Querying Graphs with Data

A

Querying Graphs with Data

LEONID LIBKIN, University of Edinburgh
WIM MARTENS, Universität Bayreuth
DOMAGOJ VRGOČ, PUC Chile and Center for Semantic Web Research

Graph databases have received much attention as of late due to numerous applications in which data is naturally viewed as a graph; these include social networks, RDF and the Semantic Web, biological databases, and many others. There are many proposals for query languages for graph databases that mainly fall into two categories. One views graphs as a particular kind of relational data and uses traditional relational mechanisms for querying. The other concentrates on querying the topology of the graph. These approaches, however, lack the ability to *combine* data and topology, which would allow queries asking how data changes along paths and patterns enveloping it.

In this paper we present a comprehensive study of languages that enable such combination of data and topology querying. These languages come in two flavors. The first follows the standard approach of path queries, which specify how labels of edges change along a path, but now we extend them with ways of specifying how both labels and data change. From the complexity point of view, the right type of formalisms are subclasses of register automata. These, however, are not well suited for querying. To overcome this, we develop several types of extended regular expressions to specify paths with data, and study their querying power and complexity. The second approach adopts the popular XML language XPath and extends it from XML documents to graphs. Depending on the exact set of allowed features, we have a family of languages, and our study shows that it includes efficient and highly expressive formalisms for querying both the structure of the data and the data itself.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*Data Models*; H.2.3 [**Database management**]: Languages—*Query Languages*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata*; F.4.1 [**Mathematical logic and formal languages**]: Mathematical logic

General Terms: Theory, Languages, Algorithms

Additional Key Words and Phrases: Graph databases, data values, navigational queries, XPath

ACM Reference Format:
Leonid Libkin, Wim Martens and Domagoj Vrgoč, 2013. Querying Graphs with Data. *J. ACM* V, N, Article A (January YYYY), 52 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Managing and maintaining graph structured data is a topic that has gained significant popularity in the last few years. Indeed, there are several vendors offering graph database systems [Neo4j 2013; Dex 2013; Gremlin 2013] and a growing body of research literature on the subject (for recent surveys, see [Angles and Gutierrez 2008; Barceló 2013; Wood 2012]).

This work was supported by the EPSRC grants G049165, J015377 and M025268, DFG grant MA 4938/2-1 and by Millennium Nucleus Center for Semantic Web Research Grant NC120004.
Author's addresses: L. Libkin, LFCS, School of Informatics, University of Edinburgh; W. Martens, Universität Bayreuth; D. Vrgoč, Department of Computer Science, School of Engineering, PUC Chile.

DOI:http://dx.doi.org/10.1145/0000000.0000000

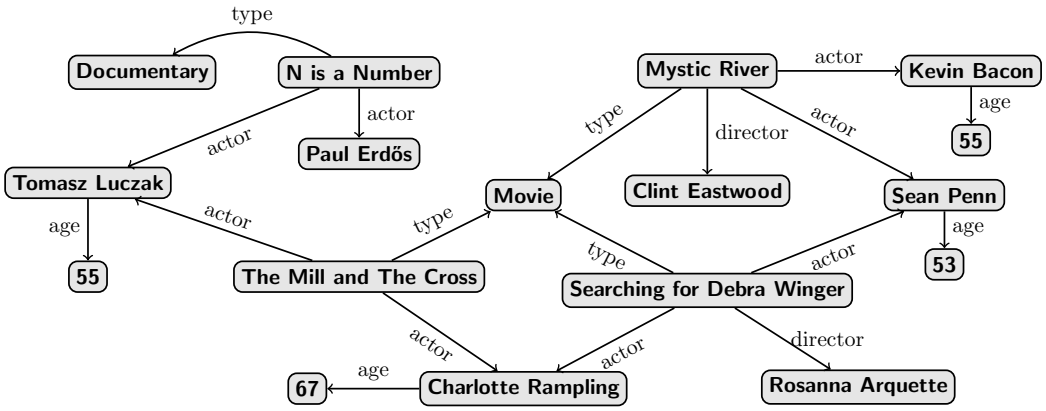


Fig. 1. A movie database represented as a graph

The model is very popular due to its numerous uses in services whose data structure is naturally represented by graphs: for instance, RDF triples are typically viewed as edges in labeled graphs [Gutierrez et al. 2011; Losemann and Martens 2012; Pérez et al. 2009] and so are connections between people in social networks [Fan 2012; Ronen and Shmueli 2009; San Martín and Gutierrez 2009]. Other application areas include biology, network traffic, crime detection, and modeling object-oriented data.

In all of these applications, data is modeled by a graph, with nodes representing entities in the database and edges representing various connections these entities can form. For example, if we are describing a social network it is natural to represent users by nodes, with edges symbolizing the connection between two users, such as *friend*, *co-worker*, *relative*, and so on. Another example would be a movie database where each node stores information about a specific movie, movie genre, or actor, while the edges of the graph tell us how entities are connected. One such database is presented in Figure 1. Since nodes can form different types of connections, it is common practice to assign labels to the edges connecting them. Finally, nodes themselves contain the actual data, such as the information about the movie title and duration, actor's names and ages, etc. This data is modeled as the usual relational data with attribute values coming from an infinite domain [Angles and Gutierrez 2008].

The most basic task for every data model, including graphs, is querying it. When designing query languages an important concern is balancing expressivity and efficiency: a language should be capable of describing a wide variety of relevant queries, while at the same time having a low complexity of main computational tasks. To achieve this for graph data, two separate approaches have been studied in the past: one focusing on the *data* stored in the graph and the other focusing on its *topology*.

The first approach treats the graph model as a relational database and uses traditional relational languages to extract the data. For example in the database above one could ask for all movies of the same duration, or all actors of the same age. Such queries are expressed in standard relational query languages, such as SQL on the practical side or relational calculus on the theory side; the particular query above is an example of a *conjunctive*, or select-project-join query [Abiteboul et al. 1995].

The second approach exploits the ability of graph databases to query intricate navigational patterns between objects, thus obtaining more information about the *topology* of the stored data. For example, considering the database in Figure 1 one might want to find pairs of actors connected by collaboration connections. This query would give us that Paul Erdős

and Charlotte Rampling are connected since they both co-starred with Tomasz Luczak. The same can be said for Kevin Bacon and Paul Erdős, but the sequence of collaborations is now longer. Taking into consideration that our databases can grow by inserting more data, it is easy to see that no fixed number of collaborators can be set in advance to answer this query, thus calling for languages that allow full transitive closure. These types of queries are not handled well by traditional DBMSs, but are of special importance for graph database querying: for instance, they underlie Facebook's *Graph Search* [Facebook 2014].

This second approach typically uses *regular path queries*, or *RPQs* as a basic building block. RPQs select nodes connected by a path described by a regular language over the labeling alphabet [Cruz et al. 1987]. One can think of them as a generalization of transitive closure by means of regular languages: indeed, in the most basic case, when the regular language permits all words, such a query expresses transitive closure. Many navigational languages for graph data are based on RPQs: those include languages with more complex patterns, backward navigation, and relations over paths, see, e.g., [Abiteboul and Vianu 1999; Barceló et al. 2012; Barceló et al. 2012; Calvanese et al. 2000; 2009; Consens and Mendelzon 1990].

Both of these approaches treat the data and the topological patterns enveloping it as two separate entities. Thus, the querying mechanisms one deals with generally fall into one of the following categories:

— queries about *data*, i.e., essentially relational queries (e.g., finding pairs of actors of the same age), or
— queries about *topology* such as finding nodes connected by a path with a certain label (e.g., actors who are connected via collaboration links).

What both of these approaches are incapable of doing is *combining data and topology*. As an example of a query that involves such a combination, one could for instance ask for people who have a finite Bacon number (that is, there is a sequence of collaboration connections linking them with Kevin Bacon). Note that here we have to test that the name attribute of the final actor in the sequence is indeed Kevin Bacon and not some arbitrary value. Another example one could imagine is a query that finds actors connected via professional links restricted to actors of the same age. In this case, comparison of data values (having the same age) is done for every node along the path. The main focus of this paper is to investigate languages that are designed for combining topology and data.

Path Queries Combining Topology and Data. As a starting point we take the approach of RPQs, where one uses regular expressions to describe the set of allowed paths in the graph and then the query itself simply extracts all pairs of nodes connected by paths whose edge labels form a word accepted by this regular expression. To extend this idea to a formalism that also deals with data values encountered along paths, we observe that such paths can be described by interchanging sequences of data values and edge labels. These objects are very close to *data words*, which are well-studied in the XML context [Bojanczyk 2010; Bojanczyk et al. 2011; Segoufin 2006; 2007]. A data word is a word in which every position is labeled by both a letter from a finite alphabet (e.g., an edge label) and a data value (e.g., the actor age, name, etc).

We can thus use multiple formalisms developed for data words (with a minor adjustment for the extra value) to specify a set of permissible paths for our query. Such formalisms abound in the literature and include first-order and monadic second-order logic with data comparisons [Bojanczyk et al. 2009; Bojanczyk et al. 2011], LTL with freeze quantifiers [Demri and Lazić 2009], XPath fragments [Bojanczyk 2010; Figueira 2009], and various automata models such as pebble and register automata [Bouyer et al. 2001; Kaminski and Francez 1994; Kaminski and Tan 2008; 2006; Neven et al. 2004].

The question is then, which one to choose? To answer this, in Section 3.1 we look at one of the fundamental problems of query language design, namely *query evaluation*. This problem asks, given a tuple $\overline{a}$, a database D, and a query q as input, if the tuple $\overline{a}$ is in the answer to the query q over the database D. This problem is often referred to as the *combined complexity* of query evaluation for a given language. Since queries are usually small compared to the database, another important measure for efficiency of a language is the so called *data complexity* of the query evaluation problem, which assumes the query q above to be fixed, that is, not part of the input. We show that as long as the formalism is capable of expressing what is perhaps the most primitive language with data value comparisons (two data values are equal) and is closed under complementation, then *data* complexity is NP-hard. Clearly one cannot tolerate such high data complexity, and this rules out most of the formalisms except *register automata*. These are the most natural analogs of nondeterministic finite automata (NFA) for data words.

We then study how register automata can be used to query paths in graphs (giving raise to *regular queries with memory*, or RQMs) and present a query evaluation algorithm based on checking non-emptiness of automata. This gives us an NL (nondeterministic logspace) data complexity bound and PSPACE-completeness for combined complexity. The bound for data complexity is good and the bound for combined complexity is completely acceptable: it is the same as for relational calculus. However, automata are not an ideal way of specifying conditions in queries, which is why one typically uses regular expressions instead. While some regular expression dialects have been considered for register automata [Kaminski and Tan 2006], they are far from intuitive. For this reason, we define a class of expressions called *regular expressions with memory* that are as expressive as register automata. The expressions use variables that store data values to mimic registers, but are otherwise similar to ordinary regular expressions. They are much easier to write than register automata are and they also inherit the same complexity bounds: NL data complexity and PSPACE combined complexity.

In an effort to lower the combined complexity, we define a class of *regular queries with data tests* (RQDs) that restrict the use of memory and only allow for testing of equality of data values in a strict stack-like manner. For this language we give a polynomial-time algorithm for combined complexity.

Beyond Path Queries. All query languages considered up to this point only query *paths*. We look beyond paths in the second part of the article. To see when queries that navigate through a single path no longer suffice, consider again the database from Figure 1. One might refine the notion of Bacon number in such a way that each collaboration witnessing it has to go through a movie; a documentary will no longer suffice. Such a query lies outside of reach of any language that navigates the graph using paths, since at each point of the path one has to check if the actors co-starred in a movie. Therefore, in order to define such queries one needs languages that allow for patterns that are no longer only paths, but allow testing if points along a path have some additional property.

The well studied XML language XPath has this ability, but was designed to query trees instead of graphs. Nevertheless, its goal seems very similar to the goal of many queries in graph databases: it describes properties of paths and patterns, taking into account both their purely navigational aspects as well as the data that is found in XML documents. The popularity of XPath is largely due to several factors:

— it defines many properties of paths and patterns that are relevant for navigational queries;
— it achieves expressiveness that relates naturally to yardstick languages for databases (such as first-order logic, its fragments, or extensions with some form of recursion); and
— it has good computational properties over XML, notably tractable combined complexity for many fragments and even linear-time complexity for some of them.

A natural question then is to see if the main ingredients that made XPath successful in the context of XML can be applied on graphs. In Section 4 we will address this issue and show that, when applied to graphs, XPath-like languages define an efficient and highly expressive class of queries. So, from a theoretical perspective, XPath-like languages seem to be very well-suited to query graphs. From a practical perspective this seems to be the case too: many users are already familiar with XPath to query (XML) trees and, therefore, using it to query graphs instead does not seem to present a steep learning curve.

To a limited extent, using XPath to query graphs was tried before. On the practical side, XPath-like languages have been used to query graph data (e.g., [Cassidy 2003; Gremlin 2013]), without any analysis of their expressiveness and complexity, however. On the theoretical side, several papers investigated XPath-like languages from the modal perspective, dropping the assumption that they are evaluated on trees [Alechina et al. 2003; Marx 2003], but most notably in [Fletcher et al. 2011] the authors consider an algebra of binary relations which is the basis of our navigational language. It is important to note that none of these approaches considered data values, thus making them suited only to ask queries about topology of the graph and not about the interplay this topology has with the stored data.

We use several versions of XPath-like languages for graph databases, all of them collectively named GXPath. Like XPath (or closely related logics such as PDL and CTL*), all versions of GXPath have node tests and path formulae, and as the basic navigational axes they use letters from the alphabet labeling graph edges. On top of that, the language is also equipped with three different kinds of data tests: the first testing if the data value in some node equals a constant, the second being the "standard" XPath data test (the equalities/inequalities studied in [Bojanczyk and Parys 2011]) and the third one checking if data values at the beginning and the end of a pattern are equal or different.

We first study the complexity of various fragments of GXPath. As it turns out, *all* GXPath fragments inherit nice properties from XPath on trees due to the "modal" nature of the language: the combined complexity is always polynomial of low degree. The data complexity is not worse than cubic when we disregard the data values and even linear for some expressive fragments. With data comparisons added, data complexity becomes cubic again. We also show that adding numerical formulas that specify length of a path connecting two nodes, although making the language exponentially more succinct [Losemann and Martens 2012], has no effect on the complexity of query evaluation.

Following this we analyze the expressive power of the language, using the usual database yardstick of first-order logic as our reference point. Just like in the tree case [ten Cate and Marx 2007], we isolate a fragment capturing first-order logic with three variables, but show that some well known results that hold over trees [Marx 2005] are no longer valid on graphs. Finally, we establish a full hierarchy of various GXPath fragments and variants and show how they compare to traditional graph languages, as well as to languages introduced in Section 3.

Conjunctive queries. To round off the study we also look at conjunctive queries that use languages from Section 3 and Section 4 as their basic building blocks and show that for these classes of queries query evaluation remains optimal. Namely, it is no harder than that of single queries when dealing with conjunctive versions of RQMs and register automata, while for RQDs and GXPath the complexity matches that of relational conjunctive queries and CRPQs.

Remark 1.1. This paper combines and extends two conference papers [Libkin and Vrgoč 2012b; Libkin et al. 2013a]. Specifically, Section 3 comes from [Libkin and Vrgoč 2012b] and parts of Section 4 from [Libkin et al. 2013a]. As most of the proofs were omitted in conference versions we present full proofs here. In addition to that we also determine the full hierarchy of GXPath fragments in Section 4.5, compare the languages in Section 4.4 and show how to use conjunctive queries based on GXPath in Section 5. Furthermore, all of

the languages presented here are equipped with the ability to use constants, a feature not present in the preliminary conference versions.

Organization. We review basic definitions in Section 2. In Section 3 we study register automata and related classes of expressions. In Section 4 we show how XPath can be adapted to work over graphs and in Section 5 we define conjunctive queries using languages from previous sections as their basic building blocks. We give some concluding remarks in Section 6.

2. PRELIMINARIES

We first describe graph databases. We assume a model in which edges are labeled by *letters* from a finite alphabet Σ and nodes can contain *data values* from a countably infinite set $\mathcal{D}$ (for instance, attributes of people in a social network). For simplicity of notation only, we assume a single data value per node, as is often done in modeling XML with *data trees* [Segoufin 2007]. This is not a restriction at all, since a node u with $k > 1$ attributes be modeled by a node u with k outgoing edges, leading to k new nodes containing the data values (again, in the same way as data trees model XML documents). For a discussion on how this can be implemented for languages considered in this paper we refer the reader to [Vrgoč 2014]. The same reference also illustrates how the approach when data values reside in the edges, or in both nodes and edges, can be reduced to the approach taken here.

Definition 2.1 (*Data graphs*). A *data graph* (over Σ and $\mathcal{D}$) is a triple $G = \langle V, E, \rho\rangle$, where:

— V is a finite set of nodes;
— $E \subseteq V \times \Sigma \times V$ is a set of labeled edges; and
— $\rho : V \to \mathcal{D}$ is a function that assigns a data value to each node in V.

When we deal with purely navigational queries, i.e., those not taking data values into account, we refer to data graphs as $\langle V, E\rangle$, omitting the function ρ. We write E_a for the set of a-labeled edges, i.e., $E_a = \{(v, v') \mid (v, a, v') \in E\}$.

A *path* from node v_1 to v_n in a graph is a sequence

$$\pi = v_1 a_1 v_2 a_2 v_3 \dots v_{n-1} a_{n-1} v_n \tag{1}$$

such that each (v_i, a_i, v_{i+1}), for $i < n$, is an edge in E. We use the notation $\lambda(\pi)$ to denote the word $a_1 \dots a_{n-1} \in \Sigma^*$ to which we refer as the *label* of path π.

Queries in this article will always select a binary relation of nodes in a data graph. That is, for a data graph G and query Q we always have that $Q(G) \subseteq V \times V$. We will be interested in the *query evaluation problem* which checks, for a query Q, a data graph G, and a pair of nodes (v, v'), whether $(v, v') \in Q(G)$. Furthermore, we study the *data-* and *combined complexity* of the query evaluation problem. For combined complexity, the input to the problem is Q, G, and (v, v'). For data complexity, the query Q is fixed, so the input only consists of G and (v, v').

Navigation languages for graph databases. Before we explain the approach for querying graphs that we take in this article, we provide an overview of existing approaches that should clarify why our approach is natural. Most navigational formalisms for querying graph databases are based on *regular path queries*, or *RPQ*s [Cruz et al. 1987], and their extensions. An RPQ is an expression of the form

$$x \xrightarrow{L} y,$$

where L is a regular language over Σ (typically represented by a regular expression or an NFA). Given a Σ-labeled graph $G = \langle V, E\rangle$, the answer to an RPQ as above is the set of pairs of nodes (v, v') such that there is a path π from v to v' with $\lambda(\pi) \in L$.

Conjunctive RPQs, or *CRPQs* [Consens and Mendelzon 1990] are the closure of RPQs under conjunction and existential quantification. Formally, they are expressions of the form

$$\varphi(\bar{x}) \;=\; \exists \bar{y} \bigwedge_{i=1}^{n} (z_i \xrightarrow{L_i} u_i), \tag{2}$$

where all variables z_i, u_i come from $\bar{x}, \bar{y}$. The semantics naturally extends the semantics of RPQs: $\varphi(\bar{a})$ is true in G iff there is a tuple $\bar{b}$ of nodes such that, for every $i \leq n$, every pair v_i, v_i' interpreting z_i and u_i is in the answer to the RPQ $z_i \xrightarrow{L_i} u_i$. These have been further extended, for instance, to 2CRPQs that allow navigation in both directions (i.e., the edges can be traversed both forwards and backwards [Calvanese et al. 2000]), U2CRPQs that allow unions, or to *extended* CRPQs, in which paths witnessing the RPQs $z_i \xrightarrow{L_i} u_i$ can be named and compared for relationships between them, defined as regular or even rational relations [Barceló et al. 2012; Barceló et al. 2012].

To formalize navigation in RDF documents, [Pérez et al. 2010] defines the class of *nested regular expressions,* or NRE, that extend ordinary regular expressions with the nesting operator and inverses. Formally their syntax is defined as follows:

$$n := \; \varepsilon \;\mid\; a \;\mid\; a^- \;\mid\; n \cdot n \;\mid\; n^* \;\mid\; n + n \;\mid\; [n]$$

where a ranges over Σ.

When evaluated on a graph G, an NRE n defines a binary relation $[\![n]\!]^G$ consisting of pairs of nodes connected by a pattern specified by the NRE. More precisely, this relation $[\![n]\!]^G$ is defined inductively as follows:

$$\begin{aligned}
[\![\varepsilon]\!]^G &= \{(v,v) \mid v \in V\} \\
[\![a]\!]^G &= \{(v,v') \mid (v,a,v') \in E\} \\
[\![a^-]\!]^G &= \{(v,v') \mid (v',a,v) \in E\} \\
[\![n \cdot n']\!]^G &= [\![n]\!]^G \circ [\![n']\!]^G \\
[\![n + n']\!]^G &= [\![n]\!]^G \cup [\![n']\!]^G \\
[\![n^*]\!]^G &= \text{the reflexive transitive closure of } [\![n]\!]^G \\
[\![[n]]\!]^G &= \{(v,v) \mid \exists v' \text{ such that } (v,v') \in [\![n]\!]^G\}.
\end{aligned}$$

Here, $\circ$ denotes the composition operator of binary relations. That is, for binary relations R_1 and R_2, we define $R_1 \circ R_2 = \{(u,v) \mid \exists z$ such that $(u,z) \in R_1$ and $(z,v) \in R_2\}$.

Path languages and graph languages. When we compare NREs to RPQs we can see that they query graphs in a rather different manner. RPQs

(1) specify a set of allowed path labels through a language L and
(2) then their semantics is defined by the existence of paths in the graph whose label belongs to L.

In particular, each answer of an RPQ is witnessed by the existence of a certain path in the graph. This is not the case for NREs, which are defined through a recursive definition that computes node pairs. These node pairs do not have to be connected by a (directed) path in the graph.

In the remainder of the paper we will use the term *path languages* to refer to the former kind (since their semantics is defined on paths that match a language) and *graph languages* to refer to the latter kind (since their semantics tests graph properties that go beyond paths). In other words, in the forthcoming sections we will be dealing with:

— **Path languages** – when the underlying idea is to describe the set of permissible path labels and then the semantics calls for finding paths in the graph whose labels belong to this set.

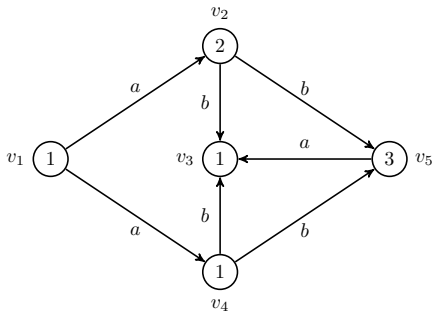


Fig. 2. Graph database with data values

— **Graph languages** – when queries are defined to operate directly on graphs and when paths alone no longer suffice to capture the intended semantics.

One difference between path- and graph languages becomes apparent when considering queries that can traverse edges bidirectionally. Whereas graph languages are not navigationally constrained by a path, path languages would travel back and forth along one path in the graph. We note that NREs, our prime example of a graph language, could also be used to define sets of words (i.e. their semantics could be adapted to paths instead of graphs), where the nesting would only look ahead (or backwards) along a single path. This approach, although interesting in its own right [Reutter 2013a], falls outside the scope of this work.

3. PATH LANGUAGES

In this section we will examine several languages that take the approach of defining a set of permissible paths when describing the answer to a query. We will show how these can be defined using standard language theoretic formalisms that take data value comparisons into consideration and examine their query evaluation properties and relative expressive power.

In order to illustrate what a suitable formalism for describing both navigational and data aspects of graphs might be, consider the data graph in Figure 2. A typical RPQ may ask for pairs of nodes connected by a path from the regular language $(ab)^*$. In the graph in Fig. 2, one possible answer is (v_1, v_3), another (v_1, v_5). To combine such a query with data values, we may ask queries of the following kind:

— Find nodes connected by a path from $(ab)^*$ such that the data values at the beginning and at the end of the path are the same. In this case, (v_1, v_3) is still in the answer but (v_1, v_5) is not.
— We may extend data value comparisons to other nodes on the path, not only to the first and the last node. For example, we may ask for nodes connected by paths along which the data value remains the same, or on which all data values are different from the first one. The pair (v_1, v_3) is in the answer to the first query (the path $v_1v_4v_3$ witnesses it), while the pair (v_1, v_5) is in the answer to the second, as witnessed by the path $v_1v_2v_5$.

What kind of languages can we use in place of regular languages to specify paths with data? To answer this, consider, for example, a path $v_1v_2v_5v_3$ in the graph. If we traverse it by starting at v_1, reading its data value, then reading the label of (v_1, v_2), then the data value in v_2, etc., we end up with the sequence $1a2b3a1$, which we refer to as a *data path*.

Definition 3.1 (*Data paths*). Let $\pi = v_1a_1v_2 \cdots v_{n-1}a_{n-1}v_n$ be a path from v_1 to v_n in a data graph. The *data path* corresponding to π is

$$w_\pi = \rho(v_1)a_1\rho(v_2)a_2\rho(v_3)\ldots\rho(v_{n-1})a_{n-1}\rho(v_n). \tag{3}$$

A data path is therefore a sequence of alternating data values and labels which starts and ends with data values. The set of all *data paths*, i.e., such alternating sequences over Σ and $\mathcal{D}$, will be denoted by $\Sigma[\mathcal{D}]^*$. For both paths and data paths, we use the notation $\lambda(\pi)$ or $\lambda(w_\pi)$ to denote their label, i.e., the word $a_1 \dots a_{n-1} \in \Sigma^*$.

Data paths are very closely related to *data words* [Bojanczyk 2010; Bojanczyk et al. 2011; Segoufin 2006; 2007], which have been actively studied in the XML context. A data word is a word in which every position is labeled by a pair $\binom{a}{d}$ where a comes from a finite alphabet and d is a data value (e.g., a natural number). Data paths are essentially data words with an extra data value. We can represent the data path $1a2b3a1$ as a data word $\binom{\#}{1}\binom{a}{2}\binom{b}{3}\binom{a}{1}$, where # is a new alphabet symbol reserved for the extra data value. It is straightforward to show that all of the expression classes and automata models that we consider in this section can be defined to operate both over data words and data paths. For a formal treatise of this see [Libkin and Vrgoč 2012b; 2012a; Vrgoč 2014].

