Recursive Programs for Document Spanners

Liat Peterfreund[1]
Technion, Haifa 32000, Israel
liatpf@cs.technion.ac.il

Balder ten Cate
Google, Inc., Mountain View, CA 94043, USA
balder.tencate@gmail.com

Ronald Fagin
IBM Research – Almaden, San Jose, CA 95120, USA
fagin@us.ibm.com

Benny Kimelfeld
Technion, Haifa 32000, Israel
bennyk@cs.technion.ac.il

Abstract

A document spanner models a program for Information Extraction (IE) as a function that takes as input a text document (string over a finite alphabet) and produces a relation of spans (intervals in the document) over a predefined schema. A well-studied language for expressing spanners is that of the regular spanners: relational algebra over regex formulas, which are regular expressions with capture variables. Equivalently, the regular spanners are the ones expressible in non-recursive Datalog over regex formulas (which extract relations that constitute the extensional database). This paper explores the expressive power of recursive Datalog over regex formulas. We show that such programs can express precisely the document spanners computable in polynomial time. We compare this expressiveness to known formalisms such as the closure of regex formulas under the relational algebra and string equality. Finally, we extend our study to a recently proposed framework that generalizes both the relational model and the document spanners.

2012 ACM Subject Classification Theory of computation → Complexity theory and logic; Information systems → Relational database model; Information systems → Data model extensions

Keywords and phrases Information Extraction, Document Spanners, Polynomial Time, Recursion, Regular Expressions, Datalog.

Digital Object Identifier 10.4230/LIPIcs.ICDT.2019.10

Funding *Liat Peterfreund*: Supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel Cyber Bureau and by the Israel Science Foundation (ISF) Grant 1295/15.
Benny Kimelfeld: Supported by the Israel Science Foundation (ISF) Grant 1295/15.

1 Introduction

The abundance and availability of valuable textual resources position text analytics as a standard component in data-driven workflows. To facilitate the incorporation of such resources, a core operation is the extraction of structured data from text, a classic task known as Information Extraction (IE). This task arises in a large variety of domains, including healthcare analysis [28], social media analysis [4], customer relationship management [2], and machine log analysis [12]. IE also plays a central role in cross-domain computational challenges such as Information Retrieval [30] and knowledge-base construction [15, 26, 27, 29].

[1] The work was done while the author was at IBM Research – Almaden.

Rule-based IE is incorporated in commercial systems and academic prototypes for text analytics, either as a standalone extraction language or within machine-learning models. IBM's SystemT [20] exposes an SQL-like declarative language, *AQL* (Annotation Query Language), for programming IE. Conceptually, AQL supports a collection of "primitive" extractors of relations from text (e.g., tokenizer, dictionary lookup, part-of-speech tagger and regular-expression matcher), together with a relational algebra for manipulating these relations. Similarly, in Xlog [25], user-defined functions are used as primitive extractors, and non-recursive Datalog is, again, allowed for relation manipulation. In DeepDive [24, 26], rules are used to generate features that are translated into the factors of a statistical model with machine-learned parameters. Feature declaration combines, once again, primitive extractors of relations alongside relational operators on these relations. In addition to the above, different Datalog-like formalisms for IE were previously suggested and studied, including monadic Datalog over trees as web-page languages [13], and a framework for annotating CSV documents [3].

The framework of *document spanners* (or just *spanners* for short) [7] captures the above IE methodology: a spanner is a function that extracts from a document a relation over text intervals, called *spans*, using either a primitive extractor (e.g., a regular expression) or a relational query on top of primitive extractors. More formally, by a *document* we refer to a string $\mathbf{s}$ over a finite alphabet, and a *span* of $\mathbf{s}$ represents a substring of $\mathbf{s}$ by its start and end positions. A spanner is a function P that maps every string $\mathbf{s}$ into a relation $P(\mathbf{s})$, over a fixed schema S_P, over the spans of $\mathbf{s}$. The most studied spanner language is that of the *regular* spanners: primitive extraction is via *regex formulas*, which are regular expressions with capture variables, and relational manipulation is via positive relational algebra: projection, natural join, and union (while difference is expressible and not explicitly needed) [7]. Equivalently, the regular spanners are the ones expressible in non-recursive Datalog, where regex formulas are playing the role of the Extensional Data Base (EDB), that is, the input database [8].

By adding string-equality selection on span variables, Fagin et al. [7] establish the extended class of *core* spanners, viewed as the core language for AQL. A syntactically different language for spanners is SpLog, which is based on the *existential theory of concatenation*, and was shown by Freydenberger [9] to have precisely the expressiveness of core spanners. Such spanners can express more than regular spanners. A simple example is the spanner that extracts from the input $\mathbf{s}$ all spans x and y such that the string $\mathbf{s}_x$ spanned by x is equal to the string $\mathbf{s}_y$ spanned by y. The class of core spanners does not behave as well as that of the regular spanners; for instance, core spanners are not closed under difference, while regular spanners are. Fagin et al. [7] prove this by showing that no core spanner extracts all spans x and y such that $\mathbf{s}_x$ is *not* a substring of $\mathbf{s}_y$. The proof is based on the *core simplification lemma*: every core spanner can be represented as a regular spanner followed by a sequence of string equalities and projections. The same technique has been used for showing that no core spanner extracts all pairs x and y of spans having the same *length* [7].

