]\!] - \mathcal{XQ}_T[\![p_2]\!].\end{aligned}$$

The variable v_0 is fresh for f_2. In Equation (+), $\beta' = subst(\text{Vars}(f_2) \cup \{v_0\}, T)$. In Equation (*), $\pi_v(\mathcal{LF}_T[\![f_2]\!])(\beta') = \pi_v(\mathcal{LF}_T[\![f_2]\!])(\beta)$, because $v_0 \notin \text{Vars}(f_2)$.

8. $p = not(p_1)$. Then, $(v, \neg Q(v)) = \overrightarrow{\mathcal{XL}}[\![not(p)]\!](v)$, where $Q(v) \leftarrow \mathsf{root}(v_0) \wedge \mathsf{child}^+(v_0, v) \wedge f_1$. Also, p and p_1 are in filter. This case is treated similarly to case 7.

$$\begin{aligned}&\pi_v(\mathcal{LF}_T[\![\neg Q(v)]\!](\beta)) = \pi_v(\beta - \mathcal{LF}_T[\![Q(v)]\!](\beta))\\ &\stackrel{1}{=} \text{Nodes}(T) - \pi_v(\{s \mid s(v) \in \pi_v(\mathcal{LF}_T[\![\mathsf{root}(v_0) \wedge \mathsf{child}^*(v_0, v) \wedge f_1]\!](\beta'))\})\\ &= \text{Nodes}(T) - \pi_v(\mathcal{LF}_T[\![\mathsf{root}(v_0) \wedge \mathsf{child}^*(v_0, v) \wedge f_1]\!](\beta'))\\ &= \text{Nodes}(T) - \pi_v(\mathcal{LF}_T[\![\mathsf{root}(v_0) \wedge \mathsf{child}^*(v_0, v)]\!](\beta') \cap \mathcal{LF}[\![f_1]\!](\beta'))\\ &\stackrel{2}{=} \text{Nodes}(T) - \pi_v(\beta' \cap \mathcal{LF}[\![f_1]\!](\beta')) \stackrel{3}{=} \text{Nodes}(T) - \pi_v(\mathcal{LF}[\![f_1]\!](\beta))\\ &= \text{Nodes}(T) - \mathcal{XF}_T[\![p_1]\!] = \mathcal{XF}_T[\![not(p_1)]\!].\end{aligned}$$

Equation (1) uses the hypothesis $v \in \text{Vars}(f_1) : \pi_v(\beta) = \text{Nodes}(T)$ and omits the intermediate step $\mathcal{LF}_T[\![Q(v)]\!](\beta) = \{s \mid s(v) \in \pi_v \mathcal{LQ}_T[\![clause(Q)]\!]$. As for case 6, in Equation (2), $\beta' = subst(\text{Vars}(f_2) \cup \{v_0\}, T)$ and in Equation (3) $\pi_v(\mathcal{LF}_T[\![f_1]\!])(\beta') = \pi_v(\mathcal{LF}_T[\![f_1]\!])(\beta)$.

We show next that the answer to any absolute XPath query p is the set of images for the variable r as computed by $\mathcal{LQ}_T[\![Q(r) \leftarrow f]\!]$, where $(v, f) = \overrightarrow{\mathcal{XL}}[\![p]\!](_)$:

$$\{y \mid \exists x : (x, y) \in \mathcal{XQ}_T[\![p]\!]\} = \mathcal{LQ}_T[\![Q(r) \leftarrow f]\!].$$

We prove this equation by using Equation (A.1), which is already proven above

$$\pi_{v,r}(\mathcal{LF}_T[\![f]\!](\beta)) = \mathcal{XQ}_T[\![p]\!].$$

Then,

$$\begin{aligned}\{y \mid \exists x : (x,y) \in \mathcal{XQ}_T[\![p]\!]\} &= \{y \mid \exists x : (x,y) \in \pi_{v,r}(\mathcal{LF}_T[\![f]\!](\beta))\} \\ &= \pi_r(\mathcal{LF}_T[\![f]\!](\beta)) = \mathcal{LQ}_T[\![Q(r) \ \leftarrow \ f]\!].\end{aligned}$$

II. *LGQ Forests* $\subseteq$ *XPath*. We conduct induction on the tree structure of any LGQ tree query $Q(v) \ \leftarrow \ b$. We show first for any tree subformula f of b that consists in all atoms from b reachable from its source x, there is an equivalent XPath query p that is the encoding of f using $\overrightarrow{\mathcal{LX}}$. More specifically, we show that

$$\begin{cases}\mathcal{XQ}[\![p]\!] = \pi_{x,v}(\mathcal{LF}[\![\eta(y)]\!](\beta)) & , x \rightsquigarrow_f v \\ \mathcal{XF}[\![p]\!] = \pi_x(\mathcal{LF}[\![\eta(y)]\!](\beta)) & , \text{otherwise,}\end{cases} \text{ where } p = \overrightarrow{\mathcal{LX}}_{v,b}[\![f]\!](x).$$

Consequently, we show that for any LGQ forest there is an equivalent XPath query.

Base Case. We show there are XPath queries equivalent to LGQ atoms.

1. $f = \eta(x)$. Then, $\overrightarrow{\mathcal{LX}}_{v,b}[\![\eta(x)]\!](x) = [\mathsf{self}::\eta]$.

$$\begin{aligned}&\mathcal{XQ}[\![[\mathsf{self}::\eta]]\!] = \{n \mid n \in \mathcal{XF}[\![\mathsf{self}::\eta]\!]\} \\ &= \{n \mid \mathit{test}(n,\eta)\} = \pi_y(\mathcal{LF}[\![\eta(y)]\!](\beta)) = \pi_y(\mathcal{LF}[\![f]\!](\beta)).\end{aligned}$$

2. $f = \alpha(x,y)$.

2a. $y = v$. Then, $\overrightarrow{\mathcal{LX}}_{v,b}[\![\alpha(x,y)]\!](x) = \mathit{pred}^{-1}::*$.

$$\begin{aligned}&\mathcal{XQ}[\![\mathit{pred}^{-1}(\alpha)::*]\!] = \{(n,m) \mid (n,m) \in \mathit{pred}^{-1}(\alpha), \mathit{test}(m,*)\} \\ &= \{(n,m) \mid (n,m) \in \mathit{pred}^{-1}(\alpha)\} = \pi_{x,v}(\mathcal{LF}[\![\alpha(x,y)]\!](\beta)).\end{aligned}$$

2b. $y \neq v$. Then, $\overrightarrow{\mathcal{LX}}_{v,b}[\![\alpha(x,y)]\!](x) = [\mathit{pred}^{-1}::*]$.

$$\begin{aligned}&\mathcal{XF}[\![[\mathit{pred}^{-1}(\alpha)::*]]\!] = \{n \mid (n,m) \in \mathit{pred}^{-1}(\alpha), \mathit{test}(m,*)\} \\ &= \{n \mid (n,m) \in \mathit{pred}^{-1}(\alpha)\} = \pi_x(\mathcal{LF}[\![\alpha(x,y)]\!](\beta)).\end{aligned}$$

Induction Hypothesis. We consider there are equivalent XPath queries for the formulas A and B.

Induction Step. We show there are equivalent XPath queries to tree formulas $f = A \wedge B$, where x is the source variable, A is an atom, and B is a formula, the latter two having equivalent XPath queries. We treat next the case when v reachable from x ($x \rightsquigarrow_b v$), the other case is similar.

1. $A = \eta(x)$, $B = f_x$, where f_x consists in all atoms from f reachable from x. Then, $\overrightarrow{\mathcal{LX}}_{v,b}[\![\eta(x) \wedge f_x]\!](x) = [\mathsf{self::}\eta]\ \mathit{left}_x$, where $\mathit{left}_x = \overrightarrow{\mathcal{LX}}_{v,b}[\![f_x]\!](x)$.

$$\mathcal{XQ}[\![[\mathsf{self::}\eta]\mathit{left}_x]\!] = \{(n,m) \mid n \in \mathcal{XF}[\![\mathsf{self::}\eta]\!], (n,m) \in \mathcal{XQ}[\![\mathit{left}_x]\!]\}$$
$$= \{(n,m) \mid n \in \pi_x(\mathcal{LF}[\![\eta(x)]\!](\beta)), (n,m) \in \pi_{x,v}(\mathcal{LF}[\![f_x]\!](\beta))\} = \pi_{x,v}(\mathcal{LF}[\![f]\!](\beta)).$$

2. $A = \alpha(x,y)$, $B = f_x \wedge f_y$, where f_x consists in all atoms from f reachable from x via any other variable but y, and f_y consists in all atoms from b reachable from y. Note that because f is a tree, f_x and f_y do not have common atoms, and $f = A \wedge B$. Let $\mathit{left}_x = \overrightarrow{\mathcal{LX}}_{v,b}[\![f_x]\!](x)$, $\mathit{left}_y = \overrightarrow{\mathcal{LX}}_{v,b}[\![f_y]\!](x)$, and $\mathit{step} = \overrightarrow{\mathcal{LX}}_{v,b}[\![\alpha(x,y)]\!](x)$.