We can thus simply use formalisms for data words (with a minor adjustment for the extra value) to specify data paths. Such formalisms abound in the literature and include first-order and monadic second-order logic with data comparisons [Bojanczyk et al. 2009; Bojanczyk et al. 2011], LTL with freeze quantifiers [Demri and Lazić 2009], XPath fragments [Bojanczyk 2010; Figueira 2009], and various automata models such as pebble and register automata [Bouyer et al. 2001; Kaminski and Francez 1994; Kaminski and Tan 2008; 2006; Neven et al. 2004].

In this section, we study *data path queries*. A data path query is an expression of the form $x \xrightarrow{L} y$, where L is a language of data paths.

3.1. Existing Languages for Paths

To specify data path queries, we need a way of defining languages of data paths. As illustrated above, data paths are essentially data words with an extra data value attached. Quite a few logics and automata models have been developed for data words over the past few years, mainly in connection with the study of XML and especially XPath. We now give a quick overview of them. A more extensive survey can be found in [Segoufin 2006].

FO($\sim$) **and** MSO($\sim$)**.** These are first-order logic and monadic second-order logic extended with the binary predicate $\sim$ saying that data values in two positions are the same. For example, $\exists x \exists y\ x \neq y \wedge a(x) \wedge a(y) \wedge x \sim y$ says that there are two a-labeled positions with the same data value. Two-variable fragments of FO($\sim$) and existential MSO with the $\sim$ predicate have been shown to have decidable satisfiability problems [Bojanczyk et al. 2009; Bojanczyk et al. 2011].

Pebble automata. These are basically finite state automata equipped with a finite set of pebbles. To ensure regular behavior, pebbles are required to adhere to a stack discipline. The automata are modeled in such a way that the last placed pebble acts as the automaton head and we are allowed to drop and lift pebbles over the current position. In addition to this we can also compare the current data value to the one that already has a pebble placed over it. Algorithmic properties and connections with logics have been extensively studied in [Neven et al. 2004].

LTL$_\downarrow$. This is the standard LTL expanded with a *freeze operator* "$\downarrow$" that allows us to store the current data value into a memory location and use it for future comparisons. The satisfiability problem for the full logic is undecidable, but various decidable restrictions are known [Demri and Lazić 2009; Demri et al. 2007].

Register automata. These are in essence finite state automata extended with a finite set of registers allowing us to store data values. Although first studied only on words over an infinite alphabet [Kaminski and Francez 1994; Neven et al. 2004; Sakamoto and Ikeda 2000], they are easily extended to handle data words, as illustrated in [Demri and Lazić

2009; Segoufin 2006]. They act as usual finite state automata in the sense that they move from one position to another by reading the appropriate letter from the finite alphabet, but are also allowed to compare the current data value with ones already stored in the registers.

XPath fragments. XPath is the standard language for navigating in XML documents and while it has been used to define data path languages in the past [Figueira 2010], we will not consider it in this context. The main reason for this is the fact that XPath is intrinsically a graph (originally tree) language, and, as we show in Section 4, even the more general graph approach already yields very efficient query evaluation algorithms (combined complexity is always PTIME and for some fragments even linear).

In deciding which formalism to investigate more deeply we look at the *data complexity* of evaluating data path queries and rule out those for which data complexity is intractable (assuming P $\neq$ NP). It turns out that most of the formalisms for data words/paths are actually not suitable for graph querying. This is implied by the following result. Let L_{eq} be the language of data paths that contain two equal data values. We will denote its complement, i.e., the language of all data paths containing pairwise different data values, by $\overline{L_{eq}}$.

THEOREM 3.2. *The data complexity of evaluating $Q = x \xrightarrow{\overline{L_{eq}}} y$ over data graphs is* NP*-complete.*

PROOF. To see that the problem is in NP it suffices to observe that, given a data graph G and two nodes (s,t), we can test if $(s,t) \in Q(G)$ by guessing a path from s to t and checking if all data values on the path are pairwise different.

The proof of the NP lower bound is by showing that with $\overline{L_{eq}}$, one can encode the 2-disjoint-paths problem which is NP-complete [Fortune et al. 1980]. This problem is to check, for a graph G and four nodes s_1, t_1, s_2, t_2 in G, whether there exist two paths in G, one from s_1 to t_1 and the other from s_2 to t_2 that have no nodes in common. First, we argue that we can assume that s_1, t_1, s_2, and t_2 are distinct. This is because we can always add two new nodes for each repeated node and connect them with all the nodes the repeated node was connected to, thus modifying our problem to have all source and target nodes different.

Assume that $G = \langle V, E\rangle$ is a digraph and s_1, t_1, s_2, t_2 are four distinct nodes in G. Recall that our query is $Q = x \xrightarrow{\overline{L_{eq}}} y$. Since Q will disregard edge labels we can take $\Sigma = \{a\}$. We will construct a data graph G' and two nodes $s, t \in G'$ such that $(s,t) \in Q(G')$ if and only if there are two disjoint paths in G from s_1 to t_1 and from s_2 to t_2.

Let $V = \{v_1, \ldots, v_n\}$. The graph G' will contain two disjoint isomorphic copies of G (extended with data values and labels) connected by a single edge. We define the two isomorphic copies $G_1 = \langle V_1, E_1, \rho_1\rangle$ and $G_2 = \langle V_2, E_2, \rho_2\rangle$ by:

— $V_1 = \{v'_1, \ldots, v'_n\}$,
— $V_2 = \{v''_1, \ldots, v''_n\}$,
— $E_1 = \{(v'_i, a, v'_j) : (v_i, v_j) \in E\}$,
— $E_2 = \{(v''_i, a, v''_j) : (v_i, v_j) \in E\}$ and
— $\rho_1(v'_i) = \rho_2(v''_i) = i$, for $i = 1 \ldots n$,

and then let $G' = \langle V', E', \rho'\rangle$, where

— $V' = V_1 \cup V_2$,
— $E' = E_1 \cup E_2 \cup \{(t'_1, a, s''_2)\}$ and
— $\rho' = \rho_1 \cup \rho_2$.

Note that ρ' is well defined since V_1 and V_2 are disjoint. Finally, we define $s = s'_1$ and $t = t''_2$.

We claim that $(s,t) \in Q(G')$ if and only if there are two disjoint paths in G from s_1 to t_1 and from s_2 to t_2 in G. To see this assume first that $(s,t) \in Q(G')$. This means that we have a path in G' which starts at s'_1 and ends at t''_2. In particular, it must pass the edge from t'_1 to s''_2, since this is the only edge connecting the two graphs. Also, since all data values on this path are different, we know that no node can repeat, i.e., the path contains no two copies of the same node in G. But then we simply split this path into two disjoint paths in G since the structure of edges in G' is the same as the one in G with the exception of edge between t'_1 and s''_2.

Conversely, assume that we have two disjoint paths from s_1 to t_1 and from s_2 to t_2 in G. Notice that we can assume these two paths to contain no loops, since loops can be removed while keeping the paths disjoint. To obtain a data path from s to t in $\overline{L_{eq}}$, we simply follow the corresponding path from s'_1 to t'_1 in G_1 (and thus in G'), traverse the edge between t'_1 and s''_2 and then follow the path in G_2 (and thus in G') from s''_2 to t''_2 corresponding to the path from s_2 to t_2 in G. Since the two paths in G have no node in common and do not have loops, all data values on the constructed data path from s to t in G' are different.

This completes the proof. □

Note that L_{eq} expresses only a very simple property of data paths/words. It seems hard to imagine a useful formalism for data paths that cannot check for the equality of data values. The corollary below effectively rules out closure under complement for such formalisms if they are to be used in graph querying.

COROLLARY 3.3. *Assume that we have a formalism for data paths that can define L_{eq} and that is closed under complement. Then the data complexity of evaluating data path queries is* NP*-hard.*

This immediately rules out FO($\sim$) and its two-variable fragment, LTL with the freeze quantifier, and pebble automata. The only hope we have among standard formalisms is *register automata*, since they are not closed under complementation [Kaminski and Francez 1994]. In the following sections we show that we can achieve good query answering complexity using register automata and some of their restrictions, while still retaining sufficient expressive power.

Remark 3.4. It is important to note that we will come back to *FO* in Section 4, where its semantics will be defined directly on graphs. As a consequence, in that context negation will be limited to the active domain, and not to the set of all data words as here, so it will no longer be possible to express that all data values along a path are different

3.2. Data path queries with register automata

From our list of candidate formalisms for path languages in Section 3.1, register automata are the only standard formalism that does not immediately lead to NP-hard data complexity for evaluating data path queries. In this section we define them and study query evaluation for data path queries based on these automata. To this end we will slightly alter the definition of register automata used in e.g. [Demri and Lazić 2009; Segoufin 2006] to work on data paths rather than data words, without affecting their desirable properties.

Register automata move from one state to another by reading the appropriate letter from the finite alphabet and comparing the currently read data value to the ones previously stored into the registers. Our version of register automata will use slightly more involved comparisons which will be boolean combinations of atomic $=, \neq$ comparisons of data values. To define such conditions formally, assume that, for each $k > 0$, we have variables $x_1, \ldots, x_k$. Then conditions in $\mathcal{C}_k$ are given by the grammar:

$$c \;:=\; x_i^{=} \mid x_i^{\neq} \mid z^{=} \mid z^{\neq} \mid c \wedge c \mid c \vee c \mid \neg c, \qquad 1 \leq i \leq k,$$

where z is a data value from $\mathcal{D}$, also referred to as the *constant*. Let $\mathcal{D}_\perp = \mathcal{D} \cup \{\perp\}$, where $\perp$ is a special symbol signifying that the register is empty. The satisfaction of a condition is defined with respect to a data value $d \in \mathcal{D}$ and a tuple $\tau = (d_1, \ldots, d_k) \in \mathcal{D}_\perp^k$ as follows:

— $d, \tau \models x_i^{=}$ iff $d = d_i$;
— $d, \tau \models x_i^{\neq}$ iff $d \neq d_i$;
— $d, \tau \models z^{=}$ iff $d = z$;
— $d, \tau \models z^{\neq}$ iff $d \neq z$;
— $d, \tau \models c_1 \wedge c_2$ iff $d, \tau \models c_1$ and $d, \tau \models c_2$ (and likewise for $c_1 \vee c_2$);
— $d, \tau \models \neg c$ iff $d, \tau \not\models c$.

In what follows, $[k]$ is a shorthand for $\{1, \ldots, k\}$ and ε for a condition that is true for any valuation and data value (e.g. $c \vee \neg c$).

Definition 3.5 (*Register data path automata*). Let Σ be a finite alphabet, and k a natural number. A *k-register data path automaton* over Σ and $\mathcal{D}$ is a tuple $\mathcal{A} = (Q, q_0, F, \tau_0, \delta)$, where:

— $Q = Q_w \cup Q_d$, where Q_w and Q_d are two finite disjoint sets of word states and data states;
— $q_0 \in Q_d$ is the initial state;
— $F \subseteq Q_w$ is the set of final states;
— $\tau_0 \in \mathcal{D}_\perp^k$ is the initial configuration of the registers;
— $\delta = (\delta_w, \delta_d)$ is a pair of transition relations:
  — $\delta_w \subseteq Q_w \times \Sigma \times Q_d$ is the word transition relation;
  — $\delta_d \subseteq Q_d \times \mathcal{C}_k \times 2^{[k]} \times Q_w$ is the data transition relation.

The intuition behind this definition is that since we alternate between data values and word symbols in data paths, we also alternate between data states (which expect a data value as the next symbol) and word states (which expect alphabet letters as the next symbol). We start with a data value, so q_0 is a data state, and end with a data value, so final states, seen after reading that value, are word states.

In a word state the automaton behaves like the usual NFA (but moves to a data state using its word transition function). In a data state, the automaton checks if the current data value and the configuration of the registers satisfy a condition, and if they do, moves to a word state and updates some of the registers with the read data value. Both functionalities are illustrated in the following picture, where in the data transition the automaton checks if the data value is different to the one stored in register seven and then moves to a word state while storing the value into registers from the set I.

Notice that we also could have modeled constants by storing them into the initial assignment (possibly using more registers). We put them into conditions to have a uniform way of handling them when we define regular queries with memory (RQMs) and regular queries with data tests (RQDs) in the following sections. When the condition ε is used, or when $I = \emptyset$ (that is, we do not store the data value into any register) we will omit them from the transition in the diagrams above.

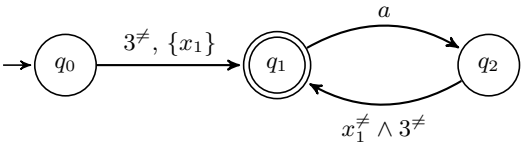
Now we formally define acceptance of a data path by a register automaton. Given a data path $w = d_0 a_1 d_2 a_3 \ldots a_{n-1} d_n$, where each d_i is a data value and each a_l is a letter, a *configuration of $\mathcal{A}$ on w* is a tuple (j, q, τ), where j is the current position of the symbol in w that $\mathcal{A}$ reads, q is the current state and $\tau \in \mathcal{D}_\perp^k$ is the current content of the registers.

The *initial configuration* is $(0, q_0, \tau_0)$ and any configuration (j, q, τ) with $q \in F$ is a *final configuration*. The automaton can move from configuration $C = (j, q, \tau)$ to configuration $C' = (j+1, q', \tau')$ on w if one of the following holds:

— $a_j \in \Sigma$, there is a transition $(q, a_j, q') \in \delta_w$, and $\tau' = \tau$; or
— $d_j \in \mathcal{D}$ and there is a transition $(q, c, I, q') \in \delta_d$ such that $d_j, \tau \models c$ and τ' coincides with τ except that the ith component of τ' is set to d_j whenever $i \in I$.

A data path w is *accepted* by $\mathcal{A}$ if $\mathcal{A}$ can move from the initial configuration to a final configuration on w. The language of data paths accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$.

Example 3.6. The following automaton recognizes the language of all data paths that have label a^*, do not contain the data value "3", and for which the first data value differs from all the others. It operates by testing if its first data value is different from 3, denoted $3^{\neq}$ and storing it into its first register x_1. It then moves to the state q_1, where it loops (by alternating between q_2 and q_1), while checking that the data value being read is different from 3 and the one stored in x_1. If this is satisfied it ends its computation in an accepting state q_1.

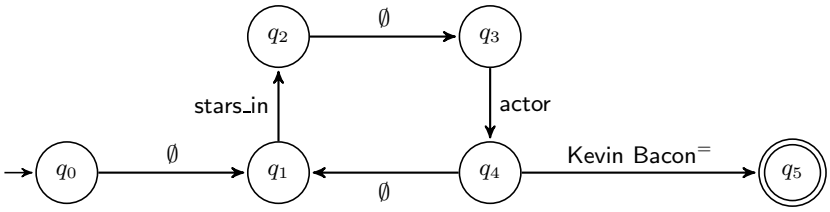


3.2.1. Regular data path queries. Our first class of queries on graphs with data is based on register data path automata.

Definition 3.7. A *regular data path query (RDPQ)* over Σ and $\mathcal{D}$ is an expression $Q = x \xrightarrow{\mathcal{A}} y$ where $\mathcal{A}$ is a register data path automaton over Σ and $\mathcal{D}$. Given a data graph G, the result of the query $Q(G)$ consists of pairs of nodes (v, v') such that there is a data path w from v to v' that belongs to $L(\mathcal{A})$.

Example 3.8. Coming back to the movie database from Figure 1, assume that, for each edge labeled actor that connects a movie or a documentary with an actor, we also have an edge going in the other direction labeled stars_in. For example we will add one such edge connecting Kevin Bacon with Mystic River, or Charlotte Rampling with The Mill and The Cross.

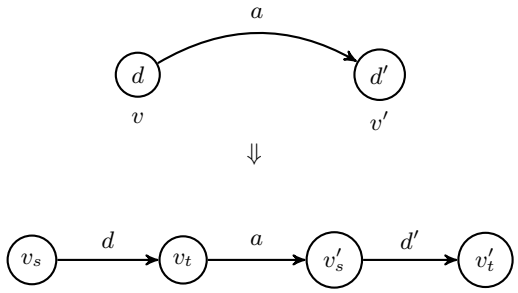
We can then ask for all people who have a finite Bacon number using the query $Q = x \xrightarrow{\mathcal{A}} y$, specified by the following register automaton $\mathcal{A}$:



The automaton works by traversing a sequence of stars_in · actor edges, which connect all pairs of actors who co-starred in a same film, but also makes sure that the last data value equals Kevin Bacon. Note that in addition to the actor with a finite Bacon number, this query also returns the node corresponding to Kevin Bacon.

To evaluate RDPQs, we transform both a data graph G and a k-register data path automaton $\mathcal{A}$ into NFAs over an extended alphabet and reduce query evaluation to NFA nonemptiness. More precisely, to evaluate $Q(G)$, we do the following:

(1) Let D be the set of all data values in G.
(2) Transform $G = \langle V, E, \rho\rangle$ into a graph $G' = \langle V', E'\rangle$ over the alphabet $\Sigma \cup D$ as follows:
— $V' = \{v_s \mid v \in V\} \cup \{v_t \mid v \in V\}$
— $E' = \{(v_t, a, v'_s) \mid (v, a, v') \in E\} \cup \{(v_s, \rho(v), v_t) \mid v \in V\}$
Basically, we split each node v with a data value d into a source node v_s and a target node v_t and add a d-labeled edge between them. After that we restore the edges from E so that they go from target to source nodes. This is illustrated below.



(3) Transform the automaton $\mathcal{A} = (Q, q_0, F, \tau_0, (\delta_w, \delta_d))$ into an NFA $\mathcal{A}_D = (Q', q'_0, F', \delta')$ over the alphabet $\Sigma \cup D$, whose intention is to accept precisely those data paths from $L(\mathcal{A})$ that have data values from D. We construct $\mathcal{A}_D$ as follows:
— $Q' = Q \times D_0^k$, with $D_0 = D \cup \{\perp\} \cup \{\tau_0(i) \mid i = 1 \dots k\}$;
— $q'_0 = (q_0, \tau_0)$;
— $F' = F \times D_0^k$;
— δ' includes two types of transitions.
  (a) Whenever we have a transition (q, a, q') in δ_w, we add transitions $((q, \tau), a, (q', \tau))$ to δ' for all $\tau \in D_0^k$.
  (b) Whenever we have a transition (q, c, I, q') in δ_d, we add transitions $((q, \tau), d, (q', \tau'))$ if $d, \tau \models c$ and τ' is obtained from τ by putting d in positions from the set I.
To see that $\mathcal{A}_D$ accepts the data paths from $L(\mathcal{A})$ that have data values from D, it suffices to show that every accepting run of $\mathcal{A}_D$ corresponds to an accepting run of $\mathcal{A}$ and vice versa, in the case of paths whose data values come from D. But this follows easily since $\mathcal{A}_D$ has all possible configurations of registers at its disposal.

For two nodes v, v' of G, we turn G' into an NFA $\mathcal{A}_{G',v,v'}$ by letting v_s be its initial state and v'_t be its final state. Then we have the following.

PROPOSITION 3.9. *Let $Q = x \xrightarrow{\mathcal{A}} y$ be an* RDPQ*, and G a data graph whose data values form a set $D \subseteq \mathcal{D}$. Then*

$$(v, v') \in Q(G) \quad \Leftrightarrow \quad L(\mathcal{A}_{G',v,v'} \times \mathcal{A}_D) \neq \emptyset.$$

PROOF. Recall that $\mathcal{A}_D$ accepts exactly the data paths from $L(\mathcal{A})$ that have data values from D. To see that the statement of Proposition 3.9 holds assume first that $(v, v') \in Q(G)$. Then there is a data path $w_\pi = d_0 a_0 d_1 a_1 \dots a_{n-1} d_n$ from v to v' such that $w_\pi \in L(\mathcal{A})$. Since this is a data path in G starting with v and ending with v' it must also be a word in the language of $\mathcal{A}_{G',v,v'}$. On the other hand, since it is in $L(\mathcal{A})$, it must also be in $L(\mathcal{A}_D)$, since $\mathcal{A}_D$ is simply the restriction of $\mathcal{A}$ to the alphabet in which data values come only from the set D. Thus $L(\mathcal{A}_{G',v,v'} \times \mathcal{A}_D) \neq \emptyset$.

Conversely, assume that $L(\mathcal{A}_{G',v,v'} \times \mathcal{A}_D) \neq \emptyset$. Then there is a data path $w_\pi = d_0 a_0 d_1 a_1 \dots a_{n-1} d_n$ such that $w_\pi \in L(\mathcal{A}_{G',v,v'})$ and $w_\pi \in L(\mathcal{A}_D)$. But then by construction w_π must be a data path in G from v to v'. Also $w_\pi \in L(\mathcal{A})$, since $L(\mathcal{A}_D)$ is simply a restriction of the language of $\mathcal{A}$ to data paths whose data values come from D. This then implies that $(v, v') \in Q(G)$. □

Thus, query evaluation, like in the case of the usual RPQs, is reduced to automata nonemptiness, although this time the automata are over larger alphabets. Since the construction is polynomial in the size of G and exponential in the size of $\mathcal{A}$ (as k gets into the exponent), we immediately get a PTIME upper bound for data complexity and an EXPTIME upper bound for combined complexity. By performing on-the-fly nonemptiness checking for the product, we can lower these bounds.

THEOREM 3.10. *The data complexity of* RDPQ*s over data graphs is* NL*-complete and the combined complexity of* RDPQ*s over data graphs is* PSPACE*-complete.*

PROOF. Concerning data complexity, the NL upper bound is obtained by guessing a word in $L(\mathcal{A}_{G',v,v'} \times \mathcal{A}_D)$ symbol by symbol and simulating $\mathcal{A}_{G',v,v'} \times \mathcal{A}_D$ on the fly. The NL lower bound is immediate from the reachability problem in graphs. For combined complexity, the upper PSPACE bound follows from essentially the same algorithm as for the NL upper bound. However, simulating $\mathcal{A}_{G',v,v'} \times \mathcal{A}_D$ on the fly now takes polynomial space because $\mathcal{A}_D$ is not constant anymore. The PSPACE lower bound is immediate from Proposition 3.13 and Theorem 3.18, which are proved for a more restricted language. □

3.3. Queries based on regular expressions with memory

Regular data path queries based on register automata have acceptable complexity bounds: data complexity is the same as for RPQs, and combined complexity, although exceeding the bounds for conjunctive queries and RPQs, is the same as for relational calculus or for RPQs extended with regular relations. Despite this, RDPQs as we defined them have no chance to lead to a practical language as it is not likely that users will specify a register automaton over data paths. Even for queries such as RPQs and their extensions, conditions are normally specified via regular expressions.

Our goal now is to introduce regular expressions that can be used in place of register automata in data path queries. Note that as long as they express languages accepted by register automata, we shall achieve an NL bound on data complexity by Theorem 3.10.

The first class of queries studied in this section is based on an extension of regular expressions with *memory* that lets us specify when data values are remembered and when they are used. The basic idea is that we can write expressions like $\downarrow x.a^+[x^=]$ saying: store the current data value in x and check that after reading a word from a^+ we see the same data value (condition $x^=$ is true). This will define data paths of the form $da \dots ad$. Such expressions are relatively easy to write and understand (typically easier than automata) and the complexity of their query evaluation will not exceed that of register automata.

Definition 3.11 (*Regular expressions with memory*). Let Σ be a finite alphabet and $x_1, \dots, x_k$ a set of variables. Then *regular expressions with memory (REM)* are defined by the grammar:

$$e \;:=\; \varepsilon \mid a \mid e + e \mid e \cdot e \mid e^+ \mid e[c] \mid \downarrow \overline{x}.e \tag{4}$$

where a ranges over alphabet letters, c over conditions in $\mathcal{C}_k$, and $\overline{x}$ over tuples of variables from $x_1, \dots, x_k$.

A regular expression with memory e is *well-formed* if it satisfies two conditions:

— Subexpressions e^+, $e[c]$, and $\downarrow \overline{x}.e$ are not allowed if e reduces to ε. Formally, e reduces to ε if it is ε, or it is $e_1 + e_2$ or $e_1 \cdot e_2$ or e_1^+ or $e_1[c]$ or $\downarrow \overline{x}.e_1$ where e_1 and e_2 reduce to ε.

— No variable appears in a condition before it appears in $\downarrow\overline{x}$.

From now on we assume that all REMs are well-formed. By $\mathrm{REM}(\Sigma[x_1, \ldots, x_k])$ we denote the set of all well-formed REMs with alphabet Σ and variables $\{x_1, \ldots, x_k\}$.

The extra condition of being well-formed is to rule out pathological cases like $\varepsilon[c]$ for checking conditions over empty subexpressions, or $a[x^{=}]$ for checking equality with a variable that has not been defined.

The intuition behind the expressions is that they process a data path in the same way that the register automaton would, by storing data values in variables, using these variables for comparisons and moving through the word by reading a letter from the finite alphabet. Note that when we bound a variable we do not specify the scope of this binding. This means that the variable can be used at any point after it was bound up to the end of the expression and is analogous to how register automata store and use data values.

Example 3.12. We give two examples of such expressions and languages they recognize, before formally defining their semantics.

(1) The language L_{eq} of data paths in which two data values are the same (see Section 3.1) is given by the expression $\Sigma^* \cdot \downarrow x.\Sigma^+[x^{=}] \cdot \Sigma^*$, where Σ is the shorthand for $a_1 + \ldots + a_l$, whenever $\Sigma = \{a_1, \ldots, a_l\}$ and Σ^* is the shorthand for $\Sigma^+ + \varepsilon$. It says: at some point, bind x, and then check that after one or more edges, we have the same data value.
(2) The language where the finite alphabet part is a sequence of as, and where each data value differs from the first one, while the second value also differs from 5, is given by $\downarrow x \cdot a[5^{\neq} \wedge x^{\neq}] \cdot (a[x^{\neq}])^*$. It starts by binding x to the first data value; then it proceeds checking that the letter is a and condition $5^{\neq} \wedge x^{\neq}$ is satisfied, which is expressed by $a[5^{\neq} \wedge x^{\neq}]$. After that it proceeds to the end by reading letters a and data values different from the first, expressed by $(a[x^{\neq}])^*$.