In this paper we explore the power of *recursion* in expressing spanners. The motivation came from the SystemT developers, who have interest in recursion for various reasons, such as programming basic natural-language parsers by means of context-free grammars [19]. Specifically, we consider the language RGXlog of spanners that are defined by means of Datalog where, again, regex formulas play the role of EDB relations, but this time recursion is allowed. More precisely, given a string $\mathbf{s}$, the regex formulas extract EDB relations from $\mathbf{s}$, and a designated relation Out captures the output of the program. Observe that such a program operates exclusively over the domain of spans of the input string. In particular,

the output is a relation over spans of $\mathbf{s}$, and hence, RGXlog is yet another representation language for spanners. As an example, the following program emits all pairs x and y of spans of equal lengths. (See Section 3 for the formal definition of the syntax and semantics.)

- $\text{EQL}(x, y) \leftarrow \langle x\{\epsilon\}\rangle, \langle y\{\epsilon\}\rangle$
- $\text{EQL}(x, y) \leftarrow \langle x\{x'\{.^*\}.\}\rangle, \langle y\{y'\{.^*\}.\}\rangle, \text{EQL}(x', y')$

The first rule states that two empty spans have same length. The second rule states that two spans x and y have equal lengths if they are obtained by adding a single symbol (represented by dot) to spans x' and y', respectively, of equal lengths.

We explore the expressiveness of RGXlog. Without recursion, RGXlog captures precisely the regular spanners [8]. With recursion, several observations are quite straightforward. First, we can write a program that determines whether x and y span the same string. Hence, we have string equality without explicitly including the string-equality predicate. It follows that every core spanner can be expressed in RGXlog. Moreover, RGXlog can express more than core spanners, an example being expressing that two spans have the same length (which the above program shows can be expressed in RGXlog, but which, as said earlier, is not expressible by a core spanner [7]). What about upper bounds? A clear upper bound is *polynomial time*: every RGXlog program can be evaluated in polynomial time (under data complexity, where the spanner is fixed and the input consists of only the string), and hence, RGXlog can express only spanners computable in polynomial time.

We begin our investigation by diving deeper into the relationship between RGXlog and core spanners. The inexpressiveness results to date are based on the aforementioned core simplification lemma [7]. The proof of this lemma heavily relies on the absence of the difference operator in the algebra. In fact, Freydenberger and Holldack [10] showed that it is unlikely that in the presence of difference, there is a result similar to the core simplification lemma. So, we extend the algebra of core spanners with the difference operator, and call a spanner of this extended language a *generalized core spanner*. We then ask whether *(a)* every generalized core spanner can be expressed in RGXlog (whose syntax is positive and excludes difference/negation), and *(b)* RGXlog can express only generalized core spanners.

The answer to the first question is positive. We establish a negative answer to the second question by deploying the theory of *Presburger arithmetic* [23]. Specifically, we consider Boolean spanners on a unary alphabet. Each such spanner can be viewed as a predicate over natural numbers: the lengths of the strings that are accepted (evaluated to **true**) by the spanner. We prove that every predicate expressible by a Boolean generalized core spanner is also expressible in Presburger arithmetic (first-order theory of the natural numbers with the addition (+) binary function and the constant 0 and 1). Yet, we show a very simple RGXlog program that expresses a predicate that is *not* expressible in Presburger arithmetic, namely being a power of two [17].

We prove that RGXlog can express *every* spanner computable in polynomial time. Formally, recall that a spanner is a function P that maps an input string $\mathbf{s}$ into a relation $P(\mathbf{s})$, over a fixed schema S_P, over the spans of $\mathbf{s}$. We prove that the following are equivalent for a spanner P: *(a)* P is expressible in RGXlog, *and (b)* P is computable in polynomial time. As a special case, Boolean RGXlog captures exactly the polynomial-time languages.

Related formalisms that capture polynomial time include the *Range Concatenation Grammars* (RCG) [5]. In RCG, the grammar defines derivation rules for reducing the input string into the empty string; if reduction succeeds, then the string is accepted. Unlike context-free and context-sensitive grammars, RCGs have predicate names in addition to variables and terminals—this allows us to maintain connections between different parts

of the input string. Another formalism that captures polynomial time is the multi-head alternating automata [16], which are finite state machines with several cursors that can perform alternating transitions. Though related, these results do not seem to imply our results on document spanners.