2a. $y \rightsquigarrow_f v$ or $y = v$. Then, $\overrightarrow{\mathcal{LX}}_{v,b}[\![\alpha(x,y) \wedge f_x \wedge f_y]\!](x) = [\mathit{left}_x]/\mathit{step}\ \mathit{left}_y$.

$$\mathcal{XQ}[\![[\mathit{left}_x]/\mathit{step}\ \mathit{left}_y]\!] = \{(n,m) \mid n \in \mathcal{XF}[\![\mathit{left}_x]\!], (n,p) \in \mathcal{XQ}[\![\mathit{step}]\!], (p,m) \in \mathcal{XQ}[\![\mathit{left}_y]\!]\}$$
$$= \{(n,m) \mid n \in \pi_x(\mathcal{LF}[\![f_x]\!](\beta)), (n,p) \in \pi_{x,y}(\mathcal{LF}[\![\alpha(x,y)]\!](\beta)), (p,m) \in \pi_{y,v}(\mathcal{LF}[\![f_y]\!](\beta))\}$$
$$= \pi_{x,v}(\mathcal{LF}[\![\alpha(x,y) \wedge f_x \wedge f_y]\!](\beta)) = \pi_{x,v}(\mathcal{LF}[\![f]\!](\beta)).$$

2b. $y \not\rightsquigarrow_f v$ or $y \neq v$. Then, $\overrightarrow{\mathcal{LX}}_{v,b}[\![\alpha(x,y) \wedge f_x \wedge f_y]\!](x) = [\mathit{step}\ \mathit{left}_y]\mathit{left}_x$.

$$\mathcal{XQ}[\![[\mathit{step}\ \mathit{left}_y]\mathit{left}_x]\!] = \{(n,m) \mid n \in \mathcal{XF}[\![\mathit{step}\mathit{left}_y]\!], (n,m) \in \mathcal{XQ}[\![\mathit{left}_x]\!]\}$$
$$= \{(n,m) \mid n \in \pi_x(\mathcal{LF}[\![\alpha(x,y) \wedge f_y]\!](\beta)), (n,m) \in \pi_{x,v}(\mathcal{LF}[\![f_x]\!](\beta))\}$$
$$= \pi_{x,v}(\mathcal{LF}[\![\alpha(x,y) \wedge f_x \wedge f_y]\!](\beta)) = \pi_{x,v}(\mathcal{LF}[\![f]\!](\beta)).$$

3. $A = \neg N(x)$, $B = f_x$, where f_x consists in all atoms from f reachable from x. Then, $\overrightarrow{\mathcal{LX}}_{v,b}[\![\neg N(x)]\!](x) = [\mathsf{self::* \text{ - }} \mathcal{X}[\![\mathit{clause}(N)]\!]]$ and $\mathit{left}_x = \overrightarrow{\mathcal{LX}}_{v,b}[\![f_x]\!](x)$.

$$\mathcal{XQ}[\![[\mathsf{self::}\eta]\mathit{left}_x]\!] = \{(n,m) \mid n \in \mathcal{XF}[\![\mathsf{self::}\eta]\!], (n,m) \in \mathcal{XQ}[\![\mathit{left}_x]\!]\}$$
$$= \{(n,m) \mid n \in \pi_x(\mathcal{LF}[\![\eta(x)]\!](\beta)), (n,m) \in \pi_{x,v}(\mathcal{LF}[\![f_x]\!](\beta))\} = \pi_{x,v}(\mathcal{LF}[\![f]\!](\beta)).$$

We show next that for any LGQ forest formula f there is an equivalent XPath query. The base case is for f a tree formula and holds due to the above proof. The induction hypothesis states there are equivalent queries for the forest formulas f_1 and f_2, where all atoms from both formulas are reachable via the source variable x. Then, it holds also for $f_1 \vee f_2$ (the induction step). Let $p_1 = \overrightarrow{\mathcal{LX}}_{v,b}[\![f_1]\!](x)$, $p_2 = \overrightarrow{\mathcal{LX}}_{v,b}[\![f_2]\!](x)$, and $\mathcal{XQ}[\![p_1]\!] = \pi_{x,v}(\mathcal{LF}[\![f_1]\!](\beta))$, $\mathcal{XQ}[\![p_2]\!] = \pi_{x,v}(\mathcal{LF}[\![f_2]\!](\beta))$. Then,

$$\mathcal{XQ}[\![p_1 \mid p_2]\!] = \mathcal{XQ}[\![p_1]\!] \cup \mathcal{XQ}[\![p_2]\!] = \pi_{x,v}(\mathcal{LF}[\![f_1]\!](\beta)) \cup \pi_{x,v}(\mathcal{LF}[\![f_2]\!](\beta))$$
$$= \pi_{x,v}(\mathcal{LF}[\![f_1 \vee f_2]\!](\beta)).$$

The equivalence for queries follows then directly from the projection of the pairs computed for formulas on the head variable.

Proof of Lemma 4.3.3

For an instance $l \to r$ of each rule (4.5) through (4.24) under an LGQ$^{\to}$ substitution $\sigma = \{\underline{x} \mapsto x, \underline{y} \mapsto y\}$, we show that (1) $l \equiv r$, and (2) $s \equiv s[r/l]$.

For the first part of the proof, we may use below some equivalences derived from definitions of base binary predicates on nodes in trees. The second part of the proof follows from Proposition 3.3.1, with the condition that the subformulas of s and t obtained by removing l, respectively r, do not contain variables appearing only in r, respectively l, and not in the other one. Indeed, both l and r have the same variables, or r does not contain variables at all.

We use the following implications ($h \in \mathsf{H}^?$, $v_1, v_2 \in \{\mathsf{fstChild}, \mathsf{child}\}$)

$$v_1(y,x) \wedge v_2(z,x) \Rightarrow \mathsf{self}(y,z) \qquad (Treeness) \qquad (1)$$
$$\mathsf{nextSibl}(y,x) \wedge \mathsf{nextSibl}(z,x) \Rightarrow \mathsf{self}(y,z) \qquad (Treeness) \qquad (2)$$
$$h(x,y) \Rightarrow \mathsf{child}(z,x) \wedge \mathsf{child}(z,y) \qquad (Siblings) \qquad (3)$$

and the equivalence ($\alpha \in \mathsf{V} \cup \mathsf{H}$)

$$\alpha^+(x,y) \equiv \alpha^*(x,z) \wedge \alpha(z,y) \qquad (Closure) \qquad (4).$$

Note that the variables x and y appearing on the left-side of $\Rightarrow$ or $\equiv$ are universally quantified, the other (z) is a fresh variable existentially quantified.

Rules (4.5) and (4.6). Let $v \in \{\mathsf{fstChild}, \mathsf{child}\}$.

$$v(x,y) \wedge \mathsf{par}(y,z) \equiv v(x,y) \wedge \mathsf{child}(z,y) \stackrel{1}{\equiv} v(x,y) \wedge v(z,y) \wedge \mathsf{self}(x,z) \equiv v(z,y) \wedge \mathsf{self}(x,z).$$

Rules (4.7) and (4.19). Let $f \in \{\mathsf{child}, \mathsf{nextSibl}\}$.

$$\begin{aligned} f^+(x,y) \wedge f^{-1}(y,z) &\stackrel{4}{\equiv} f^*(x,p) \wedge f(p,y) \wedge f(z,y) \\ &\stackrel{1,2}{\equiv} \mathsf{self}(p,z) \wedge f^*(x,z) \wedge f(z,y) \equiv f^*(x,y) \wedge f(z,y). \end{aligned}$$

Rules (4.8) and (4.9). Let $h \in \{\mathsf{nextSibl}, \mathsf{nextSibl}^+\}$.

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

Rules (4.10), (4.11), and (4.23).