Semantics of REMs. First, we define the *concatenation* of two data paths $w = d_1 a_1 \ldots a_{n-1} d_n$ and $w' = d_n a_n \ldots a_{m-1} d_m$ as $w \cdot w' = d_1 a_1 \ldots a_{n-1} d_n a_n \ldots a_{m-1} d_m$. Notice that it is only defined if the last data value of w equals the first data value of w'. The definition naturally extends to concatenation of several data paths. If $w = w_1 \cdots w_l$, we shall refer to $w_1 \cdots w_l$ as a *splitting* of a data path w (into $w_1, \ldots, w_l$).

The semantics of REMs is defined by means of a relation $(e, w, \sigma) \vdash \sigma'$, where $e \in \mathrm{REM}(\Sigma[x_1, \ldots, x_k])$ is a regular expression with memory, w is a data path, and both σ and σ' are k-tuples over $\mathcal{D} \cup \{\bot\}$ (the symbol $\bot$ means that a register has not been assigned yet). The intuition is as follows: one can start with a memory configuration σ (i.e., values of $x_1, \ldots, x_k$) and parse w according to e in such a way that at the end the memory configuration is σ'. The language of e is then defined as

$$L(e) = \{w \mid (e, w, \overline{\bot}) \vdash \sigma \text{ for some } \sigma\},$$

where $\overline{\bot}$ is the tuple of k values $\bot$.

The relation $\vdash$ is defined inductively on the structure of expressions. Recall that the empty word corresponds to a data path with a single data value d (i.e., a single node in a data graph). We use the notation $\sigma_{\overline{x}=d}$ for the valuation obtained from σ by setting all the variables in $\overline{x}$ to d.

— $(\varepsilon, w, \sigma) \vdash \sigma'$ iff $w = d$ for some $d \in \mathcal{D}$ and $\sigma' = \sigma$.
— $(a, w, \sigma) \vdash \sigma'$ iff $w = d_1 a d_2$ and $\sigma' = \sigma$.
— $(e_1 \cdot e_2, w, \sigma) \vdash \sigma'$ iff there is a splitting $w = w_1 \cdot w_2$ of w and a valuation σ'' such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$.
— $(e_1 + e_2, w, \sigma) \vdash \sigma'$ iff $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$.

— $(e^+, w, \sigma) \vdash \sigma'$ iff there are a splitting $w = w_1 \cdots w_m$ of w and valuations $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_m = \sigma'$ such that $(e, w_i, \sigma_{i-1}) \vdash \sigma_i$ for all $i \in [m]$.
— $(\downarrow \overline{x}.e, w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma_{\overline{x}=d}) \vdash \sigma'$, where d is the first data value of w.
— $(e[c], w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma) \vdash \sigma'$ and $\sigma', d \models c$, where d is the last data value of w.

Take note that in the last item we require that σ', and not σ, satisfies c. The reason for this is that our initial assignment might change before reaching the end of the expression and we want this change to be reflected when we check that condition c holds.

Translation into automata. We now show that regular expressions with memory can be efficiently translated into register automata.

PROPOSITION 3.13. *For each regular expression with memory $e \in REM(\Sigma[x_1, \ldots, x_k])$ one can construct, in* PTIME*, a k-register data path automaton $\mathcal{A}_e$ such that $L(e) = L(\mathcal{A}_e)$.*

More precisely, the automaton $\mathcal{A}_e = (Q, q_0, F, \overline{\bot}, \delta)$ (over data domain $\mathcal{D} \cup \{\bot\}$) has the property that for any two valuations σ, σ' and a data path w, we have $(e, w, \sigma) \vdash \sigma'$ iff the automaton $(Q, q_0, F, \sigma, \delta)$ has an accepting run on w that ends with the register configuration σ'.

The proof is not difficult but a bit technical. We refer the reader to the Online Appendix for details.

A natural question to ask is do regular expressions with memory define the same class of queries as register automata. As shown in [Libkin and Vrgoč 2012a] this is indeed the case over data words. The proof for data paths is virtually identical to the one presented there. We thus obtain:

COROLLARY 3.14. *Register data path automata and regular expressions with memory define the same class of data path languages.*

Remark 3.15. It is important to note here that, although regular expressions with memory share many syntactic similarities with regular expressions with back-referencing [Aho 1990], they do differ significantly in expressive power and the intended domain. First of all, regular expressions with back-referencing, just as ordinary regular expressions, were designed to operate over finite alphabets and thus can talk only about edge labels in the graph. Over a finite set of graphs, we could use a bounded alphabet to simulate the data values, thus allowing regular expressions with back-referencing to define data path languages, however, they would still not be capable of capturing languages over an arbitrary class of graphs. If regular expressions with back-referencing were extended to function over arbitrary alphabets (using a coding to distinguish between letters from a finite alphabet and data values), they would still fall short of the power of REMs as they lack the ability to test for data value inequality.

3.3.1. Queries based on regular expressions with memory. We now deal with the following class of queries.

Definition 3.16. A *regular query with memory* is an expression $Q = x \xrightarrow{e} y$, where e is a regular expression with memory. Given a data graph G, the result of the query $Q(G)$ consists of pairs of nodes (v, v') such that there is a data path w from v to v' that belongs to $L(e)$. The class of these queries is denoted by RQM.

Example 3.17. To illustrate some interesting queries expressed by RQMs we again turn to the movie database from Figure 1. As in Example 3.8 we will assume that each actor edge has a corresponding stars_in edge going in the other direction.

— To express the query from Example 3.8 returning actors that have a finite Bacon number we can use $Q = x \xrightarrow{e} y$, where e is given by $(\mathsf{stars\_in} \cdot \mathsf{actor})^+[\mathsf{Kevin\ Bacon}^=]$.

— To find movies having at least two different actors staring in them we would use the RQM $Q = x \xrightarrow{e} y$, where e is $\downarrow x.\ \mathsf{actor} \downarrow y.\ \mathsf{stars\_in}\ [x^{=}] \cdot \mathsf{actor}[y^{\neq}]$. Note that here, in addition to the movie we also return one of the actors. The expression first stores the movie name into the variable x and after that moves to one of the actors. Following this it stores the actor's name into y and moves back to the movie using a stars_in edge and checking that it arrived at the same movie by comparing the data value with the one stored into x. Following that the expression simply traverses another actor edge, ensuring it reaches a different actor by comparing the value in the node to y.

We now analyze the complexity of query evaluation for RQMs. Unsurprisingly, we can show that data complexity matches that of register automata.

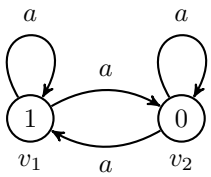
THEOREM 3.18. *The data complexity of RQMs over data graphs is* NL*-complete and the combined complexity of RQMs over data graphs is* PSPACE*-complete.*

PROOF. We first consider data complexity. The NL lower bound is immediate from reachability in graphs. The NL upper bound immediately follows from Corollary 3.14 and Theorem 3.10. Indeed, Corollary 3.14 states that each REM can be translated into some (constant-size) register data path automaton, for which data complexity is in NL according to Theorem 3.10.

We now turn to combined complexity. The PSPACE upper bound follows from Theorem 3.10 and Proposition 3.13. Thus we only have to prove PSPACE-hardness. For this we do a reduction from the non-universality problem for regular automata. The idea is to simulate on the fly reachability testing in the powerset automaton by using two sets of variables, each of the size of the automaton, for coding the current and the next state.

Let $\mathcal{A} = (Q, \Sigma, \delta, q_1, F)$ be a finite state automaton, where $Q = \{q_1, \ldots, q_n\}$ and $F = \{q_{i_1}, \ldots, q_{i_k}\}$. We will construct a fixed graph G with 2 nodes, v_1 and v_2, and a regular expression with memory e, of length $O(n \times |\Sigma|)$, such that $(v_1, v_2) \in Q(G)$ if and only if $L(\mathcal{A}) \neq \Sigma^*$, where $Q = x \xrightarrow{e} y$.

The graph G is shown below:



Since we are trying to demonstrate non-universality of the automaton $\mathcal{A}$ we simulate reachability checking in the powerset automaton for $\overline{\mathcal{A}}$ (that is, the automaton recognizing the complement of the language of the automaton $\mathcal{A}$, which is obtained using the powerset construction). To do so we designate two distinct data values, t and f, and code each state of the powerset automaton as an n-bit sequence of t/f values, where the ith bit of the sequence is set to t if the state q_i is included in our state of $\overline{\mathcal{A}}$. As we are checking reachability we will need only to remember the current and the next state of $\overline{\mathcal{A}}$.

In what follows we will code those two states using variables $s_1, \ldots, s_n$ and $w_1, \ldots, w_n$ and refer to them as the stable tape and the work tape. Our expression e will code data paths that describe successful runs of $\overline{\mathcal{A}}$ by demonstrating how one can move from one state of this automaton to another (as witnessed by their codes in the stable and work tapes), starting with the initial and ending in a final state. The graph G is designed so that it allows us to update the variables as necessary since we can reach any of the two data values in one step.

We will define several expressions and explain their role. We will use two sets of variables, s_1 through s_n and $w_1, \ldots, w_n$ to denote the stable and the work tape (i.e. current and next state in the powerset automaton). All of these variables will only contain two values, t and f, which are bound in the beginning and will correspond to 0 and 1 in the graph G.

The first expression we need is:

$$\texttt{init} := \downarrow t.a[t^{\neq}]\downarrow f.a[t^{=}]\downarrow s_1.a[f^{=}]\downarrow s_2.\ldots.a[f^{=}]\downarrow s_n.a.$$

This expression codes two different values as t and f and initializes the stable tape to contain the encoding of the initial state (the one where only the initial state from $\mathcal{A}$ can be reached). That is, a data path is in the language of this expression if and only if it starts with two different data values, c and d, and continues with n data values that form a sequence in cd^*.

$$\texttt{end} := a[f^{=} \wedge s_{i_1}^{=}] \cdot a[f^{=} \wedge s_{i_2}^{=}] \cdots a[f^{=} \wedge s_{i_k}^{=}], \text{ where } F = \{q_{i_1}, \ldots, q_{i_k}\}.$$

This expression is used to check that we have reached a state not containing any final state from the original automaton. That is, a data path is in $L(\texttt{end})$ if and only if it consists of k data values, all equal to f and where the value stored in s_{i_j} also equals f, for $j = 1 \ldots k$.

Next we define expressions that will reflect updating of the work tape according to the transition function of $\mathcal{A}$. Assume that $\delta(q_i, b) = \{q_{j_1}, \ldots, q_{j_l}\}$. We define

$$u_{\delta(q_i,b)} := \big(a[t^{=} \wedge s_i^{=}] \cdot a[t^{=}]\downarrow w_{j_1}.\ldots.a[t^{=}]\downarrow w_{j_l}.a\big) + a[f^{=} \wedge s_i^{=}].$$

Also, if $\delta(q_i, b) = \emptyset$ we simply put $u_{\delta(q_i,b)} := \varepsilon$.

This expression will be used to update the work tape by writing true to corresponding variables if the state q_i is tagged with t on the work tape (and thus contained in the current state of $\overline{\mathcal{A}}$). If it is false we skip the update.

Since we have to define an update according to all transitions from all the states corresponding to a chosen letter we get:

$$\texttt{update} := \bigvee_{b \in \Sigma} \bigwedge_{q_i \in Q} u_{\delta(q_i,b)}.$$

Here we use $\bigvee$ for $+$ and $\bigwedge$ for the concatenation of expressions. This simply states that we non deterministically pick the next symbol of the word we are guessing and move to the next state accordingly.

We still have to ensure that the tapes are copied at the beginning and end of each step, so we define:

$$\texttt{step} := (a[f^{=}]\downarrow w_1.\ldots.a[f^{=}]\downarrow w_n.a) \cdot \texttt{update} \cdot (a[w_1^{=}]\downarrow s_1.\ldots.a[w_n^{=}]\downarrow s_n.a).$$

This simply initializes the work tape at the beginning of each step, proceeds with the update and copies the new state to the stable tape. Note the few odd a's at the end of the expressions. These will not affect what we want to achieve and are here for syntactical reasons (to get a proper expression).

Finally we have

$$e := \texttt{init} \cdot (\texttt{step})^* \cdot \texttt{end}.$$

Here we use $\texttt{step}^*$ as abbreviation for $\texttt{step}^+ + \varepsilon$.

We claim that for $Q = x \xrightarrow{e} y$, we have $(v_1, v_2) \in Q(G)$ if and only if $L(\mathcal{A}) \neq \Sigma^*$.

Assume first that $L(\mathcal{A}) \neq \Sigma^*$. This means that there is a path from the initial to the final state in the powerset automaton for $\overline{\mathcal{A}}$. That is, there is a word w from Σ^* not in the language of $\mathcal{A}$. This path can in turn be described by pairs of assignments of values t/f to the stable and the work tape, where each transition is witnessed by the corresponding letter of the alphabet. But then the path from v_1 to v_2 in G that belongs to $L(e)$ is the

one that first initializes the stable tape (i.e. the variables $s_1, \ldots, s_n$) to the initial state of the powerset automaton, then runs the updates of the tape according to w and finally ends in a state where all variables corresponding to end states of $\mathcal{A}$ are tagged f. Note that we can describe this path in G, since we start in v_1, store 1 into t, and proceed to v_2 and store 0 into the variable f. After that 1 is assigned to s_1 in v_1 and 0 to $s_2, \ldots, s_n$ by looping through v_2. After that, each transition is reflected by going through v_1 and v_2 as necessary, to update tapes with t/f and finally going to v_2 and looping there to check that all s_i's corresponding to end states are tagged with the value of f.

Conversely, each path from v_1 to v_2 in $L(e)$ corresponds to a run of the powerset automaton for $\overline{\mathcal{A}}$. That is, the part of the path corresponding to `init` sets the initial state. Then the part of this path that corresponds to $\texttt{step}^*$ corresponds to updating our tapes in a way that properly codes one step of the powerset automaton. Finally, `end` denotes that we have reached a state where all end states of $\mathcal{A}$ have been tagged by f, thus, an accepting state for $\overline{\mathcal{A}}$. □

The question is whether we can reduce the PSPACE combined complexity, ideally to PTIME, but at least to NP, to match the combined complexity of conjunctive queries. The following corollary (to the proof of Theorem 3.18) shows that many restrictions will not work.

COROLLARY 3.19. *The combined complexity of evaluating RQM queries remains* PSPACE*-hard for expressions that use at most one* $^+$ *and* $\neq$ *symbol, are specified over a singleton alphabet* $\Sigma = \{a\}$*, and are evaluated over a fixed graph.*

However, we can lower the complexity when we disallow subexpressions of the form e^+, as the following proposition shows.

PROPOSITION 3.20. *The combined complexity of RQM queries whose regular expressions do not have subexpressions of the form* e^+ *is* NP*-complete.*

PROOF. For $e \in \text{REM}(\Sigma[x_1, \ldots, x_k])$, by $\text{Proj}(e)$ we will denote the regular expression obtained by removing symbols not in Σ from e. For instance, for $e = \downarrow x.a[x^=] \cdot b[x^=]$, we have $\text{Proj}(e) = a \cdot b$.

First we show NP-membership. Since we do not use $^+$ we know that every data path in the language of expression e uses at most $|\text{Proj}(e)|$ letters and $|\text{Proj}(e)| + 1$ data values. Assume now that we are given a data graph G, two nodes $s, t \in G$ and an expression with memory e. To see if $(s,t) \in Q(G)$, for $Q = x \xrightarrow{e} y$, we use the following algorithm. First compute the register automaton $\mathcal{A}_e$ for e. Note that this can be done in PTIME. Then nondeterministically guess a data path w_π in G from s to t that is of length at most $|\text{Proj}(e)|$. Now also guess $2|\lambda(w_\pi)| + 1$ states of $\mathcal{A}_e$ and check that the path w_π is accepted by $\mathcal{A}_e$, as witnessed by this sequence of states, and thus is in $L(e)$. It is straightforward to see that this can be done in polynomial time and since our guesses are of polynomial (in fact linear) size we get the desired result.

For hardness we do a reduction from k-CLIQUE. This problem asks, for a given unlabeled undirected graph G and a number k, to determine if G has a clique of size at least k.

Suppose we are given an undirected graph G and a number k. We will construct a data graph G' with $|G| + 2$ nodes, select two nodes $s, t \in G'$ and construct a regular expression with memory e_k of size $O(k^2)$ such that G contains a k-clique if and only if there is a data path from s to t in G' that satisfies e_k.

Take $\Sigma = \{a, b\}$ and make G directed by adding edges in both directions for every edge in G. Label all the edges by a and add two more nodes s and t. Add an edge from s to every other node except s, t and label them with b. Also add an edge from every node in G to t and label them by b. To finish the construction just add a different data value to every node. We call the resulting graph G'.

To define e_k we use an auxiliary expression δ_i defined as:

$$\delta_i := a[x_1^=] \cdot a[x_i^=] \cdot a[x_2^=] \cdot a[x_i^=] \ldots a[x_{i-1}^=] \cdot a[x_i^=].$$

This expression will simply allow us to test that the current node is connected to all nodes previously selected in our potential clique.
Now we can define e_k inductively as follows:

— $e_1 := b \cdot \downarrow x_1.a[x_1^{\neq}]$,
— $e_2 := e_1 \cdot \downarrow x_2.a[x_1^{\neq} \wedge x_2^{\neq}]$,
— $e_i := e_{i-1} \cdot \downarrow x_i.\delta_i \cdot a[x_1^{\neq} \wedge \ldots \wedge x_i^{\neq}]$, for $i = 3, \ldots, k-1$ and
— $e_k := e_{k-1} \cdot \downarrow x_k.\delta_k \cdot b$.

Next we show that there is a k-clique in G iff there is a data path from s to t in G' that satisfies e_k.

Suppose first that there is a k-clique in G. Then we simply move from s to an arbitrary point in that clique using the b labeled edge and traverse the clique back and forth until we reach the k-th element of the clique. Note that starting from the third element, whenever we select a different node in the clique we have to move back and forth between this node and all previously selected ones to satisfy δ_i, but since we have a clique this is possible. Finally, after selecting the last node and verifying that it is connected to all the others we move to t using a b labeled edge.

Now suppose that there is a data path from s to t in G' that satisfies e_k. This means that we will be able to select k different nodes $n_1, \ldots, n_k$ in G with data values stored in $x_1, \ldots, x_k$. Since all data values in the graph are different they also act as ids. Now take any two n_i, n_j with $i < j \leq k$. Then we know that n_i and n_j are connected in G because after selecting n_j we have to go through δ_j which contains $a[x_i^=] \cdot a[x_j^=]$ and since no two data values in G are the same this means that we have an edge between n_i and n_j. This completes the proof. □

Although the restriction from Proposition 3.20 achieves better combined complexity, it is too strong, as it effectively restricts one to languages of data paths whose projections on Σ^* are finite. All the examples we saw earlier use subexpressions of the form e^+. It seems that, if we want to achieve tractability, we need to look at a very different way of restricting expressions. This is what we do in the next section.

3.4. Queries based on regular expressions with equality

We present a class of regular expressions for data paths that lets us lower the combined complexity of queries to PTIME. It permits testing for equality or inequality of data values at the beginning or the end of a data (sub)path. For example, $(\Sigma^+)_{\neq}$ denotes the set of all data paths having different first and last data values. The language L_{eq} of data paths on which two data values are the same is given by $\Sigma^* \cdot (\Sigma^+)_= \cdot \Sigma^*$: it checks for the existence of a nonempty subpath (with label in Σ^+) such that the nodes at the beginning and at the end of this subpath have the same data value, indicated by subscript $=$.

To allow for constants we will use *simplified conditions*. These are simply conjunctions of expressions of the form $z_=$ and $z_{\neq}$, where z ranges over $\mathcal{D}$. Then a data value d satisfies a simplified condition c, denoted $d \models c$, if $\tau, d \models c$, where τ is an empty assignment. Note that the valuation itself is irrelevant here.

Definition 3.21 (*Expressions with equality*). Let Σ be a finite alphabet. Then *regular expressions with equality (REE)* are defined by the grammar:

$$e \;:=\; \varepsilon \mid a \mid e + e \mid e \cdot e \mid e^+ \mid e[c] \mid e_= \mid e_{\neq} \tag{5}$$

where a ranges over alphabet letters and c is a simplified condition.

The language $L(e)$ of data paths denoted by a regular expression with equality e is defined as follows.

— $L(\varepsilon) = \{d \mid d \in \mathcal{D}\}$.
— $L(a) = \{dad' \mid d, d' \in \mathcal{D}\}$.
— $L(e \cdot e') = L(e) \cdot L(e')$.
— $L(e + e') = L(e) \cup L(e')$.
— $L(e^+) = \{w_1 \cdots w_k \mid k \geq 1 \text{ and each } w_i \in L(e)\}$.
— $L(e[c]) = \{d_1 a_1 \dots a_{n-1} d_n \in L(e) \mid d_n \models c\}$.
— $L(e_=) = \{d_1 a_1 \dots a_{n-1} d_n \in L(e) \mid d_1 = d_n\}$.
— $L(e_{\neq}) = \{d_1 a_1 \dots a_{n-1} d_n \in L(e) \mid d_1 \neq d_n\}$.

Regular expressions with equality sacrifice the ability to store and compare data values at arbitrary places in the expression and can instead only check for (in)equality at the start and the end of chosen subexpressions. (Comparisons against constants, though, can be made everywhere.) Looking at Example 3.12, the first language can be defined by regular expressions with equality, but the second one cannot. We already saw how to do the language L_{eq}; the expression $\downarrow x.(ab)^+[x^{\neq}]$ is equivalent to $(ab)^+_{\neq}$. A simpler version of the second language would be $\downarrow x.(a[x^{\neq}])^+$, describing the language of data paths in which all data values are different from the first one. It requires checking a condition multiple times. We now show that this goes beyond the power of expressions with equality, which are strictly weaker than expressions with memory.

Proposition 3.22.

(1) *For each regular expression with equality, there is an equivalent regular expression with memory.*
(2) *For the regular expression with memory $\downarrow x.(a[x^{\neq}])^+$ there is no equivalent regular expression with equality.*

Proof. For the first item it is enough to observe that for expressions of the kind $e_=$ and $e_{\neq}$, where e is an ordinary regular expression, the expressions with memory $\downarrow x.e[x^{=}]$ and $\downarrow x.e[x^{\neq}]$ denote the same language of data paths. From this it is straightforward to construct a translation of an arbitrary regular expression with equality e to a regular expression with memory by doing the above-mentioned construction bottom-up, starting from subexpressions of e and using a new variable for each subexpression of the form $e'_=$ or $e'_{\neq}$.

To prove the second claim we introduce a new kind of automata, called weak register automata, show that they capture regular expressions with equality and that they can not express the language $\downarrow x.(a[x^{\neq}])^+$ of a-labeled data paths on which all data values are different from the first one.

The main idea behind weak register automata is that they erase the data value that was stored in the register once they make a comparison, thus rendering the register empty. We denote this by putting a special symbol $\perp$ from $\mathcal{D}$ in the register. Since they have a finite number of registers, they can keep track of only finitely many positions in the future, so in the case of our language, they can only check that a fixed finite number of data values is different from the first one. We proceed with formal definitions.

The definition of weak k-register data path automaton is the same as in Definition 3.5. The only explicit change we make is that we now assume that $\mathcal{C}_k$ contains a special symbol ε, that will allow us to simply skip the data value, without doing any comparisons. (Notice that previously we have been using a simple tautology such as $x_1^{=} \vee x_1^{\neq}$, or an additional register to emulate this. Here, emulation is not possible because those registers would be

reset.) Thus we simply add $\tau, d \models \varepsilon$, for every valuation τ and data value d, to the semantics of $\mathcal{C}_k$. We will also assume that the initial configuration is always empty.

The definition of configurations remains the same as before, but the way we move from one configuration to another changes.

From a configuration $c = (j, q, \tau)$ we can move to a configuration $c' = (j+1, q', \tau')$ if one of the following holds:

— the jth symbol is a letter a, there is a transition $(q, a, q') \in \delta_w$, and $\tau' = \tau$; or
— the current symbol is a data value d and there is a transition $(q, c, I, q') \in \delta_d$ such that $d, \tau \models c$ and τ' coincides with τ except that
  — the ith component of τ' is set to d whenever $i \in I$ and
  — every register that is not in I but mentioned in c is set to be empty (i.e. to contain $\perp$).

The second item simply tells us that if we used a condition like $c = x_3^{=} \wedge x_7^{\neq}$ in our transition, we would afterward erase data values that were stored in registers 3 and 7. Note that we can immediately rewrite these registers with the current data value.

The notion of acceptance and an accepting run is the same as before.

We now show that weak register automata can not recognize the language L of all data paths where the first data value is different from all other data values, i.e. the language denoted by the expression $\downarrow x.(a[x^{\neq}])^+$.

Assume to the contrary, that there is some weak k-register data path automaton $\mathcal{A}$ recognizing L. Since data path $w_\pi = d_1 a d_2 a \dots d_k a d_{k+1} a d_{k+2}$, where the d_is are pairwise different and do not appear in any condition in $\mathcal{A}$, belongs to L, there is an accepting run of $\mathcal{A}$ on w_π. The idea behind the proof is that $\mathcal{A}$ can only check that the first $k+1$ positions have different data values from the first one.

First we note a few things. Since every data value in the path w_π is different, no $=$ comparisons can be used in conditions appearing in this run (otherwise the condition test would fail and the automaton would not accept). This must also hold for constants appearing in the conditions, since no d_is appear in them.