We prove equivalence to polynomial time via a result by Papadimitriou [22], stating that semipositive Datalog (i.e., Datalog where only EDB relations can be negated) can express every database property computable in polynomial time, under certain assumptions: *(a)* the property is invariant under isomorphism, *(b)* a successor relation that defines a linear order over the domain is accessible as an EDB, and *(c)* the first and last elements in the database are accessible as constants (or single-element EDBs). We show that in the case of RGXlog, we get all of these for free, due to the fact that our EDBs are regex formulas. Specifically, in string logic (over a finite alphabet), isomorphism coincides with identity, negation of EDBs (regex formulas) are expressible as EDBs (regex formulas), and we can express a linear order by describing a successor relation along with its first and last elements.

Interestingly, our construction shows that, to express polynomial time, it suffices to use regex formulas with only two variables. In other words, binary regex formulas already capture the entire expressive power. Can we get away with only unary regex formulas? Using past results on monadic Datalog [14] and non-recursive RGXlog [7] we conclude a negative answer—Boolean RGXlog with unary regex formulas can express *precisely* the class of Boolean regular spanners. In fact, we can characterize explicitly the class of spanners expressible by RGXlog with unary regex formulas.

Lastly, we analyze recursive Datalog programs in a framework that generalizes both the relational and the spanner model. The framework, introduced by Nahshon, Peterfreund and Vansummeren [21] and referred to as Spannerlog⟨RGX⟩, aims to establish a unified query language for combining structured and textual data. In this framework, the input and output databases consist of relations that have two types of attributes: strings and spans. In the associated Datalog program, we refer to the relations of the input database as EDB relations (that is, *extensional*) and to those of the output database as IDB relations (that is, *intensional*, or *inferred*). The body of a Datalog rule may have three types of atoms: EDB, IDB, and regex formulas over string attributes. We prove that Spannerlog⟨RGX⟩ with stratified negation, restricted to string EDB relations, can express *precisely* the queries that are computable in polynomial time.

The remainder of the paper is organized as follows. We provide basic definitions and terminology in Section 2, and introduce RGXlog in Section 3. In Section 4, we illustrate RGXlog in the context of a comparison with (generalized) core spanners. Our main result (equivalence to polynomial time) is proved in Section 5. We describe the generalization of our main result to Spannerlog⟨RGX⟩ in Section 6, and conclude in Section 7.

2 Preliminaries

We first introduce the basic terminology and notation that we use throughout the paper.

2.1 Document Spanners

We begin with the basic terminology from the framework of document spanners [7].

Strings and spans. We fix a finite alphabet Σ of symbols. A *string* $\mathbf{s}$ is a finite sequence $\sigma_1 \cdots \sigma_n$ over Σ (i.e., each $\sigma_i \in \Sigma$). We denote by Σ^* the set of all strings over Σ. A *language* over Σ is a subset of Σ^*. A *span* identifies a substring of $\mathbf{s}$ by specifying its bounding indices.

```
C a i n ␣ s o n ␣ o f ␣ A d a m , ␣ A b e l ␣ s o n ␣ o f ␣ A d a m , ␣ E n o c h ␣ s o n ␣ o f ␣ C a i n , ␣
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
```

Figure 1 The input string $\mathbf{s}$ in our running example

Formally, a span of $\mathbf{s}$ has the form $[i, j\rangle$ where $1 \leq i \leq j \leq n+1$. If $[i, j\rangle$ is a span of $\mathbf{s}$, then $\mathbf{s}_{[i,j\rangle}$ denotes the substring $\sigma_i \cdots \sigma_{j-1}$. Note that $\mathbf{s}_{[i,i\rangle}$ is the empty string, and that $\mathbf{s}_{[1,n+1\rangle}$ is $\mathbf{s}$. Note also that the spans $[i, i\rangle$ and $[j, j\rangle$, where $i \neq j$, are different, even though $\mathbf{s}_{[i,i\rangle} = \mathbf{s}_{[j,j\rangle} = \epsilon$ where ϵ stands for the empty string. We denote by Spans the set of all spans of all strings, that is, all expressions $[i, j\rangle$ where $1 \leq i \leq j$. By $\mathsf{Spans}(\mathbf{s})$ we denote the set spans of string $\mathbf{s}$ (and in this case we have $j \leq n+1$).

▶ **Example 1.** In all of the examples throughout the paper, we use the alphabet Σ that consists of the lowercase and capital letters from the English alphabet (i.e., $\mathtt{a}, \ldots, \mathtt{z}$ and $\mathtt{A}, \ldots, \mathtt{Z}$), the comma symbol ",", and the symbol "␣" that stands for whitespace. Figure 1 depicts an example of a prefix of a string $\mathbf{s}$. (For convenience, it also depicts the position of each of the characters in $\mathbf{s}$.) Observe that the spans $[13, 17\rangle$ and $[31, 35\rangle$ are different, yet they span the same substring, that is, $\mathbf{s}_{[13,17\rangle} = \mathbf{s}_{[31,35\rangle} = \mathtt{Adam}$. ◀