Let $\alpha \in \{\mathsf{par}, \mathsf{prevSibl}\}$. We allow also $\mathsf{par}^{-1} = \mathsf{fstChild}$ (strictly, $\mathsf{par}^{-1} \supseteq \mathsf{fstChild}$).

$$\begin{aligned} & \alpha^{-1}(x,y) \wedge \alpha^+(y,z) \wedge \equiv (\alpha^{-1})^+(z,y) \wedge \alpha(y,x) \stackrel{+}{\equiv} (\alpha^{-1})^*(z,x) \wedge \alpha^{-1}(x,y) \\ & \stackrel{4}{\equiv} (\alpha^{-1})^+(z,x) \wedge \alpha^{-1}(x,y) \vee \mathsf{self}(z,x) \wedge \alpha^{-1}(x,y) \equiv \alpha^+(x,z) \wedge \alpha^{-1}(x,y) \vee \mathsf{self}(x,z) \wedge \alpha^{-1}(x,y). \end{aligned}$$

Equivalence (+) holds due to Rules (4.7) and (4.19).

Rules (4.12) and (4.24). Let $\alpha \in \mathsf{F}^{+}$.

$$\alpha(x, y) \wedge \alpha^{-1}(y, z).$$

Consider an LGQ substitution t consistent with the formula and a tree instance. The images of all variables are along a same path from the root and there is a partial order between them: $t(x) \ll t(y)$, $t(z) \ll t(y)$. The possibilities for the order between $t(x)$ and $t(z)$ are (1) $t(z) \ll t(x)$, (2) $t(x) = t(z)$, and (3) $t(x) \ll t(z)$. This reads also (1) $t(z)$ is an ancestor (preceding sibling) of $t(x)$, (2) $t(x)$ is the same as $t(z)$, (3) $t(x)$ is an ancestor (preceding sibling) of $t(z)$, thus $t(z)$ lies between $t(x)$ and $t(y)$. The LGQ encoding of all these possibilities is:

$$\alpha(x, y) \wedge \alpha^{-1}(y, z) \equiv \alpha(x, y) \wedge \alpha^{-1}(x, z) \vee \alpha(x, y) \wedge \mathsf{self}(x, z) \vee \alpha(x, z) \wedge \alpha(z, y).$$

Rules (4.13) and (4.14). Let $h \in \{\mathsf{nextSibl}, \mathsf{nextSibl}^{+}\}$.

$$\begin{aligned}
h(x, y) \wedge \mathsf{par}^{+}(y, z) &\overset{2}{\equiv} h(x, y) \wedge \mathsf{child}(p, x) \wedge \mathsf{child}(p, y) \wedge \mathsf{par}^{+}(y, z) \\
&\overset{4}{\equiv} h(x, y) \wedge \mathsf{child}(p, x) \wedge \mathsf{child}(p, y) \wedge \mathsf{par}^{*}(r, z) \wedge \mathsf{par}(y, r) \\
&\equiv h(x, y) \wedge \mathsf{child}(p, x) \wedge \mathsf{child}(p, y) \wedge \mathsf{par}^{*}(r, z) \wedge \mathsf{child}(r, y) \\
&\overset{1}{\equiv} \mathsf{self}(p, r) \wedge h(x, y) \wedge \mathsf{child}(r, x) \wedge \mathsf{child}(r, y) \wedge \mathsf{par}^{*}(r, z) \\
&\overset{4}{\equiv} \mathsf{self}(p, r) \wedge h(x, y) \wedge \mathsf{child}(r, y) \wedge \mathsf{par}^{+}(x, z) \\
&\equiv h(x, y) \wedge \mathsf{par}^{+}(x, z).
\end{aligned}$$

Rules (4.15) and (4.20) follow directly from the definition of $\mathsf{fstChild}(n, m)$: m is the first child of n, thus, for $h \in \{\mathsf{prevSibl}, \mathsf{prevSibl}^{+}\}$ holds

$$\mathsf{fstChild}(x, y) \wedge h(y, z) \equiv \perp.$$

Rules (4.16), (4.21). Let $h \in \{\mathsf{prevSibl}, \mathsf{prevSibl}^{+}\}$.

$$\begin{aligned}
\mathsf{child}(x, y) \wedge h(y, z) &\equiv \mathsf{child}(x, y) \wedge h^{-1}(z, y) \\
&\overset{3}{\equiv} \mathsf{child}(x, y) \wedge h^{-1}(z, y) \wedge \mathsf{child}(p, z) \wedge \mathsf{child}(p, y) \\
&\overset{1}{\equiv} \mathsf{self}(p, x) \wedge \mathsf{child}(x, z) \wedge h^{-1}(z, y) \wedge \mathsf{child}(x, y) \\
&\equiv \mathsf{child}(x, z) \wedge h^{-1}(z, y).
\end{aligned}$$

Rules (4.17) and (4.22). Let $h \in \{\mathsf{prevSibl}, \mathsf{prevSibl}^{+}\}$.

$$\begin{aligned}
\mathsf{child}^{+}(x, y) \wedge h(y, z) &\overset{4}{\equiv} \mathsf{child}^{*}(x, p) \wedge \mathsf{child}(p, y) \wedge h^{-1}(z, y) \\
&\overset{3}{\equiv} \mathsf{child}^{*}(x, p) \wedge \mathsf{child}(p, y) \wedge h^{-1}(z, y) \wedge \mathsf{child}(r, z) \wedge \mathsf{child}(r, y) \\
&\overset{1}{\equiv} \mathsf{self}(p, r) \wedge \mathsf{child}^{*}(x, r) \wedge \mathsf{child}(r, y) \wedge h^{-1}(z, y) \wedge \mathsf{child}(r, z) \\
&\overset{4}{\equiv} \mathsf{child}^{+}(x, z) \wedge h^{-1}(z, y).
\end{aligned}$$

Rule (4.18).

$$\mathsf{nextSibl}(x, y) \wedge \mathsf{prevSibl}(y, z) \equiv \mathsf{nextSibl}(x, y) \wedge \mathsf{nextSibl}(z, y) \overset{2}{\equiv} \mathsf{nextSibl}(z, y) \wedge \mathsf{self}(x, z).$$

Proof of Theorem 4.4.3

We prove here the local confluence property of the term rewriting systems TRS_i ($1 \leq i \leq 3$). We need to show that, given one term x that can rewrite (in one step) into y_1 and y_2 using different rewrite rules, y_1 and y_2 are joinable, i.e., they reduce to the same term after a finite rewriting sequence: $y_1 \leftarrow x \rightarrow y_2 \Rightarrow y_1 \downarrow y_2$.

TRS_1 consists of a single rule, namely Rule 4.1, and there is no critical pair created by this rule with itself or any of the AC-identities.

TRS_2. Recall that the rewrite rules of TRS_2 are defined by Lemmas 4.3.2, 4.3.3, 4.3.5, and 4.3.6, that define interactions between each forward and reverse formula.

First, we show that for LGQ general graphs, and even for the restricted version of single-join DAGs, TRS_2 is not locally confluent. Consider the single-join DAG formula x

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

We follow the two rewriting sequences

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

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

It is clear that the final contractions in both cases can not be rewritten anymore, and that they are different.

This concludes one half of the proof. We focus now on proving that TRS_2 is locally confluent for LGQ forests and its simpler derivates, i.e., trees and paths.

There are two cases regarding the interference of multiple redexes: either they do not interfere at all, or they have a common prefix. An example for the former case is: $\mathsf{child}(a,b) \land \mathsf{par}(b,c) \land \mathsf{child}(c,d) \land \mathsf{prevSibl}(d,e)$, where the first two binary atoms constitute a redex for Rule (4.6) and the last two binary atoms constitute a redex for Rule (4.16). It is clear that in this case the contracting order does not matter and the same rewrite is produced. In the latter case, there are critical pairs. More precisely, the lhs of each rule of Lemma 4.3.3 unifies with a subterm of the lhs of the built-in A-identity for $\land$. We treat next this case in detail.

We consider the LGQ formula $\alpha_1(a,b) \land \alpha_2(b,c) \land \alpha_3(b,d)$, where α_1 is a forward predicate and α_2 and α_3 are reverse predicates. Note that this term corresponds to a LGQ tree formula. Terms like $\alpha_1(a,b) \land \alpha_2(c,b) \land \alpha_3(b,d)$ can not appear because such terms are not anymore tree formulas, but DAGs.