Now note that since we have only k registers, and with every comparison we empty the corresponding registers one of the following must occur:

— There is a position $1 < i < k+2$ such that the condition used when processing data value d_i is ε. In this case we simply replace d_i by d_1 and get an accepting run on a word that has the first data value repeated – a contradiction. Note that we could store d_i in that transition, but since afterward we only test for inequality this will not alter the rest of the computation.
— There is a data value such that, when the automaton reads it, it does not use any register with the first data value, i.e. d_1, stored. Note that this must happen, because at best we can store the first data value in all the registers at the beginning of our run, but after that each time we read a data value and compare it to the first we lose the first data value in this register. But then again we can simply replace this data value with d_1 and get an accepting run (just as before, if this data value gets stored in this transition and then used later it can only be used in a $\neq$ comparison, which is also true for d_1, so the run remains accepting). Again we arrive at a contradiction.

This shows that no weak register automaton can recognize the language L.

To complete the proof of Proposition 3.22 we still have to show the following:

LEMMA 3.23. *For every regular expression with equality e there exists a weak k-register automaton $\mathcal{A}_e$, recognizing the same language of data paths, where k is the number of times $=, \neq$ symbols appear in e.*

The proof of the lemma is almost identical to the proof of Proposition 3.13. We can view this as introducing a new variable for every $=, \neq$ comparison in e and act as the subexpression $e'_=$ reads $\downarrow x.e'[x^=]$ and analogously for $\neq$. Note that in this case all variables come with their scope, so we do not have to worry about transferring register configurations from one side of the construction to another (for example when we do concatenation). The underlying automata remain the same. □

3.4.1. Queries based on Regular expressions with equality. We now deal with the following queries.

Definition 3.24. A *regular query with data tests* is an expression $Q = x \xrightarrow{e} y$, where e is a regular expression with equality. Given a data graph G, the result of the query $Q(G)$ consists of pairs of nodes (v, v') such that there is a data path w from v to v' that belongs to $L(e)$. The class of these queries is denoted by RQD.

Example 3.25. Coming back to the database from Figure 1, we can now ask the following queries.

— The query asking for people with a finite Bacon number is again the same as in Example 3.17.
— The query that checks if there is a movie in the database with at least two different actors is defined by $Q = x \xrightarrow{e} y$, with $e := (\mathsf{stars\_in} \cdot \mathsf{actor})_{\neq}$. Note that a nonempty answer to this query merely signifies that such a movie exists. To actually retrieve the movie we would need to use conjunctive queries with RQDs as atoms (Section 5.1).

THEOREM 3.26. *The data complexity of RQDs over data graphs is* NL*-complete and the combined complexity of RQDs over data graphs is in* PTIME.

PROOF. For data complexity, the proof is analogous to the one in Theorem 3.18. The bound for combined complexity follows from Theorem 4.3 which is shown for a fragment of the GXPath language which is strictly stronger. □

4. GRAPH LANGUAGES

In this section we explore querying beyond paths and investigate graph languages. We start from the idea of adapting the well known XML language XPath to the graph setting. XPath has many desirable properties such as the ability to define patterns that can not be captured by paths, close connections to first-order logic, and efficient evaluation algorithms. Here we show how to extend the language to operate over graph databases while still retaining these properties. Since XPath is already a well-established language in practice, we believe that this adaptation has excellent potential to become a widely used formalism for querying graph data.

4.1. The language GXPath and its many variants

We follow the standard way of defining XPath fragments [Bojanczyk and Parys 2011; Calvanese et al. 2009; Figueira 2010; Gottlob et al. 2005; Marx 2005; ten Cate and Marx 2007] and introduce some variants of *graph XPath*, or GXPath, to be interpreted over data graphs. As usual, XPath formulae are divided into *path formulae*, producing sets of pairs of nodes, and *node tests*, producing sets of nodes. Path formulae will be denoted by letters from the beginning of the Greek alphabet $(\alpha, \beta, \ldots)$ and node formulae by letters from the end of the Greek alphabet $(\varphi, \psi, \ldots)$.

Since we deal with data values, we need to define *data tests* permitted in our formulas. There will be three kinds of them.

(1) *Constant tests*: For each data value $c \in \mathcal{D}$, we have two tests $c^{=}$ and $c^{\neq}$. The intended meaning is to test whether the data value in the current node is equal to, or differs from, a constant c. The fragment of GXPath that uses constant tests will be denoted by GXPath(c).
(2) *Equality/inequality tests*: These are typical XPath (in)equality tests of the form $\langle \alpha = \beta \rangle$ and $\langle \alpha \neq \beta \rangle$, where α and β are path expressions. The intended meaning is to check for the existence of two paths, one satisfying α and the other satisfying β, which end with equal (resp., different) data values. The appropriate fragment will be denoted by GXPath(eq).
(3) *Subexpression tests*: These are used to test if a path or a subpath starts and ends with the same or different data value. The fragment in question is obtained by adding $\alpha_{=}$ and $\alpha_{\neq}$ to path expressions of our language. These tests will be needed to provide a logical kernel for GXPath. The corresponding fragment is denoted GXPath($\sim$).

We also consider languages that combine the above features, which will be denoted by simply listing the features. For example, GXPath(c, eq) uses constant- and equality/inequality tests.

Next we define expressions of GXPath. We look at several versions, inspired by *Core XPath* and *Regular XPath* [Marx 2005]. They both have node and path expressions. Node expressions in all fragments are given by the grammar:

$$\varphi, \psi := \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle \alpha \rangle$$

where α is a path expression.

The path formulae of the two flavors of GXPath are given below. In both cases a ranges over Σ. Path expressions of *Regular graph XPath*, denoted by $\mathsf{GXPath}_{\mathrm{reg}}$, are given by:

$$\alpha, \beta := \varepsilon \mid \_ \mid a \mid a^{-} \mid [\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha} \mid \alpha^{*}$$

Path expressions of *Core graph XPath* denoted by $\mathsf{GXPath}_{\mathrm{core}}$ are given by:

$$\alpha, \beta := \varepsilon \mid \_ \mid a \mid a^{-} \mid a^{*} \mid a^{-^{*}} \mid [\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha}$$

We call this fragment "Core graph XPath", since it is natural to view edge labels (and their reverses) in data graphs as the single-step axes of the usual XPath on trees. For instance, a and a^{-} can correspond to "child" and "parent", respectively. Thus, in the core fragment, we only allow transitive closure over navigational single-step axes, as is done in Core XPath on trees. Note that we did not explicitly define the counterpart of node label tests in GXPath node expressions to avoid notational clutter, but all the results remain true if we add them.

Finally, we consider another feature that was recently proposed in the context of navigational languages on graphs (such as in SPARQL 1.1 property paths [Harris and Seaborne 2013] and in Neo4J's *Cypher* patterns [Neo4j 2013]), namely counters. The idea is to extend all grammars defining path formulae with new path expressions

$$\alpha^{n,m}$$

for $n, m \in \mathbb{N}$ and $n < m$. Informally, this means that we have a path that consists of some k chunks, each satisfying α, with $n \leq k \leq m$. When counting is present in the language, we denote it by #GXPath, e.g., $\#\mathsf{GXPath}_{\mathrm{core}}$.

Given these path and node formulae, we can combine $\mathsf{GXPath}_{\mathrm{core}}$ and $\mathsf{GXPath}_{\mathrm{reg}}$ with different flavors of data tests or counting, starting with purely navigational fragments (neither c, eq, nor $\sim$ tests are allowed) and up to fragments allowing any combination of such tests. For example, $\#\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c}, \mathsf{eq})$ is defined by mutual recursion as follows:

$$\alpha, \beta := \varepsilon \mid \_ \mid a \mid a^{-} \mid [\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha} \mid \alpha^{*} \mid \alpha^{n,m}$$
$$\varphi, \psi := \neg\varphi \mid \varphi \wedge \psi \mid \langle \alpha \rangle \mid c^{=} \mid c^{\neq} \mid \langle \alpha = \beta \rangle \mid \langle \alpha \neq \beta \rangle$$

Path expressions (denoted α, β)

$$
\begin{aligned}
[\![\varepsilon]\!]^G &= \{(v,v) \mid v \in V\} \\
[\![\_]\!]^G &= \{(v,v') \mid (v,a,v') \in E \text{ for some } a\} \\
[\![a]\!]^G &= \{(v,v') \mid (v,a,v') \in E\} \\
[\![a^-]\!]^G &= \{(v,v') \mid (v',a,v) \in E\} \\
[\![\alpha^*]\!]^G &= \text{the reflexive transitive closure of } [\![\alpha]\!]^G \\
[\![\alpha \cdot \beta]\!]^G &= [\![\alpha]\!]^G \circ [\![\beta]\!]^G \\
[\![\alpha \cup \beta]\!]^G &= [\![\alpha]\!]^G \cup [\![\beta]\!]^G \\
[\![\overline{\alpha}]\!]^G &= V \times V - [\![\alpha]\!]^G \\
[\![[\varphi]]\!]^G &= \{(v,v) \mid v \in [\![\varphi]\!]^G\} \\
[\![\alpha^{n,m}]\!]^G &= \bigcup_{k=n}^{m} ([\![\alpha]\!]^G)^k \\
[\![\alpha_=]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) = \rho(v')\} \\
[\![\alpha_{\neq}]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) \neq \rho(v')\}
\end{aligned}
$$

Node tests (denoted φ, ψ)

$$
\begin{aligned}
[\![\langle\alpha\rangle]\!]^G &= \pi_1([\![\alpha]\!]^G) = \{v \mid \exists v' \ (v,v') \in [\![\alpha]\!]^G\} \\
[\![\top]\!]^G &= V \\
[\![\neg\varphi]\!]^G &= V - [\![\varphi]\!]^G \\
[\![\varphi \wedge \psi]\!]^G &= [\![\varphi]\!]^G \cap [\![\psi]\!]^G \\
[\![\varphi \vee \psi]\!]^G &= [\![\varphi]\!]^G \cup [\![\psi]\!]^G \\
[\![c^=]\!]^G &= \{v \in V \mid \rho(v) = c\} \\
[\![c^{\neq}]\!]^G &= \{v \in V \mid \rho(v) \neq c\} \\
[\![\langle\alpha = \beta\rangle]\!]^G &= \{v \in V \mid \exists v', v'' \ (v,v') \in [\![\alpha]\!]^G, \ (v,v'') \in [\![\beta]\!]^G, \rho(v') = \rho(v'')\} \\
[\![\langle\alpha \neq \beta\rangle]\!]^G &= \{v \in V \mid \exists v', v'' \ (v,v') \in [\![\alpha]\!]^G, \ (v,v'') \in [\![\beta]\!]^G, \rho(v') \neq \rho(v'')\}
\end{aligned}
$$

Fig. 3. Semantics of Graph XPath expressions with respect to $G = \langle V, E, \rho\rangle$

with c ranging over constants.

We define the semantics with respect to a data graph $G = \langle V, E, \rho\rangle$. The semantics $[\![\alpha]\!]^G$ of a path expression α is a set of pairs of vertices and the semantics of a node test $[\![\varphi]\!]^G$ is a set of vertices. The definitions are given in Figure 3. In that definition we denote by R^k the k-fold composition of a binary relation R, i.e., $R \circ R \circ \ldots \circ R$, with R occurring k times.

Remark. Note that each path expression α can be transformed into a node test by means of the $\langle\alpha\rangle$ operator. In particular, we can test if a node has a b-successor by writing, for instance, $\langle b\rangle$. To reduce the clutter when using such tests in path expressions, we shall often omit the $\langle\rangle$ braces and write e.g. $a[b]$ instead of $a[\langle b\rangle]$.

Basic expressiveness results. Some expressions are readily definable with those we have. For instance, Boolean operations $\alpha \cap \beta$ and $\alpha - \beta$ with the natural semantics are definable using union and complement. Indeed, $\alpha - \beta$ is equivalent to $\overline{\overline{\alpha} \cup \beta}$, and $\alpha \cap \beta$ is equivalent to $\overline{\overline{\alpha} \cup \overline{\beta}}$. So when necessary, we shall use intersection and set difference in path expressions.

Counting expressions $\alpha^{n,m}$ are definable too: they abbreviate $\alpha \cdots \alpha \cdot (\alpha \cup \varepsilon) \cdots (\alpha \cup \varepsilon)$, where we have a concatenation of n times α and $m - n$ times $(\alpha \cup \varepsilon)$. Thus, adding counters does not influence expressivity of any of the fragments, since we always allow concatenation and union. However, counting expressions can be exponentially more succinct than their smallest equivalent regular expressions (independent of whether n and m are represented in binary or in unary) [Losemann and Martens 2012]. We will exhibit a query evaluation algo-

rithm with polynomial-time complexity even for such expressions with counters represented in binary.

As another observation on the expressiveness of the language, note that we can define a test $\langle \alpha = c \rangle$, with the semantics $\{v \mid \exists v' \ (v, v') \in [\![\alpha]\!]^G$ and $\rho(v') = c\}$, by using the expression $\langle \alpha[c^{=}] \rangle$.

Finally, node expressions can be defined in terms of path operators. For example $\varphi \wedge \psi$ is defined by the expression $\langle [\varphi] \cdot [\psi] \rangle$, while $\neg\varphi$ is defined by $\langle \overline{[\varphi]} \cap \varepsilon \rangle$.

Example 4.1. To illustrate some more involved queries we come back to our introductory example of a movie database presented in Figure 1.

(1) To find people who do not have a finite Bacon number we use the query

$$e_1 = \neg\langle(\mathsf{actor}^- \cdot \mathsf{actor})^*[\mathsf{Kevin\ Bacon}^{=}]\rangle.$$

(The query within the negation simply returns all people with a finite Bacon number.)

(2) We can also ask for people with a finite Bacon number such that collaboration is always established by co-starring in movies and not documentaries:

$$e_2 = \langle(\mathsf{actor}^-[\mathsf{type}[\mathsf{Movie}^{=}]] \cdot \mathsf{actor})^*[\mathsf{Kevin\ Bacon}^{=}]\rangle.$$

This expression works in a similar way as the one for finding the Bacon number, but using the nesting capabilities of GXPath it also checks that the actors appear in a movie.

(3) One might also be interested to find out if there are actors who have a finite Bacon number and the same age as Kevin Bacon. They can be retrieved using the following query:

$$e_3 = \langle \mathsf{age}(\mathsf{age}^-(\mathsf{actor}^- \cdot \mathsf{actor})^*[\mathsf{Kevin\ Bacon}^{=}]\mathsf{age})_{=}\rangle.$$

(4) As a last example we might want to check if a movie or a documentary has at least two actors starring in it. Such a query is defined by:

$$e_4 = \langle \mathsf{actor} \neq \mathsf{actor}\rangle.$$

Here we simply check if there are two actor edges leading from the movie such that the actors names are different.

Complement and positive fragments. In standard XPath dialects on trees, the binary complementation operator is not included and one usually shows that languages are closed under negation [Marx 2005]. This is no longer true for arbitrary graphs, due to the following.

PROPOSITION 4.2. *Path complementation $\overline{\alpha}$ is not definable in GXPath$_{reg}$ without complement on path expressions.*

The proof is an immediate consequence of the following observation. Given a data graph G, let $V_1, \ldots, V_m$ be the sets of nodes of its (maximal) connected components (with respect to the edge relation $\bigcup_{a\in\Sigma} E_a$). Here we compute the connected components with respect to the undirected graph induced by the edge relation $\bigcup_{a\in\Sigma} E_a$. Namely, we treat each E_a as if it were undirected. Then a simple induction on the structure of the expressions of $\mathsf{GXPath}_{\mathrm{reg}}$ without complement on path expressions shows that for each expression α, we have $[\![\alpha]\!]^G \subseteq \bigcup_{i\leq m} V_i \times V_i$. However, path complementation $\overline{\alpha}$ clearly violates this property.

In what follows, we consider fragments of our languages that restrict complementation and negation. There are two kinds of them, the first corresponding to the well-studied notion of positive XPath [Benedikt et al. 2008].

— The *positive fragments* are obtained by removing $\neg\varphi$ and $\overline{\alpha}$ from the definitions of node and path formulae. We use the superscript pos to denote them, i.e., we write $\mathsf{GXPath}^{\mathrm{pos}}_{\mathrm{core}}$ and $\mathsf{GXPath}^{\mathrm{pos}}_{\mathrm{reg}}$.

— The *path-positive fragments* are obtained by removing $\overline{\alpha}$ from the definitions of path formulae, but keeping $\neg\varphi$ in the definitions of node formulae. We use the superscript path-pos to denote them, i.e., we write $\mathsf{GXPath}_{\text{core}}^{\text{path-pos}}$ and $\mathsf{GXPath}_{\text{reg}}^{\text{path-pos}}$.

There are two main reasons why we consider these fragments that restrict complementation and negation. First of all, these fragments are the most studied fragments of XPath, so they allow us to better compare results on XPath with GXPath. Second, the path-positive fragments are the largest fragments for which we can prove linear-time query evaluation.

4.2. Query evaluation

In this section we investigate the complexity of querying graph databases using variants of GXPath. We consider two problems. One is Query Evaluation, which is essentially model checking: we have a graph database, a query (i.e., a GXPath path expression), and a pair of nodes, and we want to check if the pair of nodes is in the query result. That is, we deal with the following decision problem.

Problem:	Query Evaluation
Input:	A graph $G = (V, E)$, a GXPath path expression α, nodes $v, v' \in V$.
Question:	Is $(v, v') \in [\![\alpha]\!]^G$?

The second problem we consider is Query Computation, which actually computes the result of a query and outputs it. Normally, when one deals with path expressions, one fixes a so-called *context node* v and looks for all nodes v' such that (v, v') satisfies the expression (see, e.g., [Pérez et al. 2010]). We deal with a slightly more general version here, where there can be a set of context nodes instead of just a single one.

Problem:	Query Computation
Input:	A graph $G = (V, E)$, a GXPath path expression α, and a set of nodes $S \subseteq V$.
Output:	All $v' \in V$ such that there exists a $v \in S$ with $(v, v') \in [\![\alpha]\!]^G$.

Note that in both problems we deal with *combined complexity*, as the query is a part of the input. Furthermore, notice that in Query Computation as we define it, does not ask to compute which nodes in the output correspond to which nodes in the input. The present definition of Query Computation, however, allows for a linear-time algorithm for $\mathsf{GXPath}_{\text{reg}}^{\text{path-pos}}$ (Fact 4.4). This would no longer be true if we would require to compute the (possibly quadratic-size) relation between input nodes and output nodes.

Unlike in the previous section, here we are not interested in describing complexity in terms of complexity classes. This is because both problems are already tractable even for the most expressive variant of the language. Instead we concentrate on establishing running time for algorithms for query evaluation and computation problems. For this, we need to explain how exactly we measure the size of the input. The size $|G|$ of a data graph G is defined as $|V| + |E|$, where $|V|$ is its number of nodes and $|E|$ its number of edges. We denote the sizes of path expressions α and node expressions φ by $|\alpha|$, resp., $|\varphi|$ and define them to be the number of symbols in α, resp., φ (equivalently, we could define them as the sizes of parse trees of those expressions). The size of a counter is the number of bits representing it, so $|\alpha^{n,m}| = |\alpha| + \log n + \log m$.

The main result of this section is that the combined complexity remains in polynomial time for all fragments we defined in Section 4.1. Not only that, but the exponents are low,

ranging from linear to cubic. Notice that for navigational fragments, the low (and even linear) complexity should not come as a surprise. Note that $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$ is essentially Propositional Dynamic Logic (PDL), for which global model checking is known to have linear-time complexity [Alechina and Immerman 2000; Cleaveland and Steffen 1993]. Also, polynomial-time combined complexity results are known for pure navigational $\mathsf{GXPath}_{\mathrm{reg}}$ from the PDL perspective as well [Lange 2006].

Our main contribution is thus to establish the low combined complexity bounds for fragments that handle two new features we added on top of navigational languages: *data value comparisons* and *counters*. The former increases expressiveness; the latter does not, but it can make expressions exponentially more succinct. Thus, some work is needed to keep combined complexity polynomial when counters are added.

We first present a general upper bound that shows that combined complexity of both problems is polynomial for the most expressive language we have: regular graph XPath with counting, constant tests, equality tests and subexpression tests. In the algorithm analysis, we assume that we can test equality between data values in constant time. Moreover, we assume that we have an ordering on the nodes of G. (Such an ordering can be obtained by performing a single depth-first traversal through G and annotating each node with the order in which it is visited. This costs linear time in $|G|$. A reverse index that points us to a node, given its number, can be constructed at the same time.)

THEOREM 4.3. *Both* QUERY EVALUATION *and* QUERY COMPUTATION *for* $\#\mathsf{GXPath}_{reg}(\mathsf{c}, \mathsf{eq}, \sim)$ *can be solved in polynomial time, specifically,* $O(|\alpha| \cdot |V|^3)$.

PROOF. Both problems can be solved in the required time by a dynamic programming algorithm that processes the parse tree of α in bottom-up fashion and computes, for every path subexpression β of α, the binary relation $[\![\beta]\!]^G$. Similarly, we compute, for every node subexpression φ of α, the set $[\![\varphi]\!]^G$. Clearly, if each such relation can be computed within time $O(|V|^3)$ (using previously computed relations), both problems can be solved within the required time. We make one exception: we allow $O(|V|^3 \log m)$ time for computing $[\![\beta^{n,m}]\!]^G$ from $[\![\beta]\!]^G$. This is not problematic, since the size of $\beta^{n,m}$ is $O(|\beta| + |\log m|)$.

We discuss how to obtain the desired time bound. The algorithm is similar to an algorithm used for evaluating regular expressions with counters on graphs (Theorem 3.4 in [Losemann and Martens 2012]).

The base cases for path expressions, that is, computing $[\![\beta]\!]^G$ where β is one of ε, $\_$, a, or a^-, are trivial. Similarly, the base cases for node expressions, that is, computing $[\![\varphi]\!]^G$ where φ is either $\top$, $c^=$, or $^{\neq}$ are trivial as well.

For the induction step we need to consider path expressions of the form $[\varphi]$, $\beta_1 \cdot \beta_2$, $\beta_1 \cup \beta_2$, $\overline{\beta}$, β^*, $\beta^{n,m}$, $\beta_=$, and $\beta_{\neq}$. Also, we need to consider node expressions of the form $\neg\varphi$, $\varphi \wedge \psi$, $\langle\beta\rangle$, $\langle\beta_1 = \beta_2\rangle$, and $\langle\beta_1 \neq \beta_2\rangle$.

In the case of path expressions, the cases $[\varphi]$, $\beta_1 \cup \beta_2$, $\beta_=$, and $\beta_{\neq}$ are trivial because $[\![\varphi]\!]^G$ contains at most $|V|$ elements and $[\![\beta]\!]^G$ at most $|V|^2$ pairs. For example, for $\beta_=$ we can iterate through $[\![\beta]\!]^G$, testing each of its pairs (u, v) and putting a pair in $[\![\beta_=]\!]^G$ if and only if $\rho(u) = \rho(v)$. For $\beta_1 \cup \beta_2$ we can first sort both relations $[\![\beta_1]\!]^G$ and $[\![\beta_2]\!]^G$ (costing $O(|V|^2 \log |V|)$ time) and then compute $[\![\beta_1 \cup \beta_2]\!]^G$ while performing a single pass over $[\![\beta_1]\!]^G$ and $[\![\beta_2]\!]^G$. For $\beta_1 \cdot \beta_2$ the relation $[\![\beta_1 \cdot \beta_2]\!]^G$ is the composition $[\![\beta_1]\!]^G \circ [\![\beta_2]\!]^G$, which can be obtained by computing the natural join of $[\![\beta_1]\!]^G$ with $[\![\beta_2]\!]^G$.

Computing $[\![\beta^*]\!]^G$ amounts to computing the reflexive-transitive closure of $[\![\beta]\!]^G$ which can be done in time $|V|^3$ by Warshall's algorithm. Computing $[\![\beta^{n,m}]\!]$ within time $O(|V|^3 \log m)$ can be done by fast squaring, as was done in Theorem 3.4 in [Losemann and Martens 2012]. Indeed, computing $[\![\beta^2]\!]^G$, given $[\![\beta]\!]^G$, takes time $O(|V|^3)$. Computing $[\![\beta^n]\!]^G$ by fast squaring means recursively computing the composition $[\![\beta^{n/2}]\!]^G \circ [\![\beta^{n/2}]\!]^G$ if n is even and $[\![\beta^{(n-1)/2}]\!]^G \circ [\![\beta^{(n-1)/2}]\!]^G \circ [\![\beta]\!]^G$ if n is odd. Notice that we only need to compute $[\![\beta^{n/2}]\!]^G$

once for computing $[\![\beta^{n/2}]\!]^G \circ [\![\beta^{n/2}]\!]^G$ (and likewise if n is odd). Using fast squaring, we need $O(\log n)$ such operations to compute $[\![\beta^n]\!]^G$. Extending this to $[\![\beta^{n,m}]\!]^G$ is straightforward. The case $[\![\overline{\beta}]\!]^G$ can be solved by first sorting the pairs from $[\![\beta]\!]^G$ and then performing a single pass over the sorted relation, which costs $O(|V|^2 \log |V|)$ time.

In the case of node expressions the most interesting cases are $\langle \beta_1 = \beta_2 \rangle$ and $\langle \beta_1 \neq \beta_2 \rangle$. Computing $[\![\langle \beta_1 = \beta_2 \rangle]\!]^G$ and $[\![\langle \beta_1 \neq \beta_2 \rangle]\!]^G$ from $[\![\beta_1]\!]^G$ and $[\![\beta_2]\!]^G$ in time $O(|V|^3)$ can be done as follows. For $\langle \beta_1 = \beta_2 \rangle$ we need to search if there exist $(v, v_1) \in [\![\beta_1]\!]^G$ and $(v, v_2) \in [\![\beta_2]\!]^G$ such that $\rho(v_1) = \rho(v_2)$. This can be tested in time $O(|V|^3)$ similarly to how one performs a sort-merge join. First, sort relations β_1 and β_2 on the left attribute, which costs time $O(|V|^2 \log |V|)$. Then, for each of the $|V|$ possible values v of the join attribute (in increasing order), we can compute in time $O(|V|)$ the sets $D_{v,1} = \{\rho(v_1) \mid (v, v_1) \in [\![\beta_1]\!]^G\}$ and $D_{v,2} = \{\rho(v_2) \mid (v, v_2) \in [\![\beta_2]\!]^G\}$. Since both $D_{v,1}$ and $D_{v,2}$ have at most $|V|$ elements, it can be tested in time $O(|V|^2)$ if they have a common data value. The result $[\![\langle \beta_1 = \beta_2 \rangle]\!]^G$ contains all v such that $D_{v,1} \cap D_{v,2} \neq \emptyset$ and can therefore be computed in time $O(|V|^3)$. The case $\langle \beta_1 \neq \beta_2 \rangle$ is similar. □

The algorithm for Theorem 4.3 uses cubic time in $|V|$ because it computes the relations $[\![\beta]\!]^G$ for larger and larger subexpressions β of the given input expression. In particular, the algorithm uses steps that are at least as difficult as multiplication of $|V| \times |V|$ matrices or computing the reflexive-transitive closure of a graph with $|V|$ nodes. However, if one can avoid computing the relations $[\![\beta]\!]^G$ for subexpressions β, the time bound can be improved, as we will see next.