Document spanners. We assume an infinite collection Vars of *variables* such that Vars and Σ are disjoint. Let $\mathbf{s}$ be a string and $V \subset \mathsf{Vars}$ a finite set of variables. A $(V, \mathbf{s})$-*record*[2] is a function $r : V \to \mathsf{Spans}(\mathbf{s})$ that maps the variables of V to spans of $\mathbf{s}$. A $(V, \mathbf{s})$-*relation* is a set of $(V, \mathbf{s})$-records. A *document spanner* (or just *spanner* for short) is a function P that maps strings $\mathbf{s}$ to $(V, \mathbf{s})$-relations $P(\mathbf{s})$, for a predefined finite set V of variables that we denote by $\mathsf{Vars}(P)$. As a special case, a *Boolean spanner* is a spanner P such that $\mathsf{Vars}(P) = \emptyset$; in this case, $P(\mathbf{s})$ can be either the singleton that consists of the empty function, a situation denoted by $P(\mathbf{s}) = \mathbf{true}$, or the empty set, a situation denoted by $P(\mathbf{s}) = \mathbf{false}$. A Boolean spanner P *recognizes* the language $\{\mathbf{s} \in \Sigma^* \mid P(\mathbf{s}) = \mathbf{true}\}$.

By a *spanner representation language*, or simply *spanner language* for short, we refer to a collection L of finite expressions p that represent a spanner. For instance, we next define the spanner language RGX of regex formulas. For an expression p in a spanner language, we denote by $[\![p]\!]$ the spanner that is defined by p, and by $\mathsf{Vars}(p)$ the variable set $\mathsf{Vars}([\![p]\!])$. Hence, for a string $\mathbf{s}$ we have that $[\![p]\!](\mathbf{s})$ is a $(\mathsf{Vars}(p), \mathbf{s})$-relation. We denote by $[\![L]\!]$ the class of all spanners $[\![p]\!]$ definable by expressions p in L.

Regex formulas. A *regex formula* is a representation of a spanner by means of a regular expression with *capture variables*. It is defined by $\gamma := \emptyset \mid \epsilon \mid \sigma \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^* \mid x\{\gamma\}$. Here, ϵ stands for the empty string, $\sigma \in \Sigma$, and the alternative beyond regular expressions is $x\{\gamma\}$ where x is a variable in Vars. We denote the set of variables that occur in γ by $\mathsf{Vars}(\gamma)$. Intuitively, every *match* of a regex formula in an input string $\mathbf{s}$ yields an assignment of spans to the variables of γ. A crucial assumption we make is that the regex formula is *functional* [7], which intuitively means that every match assigns precisely one span to each variable in $\mathsf{Vars}(\gamma)$. For example, the regex formula $\mathtt{a}^* \cdot x\{\mathtt{a} \cdot \mathtt{b}^*\} \cdot \mathtt{a}$ is functional, but $\mathtt{a}^* \cdot (x\{\mathtt{a} \cdot \mathtt{b}\})^* \cdot \mathtt{a}$ is not; similarly, $(x\{\mathtt{a}\}) \vee (\mathtt{b} \cdot x\{\mathtt{a}\})$ is functional, but $(x\{\mathtt{a}\}) \vee (\mathtt{b} \cdot \mathtt{a})$ is not. A regex formula γ defines a spanner, where the matches produce the $(V, \mathbf{s})$-records for $V = \mathsf{Vars}(\gamma)$. We refer the reader to Fagin et al. [7] for the precise definition of functionality, including its polynomial-time verification, and for the precise definition of the spanner $[\![\gamma]\!]$

[2] Fagin et al. [7] refer to $(V, \mathbf{s})$-records as $(V, \mathbf{s})$-*tuples*; we use "record" to avoid confusion with the concept of "tuple" that we later use in ordinary relations.

represented by γ. As previously said, we denote by RGX the spanner language of (i.e., the set of all) regex formulas.

Throughout the paper, we use the following abbreviations when we define regex formulas. We use "." instead of "$\vee_{\sigma\in\Sigma}\sigma$" (e.g., we use ".*" instead of "$(\vee_{\sigma\in\Sigma}\sigma)^*$"). For convenience, we put regex formulas in brackets and write $\langle\gamma\rangle$ (using angular instead of ordinary brackets) to denote that γ can occur anywhere in the document; that is, $\langle\gamma\rangle := [.^*\,\gamma\,.^*]$.

▶ **Example 2.** Following are examples of regex formulas that we use later on.