The following combinations are to be considered (note that $\land$ is commutative and therefore the symmetrical cases for α_2 and α_3 are not necessary):

Case	$(\alpha_1$	,	α_2	,	α_3)
1.	$(\mathsf{HF}^?$	,	$\mathsf{VR}^?$	,	$\mathsf{VR}^?)$
2.	$(\mathsf{HF}^?$	,	$\mathsf{HR}^?$	,	$\mathsf{VR}^?)$
3.	$(\mathsf{VF}^?$	,	$\mathsf{HR}^?$	,	$\mathsf{HR}^?)$
4.	$(\mathsf{VF}^?$	,	$\mathsf{HR}^?$	,	$\mathsf{VR}^?)$

To follow up the rewritings easier, we may rename specifically to each case the relations 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 relations.

Case 1 is similar to the previous case, because it uses the interaction type $(\mathsf{HF}, \mathsf{VR})^?$, which behaves identical to $(\{\mathsf{self}\},\mathsf{R}^?)$ (see Figure 4.5).

Case 2. Only interactions of type $(\mathsf{HF},\mathsf{VR}^?)$ (branch I) or of type $\mathsf{H}(\mathsf{F}, \mathsf{R})^?$ (branch II) are considered first. We rename the relations accordingly: $hf = \alpha_1$, $hr = \alpha_2$, $vr = \alpha_3$.

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

Now we can consider for both branches only the interaction $\mathsf{H}(\mathsf{F},\mathsf{R})^?$. After several rewrite steps, the term $hf(a,b) \wedge hr(b,c)$ is contracted to a term t containing only horizontal formulas and no further contraction can be performed on this term. Proposition 4.3.2 ensures that the connections of the non-sink variable a are preserved, in particular $a \rightsquigarrow_t b$. Only interactions $(\mathsf{HF},\mathsf{VR}^?)$ can be conducted now, and they push each variable that is connected to b as the first variable of vr, particularly also a. Because the rule applications preserve a as non-sink, this variable can not be replaced and the $vr(a,d)$ is obtained.

Case 3. Only interactions of type $(\mathsf{VF},\mathsf{HR})^?$ can be considered first. We rename the relations accordingly: $vf = \alpha_1$, $hr_1 = \alpha_2$, $hr_2 = \alpha_3$. We consider first that $vf = \mathsf{fstChild}$.

$$\text{I.} \quad \underline{vf(a,b) \wedge hr_1(b,c)} \wedge hr_2(b,d) \rightarrow \bot \wedge hr_2(b,d) \rightarrow \bot.$$
$$\text{II.} \quad \underline{vf(a,b)} \wedge hr_1(b,c) \wedge \underline{hr_2(b,d)} \rightarrow \bot \wedge hr_1(b,c) \rightarrow \bot.$$

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

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

Subcase 1: $hr_1 = hr_2 = \mathsf{prevSibl}$. Then, $hr_1^{-1} = hr_2^{-1} = \mathsf{nextSibl}$.

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

Both contractions are identical up to the variable equality $c = d$ that is ensured by the the rewriting modulo equational theory including $\mathsf{self}(v_1, v_2) \wedge \alpha(v_2, v_3) \approx \mathsf{self}(v_1, v_2) \wedge \alpha(v_1, v_3)$.

Subcase 2: $hr_1 = \mathsf{prevSibl}^+$, $hr_2 = \mathsf{prevSibl}$. Then, $hr_1^{-1} = \mathsf{nextSibl}^+$, $hr_2^{-1} = \mathsf{nextSibl}$.

$$\begin{aligned}
\text{I.}\quad & vf(a,c) \wedge \underline{\mathsf{nextSibl}^+(c,b) \wedge \mathsf{prevSibl}(b,d)} \\
\rightarrow\ & vf(a,c) \wedge \underline{\mathsf{nextSibl}^*(c,d)} \wedge \mathsf{nextSibl}(d,b) \\
\rightarrow\ & vf(a,c) \ \underline{\wedge}\ (\mathsf{nextSibl}^+(c,d) \ \underline{\vee}\ \mathsf{self}(c,d)) \wedge \mathsf{nextSibl}(d,b) \\
\rightarrow\ & vf(a,c) \wedge \mathsf{nextSibl}^+(c,d) \wedge \mathsf{nextSibl}(d,b) \vee vf(a,c) \wedge \mathsf{self}(c,d) \wedge \mathsf{nextSibl}(d,b). \\
\text{II.}\quad & vf(a,d) \wedge \underline{\mathsf{prevSibl}^+(b,c) \wedge \mathsf{nextSibl}(d,b)} \\
\rightarrow\ & vf(a,d) \ \underline{\wedge}\ (\mathsf{prevSibl}^+(d,c) \wedge \mathsf{nextSibl}(d,b) \ \underline{\vee}\ \mathsf{self}(d,c) \wedge \mathsf{nextSibl}(d,b)) \\
\rightarrow\ & \underline{vf(a,d) \wedge \mathsf{prevSibl}^+(d,c)} \wedge \mathsf{nextSibl}(d,b) \vee vf(a,d) \wedge \mathsf{self}(d,c) \wedge \mathsf{nextSibl}(d,b) \\
\rightarrow\ & vf(a,c) \wedge \mathsf{nextSibl}^+(c,d) \wedge \mathsf{nextSibl}(d,b) \vee vf(a,d) \wedge \mathsf{self}(d,c) \wedge \mathsf{nextSibl}(d,b).
\end{aligned}$$

Both contractions are identical up to the variable equality $c = d$ in the second conjunct. Subcase 3: $hr_1 = \mathsf{prevSibl}$, $hr_2 = \mathsf{prevSibl}^+$. Then, $hr_1^{-1} = \mathsf{nextSibl}$, $hr_2^{-1} = \mathsf{nextSibl}^+$.

$$\begin{aligned}
\text{I.}\quad & vf(a,c) \wedge \underline{\mathsf{nextSibl}(c,b) \wedge \mathsf{prevSibl}^+(b,d)} \\
\rightarrow\ & vf(a,c) \ \underline{\wedge}\ (\mathsf{nextSibl}(c,b) \wedge \mathsf{prevSibl}^+(c,d) \wedge \ \ \underline{\vee}\ \mathsf{nextSibl}(c,b) \wedge \mathsf{self}(c,d)) \\
\rightarrow\ & \underline{vf(a,c)} \wedge \mathsf{nextSibl}(c,b) \wedge \underline{\mathsf{prevSibl}^+(c,d)} \vee vf(a,c) \wedge \mathsf{nextSibl}(c,b) \wedge \mathsf{self}(c,d) \\
\rightarrow\ & vf(a,d) \wedge \mathsf{nextSibl}(c,b) \wedge \mathsf{nextSibl}^+(d,c) \vee vf(a,c) \wedge \mathsf{nextSibl}(c,b) \wedge \mathsf{self}(c,d). \\
\text{II.}\quad & vf(a,d) \wedge \underline{\mathsf{prevSibl}(b,c) \wedge \mathsf{nextSibl}^+(d,b)} \\
\rightarrow\ & vf(a,d) \wedge \mathsf{nextSibl}(c,b) \wedge \mathsf{nextSibl}^*(d,c) \\
\rightarrow\ & vf(a,d) \wedge \mathsf{nextSibl}(c,b) \ \underline{\wedge}\ (\mathsf{nextSibl}^+(d,c) \ \underline{\vee}\ \mathsf{self}(d,c)) \\
\rightarrow\ & vf(a,d) \wedge \mathsf{nextSibl}(c,b) \wedge \mathsf{nextSibl}^+(d,c) \vee vf(a,d) \wedge \mathsf{nextSibl}(c,b) \wedge \mathsf{self}(d,c).
\end{aligned}$$

Both contractions are identical up to the variable equality $c = d$ in the second conjunct. Subcase 4: $hr_1 = hr_2 = \mathsf{prevSibl}^+$. Then, $hr_1^{-1} = hr_2^{-1} = \mathsf{nextSibl}^+$.