For the remainder of the section, we assume that there is an ordering on labels of edges and that graphs are represented as adjacency lists such that we can obtain, for a given node v, the outgoing edges or the incoming edges, sorted in increasing order of labels, in constant time. (We note that the linear-time algorithm from [Alechina and Immerman 2000] for PDL model checking also assumes that adjacency lists are sorted.) The following result is immediate from PDL model checking techniques:

FACT 4.4. *Both* QUERY EVALUATION *and* QUERY COMPUTATION *for* $\mathsf{GXPath}_{reg}^{path\text{-}pos}$ *can be solved in time* $O(|\alpha| \cdot |G|)$.

PROOF SKETCH. Since global model checking for a PDL formula φ over a model $\mathcal{M}$ can be done in time $O(|\varphi| \cdot |\mathcal{M}|)$ [Alechina and Immerman 2000; Cleaveland and Steffen 1993], it is immediate that QUERY EVALUATION is in time $O(|\alpha| \cdot |G|)$. From this, the same bound for QUERY COMPUTATION can also be derived. Given a query α and a set S, we can mark the nodes in S with a special predicate that occurs nowhere in α. We can then modify query α and use the algorithm for global model checking for PDL to obtain the required output of QUERY COMPUTATION. □

It is straightforward to extend the algorithm of Fact 4.4 to constant tests (c-tests), since these can be treated analogously as edge labels.

COROLLARY 4.5. *Both* QUERY EVALUATION *and* QUERY COMPUTATION *for* $\mathsf{GXPath}_{reg}^{path\text{-}pos}(c)$ *can be solved in linear time, i.e.,* $O(|\alpha| \cdot |G|)$.

4.3. Expressive power

When gauging the expressive power of query languages, the most common yardstick is that of first-order logic (FO) [Abiteboul et al. 1995]. Indeed, first-order logic is well established as the core of all relational database queries and it is often one of the query language design goals to achieve some sort of completeness with respect to a fragment of FO. For example one of the governing principles when refining the syntax of the XML query language XPath [ten Cate and Marx 2007; Kay 2004] was to make it equivalent to FO over trees, as this

provides a well established base for adding new features, while keeping the language compact and easy to understand.

To this end, we will study the expressive power of GXPath and its many dialects when compared to first-order logic. We begin by showing that the core fragment $\mathsf{GXPath}_{\mathrm{core}}$ with no data value comparisons captures FO^3, like its analogue (core XPath 2.0) does on trees. The main difference here is that FO^3 over trees equals full FO, while over graphs this is not the case. After that, we also show that, for the regular fragment, an equivalent statement holds for FO^3 enriched with binary transitive closure. Following that, we move on to data fragments and show that, although standard XPath-like data tests fall short of the full power of FO with data value comparisons, the equivalence can be obtained by allowing tests of the kind used in regular queries with data tests (RQDs).

4.3.1. Expressiveness of navigational languages. We provide a detailed analysis of expressiveness for navigational features of dialects of GXPath. To understand the expressive power of navigational GXPath, we will do two types of comparisons:

— We compare them with FO, fragments and extensions. The core language will capture FO^3. This is similar to a capture result for trees [Marx 2005]; the main difference is that on graphs, unlike on trees, this falls short of full FO. We also provide a counterpart of this result for $\mathsf{GXPath}_{\mathrm{reg}}$, adding the transitive closure operator.
— We look at the analog of conditional XPath [Marx 2005] which captures FO over trees and show that, in contrast, over graph databases, it can express queries that are not FO-definable.

Comparisons with FO *and relatives.* To compare expressiveness of GXPath fragments with first-order logic, we need to explain how to represent graph databases as FO structures. Since all the formalisms can express reachability queries (at least with respect to a single label), we view graphs as FO structures

$$G = \langle V, (E_a, E_{a^*})_{a\in\Sigma}\rangle$$

where $E_a = \{(v, v') \mid (v, a, v') \in E\}$ and E_{a^*} is its reflexive-transitive closure.

Recall that FO^k stands for the k-variable fragment of FO, i.e., the set of all FO formulae that use variables from a fixed set $x_1, \ldots, x_k$. As we mentioned, on trees, the core fragment of XPath 2.0 was shown to capture FO^3. We now prove that the same remains true without restriction to trees.

THEOREM 4.6. *$\mathsf{GXPath}_{core} = \mathrm{FO}^3$ with respect to both path queries and node tests.*

PROOF. We use a result of Tarski and Givant stating that relation algebra with the basis A of binary relations has the same expressive power as first order logic with three variables over the signature A of binary relations and equality [Tarski and Givant 1987]. As we will be using a slight modification of the result, we give the precise formulation here. The proof of this version can be found in [Andréka et al. 2001] (see Theorem 1.9 and Theorem 1.10).

First we formalize relation algebras. Let $A = \{R_1, \ldots, R_n\}$ be a set of binary relation symbols. The syntax of relation algebra over A is defined as all expressions built from base relations in A using the operators $\cup, \overline{(\cdot)}, \circ, (\cdot)^-$, denoting union, complement, composition of relations and the reverse relation. We are also allowed to use an atomic symbol Id denoting identity.

Our algebra is then interpreted over a structure $M = (V, R_1^M, \ldots, R_n^M)$ where all R_i^M are binary relations over V. Interpretations of the symbols $\cup, \overline{(\cdot)}, \circ, (\cdot)^-$ and Id is the standard union, complement (with respect to V^2), composition, and reverse of binary relations. Id is simply the set of all (v, v) where $v \in V$. We will write $(a, b) \in R^M$, or aR^Mb, when the pair (a, b) belongs to relation R defined over V with relations R_i interpreted as R_i^M. In

the statement of the following fact, use the (standard) notation $\varphi[x/a, y/b]$ to denote the formula obtained from φ by substituting x by a and y by b.

FACT 4.7 ([ANDRÉKA ET AL. 2001]). *Let $A = \{R_1, \ldots, R_n\}$ be a set of binary relation symbols.*

— *For every expression R in the relation algebra $(A, \cup, \overline{(\cdot)}, \circ, (\cdot)^-, Id)$, there is an FO^3 formula in two free variables $\varphi_R(x, y)$ such that, for every structure $M = (V, R_1^M, \ldots, R_n^M)$, we have*

$$\{(a, b) \mid aR^M b\} = \{(a, b) \mid M \models \varphi_R[x/a, y/b]\}.$$

— *Conversely, for every FO^3 formula $\varphi(x, y)$ in two free variables, there exists a relation algebra expression R_φ such that, for every structure $M = (V, R_1^M, \ldots, R_n^M)$, we have*

$$\{(a, b) \mid M \models \varphi[x/a, y/b]\} = \{(a, b) \mid aR_\varphi^M b\}.$$

Note that we view a graph database $G = (V, E)$ as a structure over the alphabet of binary relations E_a, E_{a^*}, where $a \in \Sigma$. Then, a graph database is interpreted as a model

$$M = (V, (E_a^M, E_{a^*}^M)_{a\in\Sigma}), \text{ where}$$

$$E_a = \{(v, v') \mid (v, a, v') \in E\}$$

and E_{a^*} is its reflexive transitive closure. Note that the Tarski-Givant result states something stronger, namely that the equivalence will hold over any structure, no matter if a^* is interpreted as the transitive closure of a or not. This means that it will in particular hold on all the structures where it is, and those are our graph databases.

First, we give a translation from $\mathsf{GXPath}_{\mathrm{core}}$ into FO^3. That is, for every path expression e, we provide a formula $F_e(x, y)$ in two free variables such that, for every graph database $G = (V, E)$, we have $[\![e]\!]^G = \{(v, v') \in V^2 \mid M \models F_e[x/v, y/v']\}$, where $M = (V, (E_a^M, E_{a^*}^M)_{a\in\Sigma})$ and $E_a = \{(v, v') \mid (v, a, v') \in E\}$ and E_{a^*} its reflexive transitive closure. Similarly, for every node expression φ, we define a formula $F_\varphi(x)$ in one free variable. The definition is by induction on the structure of $\mathsf{GXPath}_{\mathrm{core}}$ expressions.

The base cases are those of expressions $a, a^*, a^-, (a^-)^*$ and $\top$, and the formulae F_e are: $F_a(x, y) \equiv E_a(x, y)$; $F_{a^*}(x, y) \equiv E_{a^*}(x, y)$; $F_{a^-}(x, y) \equiv E_a(y, x)$; $F_{(a^*)^-}(x, y) \equiv E_{a^*}(y, x)$; and $F_\top(x) \equiv x = x$. The induction cases are as follows:

— If $e = [\varphi]$, then $F_e(x, y) \equiv (x = y) \wedge F_\varphi(x)$.
— If $e = \alpha \cdot \beta$, then $F_e(x, y) \equiv \exists z(F_\alpha(x, z) \wedge \exists x(x = z \wedge F_\beta(x, y)))$.
— If $e = \alpha \cup \beta$, then $F_e(x, y) \equiv F_\alpha(x, y) \vee F_\beta(x, y)$.
— If $\varphi = \neg\psi$, then $F_\varphi(x) \equiv \neg F_\psi(x)$.
— If $\varphi = \psi \wedge \psi'$, then $F_\varphi(x) \equiv F_\psi(x) \wedge F_{\psi'}(x)$.
— $\varphi = \langle\alpha\rangle$, then $F_\varphi(x) \equiv \exists y F_\alpha(x, y)$.
— If $e = \overline{\alpha}$, then $F_e(x, y) \equiv \neg F_\alpha(x, y)$.

The claim easily follows. Note that we have shown that our expressions can be converted into FO^3 over a fixed interpretation of relation symbols appearing in our alphabet (that is, when $E_{a^*} = (E_a)^*$). The result by Tarski and Givant is stronger, since it holds for any interpretation. Note that this does not invalidate our result, since we are interested only in this fixed interpretation of graph predicates.

To prove the equivalence of $\mathsf{GXPath}_{\mathrm{core}}$ with FO^3, we now show that every relation algebra expression has an equivalent $\mathsf{GXPath}_{\mathrm{core}}$ path expression. Namely, we show that, for every relation algebra expression R over the signature $(E_a, E_{a^*})_{a\in\Sigma}$, there is a path expression e_R of $\mathsf{GXPath}_{\mathrm{core}}$ such that, for every graph database $G = (V, E)$, it holds that $[\![e_R]\!]^G = \{(a, b) \in R^M\}$. Here M is obtained from G as before. In particular, we assume that E_{a^*}

is the reflexive transitive closure of E_a. We do this inductively on the structure of RA expressions R.

For the base cases we have $e_{E_a} = a$; $e_{E_{a^*}} = a^*$; and $e_{Id} = \varepsilon$. Induction cases follow the structure of the expressions:

$$e_{R_1 \cup R_2} = e_{R_1} \cup e_{R_2}; \quad e_{R_1 \circ R_2} = e_{R_1} \cdot e_{R_2}; \quad e_{S^-} = (e_S)^-; \quad e_{\overline{S}} = \overline{e_S}.$$

To show the equivalence between $R = S^-$ and $e_R = (e_S)^-$ we need the following claim.

CLAIM 4.8. *For every GXPath$_{core}$ path expression e, there is a GXPath$_{core}$ expression e^- such that, for every graph G, $[\![e^-]\!]^G = \{(v, v') \mid (v', v) \in [\![e]\!]^G\}$.*

The proof of the claim is an easy induction on expressions. We simply push the reverse onto atomic statements. Notice that this is the reason why we can not simply drop the converse operators from our syntax. All the other equivalences follow from the definition and the inductive hypothesis.

Now, let $\varphi(x, y)$ be an arbitrary FO^3 formula. By Fact 4.7 we know that there is a relation algebra expression R_φ equivalent to φ over all structures that interpret $\{E_a, E_{a^*} \mid a \in \Sigma\}$. In particular it is true over all the structures where $E_{a^*} = (E_a)^*$. By the previous paragraph we know that there is a GXPath$_{\text{core}}$ expression e_{R_φ} equivalent to R_φ. In particular, this means that, for every graph database $G = (V, E)$, it holds that for the model $M = (V, (E_a, E_{a^*}) \mid a \in \Sigma)$, derived from G, we have the following:

$$\{(a, b) \mid M \models \varphi[x/a, y/b]\} = \{(a, b) \mid (a, b) \in R_\varphi^M\}.$$

On the other hand, we also have:

$$[\![e_{R_\varphi}]\!]^G = \{(a, b) \mid (a, b) \in R_\varphi^M\}.$$

Thus we conclude that

$$\{(a, b) \mid M \models \varphi[x/a, y/b]\} = [\![e_{R_\varphi}]\!]^M.$$

The previous part shows equivalence between path expressions and formulas with two free variables. To deal with formulas with a single free variable $F(x)$ we do the following. Define $F'(x, y) = x = y \wedge F(x)$. Note that F' selects all pairs (v, v) such that $F(v)$ holds. Now find an equivalent path expression α (we know we can do this by going through relation algebra) and let $e = \langle \alpha \rangle$. □

Not all results about the expressiveness of XPath on trees extend to graphs. For instance, on trees, the regular fragment with no negation on paths (i.e., the path-positive fragment) can express all of FO [Marx 2005]. This fails over graphs: GXPath$_{\text{reg}}$ fails to express even all of FO^2 when restricted to its path-positive fragment (i.e., the fragment that still permits unary negation).

PROPOSITION 4.9. *There exists a binary* FO^2 *query that is not definable in GXPath$_{reg}^{path\text{-}pos}$.*

PROOF SKETCH. The idea is to observe that path-positive fragments of GXPath cannot define the universal binary relation on an input graph. The query not definable in GXPath$_{\text{reg}}^{\text{path-pos}}$ is then the one saying that there are at least two nodes in a given graph.

Formally, let $\psi(x, y) \equiv \exists x \exists y(\neg(x = y))$. It is easy to see that $[\![\psi]\!]^G = \{(x, y) \mid (x, y) \in V^2\}$ if $G = \langle V, E \rangle$ has at least two nodes and $[\![\psi]\!]^G = \emptyset$ otherwise. (Notice that the variables x, y in ψ are immediately "overwritten" by the existential quantification, which is why $[\![\psi]\!]^G \neq \{(x, y) \mid (x, y) \in V^2$ and $x \neq y\}$.) Consider the graphs $G_1 = \langle \{v, v'\}, \emptyset \rangle$ and $G_2 = \langle \{v\}, \emptyset \rangle$. That is, we have no edges. It follows that $[\![\psi]\!]^{G_1} = \{(v, v'), (v', v)\}$ and $[\![\psi]\!]^{G_2} = \emptyset$. It can be shown by induction on the structure of path GXPath$_{\text{reg}}^{\text{path-pos}}$ expressions

that we either have that $[\![\alpha]\!]^{G_1} = \{(v,v),(v',v')\}$ and $[\![\alpha]\!]^{G_2} = \{(v,v)\}$, or $[\![\alpha]\!]^{G_1} = \emptyset$ and $[\![\alpha]\!]^{G_2} = \emptyset$. Similarly for node expressions it can be shown that either $[\![\varphi]\!]^{G_1} = \{v, v'\}$ and $[\![\varphi]\!]^{G_2} = \{v\}$, or $[\![\varphi]\!]^{G_1} = \emptyset$ and $[\![\varphi]\!]^{G_2} = \emptyset$. □

We now move to $\mathsf{GXPath}_{\mathrm{reg}}$ and relate it to a fragment of FO*, the parameter-free fragment of the transitive-closure logic. The language of FO* extends the one of FO with a transitive closure operator that can be applied to formulas with precisely two free variables. That is, for every FO formula $F(x,y)$, the formula $F^*(x,y)$ is also an FO* formula. The semantics is the reflexive-transitive closure of the semantics of F. That is, $G \models F^*(a,b)$ iff $a = b$ or there is a sequence of nodes $a = v_0, v_1, \ldots, v_n = b$ for $n > 0$ such that $G \models F(v_i, v_{i+1})$ whenever $0 \leq i < n$.

By $(\mathrm{FO}^*)^k$ we mean the k-variable fragment of FO*. Note that when we deal with FO* and $(\mathrm{FO}^*)^k$, we can view graphs as structures of the vocabulary $(E_a)_{a\in\Sigma}$, since all the E_{a^*} are definable and there is no reason to include them in the language explicitly.

Over trees, regular XPath is known to be equal to $(\mathrm{FO}^*)^3$ in terms of expressiveness [ten Cate 2006]. The next theorem shows that over graphs, these logics coincide as well.

THEOREM 4.10. *$\mathsf{GXPath}_{reg} = (\mathrm{FO}^*)^3$.*

PROOF SKETCH. The containment of $\mathsf{GXPath}_{\mathrm{reg}}$ in $(\mathrm{FO}^*)^3$ is done by a routine translation. To show the converse, we use techniques similar to those in the proof of Theorem 4.6: we extend $(\mathrm{FO}^*)^3$ and relation algebra equivalence to state that relation algebra with the transitive closure operator has equal expressive power to $(\mathrm{FO}^*)^3$ over the class of all labeled graphs. For this one can simply check that the inductive proof from [Andréka et al. 2001] can be extended by adding two extra inductive clauses. Namely, when going from relation algebra to FO^3 we state that expressions of the form R^* are equivalent to $F^*_R(x,y)$, where F_R is the formula equivalent to R. In the other direction we state that $F^*(x,y)$ is equivalent to $(R_F(x,y))^*$. Here, by $R_F(x,y)$ we denote the expression equivalent to $F(x,y)$, when the variables are used in that particular order. After that one verifies that the correctness proof of [Andréka et al. 2001] applies. □

What about the relative expressive power of $\mathsf{GXPath}_{\mathrm{core}}$ and $\mathsf{GXPath}_{\mathrm{reg}}$? For positive fragments, known results on trees (see [ten Cate and Marx 2007]) imply the following.

COROLLARY 4.11. *$\mathsf{GXPath}^{pos}_{core} \subsetneq \mathsf{GXPath}^{pos}_{reg}$.*

We shall now see that the strict separation applies to the full languages. This is not completely straightforward even though $\mathsf{GXPath}_{\mathrm{core}}$ is equivalent to a fragment of FO, since the latter uses the vocabulary with transitive closures. This makes it harder to apply standard techniques, such as locality, directly. We shall see how to establish separation by taking a detour through conditional XPath.

Conditional GXPath. To capture FO over XML trees, Marx showed that one can use *conditional* XPath, which essentially adds the temporal *until operator* [Marx 2005]. That is, it expands the core-XPath's a^* with $(a[\varphi])^*$, which checks that the test $[\varphi]$ is true on every node of an a^*-labeled path. Formally, its path formulae are given by:

$$\alpha, \beta \quad := \quad \varepsilon \mid \_ \mid a \mid a^- \mid a^* \mid a^{-*} \mid (a[\varphi])^* \mid (a^-[\varphi])^* \mid [\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha}$$

We refer to this language as $\mathsf{GXPath}_{\mathrm{cond}}$. We now show that the FO capture result fails rather dramatically over graphs: there are even positive $\mathsf{GXPath}_{\mathrm{cond}}$ queries not expressible in FO.

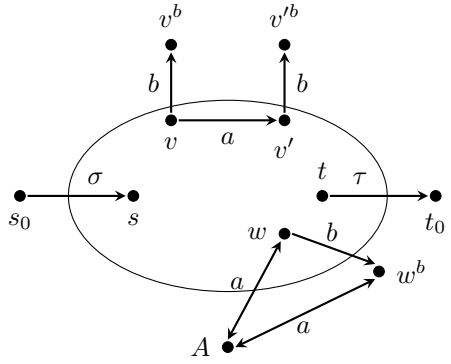
THEOREM 4.12. *There is a $\mathsf{GXPath}^{pos}_{cond}$ query not expressible in* FO.

Note that the standard inexpressibility tools for FO, such as locality, cannot be applied straightforwardly since the vocabulary of graphs already contains all the transitive closures

E_{a^*}. In fact, this means that in $\mathsf{GXPath}^{\text{pos}}_{\text{cond}}$ the query asking for transitive closures of base relations is trivially definable, even though it is not definable in FO over the E_as. The way around this is to combine locality with the composition method: we use locality to establish a winning strategy for the duplicator in a game that does not involve transitive closures and then use composition to extend the winning strategy to handle transitive closures.

PROOF. To prove Theorem 4.12 we will need several auxiliary results.

Let $\Sigma = \{a, b, \sigma, \tau\}$ be an alphabet of labels. For an arbitrary graph $G = (V, E)$ over the singleton alphabet $\{a\}$ and two fixed nodes $s, t \in V$ we define a Σ-labeled graph $G(s, t)$ as follows. First, it contains all the nodes and edges of G. For every node $v \notin \{s, t\}$ in V we add a new node v^b and an edge (v, b, v^b) to $G(s, t)$. We also add two new nodes, s_0 and t_0, together with edges (s_0, σ, s) and (t, τ, t_0), coming into s and leaving t. These nodes and edges are added to distinguish s and t in our graph. Finally, we add one extra node called A and, for every other node in $G(s, t)$, we add two edges: one going into A and the other returning from A to the same node, both labeled a. The modifications are illustrated in the following image.



By $\mathcal{C}$ we denote the class of all $G(s, t)$, obtained from G and $s, t \in V$ as above, for every G over $\{a\}$ and $s, t \in V$. Define $\mathcal{C}^-$ to be the class of graphs that are obtained from the graphs in $\mathcal{C}$ by removing the node A and all the associated edges. Let the property P stand for

$$t \text{ is reachable from } s \text{ via a path labeled with } (a[b])^*.$$

That is, t is reachable from s by a path that proceeds forwards by a-labeled edges, but also has to have a b labeled edge leaving every internal node on the path. To obtain the desired result we will first prove the following claim.

CLAIM 4.13. *The property P is not expressible in* FO *with vocabulary $\{E_a, E_b, E_\sigma, E_\tau, E_{a^*}, E_{b^*}, E_{\sigma^*}, E_{\tau^*}\}$ over the class $\mathcal{C}$. Here, as before, we assume that E_{ℓ^*} is the reflexive transitive closure of E_ℓ, for $\ell \in \{a, b, \sigma, \tau\}$.*

The main idea of the proof is as follows. Assume that P is expressible in FO, over the full vocabulary, by a formula of quantifier rank m. Then we will show that there exist graphs $\mathbf{G}^1_d$ and $\mathbf{G}^2_d$ (parametrized by $d \geq 0$) such that $\mathbf{G}^1_d \equiv_m \mathbf{G}^2_d$. However, by construction, $\mathbf{G}^1_d$ and $\mathbf{G}^2_d$ will not agree on P, which is a contradiction.

More precisely, the proof goes through three lemmas. We will use Hanf-locality and composition of games.

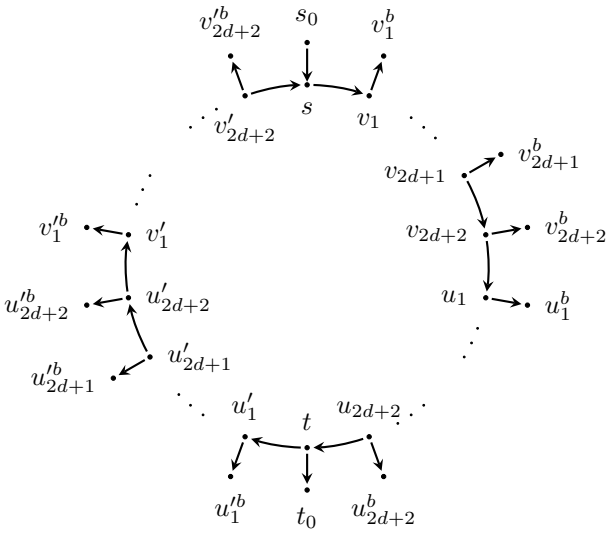
In the first lemma, we use the standard notion of a *neighborhood* of an element in a structure, and the notion of Hanf-locality. For details, see [Libkin 2004]. Specifically, for two graphs G^1, G^2, we write $G^1 \leftrightarrows_d G^2$ if there is a bijection f between nodes of G^1 and

nodes of G^2 (in particular, the sets of nodes must have the same cardinality) such that the radius-d neighborhoods of each node v in G^1 and $f(v)$ in G^2 are isomorphic. The radius-d neighborhood around v is the substructure generated by all nodes reachable from v by a path (using all types of edges and going both forwards and backwards) of length at most d, with v interpreted as a distinguished constant (which thus must be preserved by isomorphisms between neighborhoods).

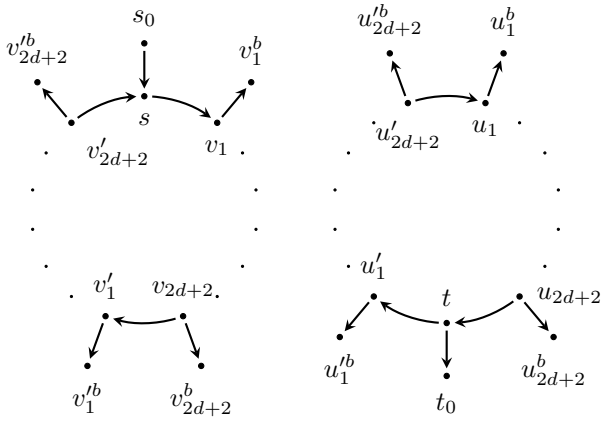
Locality is meaningless over structures in $\mathcal{C}$, since every two nodes are connected by a path of length 2, so $\leftrightarrows_2$ is an isomorphism. This is why we use several steps to prove our result.