- $\gamma_{\mathsf{token}}(x) := \langle\ \sqcup\ x\{(\mathtt{a}-\mathtt{zA}-\mathtt{Z})^*\}\ (\sqcup \vee\ ,)\ \rangle$
- $\gamma_{\mathsf{cap}}(x) := \langle\ \sqcup\ x\{(\mathtt{A}-\mathtt{Z})(\mathtt{a}-\mathtt{zA}-\mathtt{Z})^*\}\ (\sqcup \vee\ ,)\ \rangle$
- $\gamma_{\mathsf{prnt}}(x,y) := \langle y\{.^*\}\sqcup\mathtt{son}\sqcup\mathtt{of}\sqcup x\{.^*\}\rangle$

The regex formula $\gamma_{\mathsf{token}}(x)$ extracts the spans of tokens (defined simplistically for presentation sake), $\gamma_{\mathsf{cap}}(x)$ extracts capitalized tokens, and $\gamma_{\mathsf{prnt}}(x,y)$ extracts spans separated by $\sqcup\mathtt{son}\sqcup\mathtt{of}\sqcup$ (where prnt stands for "parent"). For illustration, applying $[\![\gamma_{\mathsf{cap}}]\!]$ to $\mathbf{s}$ of Figure 1 results in a set of $(\{x\},\mathbf{s})$-records that includes the record r that maps x to $[19,23\rangle$. ◀

2.2 Spanner Algebra

The algebraic operators *union*, *projection*, *natural join*, and *difference* are defined in the usual way, for all spanners P_1 and P_2 and strings $\mathbf{s}$, as follows. For a $(V,\mathbf{s})$-record r and $Y\subseteq V$, we denote by $r\restriction Y$ the $(Y,\mathbf{s})$-record obtained by restricting r to the variables in Y. We say that P_1 and P_2 are *union compatible* if $\mathsf{Vars}(P_1)=\mathsf{Vars}(P_2)$.

- **Union:** Assuming P_1 and P_2 are union compatible, the union $P=P_1\cup P_2$ is defined by $\mathsf{Vars}(P):=\mathsf{Vars}(P_1)$ with $P(\mathbf{s}):=P_1(\mathbf{s})\cup P_2(\mathbf{s})$.
- **Projection:** For $Y\subseteq\mathsf{Vars}(P_1)$, the projection $P=\pi_Y P_1$ is defined by $\mathsf{Vars}(P):=Y$ with $P(\mathbf{s})=\{r\restriction Y \mid r\in P_1(\mathbf{s})\}$.
- **Natural join:** Let $V_i:=\mathsf{Vars}(P_i)$ for $i\in\{1,2\}$. The *(natural) join* $P=(P_1\bowtie P_2)$ is defined by $\mathsf{Vars}(P):=\mathsf{Vars}(P_1)\cup\mathsf{Vars}(P_2)$ with $P(\mathbf{s})$ consisting of all $(V_1\cup V_2,\mathbf{s})$-records r such that there exist $r_1\in P_1(\mathbf{s})$ and $r_2\in P_2(\mathbf{s})$ with $r\restriction V_1=r_1$ and $r\restriction V_2=r_2$.
- **Difference:** Assuming P_1 and P_2 are union compatible, the difference $P=P_1\setminus P_2$ is defined by $\mathsf{Vars}(P_1\setminus P_2):=\mathsf{Vars}(P_1)$ with $P(\mathbf{s}):=P_1(\mathbf{s})\setminus P_2(\mathbf{s})$.
- **String-equality selection:** For variables x and y in $\mathsf{Vars}(P_1)$, the string-equality selection $P:=\zeta^{=}_{x,y}P_1$ is defined by $\mathsf{Vars}(P):=\mathsf{Vars}(P_1)$ with $P(\mathbf{s})$ consisting of all records $r\in P_1(\mathbf{s})$ such that $\mathbf{s}_{r(x)}=\mathbf{s}_{r(y)}$.

If L is a spanner language and O is a set of operators in a spanner algebra, then L^O denotes the spanner language obtained by closing L under the operations of O.

2.3 Regular and (Generalized) Core Spanners

Following Fagin et al. [7], we define a *regular* spanner to be one definable in $\mathsf{RGX}^{\{\cup,\pi,\bowtie\}}$, that is, a spanner P such that $P=[\![p]\!]$ for some p in $\mathsf{RGX}^{\{\cup,\pi,\bowtie\}}$. Similarly, we define a *core* spanner to be a spanner definable in $\mathsf{RGX}^{\{\cup,\pi,\bowtie,\zeta^{=}\}}$.

▶ **Example 3.** Consider the regex formulas of Example 2. We can take their join and obtain a regular spanner: $\gamma_{\mathsf{prnt}}(x,y)\bowtie\gamma_{\mathsf{cap}}(x)\bowtie\gamma_{\mathsf{cap}}(y)$. This spanner extracts a set of $(\{x,y\},\mathbf{s})$-records r such that r maps x and y to strings that begin with a capital letter and are separated by $\sqcup\mathtt{son}\sqcup\mathtt{of}\sqcup$. Assume we wish to extract a binary relation that holds the tuples (x,y) such that the span x spans the name of the grandparent of y. (For simplicity, we assume that the name is a unique identifier of a person.) For that, we can define the following

core spanner on top of the regex formulas from Example 2: $\pi_{x,w}\zeta^{=}_{y,z}\big(\gamma_{\mathsf{prnt}}(x,y) \bowtie \gamma_{\mathsf{prnt}}(z,w)\big)$. We denote this spanner by $\gamma_{\mathsf{grpr}}(x,w)$. ◀

Note that we did not include *difference* in the definition of regular and core spanners; this does not matter for the class of *regular* spanners, since it is closed to difference (i.e., a spanner is definable in $\mathsf{RGX}^{\{\cup,\pi,\bowtie,\setminus\}}$ if and only if it is definable in $\mathsf{RGX}^{\{\cup,\pi,\bowtie\}}$), but it matters for the class of *core* spanners, which is *not* closed under difference [7]. We define a *generalized core spanner* to be a spanner definable in $\mathsf{RGX}^{\{\cup,\pi,\bowtie,\zeta^{=},\setminus\}}$. We study the expressive power of the class of generalized core spanners in Section 4.