$$\begin{aligned}
\text{I.}\quad & vf(a,c) \wedge \underline{\mathsf{nextSibl}^+(c,b) \wedge \mathsf{prevSibl}^+(b,d)} \\
\rightarrow\ & vf(a,c) \ \underline{\wedge}\ (\mathsf{nextSibl}^+(c,b) \wedge \mathsf{prevSibl}^+(c,d) \ \underline{\vee}\ \mathsf{nextSibl}^*(c,d) \wedge \mathsf{nextSibl}^+(d,b)) \\
\rightarrow\ & \underline{vf(a,c)} \wedge \mathsf{nextSibl}^+(c,b) \wedge \underline{\mathsf{prevSibl}^+(c,d)} \vee vf(a,c) \wedge \mathsf{nextSibl}^*(c,d) \wedge \mathsf{nextSibl}^+(d,b) \\
\rightarrow\ & vf(a,d) \wedge \mathsf{nextSibl}^+(c,b) \wedge \mathsf{nextSibl}^+(d,c) \vee vf(a,c) \wedge \mathsf{nextSibl}^*(c,d) \wedge \mathsf{nextSibl}^+(d,b). \\
\text{II.}\quad & vf(a,d) \wedge \underline{\mathsf{prevSibl}^+(b,c) \wedge \mathsf{nextSibl}^+(d,b)} \\
\rightarrow\ & vf(a,d) \ \underline{\wedge}\ (\mathsf{prevSibl}^+(d,c) \wedge \mathsf{nextSibl}^+(d,b) \ \underline{\vee}\ \mathsf{nextSibl}^*(d,c) \wedge \mathsf{nextSibl}^+(d,b)) \\
\rightarrow\ & \underline{vf(a,d) \wedge \mathsf{prevSibl}^+(d,c)} \wedge \mathsf{nextSibl}^+(d,b) \vee vf(a,d) \wedge \mathsf{nextSibl}^*(d,c) \wedge \mathsf{nextSibl}(d,b) \\
\rightarrow\ & vf(a,c) \wedge \mathsf{nextSibl}^+(c,d) \wedge \mathsf{nextSibl}^+(d,b) \vee vf(a,d) \wedge \mathsf{nextSibl}^*(d,c) \wedge \mathsf{nextSibl}(d,b).
\end{aligned}$$

Both contractions are identical up to the variable equality $c = d$ in both conjuncts.

Case 4. Only interactions of type $(\mathsf{VF},\mathsf{HR})^?$ (branch I) or special cases of $\mathsf{V}(\mathsf{F},\mathsf{R})^?$

(branch II) are considered first. We rename the relations $vf = \alpha_1$, $hr = \alpha_2$, $vr = \alpha_3$.

$$\text{I.} \quad \underline{vf(a,b) \wedge hr(b,c)} \wedge vr(b,d) \rightarrow vf(a,c) \wedge \underline{hr^{-1}(c,b) \wedge vr(b,d)}$$
$$\rightarrow \underline{vf(a,c)} \wedge hr{-}1(c,b) \wedge \underline{vr(c,d)}.$$

For both branches, only special cases of interaction type $\mathsf{V(F,R)}^?$ can be applied further. For branch I, we consider as term of interest only the subterm $vf(a,c) \wedge vr(c,d)$ and for branch II we consider the subterm $vf(a,b) \wedge vr(b,d)$ (we leave the forward formula $hr^{-1}(c,b)$ out of discussion for a while, because it can not interact now with the other two for both branches). It should be clear that (1) both branches get the same contraction up to replacing c (from the contraction of branch I) with b (to get the contraction of branch II), and (2) both contractions have only forward formulas. For branch II, this means that only interactions of type $\mathsf{(VF, HR)}^?$ can apply. Proposition 4.3.2 ensures that the connections of the non-sink variable a are preserved, in particular $a \leadsto_t b$. The interaction $\mathsf{(VF,HR)}^?$ ensures that $a \leadsto_t b, hr(b,c) \Rightarrow a \leadsto_t c, hr^{-1}(c,b)$, b is replaced by c in the contraction for branch II, and $hr^{-1}(c,b)$ is added also.

TRS$_3$. TRS$_2$ is not confluent for input LGQ graphs. The local confluence could be obtained, however, by adding a rule for rewriting disjuncts of two forward atoms having the same sink into disjuncts of one of these forward atoms and the second forward atom replaced by its corresponding reverse one. TRS$_2$ improves on TRS$_1$ exactly in this point by adding Rule (4.25).

Because TRS$_3$ includes TRS$_2$, all interference cases of multiple redexes that appear in TRS$_2$ can appear also here, but they do not raise confluence problems, as shown already for TRS$_2$. The new interference cases that can appear are

A. $\alpha_1(a,b) \wedge \alpha_2(c,b) \wedge \alpha_3(d,b)$ with $\alpha_1, \alpha_2, \alpha_3 \in \mathsf{F}^?$

B. $\alpha_1(a,b) \wedge \alpha_2(c,b) \wedge \alpha_3(b,d)$ with $\alpha_1, \alpha_2 \in \mathsf{F}^?, \alpha_3 \in \mathsf{R}^?$.

For case A, there are three possible distinct contractions, as underlined below

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).

Note that the terms (I) to (III) are joinable between each other (the arabic numbers on the right represent identical contractions). The case B can be shown similarly.

Bibliography

[1] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik, "Aurora: a new model and architecture for data stream management," *VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[2] Serge Abiteboul, "Querying semistructured data," in *Proc. of Int. Conf. on Database Theory (ICDT)*, 1997, pp. 1–18.

[3] Serge Abiteboul, Peter Buneman, and Dan Suciu, *Data on the Web*, Morgan Kaufmann, 2000.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu, *Foundations of Databases*, Addison Wesley, 1995.

[5] Serge Abiteboul, Dallas Quass, John McHugh, Jenifer Widom, and Janet Wiener, "The Lorel query language for semistructured data," *International Journal on Digital Libraries*, vol. 1, no. 1, pp. 68–88, 1997.

[6] David K. Gifford Alex C. Snoeren, Kenneth Conley, "Mesh-based content routing using XML," in *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2001, pp. 160–173.

[7] Mehmet Altinel and Michael J. Franklin, "Efficient filtering of XML documents for selective dissemination of information," in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2000, pp. 53–64.

[8] Rajeev Alur and P. Madhusudan, "Visibly pushdown languages," in *Proc. of Annual ACM Symposium on Theory of Computing (STOC)*, 2004, pp. 202–211.

[9] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava, "Tree pattern query minimization," *VLDB Journal*, vol. 11, no. 4, pp. 315–331, 2002.

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

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

[12] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAllister, and Jenifer Widom, "Characterizing memory requirements for queries over continuous data streams," in *Proc. of ACM SIGMOD/SIGART Symposium on Principles of Database Systems (PODS)*, 2002, pp. 221–232.

[13] Andrei Arion, Angela Bonifati, Gianni Costa, Ioana Manolescu Sandra D'Aguanno, and Andrea Pugliese, "Efficient query evaluation over compressed XML data," in *Proc. of Int. Conf. on Extending Database Technology (EDBT)*, 2004, pp. 200–218.

[14] Iliana Avila-Campillo, Ashish Gupta, Makoto Onizuka, Demian Raven, and Dan Suciu, "XMLTK: An XML toolkit for scalable XML stream processing," in *Proc. of Int. Workshop on Programming Language Technologies for XML (PLAN-X)*, 2002.

[15] Ron Avnur and Joseph M. Hellerstein, "Eddies: Continuously adaptive query processing," in *Proc. of ACM SIGMOD*. 2000, pp. 261–272, ACM Press.

[16] Franz Baader and Tobias Nipkow, *Term Rewriting and All That*, Cambridge University Press, 1998.

[17] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jenifer Widom, "Models and issues in data stream systems," in *Proc. of ACM SIGMOD/SIGART Symposium on Principles of Database Systems (PODS)*, 2002, pp. 1–16.

[18] Shivnath Babu and Jenifer Widom, "Continuous queries over data streams," *Proc. of ACM SIGMOD*, pp. 109–120, 2001.

[19] Ziv Bar-Youssef, Marcus Fontoura, and Vanja Josifovski, "On the memory requirements of XPath evaluation over XML streams," in *Proc. of ACM SIGMOD/SIGART Symposium on Principles of Database Systems (PODS)*, 2004, pp. 177–188.

[20] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura, and Vanja Josifovski, "An algorithm for streaming XPath processing with forward and backward axes," in *Proc. of Int. Workshop on Programming Language Technologies for XML (PLAN-X)*, 2002.

[21] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura, and Vanja Josifovski, "Streaming XPath processing with forward and backward axes," in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2003, pp. 455–466.

[22] Anders Berlund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon, "XML path language (XPath) 2.0," W3C working draft, 2002.

[23] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon, "XQuery 1.0: An XML query language," Working draft, World Wide Web Consortium, 2002.

[24] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler, "Extensible markup language (XML) 1.0," W3C Recommendation, World Wide Web Consortium, 1998, `http://www.w3.org/TR/REC-xml`.