LEMMA 4.14. *For every $d \geq 0$ there exist two graphs G^1_d and G^2_d, as structures of the vocabulary $\{E_a, E_b, E_\sigma, E_\tau\}$, in $\mathcal{C}^-$ such that $G^1_d \leftrightarrows_d G^2_d$ and G^1_d satisfies P, while G^2_d does not.*

PROOF. To see this take arbitrary d and let the graphs G^1_d and G^2_d be as in the following two images. All the labels on the circles are a, the incoming edges from the nodes s_0 to the nodes s are labeled σ, the outgoing edges from the nodes t to nodes t_0 are labeled τ, and the edges from the nodes v to the nodes v^b are labeled b.



Graph G^1_d



Graph G^2_d

Let $f : G^1_d \to G^2_d$ be the bijection defined by the node labels in the natural way: each node gets mapped to the one with the same name in the other graph. That is, we set $f(s) := s, f(v_i) := v_i$, then $f(v^b_i) := v^b_i$ and similarly for v'_i, u_i, etc.

To see that $G^1_d \leftrightarrows_d G^2_d$ we have to check $N_d^{G^1_d}(c) \cong N_d^{G^2_d}(f(c))$ for every c. This is now easily established, since the d neighborhood of any c and $f(c)$ will simply be extended chains of length d around c and $f(c)$. In particular, it is possible that they intersect the d neighborhood of either s or t, but never both. We thus conclude that they will always be isomorphic, giving us the desired result. □

From Lemma 4.14 and Corollary 4.21 in [Libkin 2004], which shows that Hanf-locality with a sufficiently large radius implies the winning strategy for the duplicator in an Ehrenfeucht-Fraïssé game, we obtain the following.

LEMMA 4.15. *For every $m \geq 0$ there exists $d \geq 0$ so that $G^1_d \equiv_m G^2_d$.*

As usual, by $\equiv_m$ we denote the fact that duplicator has a winning strategy in an m-round Ehrenfeucht-Fraïssé game. This game is still played on structures in the vocabulary that does not use transitive closures.

Now let $\mathbf{G}^1_d$ and $\mathbf{G}^2_d$ be obtained from G^1_d and G^2_d by adding, as in the picture above, a node A with a-edges to and from every other node. We view these graphs as structures of the vocabulary that has all the relations E_ℓ and E_{ℓ^*} for each of the four labels ℓ we have. Next, we show

LEMMA 4.16. *If $G^1_d \equiv_m G^2_d$, then $\mathbf{G}^1_d \equiv_m \mathbf{G}^2_d$.*

The strategy is very simple: the duplicator plays by copying the moves from the game $G^1_d \equiv_m G^2_d$ as long as the spoiler does not play the A-node. If the spoiler plays the A-node in one structure, the duplicator responds with the A-node in the other. We now need to show that this preserves all the relations. Clearly this strategy preserves all the relations E_ℓ among nodes other than the A-node, simply by assumption. Moreover, since $E_{\ell^*} = E_\ell$ for $\ell \neq a$, we have preservation of the transitive closures other than that of E_a as well. So we need to prove that the strategy preserves E_{a^*}, but this is immediate since in both graphs E_{a^*} is interpreted as the total relation. This proves the lemma.

The claim now follows from the lemmas: assume that P is expressible in FO, over the full vocabulary, by a formula of quantifier rank m. Pick a sufficiently large d to ensure that $\mathbf{G}^1_d \equiv_m \mathbf{G}^2_d$. Then $\mathbf{G}^1_d$ and $\mathbf{G}^2_d$ must agree on P, but they clearly do not, since the extra

paths introduced in these graphs compared to G_d^1 and G_d^2 go via the A-node, which does not have a b-successor.

To conclude the proof of Theorem 4.12, consider a conditional graph XPath expression $\sigma(a[b])^*[\tau]$. Over graphs as considered here it defines precisely the property P, which, as just shown, is not FO-expressible in the full vocabulary. □

We can now fulfill our promise and establish separation between $\mathsf{GXPath}_{\mathrm{core}}$ and $\mathsf{GXPath}_{\mathrm{reg}}$. Since $\mathsf{GXPath}_{\mathrm{core}} \subseteq \mathrm{FO}$ and we just saw a conditional (and thus regular) GXPath query not expressible in FO, we have:

COROLLARY 4.17. *$\mathsf{GXPath}_{core} \subsetneq \mathsf{GXPath}_{reg}$.*

4.3.2. Expressiveness of data languages. We saw that, for navigational features, core graph XPath captures FO^3. The question is whether this continues to be so in the presence of data tests. First, we need to explain how to describe data graphs as FO-structures to talk about FO with data tests.

Following the standard approach for data words and data trees [Segoufin 2007], we do so by adding a binary predicate for testing if two nodes hold the same data value. That is, a data graph is then viewed as a structure $G = \langle V, (E_a, E_{a^*})_{a\in\Sigma}, \sim\rangle$ where $v \sim v'$ iff $\rho(v) = \rho(v')$. To be clear that we deal with FO over that vocabulary, we shall write $\mathrm{FO}(\sim)$. If we want to talk about constant data tests (i.e., $c^=$), we add a new unary symbol D_c, for each $c \in \mathcal{D}$ that will denote that the data value equals c. In that case we shall refer to $\mathrm{FO}(\mathsf{c}, \sim)$. It turns out that the equivalence with FO^3 breaks when we consider XPath style data tests, as the following theorem shows.

THEOREM 4.18.

— $\mathsf{GXPath}_{core}(\mathsf{eq}) \subsetneq \mathrm{FO}^3(\sim)$;
— $\mathsf{GXPath}_{core}(\mathsf{c}, \mathsf{eq}) \subsetneq \mathrm{FO}^3(\mathsf{c}, \sim)$.

PROOF SKETCH. The first containment uses the translation into FO^3 shown in the proof of Theorem 4.6. For the new data operators, we use the following. If $e = \langle \alpha = \beta \rangle$ then

$$F_e(x) \equiv \exists y, z(y \sim z \wedge F_\alpha(x, y) \wedge \exists y(z = y \wedge F_\beta(x, y)))$$

and likewise for the inequality comparison. Translation of constants is self-evident.

To prove strictness we show that the FO^3 query $F(x, y) \equiv x \sim y$ is not definable in $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c}, \mathsf{eq})$. Note that F defines the set of all pairs of nodes carrying the same data value. The proof of this is implicit in the proof of Proposition 4.19. □

Thus, the standard XPath data tests are insufficient for capturing FO^3 over data graphs. This naturally leads to a question: what can be added to data tests to capture the full power of FO^3? The answer, as it turns out, is quite simple: we need to use the same sort of data value tests as in regular queries with data tests (RQDs). Recall that these are defined by adding two expressions to the grammar for α: one is $\alpha_=$, the other is $\alpha_{\neq}$. The semantics, over data graphs, is

$$\begin{aligned} [\![\alpha_=]\!]^G &= \{(v, v') \in [\![\alpha]\!]^G \mid \rho(v) = \rho(v')\} \\ [\![\alpha_{\neq}]\!]^G &= \{(v, v') \in [\![\alpha]\!]^G \mid \rho(v) \neq \rho(v')\} \end{aligned}$$

In other words, we test whether data values at the beginning and at the end of a path are the same, or different. We recall that such an extension is denoted by $\sim$, i.e., we talk about languages $\mathsf{GXPath}(\sim)$ (with the usual sub- and superscripts). The first observation is that these tests indeed add to the expressiveness of the languages.

PROPOSITION 4.19. *The path query $a_=$, for $a \in \Sigma$, is not definable in $\mathsf{GXPath}_{reg}(\mathsf{c}, \mathsf{eq})$.*

Note that the query $a_=$ is definable on trees by the $\mathsf{GXPath}_{\mathrm{core}}(\mathsf{eq})$ query $[\langle \varepsilon = a\rangle]\cdot a\cdot[\langle \varepsilon = a^-\rangle]$. This is because the parent of a given node is unique. However, on graphs this is not always the case, and thus new equality tests add power.

The idea of the proof is that even though $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c},\mathsf{eq})$ can test if a node has an a-successor with the same data value by the means of expression $\langle \varepsilon = a\rangle$, which will return the set $\{v \in V \mid \exists v' \in V \text{ and } (v,v') \in [\![a_=]\!]^G\}$, it has no means of retrieving that particular successor. Specifically, we show that it is possible to find two data graphs G_1 and G_2, such that $[\![a_=]\!]^{G_1} \neq [\![a_=]\!]^{G_2}$, but for every $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c},\mathsf{eq})$ query e we have $[\![e]\!]^{G_1} = [\![e]\!]^{G_2}$. Both G_1 and G_2 have K_6 as their underlying graph, with different data values attached. The proof is essentially by mundane case analysis and is present in the online appendix.

With the extra power obtained from equality tests, we can capture FO^3 over data graphs.

THEOREM 4.20. $\mathit{GXPath}_{\mathit{core}}(\sim) = \mathrm{FO}^3(\sim)$.

PROOF. We follow the technique of the proof of Theorem 4.6. All of the translations used there still apply. The proof that relation algebra is contained in the language $\mathsf{GXPath}_{\mathrm{core}}(\sim)$ is the same as without data values. We only have to add conversion of the new symbol $\sim$: if $R = \sim$, then $e = \varepsilon \cup (\overline{\varepsilon})_=$.

For the other direction we have to show how to translate new path expressions $\alpha_=$ and $\alpha_{\neq}$ into $\mathrm{FO}^3(\sim)$. This is done as follows: if $e = \alpha_=$ then $F_e(x,y) \equiv F_\alpha(x,y) \wedge x \sim y$ and likewise for inequality. The equivalences easily follow. Now the theorem follows from the equivalence of relation algebra and FO^3 [Tarski and Givant 1987]. □

By adopting the technique used in Theorem 4.10 it is straightforward to see that the previous result extends to $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$.

THEOREM 4.21. $\mathit{GXPath}_{\mathit{reg}}(\sim) = (\mathrm{FO}^*)^3(\sim)$.

As mentioned before, one could also allow constant tests in the language. It is then easy to see that the equivalence extends to FO with constants.

COROLLARY 4.22.

— $\mathit{GXPath}_{\mathit{core}}(\mathsf{c},\sim) = \mathrm{FO}^3(\mathsf{c},\sim)$.
— $\mathit{GXPath}_{\mathit{reg}}(\mathsf{c},\sim) = (\mathrm{FO}^*)^3(\mathsf{c},\sim)$.

4.4. Comparing GXPath with other languages

Here we compare GXPath to path languages introduced in Section 3 as well as to traditional navigational languages such as RPQs, CRPQs and NREs. Note that GXPath enriches RPQs with new navigational abilities and it is therefore worthwhile examining how the navigational part of the language fares when compared to other extensions of RPQs.

Relative expressiveness of navigational fragments. Our first goal is to compare the expressiveness of navigational GXPath fragments with that of traditional graph languages. We start with *nested regular expressions* and, after that, look at path languages such as RPQs, CRPQs, and relatives.

As expected, $\mathsf{GXPath}_{\mathrm{reg}}$ is strictly more expressive than NREs. However, we show that NREs do capture the positive fragment of $\mathsf{GXPath}_{\mathrm{reg}}$.

THEOREM 4.23. $\mathit{GXPath}_{\mathit{reg}}^{\mathit{pos}} = \mathit{NRE} \subsetneq \mathit{GXPath}_{\mathit{reg}}^{\mathit{path\text{-}pos}}$.

PROOF. First, we show that $\mathsf{NRE} \subsetneq \mathsf{GXPath}_{\mathrm{reg}}$. Using a straightforward inductive construction one can show how to convert a nested regular expression into an equivalent path expression of $\mathsf{GXPath}_{\mathrm{reg}}$. Note that all the operations can be written down verbatim, minus the $[n]$ expression whose $\mathsf{GXPath}_{\mathrm{reg}}$ equivalent is $[\langle e_n\rangle]$, where e_n is an expression equivalent to n.

Next we show that $\mathsf{GXPath}_{\text{core}}$ query $q = a[\neg\langle b\rangle]$ is not expressible by an NRE. Consider the data graph G with two nodes v, v' and both a and b-labeled edges, going both from v to v' and v' to v. It is easy to see that $[\![q]\!]^G = \emptyset$. In contrast, it can be shown by an easy induction on the structure of NREs n that there exist nodes $x_1, x_2, y_1, y_2 \in \{v, v'\}$ such that $(v, x_1), (v', x_2), (y_1, v), (y_2, v') \in [\![n]\!]^G$. This means that, for every NRE n, we have $[\![n]\!]^G \neq \emptyset$ which means that no NRE equivalent to q exists. (Notice that the proof works for any alphabet we would take for the NRE.)

We now show that $\mathsf{GXPath}_{\text{reg}}^{\text{pos}} = \mathsf{NRE}$. We already know that nested regular expressions can be expressed as GXPath queries. Since no negation operators are used to obtain this we obtain the inclusion of NRE in $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}$.

To complete the proof we now show how to convert an arbitrary $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}$ expression into an equivalent nested regular expression. More precisely, we show that for every path expression α of our fragment there exists a nested regular expression n_α such that for every graph G we have $(x, y) \in [\![\alpha]\!]^G$ iff $(x, y) \in [\![n_\alpha]\!]^G$. Moreover, for every node expression φ we define a nested regular expression n_φ such that $x \in [\![\varphi]\!]^G$ iff $(x, x) \in [\![n_\varphi]\!]^G$. We do this by induction on the structure of our $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}$ expressions. If e is a or a^-, or ε, then $n_e = e$, and if $e = \top$, then $n_e = \varepsilon$. The inductive step is as follows:

— If $e = [\varphi]$, then $n_e = [n_\varphi]$
— If $e = \alpha \cdot \beta$, then $n_e = n_\alpha \cdot n_\beta$
— If $e = \alpha \cup \beta$, then $n_e = n_\alpha + n_\beta$
— If $e = \varphi \wedge \psi$, then $n_e = \varepsilon[n_\varphi] \cdot \varepsilon[n_\psi]$
— If $e = \varphi \vee \psi$, then $n_e = \varepsilon[n_\varphi + n_\psi]$
— If $e = \langle\alpha\rangle$, then $n_e = \varepsilon[n_\alpha]$.

It is easy to see the equivalence between defined expressions. □

We will now show that XPath-like formalisms are incomparable with CRPQs and similar queries in terms of their navigational expressiveness. The simple restriction, $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}$, is not subsumed by CRPQs. In fact, it is not even subsumed by unions of two-way CRPQs (which allow navigation in both ways). On the other hand, CRPQs are not subsumed by the strongest of our navigational languages, $\mathsf{GXPath}_{\text{reg}}$.

THEOREM 4.24. *CRPQs and GXPath fragments are incomparable:*

— $\mathsf{GXPath}_{reg}^{pos} \not\subseteq \mathsf{CRPQ}$ *(even stronger, there are* $\mathsf{GXPath}_{reg}^{pos}$ *queries not definable by U2CRPQs);*
— $\mathsf{CRPQ} \not\subseteq \mathsf{GXPath}_{reg}$.

PROOF. The first item follows from Theorem 4.23 and Theorem 1 in [Barceló et al. 2012], where it was shown that there exists a NRE not expressible by U2CRPQs.

To see that the second item holds we first show that for every $\mathsf{GXPath}_{\text{reg}}$ expression e there exists an $\mathcal{L}^3_{\infty\omega}$ formula F_e equivalent to it (i.e., a formula in infinitary logic with 3 variables). After that we give an example of a CRPQ that is not expressible in this logic using a standard multi-pebble games argument. To be more precise, we will be working with $\mathcal{L}^3_{\infty\omega}$ formulas over the alphabet $\{E_a \mid a \in \Sigma\}$ (and with the equality symbol). All the relations are binary and simply represent a labeled edge between two nodes. We will denote data graphs as structures for this logic by $G = \langle V, (E_a)_{a\in A}\rangle$.

Now for every path expression α we will define a formula $F_\alpha(x, y)$ such that $(v, v') \in [\![\alpha]\!]^G$ iff $G \models F_\alpha[x/v, y/v']$. Likewise, for a node expression φ, we define a formula $F_\varphi(x)$ such that $v \in [\![\varphi]\!]^G$ iff $G \models F_\varphi[x/v]$. We do this by induction on $\mathsf{GXPath}_{\text{reg}}$ expressions.

Basis:
— $\alpha = a$ then $F_\alpha(x, y) \equiv E_a(x, y)$
— $\alpha = a^-$ then $F_\alpha(x, y) \equiv E_a(y, x)$

— $\alpha = \varepsilon$ then $F_\alpha(x,y) \equiv x = y$
— $\varphi = \top$ then $F_\alpha(x) \equiv x = x$
Inductive step:
— $\alpha' = [\varphi]$ then $F_{\alpha'}(x,y) \equiv x = y \wedge F_\varphi(x)$
— $\alpha' = \alpha \cdot \beta$ then $F_{\alpha'}(x,y) \equiv \exists z(\exists y\ (y = z \wedge F_\alpha(x,y)) \wedge \exists x\ (x = z \wedge F_\beta(x,y)))$
— $\alpha' = \alpha \cup \beta$ then $F_{\alpha'}(x,y) \equiv F_\alpha(x,y) \vee F_\beta(x,y)$
— $\alpha' = \alpha^*$ then define
  — $\varphi^1_\alpha(x,y) \equiv F_\alpha(x,y)$,
  — $\varphi^{n+1}_\alpha(x,y) \equiv \exists z\ (\exists y\ (y = z \wedge F_\alpha(x,y)) \wedge \exists x\ (x = z \wedge \varphi^n_\alpha(x,y)))$
  — Finally, set $F_{\alpha'}(x,y) \equiv \bigvee_{n\in\omega} \varphi^n_\alpha(x,y)$
— $\alpha' = \overline{\alpha}$ then $F_{\alpha'}(x,y) \equiv \neg F_\alpha(x,y)$
— $\varphi' = \neg\varphi$ then $F_{\varphi'}(x) \equiv \neg F_\varphi(x)$
— $\varphi' = \varphi \wedge \psi$ then $F_{\varphi'}(x) \equiv F_\varphi(x) \wedge F_\psi(x)$
— $\varphi' = \langle\alpha\rangle$ then $F_{\varphi'}(x) \equiv \exists y F_\alpha(x,y)$.

It is straightforward to show that the translation has the desired property.

Next we define a binary CRPQ $\varphi(x,y)$ that has no $\mathsf{GXPath}_{\mathrm{reg}}$ equivalent. For the separation argument it will suffice to assume that all edges are labeled with the same symbol a. The CRPQ $\varphi(x,y)$ states that there is a K_4 subgraph, with x, y as two of its nodes.

$$\begin{aligned}\varphi(x,y) := (x,a,y) \wedge (x,a,z) \wedge (x,a,w) \wedge \\ (y,a,x) \wedge (z,a,x) \wedge (w,a,x) \wedge \\ (y,a,z) \wedge (y,a,w) \wedge \\ (z,a,y) \wedge (w,a,y) \wedge \\ (z,a,w) \wedge (w,a,z).\end{aligned}$$

Note that $\varphi(K_3) = \emptyset$, while $\varphi(K_4) \neq \emptyset$. However, it is well known that no $\mathcal{L}^3_{\infty\omega}$ sentence F can distinguish K_3 and K_4 (cf. [Libkin 2004]), since the duplicator has a winning strategy in an infinite 3-pebble game on these graphs, simply by preserving equality of pebbled elements. Thus, φ cannot be expressed by a $\mathsf{GXPath}_{\mathrm{reg}}$ query. □

On the other hand, the positive fragment of $\mathsf{GXPath}_{\mathrm{core}}$ can be captured by unions of two-way CRPQs.

PROPOSITION 4.25. *$\mathsf{GXPath}^{pos}_{core} \subsetneq$ U2CRPQ.*

PROOF. We have seen an example of U2CRPQ not expressible by GXPath in Theorem 4.24. To see that the inclusion holds we show that for every $\mathsf{GXPath}^{\mathrm{pos}}_{\mathrm{core}}$ expression e we can construct an equivalent U2CRPQ. That is, for every path expression α we define a U2CRPQ, named $\psi_\alpha(x,y)$, in two free variables, x and y, such that for every graph database G we have $[\![\alpha]\!]^G = \psi_\alpha(G)$. Similarly for every node expression φ we define a U2CRPQ $\psi_\varphi(x)$. We do so by induction on the structure of $\mathsf{GXPath}^{\mathrm{pos}}_{\mathrm{core}}$ expressions.

Basis:
— For $\alpha = \varepsilon$ we have $\psi_\alpha(x,y) := (x,\varepsilon,y)$.
— For $\alpha = \_$ we have $\psi_\alpha(x,y) := \bigvee_{a\in\Sigma}(x,a,y)$.
— For $\alpha = a$ we have $\psi_\alpha(x,y) := (x,a,y)$.
— For $\alpha = a^-$ we have $\psi_\alpha(x,y) := (x,a^-,y)$.
— For $\alpha = a^*$ we have $\psi_\alpha(x,y) := (x,a^*,y)$.
— For $\alpha = a^{-*}$ we have $\psi_\alpha(x,y) := (x,a^{-*},y)$.
— For $\varphi = \top$ we have $\psi_\varphi(x) := \exists y(x,\varepsilon,y)$.
Inductive step:
— For $\alpha = [\varphi]$ we have $\psi_\alpha(x,y) := (x,\varepsilon,y) \wedge \psi_\varphi(y)$.

— For $\alpha = \alpha' \cdot \beta'$ we have $\psi_\alpha(x,y) := \exists z \psi_{\alpha'}(x,z) \wedge \psi_{\beta'}(z,y)$.
— For $\alpha = \alpha' \cup \beta'$ we have $\psi_\alpha(x,y) := \psi_{\alpha'}(x,y) \vee \psi_{\beta'}(x,y)$.
— For $\varphi = \varphi_1 \wedge \varphi_2$ we have $\psi_\varphi(x) := \psi_{\varphi_1}(x) \wedge \psi_{\varphi_2}(x)$.
— For $\varphi = \varphi_1 \vee \varphi_2$ we have $\psi_\varphi(x) := \psi_{\varphi_1}(x) \vee \psi_{\varphi_2}(x)$.
— For $\varphi = \langle \alpha \rangle$ we have $\psi_\varphi(x) := \exists y \psi_\alpha(x,y)$.

It is straightforward to show that the defined expressions are equivalent. □

GXPath and path languages. When comparing GXPath with path languages from Section 3 we will consider the regular fragment with $\sim$ type data tests, since they subsume classical XPath-style tests. While it is apparent from the definition of $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c}, \sim)$ that it contains RQDs, we can also show that the containment is strict.

PROPOSITION 4.26. *The class of RQD queries is strictly contained in* $\mathsf{GXPath}_{reg}(\mathsf{c}, \sim)$.

PROOF. To see that the containment is strict consider the GXPath query $q = (a[b])^*$. Note that this is also an NRE. To obtain a contradiction assume that there is some RQD Q_q equivalent to q. Now consider a graph G with edges (v_1, a, v_2) and (v_2, b, v_3). Data values are not important here so we do not list them explicitly. It is easily checked that $(v_1, v_2) \in [\![q]\!]^G$. By our assumption we also have that $(v_1, v_2) \in Q_q(G)$. But since Q_q is an RQD this means that there is some regular expression with equality e_q such that $Q_q = x \xrightarrow{e_q} y$ and:

— There is a path π starting with v_1 and ending with v_2, and
— $\lambda(\pi)$ belongs to $\mathcal{L}(e_q)$.

However, the only path in G connecting v_1 and v_2 is $\pi = v_1 a v_2$. Consider now the graph G' obtained from G by removing the edge (v_2, b, v_3). We now have $(v_1, v_2) \notin [\![q]\!]^{G'}$, but $\pi = v_1 a v_2$ is still a path in G' with $\lambda(\pi) \in \mathcal{L}(e_q)$. This then implies that $(v_1, v_2) \in Q_q(G')$, a contradiction. □

Comparing GXPath to more expressive path languages we can see that the ability to use variables makes them capable of expressing queries outside the reach of GXPath. We also show that the converse is true, as new navigational features allow GXPath to define patterns not captured by paths.

PROPOSITION 4.27. $\mathsf{GXPath}_{reg}(\mathsf{c}, \sim)$ *is incomparable in terms of expressive power with* RQM*s and register automata.*

PROOF SKETCH. It is easily seen that the example from Proposition 4.26 can be used to give a GXPath query not expressible by any of the path languages.

To prove the reverse it is straightforward to extend the proof of Theorem 4.24 to show that $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c}, \sim)$ is contained in three variable infinitary logic $\mathcal{L}^3_{\infty\omega}$ (with data value comparison relation $\sim$ and unary relations coding constants). It is well known that this logic cannot define models that have at least four different elements [Libkin 2004]. However, one can readily check that such a query is expressible by RQMs and register automata. □

4.5. Hierarchy of the fragments

By coupling the basic navigational languages – $\mathsf{GXPath}_{\mathrm{core}}$ and $\mathsf{GXPath}_{\mathrm{reg}}$ – with various possibilities of data tests, such as no data tests, constant tests, XPath-style equality tests, RQD equality tests, or all of them, we obtain sixteen languages, ranging from $\mathsf{GXPath}_{\mathrm{core}}$ to $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c}, \mathsf{eq}, \sim)$. Recall that adding counting does not affect expressiveness, only the complexity of query evaluation.

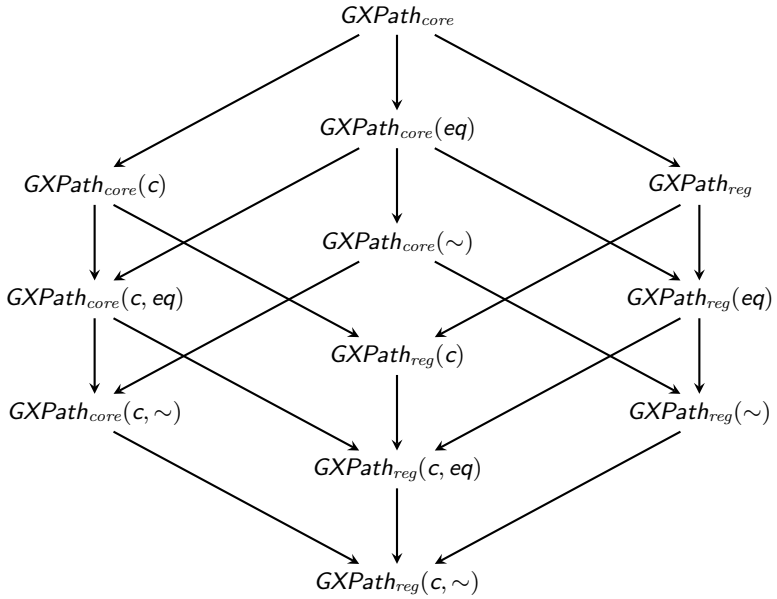
The question is then, how do these fragments compare to each other?