▶ **Example 4.** Recall the definition of $\gamma_{\mathsf{grpr}}(x,w)$ from Example 3. The generalized core spanner $\gamma_{\mathsf{cap}}(w) \setminus (\pi_w\gamma_{\mathsf{grpr}}(x,w))$ finds all spans of capitalized words w such that the text has no mentioning of any grandparent of w. ◀

2.4 Span Databases

We also use the terminology and notation of ordinary relational databases, with the exception that database values are all spans. (In Section 6 we allow more general values in the database.) More formally, a *relation symbol* R has an associated arity that we denote by $\mathsf{arity}(R)$, and a *span relation* over R is a finite set of *tuples* $\mathbf{t} \in \mathsf{Spans}^{\mathsf{arity}(R)}$ over R. We denote the ith element of a tuple $\mathbf{t}$ by $\mathbf{t}_i$. A *(relational) signature* $\mathcal{R}$ is a finite set $\{R_1,\ldots,R_n\}$ of relation symbols. A *span database* D over a signature $\mathcal{R} := \{R_1,\ldots,R_n\}$ consists of span relations R_i^D over the R_i. We call R_i^D the *instantiation* of R_i by D.

3 RGXlog: Datalog over Regex Formulas

In this section, we define the spanner language RGXlog, pronounced "regex-log," that generalizes regex formulas to (possibly recursive) Datalog programs.

Let $\mathcal{R}$ be a signature. By an *atom* over $\mathcal{R}$ we refer to an expresion of the form $R(x_1,\ldots,x_k)$ where $R \in \mathcal{R}$ is a k-ary relation symbol and each x_i is a variable in Vars. Note that a variable can occur more than once in an atom (i.e., we may have $x_i = x_j$ for some i and j with $i \neq j$), and we do not allow constants in atoms. A RGXlog *program* is a triple $\langle \mathcal{I}, \Phi, \textsc{Out}(\mathbf{x})\rangle$ where:

- $\mathcal{I}$ is a signature referred to as the *IDB* signature;
- Φ is a finite set of *rules* of the form $\varphi \leftarrow \psi_1,\ldots,\psi_m$, where φ is an atom over $\mathcal{I}$, and each ψ_i is either an atom over $\mathcal{I}$ or a regex formula;
- $\textsc{Out} \in \mathcal{I}$ is a designated *output* relation symbol;
- $\mathbf{x}$ is a sequence of k distinct variables in Vars, where k is the arity of $\textsc{Out}$.

If ρ is the rule $\varphi \leftarrow \psi_1,\ldots,\psi_m$, then we call φ the *head* of ρ and $\psi_1,\ldots,\psi_m$ the *body* of ρ. Each variable in φ is called a *head variable* of ρ. We make the standard assumption that each head variable of a rule occurs at least once in the body of the rule.

We now define the semantics of evaluating a RGXlog program over a string. Let $Q = \langle \mathcal{I}, \Phi, \textsc{Out}(\mathbf{x})\rangle$ be a RGXlog program, and let $\mathbf{s}$ be a string. We evaluate Q on $\mathbf{s}$ using the usual fixpoint semantics of Datalog, while viewing the regex formulas as extensional-database (EDB) relations. More formally, we view a regex formula γ as a logical assertion over assignments to $\mathsf{Vars}(\gamma)$, stating that the assignment forms a tuple in $[\![\gamma]\!](\mathbf{s})$. The span database with signature $\mathcal{I}$ that results from applying Q to $\mathbf{s}$ is denoted by $Q(\mathbf{s})$, and it is the minimal span database that satisfies all rules, when viewing each left arrow ($\leftarrow$) as a logical implication with all variables being universally quantified.

Next, we define the semantics of RGXlog as a spanner language. Let $Q = \langle \mathcal{I}, \Phi, \textsc{Out}(\mathbf{x})\rangle$ be a RGXlog program. As a spanner, the program Q constructs $D = Q(\mathbf{s})$ and emits the

relation $\textsc{Out}^D$ as assignments to $\mathbf{x}$. More precisely, suppose that $\mathbf{x} = x_1, \ldots, x_k$. The spanner $P = [\![Q]\!]$ is defined as follows.