[25] François Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, and Markus Spannagel:, "The XML stream query processor SPEX," in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2005, to appear.

[26] François Bry, Tim Furche, and Dan Olteanu, "Aktuelles schlagwort: Datenströme," *Informatik Spektrum*, vol. 27, no. 2, pp. 168–171, 2004.

[27] François Bry, Michael Kraus, Dan Olteanu, and Sebastian Schaffert, "Aktuelles schlagwort: Semistrukturierte daten," *Informatik Spektrum*, vol. 24, no. 4, pp. 230–233, 2001.

[28] François Bry and Peer Kröger, "A computational biology database digest: Data, data analysis, and data management," *Distributed and Parallel Databases*, vol. 13, no. 1, pp. 7–42, 2003.

[29] Ahmet Bulut and Ambuj Singh, "SWAT: Hierarchical stream summarization in large networks," in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2003, pp. 303–314.

[30] Peter Buneman, Martin Grohe, and Christoph Koch, "Path queries on compressed XML," in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2003, pp. 141–152.

[31] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi, "Rewriting of regular expressions and regular path queries," in *Proc. of ACM SIGMOD/SIGART Symposium on Principles of Database Systems (PODS)*, 1999, pp. 194–204.

[32] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi, "Answering regular path queries using views," in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2000, pp. 389–398.

[33] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi, "Containment of conjunctive regular path queries with inverse," in *Proc. of Knowledge Representation (KR)*, 2000, pp. 176–185.

[34] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi, "View-based query processing for regular path queries with inverse," in *Proc. of ACM SIGMOD/SIGART Symposium on Principles of Database Systems (PODS)*, 2000, pp. 58–66.

[35] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi, "What is view-based query rewriting? (position paper)," in *Proc. of Int. Workshop on Knowledge Representation meets Databases (KRDB)*, 2000, pp. 17–27.

[36] Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik, "Monitoring streams: A new class of data management applications," in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2002, pp. 215–226.

[37] Chee-Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi, "Efficient filtering of XML documents with XPath expressions," in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2002, pp. 235–244.

[38] A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational databases," in *Proc. of Annual ACM Symposium on Theory of Computing (TOC)*, 1977, pp. 77 – 90.

[39] Sirish Chandrasekaran and Michael J. Franklin, "Streaming queries over streaming data," in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2002, pp. 203–214.

[40] Jianjun Chen, David J. DeWitt, , and Jeffrey F. Naughton, "Design and evaluation of alternative selection placement strategies in optimizing continuous queries," in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2002, pp. 345–356.

[41] Shu-Yao Chien, *Managing and querying multiversion XML documents*, Ph.D. thesis, University of California, Los Angeles, 2001.

[42] Christian Choffrut and Karel Culik II, "Properties of finite and pushdown transducers," *SIAM Journal of Computing*, vol. 12, no. 2, pp. 300–315, 1983.

[43] Byron Choi, "DTD Inquisitor 2," Tech. Rep., Univ. of Pennsylvania, `http://db.cis.upenn.edu/~kkchoi/DTDI2/`, 2001.

[44] Byron Choi, "What are real DTDs like," in *Proc. of Int. Workshop on the Web and Databases (WebDB)*, 2001, pp. 43–48.

[45] James Clark, "XSL transformations (XSLT) version 1.0," W3C Recommendation, World Wide Web Consortium, 1999.

[46] James Clark and Steve DeRose, "XML path language (XPath) version 1.0," W3C Recommendation, World Wide Web Consortium, 1999.

[47] James Clark and William D. Lindsey, *XT: An XSLT Engine*, 2002, `http://www.blnz.com/xt/index.html`.

[48] James Clark and Makoto Murata, "Relax NG," Tech. Rep., OASIS Committee Specification, 2001, `http://www.relaxng.org/`.

[49] R. Cole, R. Hanharan, and P. Indyk, "Tree pattern matching and subset matching in deterministic O($nlog^3n$)-time," in *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1999, pp. 245–254.

[50] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, "Tree automata techniques and applications," `http://www.grappa.univ-lille3.fr/tata`, 1997, release October, 1st 2002.

[51] Andy Cooke, Alasdair J. G. Gray, and Wernet Nutt, "Data integration techniques in grid monitoring," Tech. Rep. HW-MACS-TR-0019, Herriot-Watt University, 2004.

[52] Corina Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith, "Hancock: A language for extracting signatures from data streams," in *Proc. of Int. Conf. on Knowledge Discovery and Data Mining*, 2000, pp. 9–17.

[53] John Cowan and Richard Tobin, "XML Information Set (second edition)," Working draft, World Wide Web Consortium, 2004.

[54] Steven DeRose, Ron Daniel Jr., Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh, "XML pointer language (XPointer)," W3C Recommendation, World Wide Web Consortium, 2002, `http://www.w3.org/TR/xptr/`.

[55] Arpan Desai, "Introduction to Sequential XPath," in *Proc. IDEAlliance XML Conference*, 2001.

[56] Alin Deutsch and Val Tannen, "Containment for classes of XPath expressions under integrity constraints," in *Proc. of Int. Workshop on Knowledge Representation meets Databases (KRDB)*, 2001.

[57] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter M. Fischer, "Path sharing and predicate evaluation for high-performance XML filtering," *ACM Transactions on Database Systems (TODS)*, vol. 28, no. 4, pp. 467–516, 2003.

[58] Yanlei Diao, Peter Fischer, Michael J. Franklin, and Raymond To, "YFilter: Efficient and scalable filtering of XML documents," in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2002, pp. 341–342.

[59] David C. Fallside and Priscilla Walmsley, "XML-Schema," W3C Recommendation, World Wide Web Consortium, 2001, `http://www.w3.org/XML/Schema`.

[60] C. Fellbaum, Ed., *WordNet – An Electronic Lexical Database*, MIT Press, 1998, `http://www.cogsci.princeton.edu/~wn/`.

[61] Mary F. Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur, "Implementing XQuery 1.0: The Galax experience," in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2003, pp. 1077–1080.

[62] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh, "XQuery 1.0 and XPath 2.0 data model," Working draft, World Wide Web Consortium, 2004.

[63] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson, “Scalable Vector Graphics (SVG) 1.1 Specification,” W3C Recommendation, World Wide Web Consortium, 2003, `http://www.w3.org/TR/SVG/`.

[64] Sergio Flesca, Filippo Furfaro, and Elio Masciari, “On the minimization of XPath queries,” in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, pp. 153–164.

[65] Daniela Florescu, Alon Levy, and Dan Suciu, “Query containment for conjunctive queries with regular expressions,” in *Proc. of ACM SIGMOD/SIGART Symposium on Principles of Database Systems (PODS)*, 1998, pp. 139–148.

[66] Markus Frick, Martin Grohe, and Christoph Koch, “Query evaluation on compressed trees,” in *Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2003, pp. 188–197.

[67] Tim Furche, “Optimizing multiple queries against XML streams,” Diploma thesis, Univ. of Munich, 2003.

[68] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds, *Temporal Logic*, Oxford University Press, 1994.

[69] Georg Gottlob and Christoph Koch, “Monadic queries over tree-structured data,” in *Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2002, pp. 189–202.

[70] Georg Gottlob, Christoph Koch, and Reinhard Pichler, “Efficient algorithms for processing XPath queries,” in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2002, pp. 95–106.

[71] Georg Gottlob, Christoph Koch, and Reinhard Pichler, “The complexity of XPath query evaluation,” in *Proc. of ACM SIGMOD/SIGART Symposium on Principles of Database Systems (PODS)*, 2003, pp. 179–190.

[72] Georg Gottlob, Christoph Koch, and Reinhard Pichler, “XPath processing in a nutshell,” *SIGMOD Record*, vol. 32, no. 1, pp. 12–19, 2003.

[73] Georg Gottlob, Christoph Koch, and Reinhard Pichler, “XPath query evaluation: Improving time and space efficiency,” in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2003, pp. 379–390.

[74] Georg Gottlob, Christoph Koch, and Klaus Schulz, “Conjunctive queries over trees,” in *Proc. of ACM SIGMOD/SIGART Symposium on Principles of Database Systems (PODS)*, 2004, pp. 189–200.

[75] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu, “Processing XML streams with deterministic automata and stream indexes,” *ACM Transactions on Database Systems (TODS)*, vol. 29, no. 4, 2004.