First thing we note is that some of the fragments collapse. Namely, from Theorem 4.20 we know that every $\mathsf{GXPath}_{\mathrm{core}}(\mathsf{eq})$ query can be expressed in $\mathsf{GXPath}_{\mathrm{core}}(\sim)$, and the same holds for regular fragments using Theorem 4.21.

To perform such a transformation explicitly we simply need to show how to convert every test of the form $\langle \alpha = \beta \rangle$ to one using only $=$ comparisons from $\mathsf{GXPath}_{\mathrm{core}}(\sim)$ and that the same can be done for inequality. It is not difficult to see that every node expression of the form $\langle \alpha = \beta \rangle$ is equivalent to $\mathsf{GXPath}_{\mathrm{core}}(\sim)$ expression $\langle \alpha \cdot (\alpha^- \cdot \beta)_= \cdot \beta^- \cap \varepsilon \rangle$, and similarly for $\neq$.

Therefore we can conclude that every fragment where both eq and $\sim$ data tests are present collapses to the one with only $\sim$. For example, $\mathsf{GXPath}_{\mathrm{core}}(\mathsf{eq}, \sim)$ is the same as $\mathsf{GXPath}_{\mathrm{core}}(\sim)$ and so on, bringing the number of possible fragments to twelve. Next we establish the full hierarchy of the remaining fragments.

THEOREM 4.28. *The relative expressive power of graph XPath languages with data comparisons is as shown below:*



Here, an arrow indicates strict containment, while the lack of an arrow indicates that the fragments are incomparable.

PROOF. The result follows from Corollary 4.17 (for navigational fragments), the fact that $\sim$ comparisons subsume usual XPath-style tests, and the following two observations which show that c tests and eq or $\sim$ tests are not mutually definable. Namely, take an alphabet Σ containing letter a. Let c be a fixed data value. Then:

— There is no $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$ expression equivalent to the $\mathsf{GXPath}_{\mathrm{core}}(\mathsf{c})$ query $q_{\mathsf{c}} := c^=$.
— There is no $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c})$ expression equivalent to the $\mathsf{GXPath}_{\mathrm{core}}(\mathsf{eq})$ query $q_{\mathsf{eq}} := \langle a \neq a \rangle$.

For the first item, simply take two single-node data graphs G_1 and G_2, with G_1's single node holding value c and G_2 holding a different value c'. Hence, $[\![q_{\mathsf{c}}]\!]^{G_1}$ selects the only node of G_1, while $[\![q_{\mathsf{c}}]\!]^{G_2} = \emptyset$. However, a straightforward induction on the structure of expressions shows that, for every $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$ query e we have $[\![e]\!]^{G_1} = [\![e]\!]^{G_2}$.

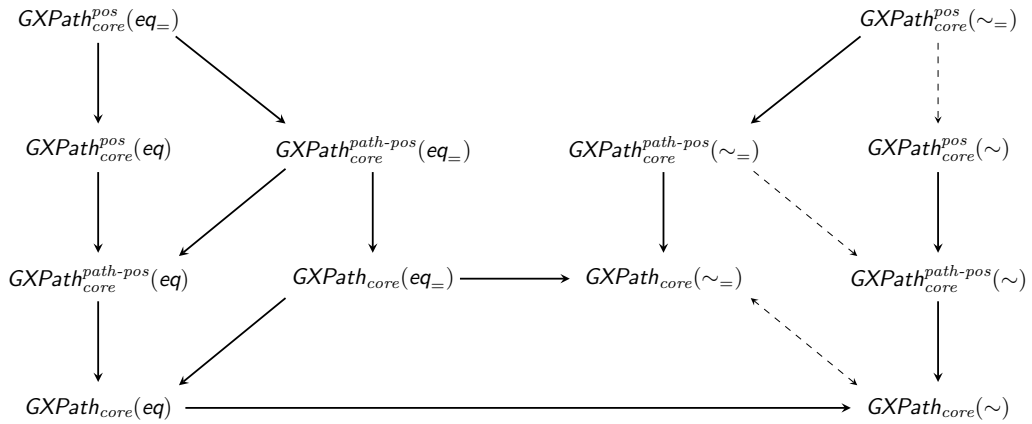
For the second item assume that there is an $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c})$ expression e_{eq} equivalent to q_{eq}. Take three arbitrary pairwise distinct data values x, y, z that are different from all the constants appearing in e_{eq} and let G_1 and G_2 be data graphs on nodes v_1, v_2, v_3, both with a-labeled edges from v_1 to v_2 and v_3, so that data values assigned to v_1 and v_2 are x and y, but in G_1, the node v_3 gets data value z while in G_2 it gets y. Then a straightforward

induction on $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c})$ expressions e that use only constants appearing in e_{eq} shows that $[\![e]\!]^{G_1} = [\![e]\!]^{G_2}$. Thus, q_{eq} cannot be equivalent to e_{eq}, since $[\![q_{\mathsf{eq}}]\!]^{G_1} \neq [\![q_{\mathsf{eq}}]\!]^{G_2}$. Note that this also shows that $\mathsf{GXPath}_{\mathrm{core}} \subsetneq \mathsf{GXPath}_{\mathrm{core}}(\mathsf{c})$ and $\mathsf{GXPath}_{\mathrm{core}} \subsetneq \mathsf{GXPath}_{\mathrm{core}}(\mathsf{eq})$. □

As shown in Proposition 4.2, the path positive and the positive fragments are strictly contained in the full language. Furthermore, Theorem 4.23 shows that the positive fragment cannot express node negation.

Another natural question here is to see how the expressive power of fragments that use both inequality and equality comparisons differs from the ones that compare data values for equality only. A subfragment of a $\sim$ fragment using only equalities (that is, subexpressions of the form $\alpha_{\neq}$ are not permitted) will be denoted by $\sim_{=}$, while the corresponding subfragment of an eq fragment will be denoted by $\mathsf{eq}_{=}$. The following theorem establishes the hierarchy of such fragments. It is important to note here that, in the absence of path negation, one can no longer simulate eq tests using the $\sim$ tests. Note that in order to avoid notational clutter we disregard constants in this comparison.

THEOREM 4.29. *The relative expressive power of GXPath_{core} fragments based on restricting negation in navigational features or data comparisons is given below.*



Here, an arrow from one fragment to another signifies that the source fragment is contained in the target one. A dashed line indicates containment and a full line proper containment. An analogous set of results holds for GXPath_{reg}.

PROOF. As just discussed, the positive fragments are strictly contained in the path-positive ones when they use the same type of tests. Furthermore, by Proposition 4.2, we know that the path-positive fragments are strictly contained in the full language allowing negation over paths when using the same type of data tests.

From Theorems 4.20 and 4.18 we also get that when path negation is present, $\sim$ fragments subsume the ones with eq tests, but not vice versa.

To show that $\mathsf{eq}_{=}$ fragments are strictly contained in corresponding eq fragments, we simply need to take a graph G_1 with two nodes holding the same data value, connected by an a-labeled edge in both directions and a graph G_2, this time with two nodes holding different data values, again connected by a-labeled edges. Both graphs also have self loops labeled a for each node. A straightforward induction on $\mathsf{GXPath}_{\mathrm{core}}(\mathsf{eq}_{=})$ expressions shows that the result of any expression is the same on both graphs. However, the $\langle a \neq \varepsilon \rangle$ differentiates the two. Note that this proof does not work for $\sim_{=}$ and $\sim$.

Proposition 4.19 and the discussion before Theorem 4.28 imply that $\mathsf{GXPath}_{\mathrm{core}}(\mathsf{eq}_{=})$ is strictly contained in $\mathsf{GXPath}_{\mathrm{core}}(\sim_{=})$.

Finally, to see that with the presence of path negation the $\sim_=$ fragment can define $\alpha_{\neq}$ observe that $\alpha_{\neq}$ is equivalent to $\overline{\alpha_=} \cap \alpha$. Naturally, this containment is not strict. □

Note that some of the inclusions in Theorem 4.29 are not proved to be strict. We do however conjecture that all of the one-way arrows denote strict inclusions.

5. CONJUNCTIVE QUERIES

A standard extension of RPQs is to that of *conjunctive RPQs*, or CRPQs [Calvanese et al. 2000; Deutsch and Tannen 2001; Florescu et al. 1998]. These add conjunctions of RPQs and existential quantification over variables, in the same way as conjunctive queries extend atomic formulae of relational calculus. We now look at similar extensions of queries from Section 3 and Section 4.

5.1. Conjunctive path queries

First we examine how the classes of conjunctive queries behave when their base atoms are queries from one of the path languages introduced in Section 3. Formally, they are defined as expression of the form

$$Ans(\overline{z}) \ := \bigwedge_{1 \leq i \leq m} x_i \xrightarrow{L_i} y_i, \tag{6}$$

where $m > 0$, each $x_i \xrightarrow{L_i} y_i$ is a query in one of the formalisms from Section 3, and $\overline{z}$ is a tuple of variables among $\overline{x}$ and $\overline{y}$. Note that the x_is and y_js are not required to be pairwise distinct. A query with the head $Ans()$ (i.e., no variables in the output) is called a *Boolean* query. To establish terminology we will talk about:

— Conjunctive regular data path queries (CRDPQs), when each $x_i \xrightarrow{L_i} y_i$ is an RDPQ,
— Conjunctive regular queries with memory (CRQMs), when each $x_i \xrightarrow{L_i} y_i$ is an RQM,
— Conjunctive regular queries with data tests (CRQDs), when each $x_i \xrightarrow{L_i} y_i$ is an RQD,

We will also use the name *conjunctive data path query (CDPQ)* for a query from any of the three classes just defined.

These queries extend their base atoms with conjunction, as well as existential quantification: variables that appear in the body but not in the head (i.e., variables in $\overline{x}$ and $\overline{y}$ but not $\overline{z}$) are assumed to be existentially quantified.

The semantics of a CDPQ Q of the form (6) over a data graph $G = \langle V, E, \rho \rangle$ is defined as follows. Given a valuation $\nu : \bigcup_{1 \leq i \leq m} \{x_i, y_i\} \to V$, we write $(G, \nu) \models Q$ if $(\nu(x_i), \nu(y_i))$ is in the answer of $x_i \xrightarrow{L_i} y_i$ on G, for each $i = 1, \ldots, m$. Then $Q(G)$ is defined as the set of all tuples $\nu(\overline{z})$ such that $(G, \nu) \models Q$, where $\nu(\overline{z})$ denotes the pointwise application of ν to $\overline{z}$. If Q is Boolean, we let $Q(G)$ be true if $(G, \nu) \models Q$ for some ν (that is, as usual, the empty tuple models the Boolean constant true, and the empty set models the Boolean constant false).

As before, we study data and combined complexity of the query evaluation problem, i.e. checking, for a CDPQ Q, a data graph G and a tuple of nodes $\overline{v}$, whether $\overline{v} \in Q(G)$ (for data complexity the query Q is fixed).

First, we show that for all the formalisms studied in the previous chapter, no cost is incurred by going from a single query to a conjunctive query as far as data complexity is concerned.

THEOREM 5.1. *The data complexity of conjunctive data path queries remains* NL*-complete if they are defined using RDPQs,RQMs, or RQDs.*

PROOF. Consider a query of the form (6) and let $\overline{z'}$ be the tuple of variables from $\overline{x}$ and $\overline{y}$ that is not present in $\overline{z}$. To check whether $\overline{v} \in Q(G)$, we need to check whether there exists a valuation $\overline{v'}$ for $\overline{z'}$ so that under that valuation each of the queries in the conjunction in (6) is true.

We know from the previous sections that checking whether $v \xrightarrow{L} v'$ evaluates to true for some nodes v, v' can be done with NL data complexity for all the formalisms mentioned in the theorem. Thus, given a data graph $G = \langle V, E, \rho \rangle$, we can enumerate all the tuples from $V^{|\overline{z'}|}$, and for each of them check the truth of all the queries in conjunction (6). Since we deal with data complexity, $|\overline{z'}|$ is fixed, and thus such an enumeration can be done in logarithmic space, showing that query evaluation remains in NL. Note that the NL algorithms can be composed here since they are independent one of another. □

For combined complexity, we have the same bounds for CRDPQs and CRQMs. For CRQDs we get NP-completeness, which matches the combined complexity of conjunctive queries and CRPQs.

THEOREM 5.2. *The combined complexity of conjunctive regular data path queries remains* PSPACE*-complete if they are specified using RDPQs or RQMs. It is* NP*-complete if they are specified using RQDs .*

PROOF. PSPACE-hardness follows from the corresponding results for *RQM*s, and NP-hardness follows from NP-hardness of relational conjunctive queries. Thus we show upper bounds. The algorithm (using notations from the proof of Theorem 5.1) is the same in all the cases: guess a tuple $\overline{v'}$ of nodes for $\overline{z'}$, and check whether all the queries in conjunction (6) are true. We know that for RDPQs and RQMs the latter can be done in PSPACE; since PSPACE is closed under nondeterministic guesses we have the PSPACE upper bound for combined complexity. For CRQDs, an NP upper bound for the algorithm follows from the PTIME bound for combined complexity for RQDs. □

5.2. Conjunctive GXPath

Conjunctive GXPath queries are defined as expressions of the form:

$$Ans(\overline{w}) \;:=\; \bigwedge_{1 \le i \le m} (x_i, \alpha_i, y_i) \wedge \bigwedge_{1 \le j \le m'} \psi_j(z_j), \tag{7}$$

where $m, m' \geq 0$, each α_i is a path expression, each ψ_j a node expression, and $\overline{w}$ is a tuple of variables among $\overline{x}$, $\overline{y}$ and $\overline{z}$. A query with the head $Ans()$ (i.e., no variables in the output) is called a *Boolean* query.

The semantics of a conjunctive GXPath query Q of the form (7) over a data graph $G = \langle V, E, \rho \rangle$ is defined as follows. Given a valuation $\nu : \bigcup_{1 \le i \le m} \{x_i, y_i\} \cup \bigcup_{1 \le j \le m'} \{z_j\} \to V$, we write $(G, \nu) \models Q$ if $(\nu(x_i), \nu(y_i))$ is in $[\![\alpha_i]\!]^G$, for each $i = 1, \ldots, m$ and $\nu(z_j) \in [\![\psi_j]\!]^G$, for $j = 1, \ldots, m'$. Then $Q(G)$ is defined as the set of all tuples $\nu(\overline{z})$ such that $(G, \nu) \models Q$. If Q is Boolean, we let $Q(G)$ be true if $(G, \nu) \models Q$ for some ν.

Example 5.3. Coming back to the example with actors and movies from Figure 1, we can now ask for people who have collaborated both with Kevin Bacon and Paul Erdős. This query is defined by:

$$Q(x) = (x, (\mathsf{actor}^- \cdot \mathsf{actor})^*[\mathsf{Kevin\ Bacon}^=], y) \wedge (x, (\mathsf{actor}^- \cdot \mathsf{actor})^*[\mathsf{Paul\ Erdős}^=], z).$$

Note that this query is expressible by GXPath with no conjunction (by using intersection), however, the syntax used by conjunctive queries is more intuitive, especially when one needs conjunction of three or more conditions. Furthermore, conjunction of four or more queries is no longer expressible, since GXPath is contained in $\mathcal{L}^3_{\infty,\omega}$ (Theorem 4.24).

Query answering	RDPQ	RQM	RQM over finite languages	RQD	GXPath
data complexity	NL-complete	NL-complete	NL-complete	NL-complete	PTIME
combined complexity	PSPACE-complete	PSPACE-complete	NP-complete	PTIME	PTIME

(a) for a single query

Query answering	CRDPQ	CRQM	CRQD	Conjunctive GXPath
data complexity	NL-complete	NL-complete	NL-complete	PTIME
combined complexity	PSPACE-complete	PSPACE-complete	NP-complete	NP-complete

(b) for conjunctive queries

Fig. 4. Summary of complexity results for classes of queries studied in this paper

If the database is further extended to include people who have co-written papers, we could also express the query returning people with a finite Erdős-Bacon number. For this, the second conjunct in the query Q would simply change to $(x, (\mathsf{author}^- \cdot \mathsf{author})^*[\mathsf{Erdős}^=], z)$, where an author edge connects each paper with one of its authors.

As before, we study data and combined complexity of the query evaluation problem, i.e. checking, for a query Q, a data graph G and a tuple of nodes $\overline{v}$, whether $\overline{v} \in Q(G)$ (for data complexity the query Q is fixed).

THEOREM 5.4.

— *Data complexity for conjunctive GXPath queries is in* PTIME.
— *Combined complexity is* NP*-complete.*

The data complexity bound easily follows from query evaluation bounds for GXPath queries. For combined complexity we do the standard guess and check algorithms, using again the fact that the language can be evaluated in PTIME. The NP lower bound follows from the result for CRPQs [Barceló et al. 2012].

6. SUMMARY AND CONCLUSIONS

Historically, querying graph data was done in two completely separate ways: either one would query the raw data residing in the graph while completely disregarding how the data is connected, or one would query only the topology of the model, determining intricate patterns connecting the data, but not doing any reasoning on the data itself. The main objective of this paper was to explore principles of good query language design that combines these two modes of querying. Namely, we propose languages that, in addition to being able to ask questions about the underlying topology of the model, also allow us to determine how the actual data changes while navigating in the graph.

Having studied how various data and navigational features affect the ability of the language to express relevant queries, as well as how they influence efficiency (a summary of this is given in Figure 4), we come to a conclusion that there are no clear winners when it comes to choosing a particular language, if the context is not known. Indeed, for a specific scenario we might value a certain set of functionalities above others, and consider a language allowing these functionalities best suited for our purposes. In a different setting we might dismiss the same language on grounds of high complexity. Because of this we can not bring one of the proposed languages forward as *the* language for graphs with data, however, we can point to good candidates when a specific capability is required.

For example, if we are interested solely in navigational queries a clear candidate would be GXPath, or some of its fragments. Not only does this language extend all of the previously studied navigational languages (with the only exception being extended RPQs [Barceló et al. 2012] which are incomparable with GXPath) but it is also closely connected to logic — both FO and PDL, and has efficient query evaluation algorithms matching those of previously studied navigational languages.

On the other hand, if we require the ability to use memory, for instance to ask queries that propagate data (in)equality along the path connecting two data points, it seems that RQMs are the way to go. Not only do these queries have high expressive power matching that of register automata, but their syntax is also clear and easily understandable. The price we have to pay in this case is high complexity — namely PSPACE-complete.

For highly efficient languages, the clear winner are RQDs, however, we might be willing to trade their somewhat limited expressive power for that of GXPath fragments with linear time query evaluation.

7. FUTURE WORK

The theoretical study that we undertook here enabled us to determine the practical potential of a query language. Most issues pertaining to the complexity cost of including certain features into a language have been settled. However, since mixing topology and data is a relatively new concept in graph databases, there are still many questions that remain open, and thus many possible directions for future research. We finish by briefly outlining some of them.

Comparisons with practical languages. Most existing languages for graph databases [Dex 2013; Neo4j 2013] do not even support navigational functionalities provided by regular path queries and have a relational mindset when it comes to comparing data values. One notable exception is SPARQL [Harris and Seaborne 2013], the native language for RDF data. Graph features of SPARQL are centered around property paths, which are similar to regular path queries with inverse [Calvanese et al. 2000]. These can be combined using standard relational operators, thus allowing the language to express unions of conjunctive regular path queries. Data value tests in SPARQL are allowed using the FILTER operator, which enables to test for data (in)equality in a set of CRPQ outputs. Determining the connection between SPARQL queries and graph languages is an important task and, as witnessed by the abundance of the literature on the problem [Libkin et al. 2013b; Reutter et al. 2015; Kostylev et al. 2015b], one that is not solved easily. There are several challenges that we are facing. To start with, RDF data is in several aspects different from graph databases and thus straightforward translations between the two settings are not always possible [Libkin et al. 2013b]. Furthermore, property paths are not precisely equivalent to RPQs since they operate over an infinite alphabet of IRIs [Kostylev et al. 2015b]. In addition, many graph constructions are not available in SPARQL due to the lack of a more general transitive closure operator. In fact, [Reutter et al. 2015] showed that there are several implementation issues that hamper the usability of SPARQL in the graph context. All of this tells us that capturing navigational SPARQL queries using graph languages is an an intriguing problem requiring much further investigation.

Besides expressivity comparisons, note also that languages such as SPARQL or SQL with transitive closure always have conjunctive queries built in, which has important complexity consequences. Since evaluation of conjunctive queries over graphs is NP-complete in combined complexity, so is the evaluation of SPARQL or SQL with transitive closure. Languages such as GXPath and NREs are fundamentally different since, by design, they can only express *tree-shaped* queries, that is, the underlying structure of GXPath-queries or NREs is always a tree. This is the main reason why GXPath queries can be evaluated in polynomial time combined complexity. The characterization of tractable queries in SPARQL and, in particular, tree-shaped SPARQL queries is still a topic of ongoing research, see, e.g. [Barceló et al. 2015].

Additional features. We have already explored how some basic add-ons, such as conjunctive queries, affect our languages. There are, of course, many other features that come into play when languages are applied, such as aggregation or allowing more freedom in manipulating the attribute data. For example we could compare string values for substrings, or

do arithmetical operations over integers. While a general theory for adding such languages to first-order logic is well developed [Kuper et al. 2000], it would be interesting to look how adding them can be accommodated into the languages we proposed. What we also hope to achieve is a syntax that would be more attractive to users who require multiple attributes per node (or edge). There are various options that present themselves here, as our languages are readily extensible to support this functionality, but some careful examination of actual requirements by various groups of users is needed to determine which syntax is better suited for such a language.

Using languages in different scenarios. We would also like to explore how our languages can be used in new application domains that require navigational and data patterns to be detected in the underlying model. One area that we had in mind is querying data and workflow provenance. Here one typically stores information about how data is created and modified and sometimes it is useful to have the ability to track the origins of such data. For example if a bug is found in a large software project it is important to locate the library, or the modification of code, that led to this bug. Another possible application is in description logics, since the underlying model there is basically that of a graph. Indeed, some preliminary results on this have already been published in [Kostylev et al. 2015a].

Static analysis. Static analysis of queries is an important component of query optimization, including tasks such as query containment or query equivalence. These problems are also relevant in other tasks, for instance virtual data integration. Some preliminary results on these problems are available in [Vrgoč 2014]. However, there are still many unresolved issues of interest, and known techniques from the XML context [Figueira 2010] do not seem to be readily transferable to graphs.

Incomplete information. Finally, it would be interesting to see how missing and incomplete data impacts graph languages. In a limited way, this problem was looked at in [Reutter 2013b; Barceló et al. 2014], which however only dealt with navigational aspects of graph languages and did not consider data values. From the study of incompleteness in XML [Abiteboul et al. 2006; Barceló et al. 2010], we know that the interaction between navigation and handling data values complicates the issue quite considerably. In fact, studies in the XML setting were restricted to very simplistic queries, essentially those returning relations, to apply the classical notion of certain answers to them. Recently, however, new techniques for understanding query answering over incomplete data in a data-model independent way have been introduced [Libkin 2014], and we hope to adapt them to the graph model.

ACKNOWLEDGMENTS

The authors would like to thank Juan Reutter, Tony Tan and the referees for their helpful comments. Part of this work was done when Vrgoč was at the University of Edinburgh. We acknowledge support of EPSRC grants G049165, J015377, and M025268, DFG grant MA 4938/2–1 and Millennium Nucleus Center for Semantic Web Research Grant NC120004.

REFERENCES

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

S. Abiteboul, L. Segoufin, and V. Vianu. 2006. Representing and querying XML with incomplete information. *ACM Transactions on Database Systems* 31, 1 (2006), 208–254.

S. Abiteboul and V. Vianu. 1999. Regular path queries with constraints. *J. Comput. Syst. Sci.* 3, 58 (1999), 428–452.

A.V. Aho. 1990. *Handbook of Theoretical Computer Science*. 255–300 pages.

N. Alechina, S. Demri, and M. de Rijke. 2003. A modal perspective on path constraints. *J. Log. Comput.* 13, 6 (2003), 939–956.

N. Alechina and N. Immerman. 2000. Reachability logic: An efficient fragment of transitive closure logic. *Logic Journal of the IGPL* 8, 3 (2000), 325–337.

H. Andréka, I. Németi, and I. Sain. 2001. *Handbook of Philosophical Logic* (2 ed.). Vol. 2. Springer, Chapter Algebraic logic.

R. Angles and C. Gutierrez. 2008. Survey of graph database models. *Comput. Surveys* 40, 1 (2008).

P. Barceló. 2013. Querying Graph Databases. In *32th ACM Symposium on Principles of Database Systems (PODS)*.

P. Barceló, D. Figueira, and L. Libkin. 2012. Graph logics with rational relations and the generalized intersection problem. In *27th Annual IEEE Symposium on Logic in Computer Science (LICS)*.

P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood. 2012. Expressive languages for path queries over graph-structured data. *ACM TODS* 38, 4 (2012).

P. Barceló, L. Libkin, A. Poggi, and C. Sirangelo. 2010. XML with incomplete information. *J. ACM* 58, 1 (2010), 1–62.

P. Barceló, L. Libkin, and J. L. Reutter. 2014. Querying Regular Graph Patterns. *J. ACM* 61, 1 (2014), 8:1–8:54.

P. Barceló, J. Pérez, and J.L. Reutter. 2012. Relative Expressiveness of Nested Regular Expressions. In *AMW*. 180–195.

P. Barceló, R. Pichler, and S. Skritek. 2015. Efficient Evaluation and Approximation of Well-designed Pattern Trees. In *ACM Symposium on Principles of Database Systems (PODS)*. 131–144.