- $\mathsf{Vars}(P) := \{x_1, \ldots, x_k\}$.
- Given $\mathbf{s}$ and $D = Q(\mathbf{s})$, the set $P(\mathbf{s})$ consists of all records $r_{\mathbf{a}}$ obtained from tuples $\mathbf{a} = (a_1, \ldots, a_k) \in \textsc{Out}^D$ by setting $r_{\mathbf{a}}(x_i) = a_i$.

Finally, *recursive* and *non-recursive* RGXlog programs are defined similarly to ordinary Datalog (e.g., using the acyclicity of the dependency graph over the IDB predicates).

▶ **Example 5.** In the following and later examples of programs, we use the cursor sign ▶ to indicate where a rule begins. Importantly, for brevity we use the following convention: $\textsc{Out}(\mathbf{x})$ is always the left hand side of the last rule.

- ▶ $\textsc{Ancstr}(x, z) \leftarrow \gamma_{\mathsf{prnt}}(x, z)$
- ▶ $\textsc{Ancstr}(x, y) \leftarrow \textsc{Ancstr}(x, z), \gamma_{\mathsf{prnt}}(z, y)$

By our convention, $\textsc{Out}(\mathbf{x})$ is $\textsc{Ancstr}(x, y)$. This program returns the transitive closure of the relation obtained by applying the regex formula $\gamma_{\mathsf{prnt}}(x, z)$ from Example 2. ◀

4 Comparison to Core Spanners

We begin the exploration of the expressive power of RGXlog by a comparison to the class of core spanners and the class of generalized core spanners. We first recall the following observation by Fagin et al. [8] for later reference.

▶ **Proposition 6.** [8] *The class of spanners definable by non-recursive* RGXlog *is precisely the class of regular spanners, namely* $[\![\mathsf{RGX}^{\{\cup,\pi,\bowtie\}}]\!]$.

In addition to RGXlog being able to express union, projection and natural join, the following program shows that RGXlog can express the string-equality selection, namely $\zeta^=$.

- ▶ $\textsc{StrEq}(x, y) \leftarrow \langle x\{\epsilon\}\rangle, \langle y\{\epsilon\}\rangle$
- ▶ $\textsc{StrEq}(x, y) \leftarrow \langle x\{\sigma\tilde{x}\{.^*\}\}\rangle, \langle y\{\sigma\tilde{y}\{.^*\}\}\rangle, \textsc{StrEq}(\tilde{x}, \tilde{y})$

Here, the second rule is repeated for every alphabet letter σ. (Note that we are using the assumption that the alphabet is finite.) It thus follows that every core spanner is definable in RGXlog. The other direction is false. As an example, no core spanner extracts all spans x and y such that $\mathbf{s}_x$ is *not* a substring of $\mathbf{s}_y$ [7], or all pairs x and y of spans having the same *length* [8]. In the following example, we construct a RGXlog program that extracts both of these relationships.

▶ **Example 7.** In the following program, rules that involve σ and τ are repeated for all letters σ and τ such that $\sigma \neq \tau$, and the ones that involve only σ are repeated for every σ.

- ▶ $\textsc{Len}_=(x, y) \leftarrow \langle x\{\epsilon\}\rangle, \langle y\{\epsilon\}\rangle$
- ▶ $\textsc{Len}_=(x, y) \leftarrow \langle x\{.\tilde{x}\{.^*\}\}\rangle, \langle y\{.\tilde{y}\{.^*\}\}\rangle, \textsc{Len}_=(\tilde{x}, \tilde{y})$
- ▶ $\textsc{Len}_>(x, y) \leftarrow \langle x\{.^+\tilde{y}\{.^*\}\}\rangle, \textsc{Len}_=(\tilde{y}, y)$

- ▶ $\textsc{NotPrfx}(x, y) \leftarrow \langle x\{\sigma.^*\}\rangle, \langle y\{\epsilon\} \vee y\{\tau.^*\}\rangle$
- ▶ $\textsc{NotPrfx}(x, y) \leftarrow \langle x\{\sigma\tilde{x}\{.^*\}\}\rangle, \langle y\{\sigma\tilde{y}\{.^*\}\}\rangle, \textsc{NotPrfx}(\tilde{x}, \tilde{y})$

- ▶ $\textsc{NotCntd}(x, y) \leftarrow \textsc{Len}_>(x, y)$
- ▶ $\textsc{NotCntd}(x, y) \leftarrow \textsc{NotPrfx}(x, y), \langle y\{.\tilde{y}\{.^*\}\}\rangle, \textsc{NotCntd}(x, \tilde{y})$

The program defines the following relations:

- $\textsc{Len}_{=}(x, y)$ contains all spans x and y of the same length.
- $\textsc{Len}_{>}(x, y)$ contains all spans x and y such that x is longer than y.
- $\textsc{NotPrfx}(x, y)$ contains all spans x and y such that $\mathbf{s}_x$ is *not* a prefix of $\mathbf{s}_y$. The rules state that $\mathbf{s}_x$ is not a prefix of $\mathbf{s}_y$ if $\mathbf{s}_x$ is nonempty but $\mathbf{s}_y$ is empty, or the two begin with different letters, or the two begin with the same letter but the rest of $\mathbf{s}_x$ is not a prefix of the rest of $\mathbf{s}_y$.
- $\textsc{NotCntd}(x, y)$ contains all spans x and y such that $\mathbf{s}_x$ is *not* contained in $\mathbf{s}_y$. The rules state that this is the case if x is longer than y, or both of the following hold: $\mathbf{s}_x$ is not a prefix of $\mathbf{s}_y$, and $\mathbf{s}_x$ is not contained in the suffix of $\mathbf{s}_y$ following the first symbol.