[76] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu, "Processing XML streams with deterministic automata," in *Proc. of Int. Conf. on Database Theory (ICDT)*, 2003, pp. 173–189.

[77] The STREAM Group, "STREAM: The Stanford Stream Data Manager," in *IEEE Data Engineering Bulletin*, `http://www-db.stanford.edu/stream/`, 2003, vol. 26.

[78] Torsten Grust, "Accelerating XPath location steps," in *Proc. of ACM SIGMOD*, 2002, pp. 109–120.

[79] Torsten Grust, Maurice van Keulen, and Jens Teubner, "Staircase join: Teach a relational DBMS to watch its (axis) steps," in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2003, pp. 524–535.

[80] Torsten Grust, Maurice van Keulen, and Jens Teubner, "Accelerating XPath evaluation in any RDBMS," *ACM Transactions on Database Systems (TODS)*, , no. 29, pp. 91–131, 2004.

[81] Ashish Kumar Gupta and Dan Suciu, "Stream processing of XPath queries with predicates," in *Proc. of ACM SIGMOD*, 2003, pp. 419–430.

[82] E. Gurari, *An Introduction to the Theory of Computation*, Computer Science Press, 1989.

[83] Alon Y. Halevy, "Answering queries using views: A survey," *VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.

[84] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte, "Optimized translation of XPath into algebraic expressions parameterized by programs containing navigational primitives," in *Proc. of Int. Conf. on Web Information Systems Engineering (WISE)*, 2002, pp. 215–224.

[85] Jan Hidders and Philippe Michiels, "Avoiding unnecessary ordering operations in XPath," in *Proc. of Int. Conf. on Data Base Programming Languages (DBPL)*, 2003, pp. 54–70.

[86] Jan Hidders and Philippe Michiels, "Efficient XPath axis evaluation for DOM data structures," in *Proc. of Int. Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004.

[87] C. M. Hoffmann and M. J. O'Donnell, "Pattern matching in trees," *Journal of ACM*, vol. 29, no. 1, pp. 68–95, 1982.

[88] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory. Languages, and Computation*, Addison Wesley, 1979.

[89] Haruo Hosoya and Benjamin C. Pierce, “Regular expression pattern matching,” in *Proc. of Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2001, pp. 67–80.

[90] Haruo Hosoya and Benjamin C. Pierce, “Xduce: A statically typed XML processing language,” *ACM Transactions on Internet Technology (TOIT)*, vol. 3, no. 2, pp. 117–148, 2003.

[91] I. Sosnoski Software Solutions, `http://www.sosnoski.com/opensrc/xmlbench`, *Java XML Models Benchmarks*, 2004.

[92] Nancy Ide, Patrice Bonhomme, and Laurent Romary, “XCES: An XML-based standard for linguistic corpora,” in *Proc. Annual Conf. on Language Resources and Evaluation (LREC)*, 2000, pp. 825–830.

[93] International Standard Organization (ISO), *Traffic and Traveller Information (TTI) – TTI messages via traffic message coding – Part 1: Coding protocol for Radio Data System – Traffic Message Channel (RDS-TMC)*, 2003, `http://www.iso.org`.

[94] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld, “An XML query engine for network-bound data,” *VLDB Journal*, vol. 11, no. 4, pp. 380–402, 2002.

[95] Zachary G. Ives, Alon Y. Levy, Daniel S. Weld, Daniela Florescu, and Marc Friedman, “Adaptive query processing for internet applications,” *IEEE Data Engineering Bulletin*, vol. 23, no. 2, pp. 19–26, 2000.

[96] Michael Kay, “XSL Transformations (XSLT) Version 2.0,” Working draft, World Wide Web Consortium, 2004.

[97] P. Kilpelainen and H. Mannila, “Ordered and unordered tree inclusion,” *SIAM Journal of Computing*, vol. 24, no. 2, pp. 340–356, 1995.

[98] Christoph Koch and Stefanie Scherzinger, “Attribute grammars for scalable query processing on XML streams,” in *Proc. of Int. Conf. on Data Base Programming Languages (DBPL)*, 2003, pp. 233–256.

[99] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier, “FluXQuery: An optimizing XQuery processor for streaming XML data,” in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2004, pp. 1309–1312, Demonstration.

[100] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier, “Schema-based scheduling of event processors and buffer minimization for queries on structured data streams,” in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2004, pp. 228–239.

[101] Nick Koudas and Divesh Srivastava, “Data stream query processing: A tutorial,” in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2003, p. 1149.

[102] Dongwon Lee, Murali Mani, and Makoto Murata, "Reasoning about XML schema languages using formal language theory," Tech. Rep. RJ 10197 Log 95071, IBM Reseach, 2000.

[103] Patrick Lincoln and Jim Christian, "Adventures in associative-commutative unification," *Journal of Symbolic Computation*, vol. 8, no. 1–2, pp. 217–240, 1989.

[104] Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou, "A transducer-based XML query processor," in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2002, pp. 227–238.

[105] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman, "Continuously adaptive continuous queries over streams," in *Proc. of ACM SIGMOD*, 2002, pp. 49–60.

[106] Amélie Marian and Jérôme Siméon, "Projecting XML documents," in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2003, pp. 213–224.

[107] José M. Martínez, "MPEG-7 overview," Tech. Rep. N4980, ISO/IEC JTC1/SC29/WG11, 2002, `http://mpeg.telecomitalialab.com/standards/mpeg-7/mpeg-7.htm`.

[108] Maarten Marx, "XCPath, the first order complete XPath dialect," in *Proc. of ACM SIGMOD/SIGART Symposium on Principles of Database Systems (PODS)*, 2004, pp. 13–22.

[109] Maarten Marx, "XPath with conditional axis relations," in *Proc. of Int. Conf. on Extending Database Technology (EDBT)*, 2004, pp. 477–494.

[110] David Megginson, *SAX: The Simple API for XML*, 1998, `http://www.saxproject.org/`.

[111] Holger Meuss, Andreas Wicenec, and S. Farrow, "Flexible storage of astronomical data in the ALMA archive," in *ASP Conf. Ser. 314: Astronomical Data Analysis Software and Systems (ADASS)*, 2004, pp. 97–+, `http://www.eso.org`.

[112] Philippe Michiels, "XQuery optimization," in *VLDB PhD Workshop*, 2003.

[113] Microsoft Corporation, *Internet Explorer 6.0*, 2002, `http://www.microsoft.com/windows/ie/worldwide/all.mspx`.

[114] Gerome Miklau, *XMLData Repository*, Univ. of Washington, 2003, `http://www.cs.washington.edu/research/xmldatasets`.

[115] Gerome Miklau and Dan Suciu, "Containment and equivalence of an XPath fragment," in *Proc. of ACM SIGMOD/SIGART Symposium on Principles of Database Systems (PODS)*, 2002, pp. 65–76.

[116] Gerome Miklau and Dan Suciu, "Containment and equivalence of a fragment of XPath," *Journal of the ACM*, vol. 51, no. 1, pp. 2–45, 2004.

[117] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma, "Query processing, approximation, and resource management in a data stream management system," in *Proc. of CIDR*, 2003.

[118] NASA, *JPL Sensor Webs Project*, `http://sensorwebs.jpl.nasa.gov`, 2004.

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

[120] Netscape, *DMOZ: Open Directory Project*, `http://dmoz.org`, 2005.

[121] Frank Neven and Thomas Schwentick, "XPath containment in the presence of disjunction, DTDs, and variables," in *Proc. of Int. Conf. on Database Theory (ICDT)*, 2003, pp. 315 – 329.

[122] M. H. A. Newman, "On theories with a combinatorial definition of 'equivalence'," *Annals of Mathematics*, vol. 43, no. 2, pp. 223 – 243, 1942.

[123] Dan Olteanu, "Answering queries using views in agora," M.S. thesis, "Politehnica" University of Bucharest, 2000.

[124] Dan Olteanu, Tim Furche, and François Bry, "An efficient single-pass query evaluator for XML data streams," in *Proc. of Annual ACM Symposium on Applied Computing (SAC)*, 2004, pp. 627–631.

[125] Dan Olteanu, Tim Furche, and François Bry, "Evaluating complex queries against XML streams with polynomial combined complexity," in *Proc. of Annual British National Conference on Databases (BNCOD)*, 2004, pp. 31–44.

[126] Dan Olteanu, Tobias Kiesling, and François Bry, "An evaluation of regular path expressions with qualifiers against XML streams," Tech. Rep. PMS-FB-2002-12, Univ. of Munich, Institute of Computer Science, 2002.