M. Benedikt, W. Fan, and F. Geerts. 2008. XPath satisfiability in the presence of DTDs. *J. ACM* 55, 2 (2008).

M. Bojanczyk. 2010. Automata for data words and data trees. In *RTA*.

M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. 2011. Two-variable logic on words with data. *ACM TOCL* 12, 4 (2011).

M. Bojanczyk, A. Muscholl, T. Schwentick, and L. Segoufin. 2009. Two-variable logic on data trees and XML reasoning. *J. ACM* 56, 3 (2009).

M. Bojanczyk and P. Parys. 2011. XPath evaluation in linear time. *J. ACM* 58, 4 (2011).

P. Bouyer, A. Petit, and D. Thérien. 2001. An algebraic characterization of data and timed languages. In *CONCUR*. 248–261.

D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. 2000. Containment of conjunctive regular path queries with inverse. In *7th International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 176–185.

D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. 2009. An automata-theoretic approach to regular XPath. In *DBPL*. 18–35.

S. Cassidy. 2003. Generalizing XPath for directed graphs. In *Extreme Markup Languages*.

R. Cleaveland and B. Steffen. 1993. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. *Formal Methods in System Design* 2, 2 (1993), 121–147.

M. Consens and A.O. Mendelzon. 1990. GraphLog: A visual formalism for real life recursion. In *9th ACM Symposium on Principles of Database Systems (PODS)*. 404–416.

I. Cruz, A.O. Mendelzon, and P. Wood. 1987. A graphical query language supporting recursion. In *ACM Special Interest Group on Management of Data 1987 Annual Conference (SIGMOD)*. 323–330.

S. Demri and R. Lazić. 2009. LTL with the freeze quantifier and register automata. *ACM TOCL* 10, 3 (2009).

S. Demri, R. Lazić, and D. Nowak. 2007. On the freeze quantifier in constraint LTL: Decidability and complexity. *Information and Computation* 205, 1 (2007), 2–24.

A. Deutsch and V. Tannen. 2001. Optimization properties for classes of conjunctive regular path queries. In *8th International Workshop on Database Programming Languages (DBPL)*. 21–39.

Dex 2013. DEX query language, Sparsity Technologies. http://www.sparsity-technologies.com/dex.php. (2013).

Facebook. 2014. *Graph Search*. https://www.facebook.com/about/graphsearch.

W. Fan. 2012. Graph pattern matching revised for social network analysis. In *ICDT*. 8–21.

D. Figueira. 2009. Satisfiability of downward XPath with data equality tests. In *28th ACM Symposium on Principles of Database Systems (PODS)*. 197–206.

D. Figueira. 2010. *Reasoning on words and trees with data*. Ph.D. Dissertation. ÉNS de Cachan.

G. H. L. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu. 2011. Relative expressive power of navigational querying on graphs. In *ICDT*. 197–207.

D. Florescu, A. Y. Levy, and D. Suciu. 1998. Query Containment for Conjunctive Queries with Regular Expressions. In *PODS*. 139–148.

S. Fortune, J. Hopcroft, and J. Wyllie. 1980. The directed homeomorphism problem. *Theoretical Computer Science* 10 (1980), 111–121.

G. Gottlob, C. Koch, and R. Pichler. 2005. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.* 30, 2 (2005), 444–491.

Gremlin 2013. Gremlin Language. https://github.com/tinkerpop/gremlin/wiki. (2013).

C. Gutierrez, C. Hurtado, A. O. Mendelzon, , and J. Pérez. 2011. Foundations of semantic web databases. *J. Comput. System Sci.* 77, 3 (2011), 520–541.

S. Harris and A. Seaborne. 2013. SPARQL 1.1 Query Language. W3C Recommendation. http://www.w3.org/TR/sparql11-query/. (March 2013).

M. Kaminski and N. Francez. 1994. Finite memory automata. *Theoretical Computer Science* 134, 2 (1994), 329–363.

M. Kaminski and T. Tan. 2006. Regular expressions for languages over infinite alphabets. *Fundamenta Informaticae* 69, 3 (2006), 301–318.

M. Kaminski and T. Tan. 2008. Tree automata over infinite alphabets. In *Pillars of Computer Science*. 386–423.

M. Kay. 2004. *XPath 2.0 Programmer's Reference*. Wrox.

E. V. Kostylev, J. L. Reutter, M. Romero, and D. Vrgoč. 2015b. SPARQL with Property Paths. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*. 3–18.

E. V. Kostylev, J. L. Reutter, and D. Vrgoč. 2015a. XPath for DL Ontologies. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. 1525–1531.

G. M. Kuper, L. Libkin, and J. Paredaens (Eds.). 2000. *Constraint Databases*. Springer.

M. Lange. 2006. Model checking propositional dynamic logic with all extras. *J. Applied Logic* 4, 1 (2006), 39–49.

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

L. Libkin. 2014. Certain answers as objects and knowledge. In *Principles of Knowledge Representation and Reasoning (KR)*. 328–337.

L. Libkin, W. Martens, and D. Vrgoč. 2013a. Querying Graph Databases with XPath. In *ICDT*.

L. Libkin, J. L. Reutter, and D. Vrgoč. 2013b. TriAL for RDF: Adapting Graph Query Languages for RDF Data. In *PODS*.

L. Libkin and D. Vrgoč. 2012a. Regular expressions for data words. In *LPAR*. 274–288.

L. Libkin and D. Vrgoč. 2012b. Regular Path Queries on Graphs with Data. In *ICDT*. 74–85.

K. Losemann and W. Martens. 2012. The complexity of evaluating path expressions in SPARQL. In *PODS*. 101–112.

M. Marx. 2003. XPath and Modal Logics of Finite DAG's. In *TABLEAUX*. 150–164.

M. Marx. 2005. Conditional XPath. *ACM Trans. Database Syst.* 30, 4 (2005), 929–959.

Neo4j 2013. Neo4j, The graph database. http://www.neo4j.org/. (2013).

F. Neven, Th. Schwentick, and V. Vianu. 2004. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.* 5, 3 (2004), 403–435.

J. Pérez, M. Arenas, and C. Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems* 34, 3 (2009).

J. Pérez, M. Arenas, and C. Gutierrez. 2010. nSPARQL: A navigational language for RDF. *Journal of Web Semantics* 8, 4 (2010), 255–270.

J. L. Reutter. 2013a. Containment of Nested Regular Expressions. CoRR abs/1304.2637. (2013).

J. L. Reutter. 2013b. *Graph Patterns: Structure, Query Answering and Applications in Schema Mappings and Formal Language Theory*. Ph.D. Dissertation. School of Informatics, University of Edinburgh.

J. L. Reutter, A. Soto, and D. Vrgoč. 2015. Recursion in SPARQL. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*. 19–35.

R. Ronen and O. Shmueli. 2009. SoQL: a language for querying and creating data in social networks. In *25th International Conference on Data Engineering (ICDE)*. 1595–1602.

H. Sakamoto and D. Ikeda. 2000. Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.* 231, 2 (2000), 297–308.

M. San Martín and C. Gutierrez. 2009. Representing, querying and transforming social networks with RDF/SPARQL. In *6th European Semantic Web Conference (ESWC)*. 293–307.

L. Segoufin. 2006. Automata and logics for words and trees over an infinite alphabet. In *CSL*. 41–57.

L. Segoufin. 2007. Static analysis of XML processing with data values. *SIGMOD Record* 36, 1 (2007), 31–38.

A. Tarski and S. Givant. 1987. *A Formalization of Set Theory Without Variables*. AMS.

B. ten Cate. 2006. The expressivity of XPath with transitive closure. In *25th ACM Symposium on Principles of Database Systems (PODS)*. 328–337.

B. ten Cate and M. Marx. 2007. Navigational XPath: calculus and algebra. *Sigmod Record* 36, 2 (2007), 19–26.

D. Vrgoč. 2014. *Querying graphs with data*. Ph.D. Dissertation. School of Informatics, University of Edinburgh.

P.T. Wood. 2012. Query Languages for Graph Databases. *Sigmod Record* 41, 1 (2012), 50–60.

Online Appendix to: Querying Graphs with Data

LEONID LIBKIN, University of Edinburgh
WIM MARTENS, Universität Bayreuth
DOMAGOJ VRGOČ, PUC Chile and Center for Semantic Web Research

A. PROOF OF PROPOSITION 3.13

We prove this by induction on the structure of e. Note that the initial assignment of $\mathcal{A}_e$ is not specified in advance. We will simply put the assignment in as needed, since it does not change the structure of the underlying automaton. In what follows we will identify the vector $\overline{x}$ of variables with the set of registers (i.e. positions) it corresponds to. For example the vector (x_3, x_5) will correspond to the set $I = \{3, 5\}$ of registers.

If $(e, w, \sigma) \vdash \sigma'$, we will write $w \in L(e, \sigma, \sigma')$ and similarly if $\mathcal{A}_e = (Q, q_0, F, \overline{\bot}, \delta)$ started with σ accepts w with σ' in the registers, we write $w \in L(\mathcal{A}_e, \sigma, \sigma')$.

— If $e = \varepsilon$, then $\mathcal{A}_e = (Q, q_0, F, \overline{\bot}, \delta)$, where $Q = \{d\} \cup \{w\}$ is the set of states, $q_0 = d$ is the initial state, $F = \{w\}$ the set of final states and the only transition is $(d, \varepsilon, \emptyset, w)$.

— If $e = a$, for some $a \in \Sigma$ then $\mathcal{A}_e = (Q, q_0, F, \overline{\bot}, \delta)$, where $Q = \{d_1, d_2\} \cup \{w_1, w_2\}$ is the set of states, $q_0 = d_1$ the initial state, $F = \{w_2\}$ the final state and the transition functions are as follows: $\delta_w = \{(w_1, a, d_2)\}$ is the word transition relation, and $\delta_d = \{(d_1, \varepsilon, \emptyset, w_1), (d_2, \varepsilon, \emptyset, w_2)\}$ is the data transition relation.

— If $e = e_1 + e_2$ then by the inductive hypothesis we already have automata $\mathcal{A}_{e_1} = (Q_1, d_1, F_1, \overline{\bot}, \delta_1)$ and $\mathcal{A}_{e_2} = (Q_2, d_2, F_2, \overline{\bot}, \delta_2)$ with the desired property. The registers of $\mathcal{A}_e$ will be the union of registers of $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$. To obtain the desired automaton we set $\mathcal{A}_e = (Q, d_0, F, \overline{\bot}, \delta)$, where
  — $Q = Q_1 \cup Q_2 \cup \{d_0\}$, where d_0 is a new data state,
  — $F = F_1 \cup F_2$,
  — To δ we add all transitions from $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ and in addition, for every transition $(d, c, I, w) \in \delta_1 \cup \delta_2$, where $d = d_1$, or $d = d_2$, we add a transition (d_0, c, I, w).

  To see that this automaton has the desired property assume that $w \in L(e_1 + e_2, \sigma, \sigma')$. This means $(e_1 + e_2, w, \sigma) \vdash \sigma'$. By definition, $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$. By the induction hypothesis it follows that either $\mathcal{A}_{e_1}$, or $\mathcal{A}_{e_2}$ accepts w and halts with σ' in the registers (when started with σ). From this it is clear that $\mathcal{A}_e$ can simulate the same accepting run when started with σ in the registers (by using the transition from d_0 to the appropriate automaton and continuing on the same run there). (Note that all conclusions here are equivalences.)

— If $e = \downarrow \overline{x}.e_1$ then again by the induction hypothesis we have $\mathcal{A}_{e_1} = (Q_1, d_1, F_1, \overline{\bot}, \delta_1)$ with the desired property. The automaton for $\mathcal{A}_e$ is defined as $\mathcal{A}_e = (Q_1 \cup \{d_0\}, d_0, F_1, \overline{\bot}, \delta)$, where d_0 is a new data state and δ contains all the transitions of $\mathcal{A}_{e_1}$ and in addition, for every transition (d_1, c, I, w), going from the initial state of $\mathcal{A}_{e_1}$, we add a transition $(d_0, c, I \cup \overline{x}, w)$ to δ. The registers of $\mathcal{A}_e$ are the union of registers of $\mathcal{A}_{e_1}$ and $|\overline{x}|$ new registers.

  To see the equivalence, assume that $w \in L(e, \sigma, \sigma')$. By definition $(e, w, \sigma) \vdash \sigma'$. It follows that $(e_1, w, \sigma_{\overline{x}=v_1}) \vdash \sigma'$, where v_1 is the first data value in w and $\sigma_{\overline{x}=v_1}$ is the same as σ except that every register in $\overline{x}$ contains v_1. By the induction hypothesis we know that $\mathcal{A}_{e_1}$ with $\sigma_{\overline{x}=v_1}$ as initial assignment has an accepting run on w ending with σ' in the

DOI:http://dx.doi.org/10.1145/0000000.0000000

registers. But then $\mathcal{A}_e$ starting with σ in the registers can go through the same run with the exception that the first transition will change σ to $\sigma_{\overline{x}=v_1}$ and since all other transitions are the same we have the desired result. (Note that all conclusions here are equivalences.)
It is important to note that potential confusion of the variables will cause no conflicts. To see this assume we have a transition (d_1, c, I, w) in $\mathcal{A}_{e_1}$ and we start with σ as initial assignment. If I and $\overline{x}$ have variables in common it will not matter, since all of them will get replaced by the same value, namely the first data value of w. This means that the first step of the run will end up with the same result. Also note that no transition in δ_d with d_1 as the first component will have $c \neq \varepsilon$, since this would amount to an expression starting with a condition, something disallowed by our syntax.

— If $e = e_1[c]$ then let $\mathcal{A}_{e_1} = (Q_1, d_1, F_1, \overline{\perp}, \delta_1)$ be an automaton for e_1 as before. We define $\mathcal{A}_e = (Q, d_1, F, \overline{\perp}, \delta)$ where $Q = Q_1 \cup \{w_f\}$, with w_f a new state, $F = \{w_f\}$ and for every transition (d, c', I, w) where $w \in F_1$ we add a transition $(d, c' \wedge c, I, w_f)$ to $\mathcal{A}_e$. We have to add a new state simply because our original automaton could have looped back from some final state.
To get the equivalence assume again that $w \in L(e, \sigma, \sigma')$. By definition $(e_1, w, \sigma) \vdash \sigma'$ and $\sigma', v \models c$, where v is the last data value in w. From the induction hypothesis we get an accepting run of $\mathcal{A}_{e_1}$ with σ as initial configuration and σ' as final one. But since $\sigma', v \models c$ instead of the last transition we can simply make a transition to w_f in $\mathcal{A}_e$ (since all other transitions are the same). We again notice that all the implications can be reversed, i.e. we can prove the equivalence.

— If $e = e_1 \cdot e_2$, take again $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ as above. The automaton for e is simply the union of the previous two automata, but in addition to the already existing transitions we add the following: for every (d, c, I, w) in $\mathcal{A}_{e_1}$, where $w \in F_1$ and for every (d_2, c', I', w') in $\mathcal{A}_{e_2}$, where d_2 is the initial state of $\mathcal{A}_{e_2}$, we add $(d, c \wedge c', I \cup I', w')$ to δ. Note that I is going to be an empty set, since we work with well formed expressions. We also make d_1 the initial state and F_2 the set of final states. The registers of $\mathcal{A}_e$ are again the union of registers of $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$.
To get the desired result once again assume that $w \in L(e, \sigma, \sigma')$. This means $(e, w, \sigma) \vdash \sigma'$, which implies that there exists some σ'' and a splitting $w = w_1 \cdot w_2$ of w such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$. By the induction hypothesis we know that there is an accepting run of $\mathcal{A}_{e_1}$ on w_1 starting with σ and ending with σ'' in the registers and also an accepting run of $\mathcal{A}_{e_2}$ on w_2 starting with σ'' and ending with σ' in the registers. But we can simply combine these two runs into an accepting run of $\mathcal{A}_e$ on w. We do so by setting σ as initial assignment and tracing the run of $\mathcal{A}_{e_1}$ till the final state. Now instead of taking the last transition we will take one of the newly added transitions from the next to final state in $\mathcal{A}_{e_1}$ to the next to first state in $\mathcal{A}_{e_2}$. Note that we can do this since we know there is an accepting run of $\mathcal{A}_{e_2}$ on w_2 and since $w = w_1 \cdot w_2$, so their last and first data value, respectively, coincide. Note that at this point we end up with σ'' in the registers and can continue the accepting run of $\mathcal{A}_{e_2}$ and thus $\mathcal{A}_e$.
Conversely, if we have an accepting run of $\mathcal{A}_e$ on w, we split the run, and thus the path, into the part before and after taking the new transition added while constructing the automaton. Note that we have to take this transition in order to pass from the initial state, which is in $\mathcal{A}_{e_1}$ part of $\mathcal{A}_e$, to a final state, which is in a $\mathcal{A}_{e_2}$ part of $\mathcal{A}_e$. From this it follows that $w \in L(e)$.

— If $e = e_1^+$, then let again $\mathcal{A}_{e_1}$ be the automaton from the induction hypothesis. Note first that this automaton has at least four states, since $\mathrm{Proj}(e_1) \neq \varepsilon$, where $\mathrm{Proj}(e)$ denotes the projection to the finite alphabet Σ, and transitions going directly from initial to final state can only accept the empty word, so they will not alter computations or acceptance. We let the automaton for e be the same as the one for e_1, but we add the following transitions: for every (d, c, I, w) with $w \in F_1$ and for every (d_1, c', I', w'), where d_1 is the initial state

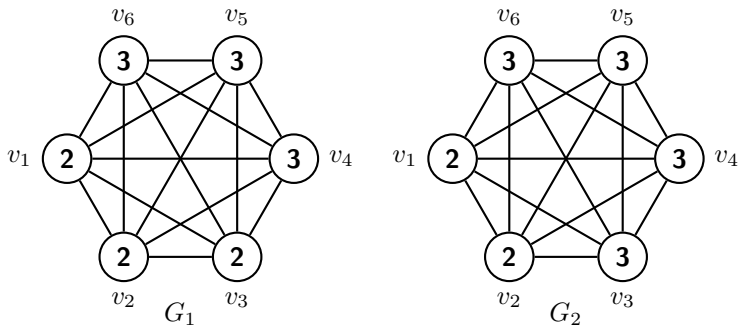


Fig. 5.

of $\mathcal{A}_{e_1}$, we add $(d, c \wedge c', I \cup I', w')$ to our transition function, thus bypassing the last and the first state.
Assume now that $(e, w, \sigma) \vdash \sigma'$. Then either $(e_1, w, \sigma) \vdash \sigma'$, so we are done by the induction hypothesis, or $w = w_1 \cdots w_k$ with $k \geq 2$ and valuations $\sigma_1, \ldots, \sigma_{k+1}$ exist such that $(e_1, w_i, \sigma_i) \vdash \sigma_{i+1}$ for $i = 1, \ldots, k$. But then by the induction hypothesis we have computations of $\mathcal{A}_{e_1}$ with σ_i as the initial assignment and σ_{i+1} as final assignment that accept w_i, for $i = 1, \ldots, k$. Note that this actually means that we start with σ, do a computation for w_1, end with σ_2 in the registers, then take the new transition bypassing the end state for this computation and thus starting the computation with σ_2 in the registers (and updating the registers as dictated by the first transition in the new cycle), etc., until we reach σ' after reading w_k, thus accepting w.
For the converse, if $\mathcal{A}_e$ accepts w when started with σ and ended with σ' then we simply split the data path for every time we take the additional transitions added in the construction of $\mathcal{A}_e$. From this we get computations of $\mathcal{A}_{e_1}$ on sub-paths with intermediate valuations. By the induction hypothesis we have acceptance of these subpaths by e_1 with appropriate valuations and thus the membership of the entire path w in $L(e, \sigma, \sigma')$.

This concludes the proof. It is straightforward to see that the construction can be done in PTIME.

B. PROOF OF PROPOSITION 4.19

PROOF. Here we prove that even though $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c}, \mathsf{eq})$ can test if a node has an a-successor with the same data value by the means of expression $\langle \varepsilon = a \rangle$, which will return the set $\{v \in V \mid \exists v' \in V \text{ and } (v, v') \in [\![a_=]\!]^G\}$, it has no means of retrieving that specific successor.

We will first prove the result without constant tests. To prove that $a_=$ is not expressible in $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{eq})$ over graphs we will give two graphs G_1 and G_2, such that $[\![a_=]\!]^{G_1} \neq [\![a_=]\!]^{G_2}$, but for every $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{eq})$ query e we have $[\![e]\!]^{G_1} = [\![e]\!]^{G_2}$. Both G_1 and G_2 are based on the graph K_6, that is, the complete graph with six vertices. We will therefore have $V = \{v_1, \ldots, v_6\}$ as the set of vertices in both graphs, and the data values attached to the vertices in G_1 are 2,2,2,3,3,3 and in G_2 they are 2,2,3,3,3,3. All the edges in both G_1 and G_2 are labeled a. The graphs G_1 and G_2 are pictured in Figure 5. It follows from the definitions that $(v_2, v_3) \in [\![a_=]\!]^{G_1}$, while $(v_2, v_3) \notin [\![a_=]\!]^{G_2}$. We conclude that $[\![a_=]\!]^{G_1} \neq [\![a_=]\!]^{G_2}$.

We now show that for every $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{eq})$ query e we have $[\![e]\!]^{G_1} = [\![e]\!]^{G_2}$. In particular we show the following:

— For every path query α one of the following holds:
  — $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = \emptyset$, or

— $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(V)$, or
— $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = V^2$, or
— $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = V^2 - Id(V)$.
— For every node query φ one of the following holds:
— $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$, or
— $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = V$.

Here $Id(V)$ stands for the set $\{(x,x) \mid x \in V\}$, with V the set of vertices in G_1 and G_2.

We prove this claim by induction on the structure of our $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{eq})$ expression e. The base cases trivially follow. For the induction step assume that our claim is true for the expressions of lower complexity. We proceed by cases.

— If $\alpha = [\varphi]$ then by the inductive hypothesis we have two cases.
— Either $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$, in which case $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = \emptyset$,
— Or $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = V$, in which case $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(V)$.
— If $\alpha = \alpha' \cup \beta'$ then the claim follows from the induction hypothesis and the fact that the set $\{\emptyset, V^2, V^2 - Id(V), Id(V)\}$ is closed under taking unions.
— If $\alpha = \alpha' \cdot \beta'$ we proceed as follows.
Note first that $[\![\alpha]\!]^{G_1} = \emptyset$ iff $[\![\alpha']\!]^{G_1} = \emptyset$ or $[\![\beta']\!]^{G_1} = \emptyset$ (this follows from the inductive hypothesis about the structure of the answers, since for every other case the sets have nonempty composition). This is now equivalent to the same being true in G_2 and thus to $[\![\alpha]\!]^{G_2} = \emptyset$.
If $[\![\alpha]\!]^{G_1} \neq \emptyset$ then we know that both $[\![\alpha']\!]^{G_1}$ and $[\![\beta']\!]^{G_1}$ belong to $\{V^2, V^2 - Id(V), Id(V)\}$. The claim now simply follows from the inductive hypothesis and the fact that the set $\{V^2, V^2 - Id(V), Id(V)\}$ is closed under composition of relations.
— If $\alpha = \overline{\alpha'}$ we have four cases.
— In case that $[\![\alpha']\!]^{G_1} = [\![\alpha']\!]^{G_2} = \emptyset$ we have $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = V^2$.
— In case that $[\![\alpha']\!]^{G_1} = [\![\alpha']\!]^{G_2} = V^2$ we have $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = \emptyset$.
— In case that $[\![\alpha']\!]^{G_1} = [\![\alpha']\!]^{G_2} = V^2 - Id(V)$ we have $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(V)$.
— In case that $[\![\alpha']\!]^{G_1} = [\![\alpha']\!]^{G_2} = Id(V)$ we have $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = V^2 - Id(V)$.
— If $\alpha = \alpha'^*$ we have the same situation as in the previous case. In particular we know that the transitive closures in each case will be the same.
— If $\varphi = \neg\varphi$ we have the following.
— In case that $[\![\varphi']\!]^{G_1} = [\![\varphi']\!]^{G_2} = V$ we have $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$.
— In case that $[\![\varphi']\!]^{G_1} = [\![\varphi']\!]^{G_2} = \emptyset$ we have $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = V$.
— If $\varphi = \varphi' \wedge \psi'$ the claim easily follows.
— If $\varphi = \langle\alpha\rangle$ we consider the value of $[\![\alpha]\!]^{G_1}$.
— In case that $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = \emptyset$ we get $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$.
— In case that $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = V^2, Id(V)$, or $V^2 - Id(V)$ we get $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = V$.
— If $\varphi = \langle\alpha = \beta\rangle$ we proceed by cases, depending on the value of $[\![\alpha]\!]^{G_1}$ and $[\![\beta]\!]^{G_1}$.
Note that if either equals $\emptyset$ we get that $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$. There are now nine possible cases remaining.
(1) $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(V)$ and $[\![\beta]\!]^{G_1} = [\![\beta]\!]^{G_2} = Id(V)$ implies that $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = V$.
(2) $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(V)$ and $[\![\beta]\!]^{G_1} = [\![\beta]\!]^{G_2} = V^2$ implies that $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = V$.
(3) $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(V)$ and $[\![\beta]\!]^{G_1} = [\![\beta]\!]^{G_2} = V^2 - Id(V)$ implies that $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = V$.
(4) All the remaining cases have the same result.
— If $\varphi = \langle\alpha \neq \beta\rangle$ we proceed by cases, depending of the value of $[\![\alpha]\!]^{G_1}$ and $[\![\beta]\!]^{G_1}$.
Note that if either equals $\emptyset$ we get that $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$. Just as for $\langle\alpha = \beta\rangle$ we have nine cases. It is easily verified that we have $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = V$ for each case, except when $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(V)$ and $[\![\beta]\!]^{G_1} = [\![\beta]\!]^{G_2} = Id(V)$. In this case we get $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$.

To extend the induction to work for constants, we assume the contrary. Let e be an expression defining $a_=$. We exchange the data values 2 and 3 in our graphs G_1 and G_2 with any two data values that do not appear as constants in e. The proof is now the same as in the case without constants. This completes the proof. □