[127] Dan Olteanu, Tobias Kiesling, and François Bry, "An evaluation of regular path expressions with qualifiers against XML streams," in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2003, pp. 702–704.

[128] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry, "XPath: Looking forward," in *Proc. of EDBT Workshop XMLDM*, 2002, pp. 109–127, LNCS 2490.

[129] Makoto Onizuka, "Light-weight xPath processing of XML stream with deterministic automata," in *Proc. of the Int. Conf. on Information and Knowledge Management (CIKM)*, 2003, pp. 342–349.

[130] Yannis Papakonstantinou and Vasilis Vassalos, "Query rewriting for semistructured data," in *Proc. of ACM SIGMOD*, 1999, pp. 455–466.

[131] Feng Peng and Sudarshan S. Chawathe, "XSQ: A Streaming XPath Engine," Tech. Rep. CS-TR-4493 (UMIACS-TR-2003-62), University of Maryland, 2003.

[132] Feng Peng and Sudarshan S. Chawathe, "XSQ: Streaming XPath Queries: A Demonstration," in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2003, pp. 780–782.

[133] Corin Pitcher, "Visibly pushdown expression effects for XML stream processing," in *Proc. of Int. Workshop on Programming Language Technologies for XML (PLAN-X)*, 2005, to appear.

[134] Dave Raggett, Arnaud Le Hors, and Ian Jacobs, "Hypertext markup language (HTML) 4.01 specification," W3C Recommendation, World Wide Web Consortium, 1999, `http://www.w3.org/TR/REC-html40/`.

[135] Prakash Ramanan, "Efficient algorithms for minimizing tree pattern queries," in *Proc. of ACM SIGMOD*, 2002, pp. 299–309.

[136] Derek Rogers, Jane Hunter, and Douglas Kosovic, "The TV-trawler project," *Journal of Imaging Systems and Technology*, pp. 289–296, 2003.

[137] Sebastian Schaffert, *Xcerpt: A Query and Transformation Language for the Web*, Ph.D. thesis, University of Munich, 2004.

[138] Steffen Schott and Markus L. Noga, "Lazy XSL transformations," in *Proc. of ACM Symposium on Document Engineering*, 2003, pp. 9–18.

[139] Dominik Schwald, "Approximate streamed evaluation of XPath under memory constraints," Project thesis, Univ. of Munich, 2003.

[140] Luc Segoufin, "Typing and querying XML documents: some complexity bounds," in *Proc. of ACM SIGMOD/SIGART Symposium on Principles of Database Systems (PODS)*, 2003, pp. 167–178.

[141] P. Seshadri, M. Livny, and R. Ramakrishnan, "The design and implementation of a sequence database system," in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 1996, pp. 99–110.

[142] R. Snodgrass and I. Ahn, "A taxonomy of time in databases," in *Proc. of ACM SIGMOD*, 1985, pp. 236–245.

[143] Anthony Vetro, "MPEG-7 applications," Tech. Rep. N3934, ISO/IEC JTC1/SC29/WG11, 2001.

[144] Jean-Yves Vion-Dury and Nabil Layaida, "Containment of XPath expressions: an inference and rewriting based approach," in *Proc. of Extreme Markup Languages*, 2003.

[145] Ray Whitmer, "Document Object Model (DOM) Level 3 XPath Specification," W3C Recommendation, World Wide Web Consortium, 2000.

[146] Peter T. Wood, "Optimising web queries using document type definitions," in *Proc. 2nd ACM Workshop on Web Information and Data Management (WIDM)*, 1999, pp. 28–32.

[147] Peter T. Wood, "On the equivalence of XML patterns," in *Computational Logic*, 2000, pp. 1152–1166.

[148] Peter T. Wood, "Minimizing simple XPath expressions," in *Proc. of Int. Workshop on Web and Databases (WebDB)*, 2001, pp. 13–18.

[149] Peter T. Wood, "Containment for XPath fragments under DTD constraints," in *Proc. of Int. Conf. on Database Theory (ICDT)*, 2003, pp. 300–314.

Index

Annotations, 106, 142
  ambiguity, 116
  in-mapping $\xrightarrow{i}$, 120, 122, 124, 125, 127, 138
  lifetime, 121, 124, 125
  mapping, 119
  operations, 106
    inclusion $\sqsubseteq$, 106, 108, 114, 121, 122
    intersect $\sqcap$, 106, 133, 135, 138
    union $\sqcup$, 106, 114, 116, 122, 138
  out-mapping $\xleftarrow{i}$, 120, 122, 124, 125, 127, 128
  scope, 121

LGQ Formulas
  absolute, 21
  binary atoms, 20
  connected, 21
  connectives, 19
  DAGs, 27, 47
  disjunctive normal form, 21
  DNF, 70
  equivalence, 23
  forests, 27, 47, 144
  graphs, 27, 47, 147
  paths, 27
    pdown, 121, 124, 143
    rdown, 121, 124, 143
    sdown, 121, 124, 143
    sdown, pdown, rdown, 28, 125, 144, 145
  predicates, 17
  predicates, classes, 18
  query, 21
  rule, body, 21
  rule, head, 21
  semantics, 23
  substitutions, 22
  substitutions,consistent, 22
  trees, 27, 47, 144
  unary atoms, 20
  unsatisfiability, 24
  variable
    forward sink-arity, 20, 31
    fresh, 40
    head, 21
    multi-sink, 20
    multi-source, 20
    sink, 20
    sink-arity, 20
    source, 20
  variable-preserving minimality, 28, 47, 77, 81, 95

Measures
  connection
    sequence, 30, 85–88
    variable, 30, 41, 64, 79, 80, 84
  DAG type factor $type^{dag}$, 31
  reverse position factor pos^{rev}, 30
  size $\mid e \mid$, 30
  type position factor $type^{rev}$, 31

Orders
  $>^{dag}_{type}$, 31, 69, 82
  $>^{rev}_{pos}$, 31, 82
  $>^{rev}_{type\times pos}$, 65, 66, 68, 82
  $>^{rev}_{type}$, 31, 59, 82
  $>_{dnf}$, 70, 82
  $>_{mul}$, 59, 65, 68–70
  $>_{size}$, 73, 82

lexicographic product, 54
on multisets, 55
well-founded, 54

Rewriting
AC matching, 57, 83
AC unification, 57
confluence, 54
critical pairs, 56
local, 56, 83
Identity $\approx$, 53
joinable terms, 55
lhs, rhs, 53
modulo
AC, 56, 73, 177
equational theory, 56
normal form, 54, 78
Rule $\rightarrow$, 53
Rules
par^+, 61
par^*, 60
child^*, 60
foll, 60
$\mathsf{nextSibl}^*$, 60
par, 60
prec, 60
$\mathsf{prevSibl}$, 61
$\mathsf{prevSibl}^+$, 61
$\mathsf{prevSibl}^*$, 60
DNF, 70
duplicate elimination, 71
relation-independent, 58
simplification, 71
unsatisfiability detection, 71
unsatisfiability propagation, 71
substitution, 53
matching, 53
mgu, 53
termination, 54, 82

Stream Processing Functions, 105, 107
α_f for predicate α, 109, 113, 114, 116, 131
in, 109, 127
out, 109, 127
$\boxed{head}$, 109, 127
annotation-merge Θ_c, 108, 119, 138
composition
parallel ($+\!\!+$), 107, 109, 110, 119, 122
sequential ($\cdot$), 107, 109, 110, 116, 122
connective $\vee_f$, 119, 138
connective $\wedge_f$, 119, 138
merge $\oplus$, 108
scope-begin $\overrightarrow{scope}$, 109, 110, 122, 125
scope-begin sdown, pdown, rdown, 125, 138
scope-end $\overleftarrow{scope}$, 109, 110, 122
symbol-filter $|$, 108

Curriculum Vitae

Personal Data

Name:	Dan Alexandru Olteanu
Date of Birth:	9th of November, 1976
Place of Birth:	Târgovişte, Romania
Marital Status:	Married

Studies

February 2005	PhD examination in Computer Science (LMU, Munich)
Spring/Summer 2000	visiting student in the Caravel project (INRIA, Rocquencourt)
October 2000	Dipl. Ing. in Computer Science (Politehnica, Bucharest)
July 1995	Baccalaureate in Math-Physics (HighSchool, Târgovişte)

Research Interests

- Semistructured Data
- XML query processing
- Data Integration
- Formal Languages and Automata