In particular, the program defines both equal-length and non-containment relationships. ◀

The impossibility proofs of Fagin et al. [7,8] are based on the *core simplification lemma* [7], which states that every core spanner can be represented as a regular spanner, followed by a sequence of string-equality selections ($\zeta^{=}$) and projections (π). In turn, the proof of this lemma relies on the absence of the difference operator in the algebra. See Freydenberger and Holldack [10] for an indication of why a result similar to the core simplification lemma is not likely to hold in the presence of difference. Do things change when we consider *generalized core spanners*, where difference is allowed? To be precise, we are interested in two questions:

1. Can RGXlog express every generalized core spanner?
2. Is it true that every spanner definable in RGXlog is a generalized core spanner?

In the next section, we show that the answer to the first question is yes. In the remainder of this section, we show that the answer to the second question is no.

We begin by constructing the following RGXlog program, which defines a Boolean is spanner that returns **true** if and only if the length of the input $\mathbf{s}$ is a power of two.

- ▶ $\textsc{Pow2}(x) \leftarrow \langle x\{.\}\rangle$
- ▶ $\textsc{Pow2}(x) \leftarrow \langle x\{x_1\{.^*\}x_2\{.^*\}\}\rangle, \textsc{Pow2}(x_1), \textsc{Len}_{=}(x_1, x_2)$
- ▶ $\textsc{Out}() \leftarrow [x\{.^*\}], \textsc{Pow2}(x)$

We prove the following.

▶ **Theorem 8.** *There is no Boolean generalized core spanner that determines whether the length of the input string is a power of two.*

Hence, we get a negative answer to the second question. In the remainder of this section, we discuss the proof of Theorem 8. We need to prove that no generalized core spanner recognizes precisely all strings whose length is a power of two.

Let a be a letter, and L_{a} the language of all strings $\mathbf{s}$ that consist of 2^n occurrences of a for $n \geq 0$, that is: $L_{\mathsf{a}} \stackrel{\text{def}}{=} \{\mathbf{s} \in \mathsf{a}^* \mid |\mathbf{s}| \text{ is a power of } 2\}$. We will restrict our discussion to generalized core spanners that accept only strings in a^*, and show that no such spanner recognizes L_{a}. This is enough, since every generalized core spanner S can be restricted into a^* by joining S with the regex formula $[\mathsf{a}^*]$. For simplicity, we will further assume that our alphabet consists of only the symbol a. Then, a language L is identified by a set of natural numbers—the set of all numbers m such that $\mathsf{a}^m \in L$. We denote this set by $\mathbb{N}(L)$.

Presburger Arithmetic (PA) is the first-order theory of the natural numbers with the addition ($+$) binary function and the constants 0 and 1 [23]. For example, the relationship $x > y$ is expressible by the PA formula $\exists z[x = y + z + 1]$ and by the PA formula $x \neq y \wedge \exists z[x = y + z]$. As another example, the set of all even numbers x is definable by the PA formula $\exists y[x = y + y]$. When we say that a set A of natural numbers is *definable in PA* we mean that there is a unary PA formula $\varphi(x)$ such that $A = \{x \in \mathbb{N} \mid \varphi(x)\}$.

It is known that being a power of two is *not* definable in PA [17]. Theorem 8 then follows from the next theorem.

▶ **Theorem 9.** *A language $L \subseteq \{\mathsf{a}\}^*$ is recognizable by a Boolean generalized core spanner if and only if $\mathbb{N}(L)$ is definable in PA.*

5 Equivalence to Polynomial Time

While RGXlog programs output relations (which are sets of tuples), the result of evaluating a spanner on $\mathbf{s}$ is given as a set of $(V, \mathbf{s})$-records. Therefore, to compare the expressiveness of RGXlog programs and spanners, in what follows we implicitly treat tuples as records and vice-versa as described now. We assume that there is a fixed predefined order on Vars (e.g., the lexicographic order on the variables' names) and denote the i'th element in this order by v_i. A tuple $\mathbf{t} \in \mathsf{Spans}^n$ is viewed as the record whose domain is $\{v_1, \ldots, v_n\}$ that maps each v_i to $\mathbf{t}_i$; a $(V, \mathbf{s})$-record r is viewed as the tuple whose i'th element equals the value of r on v where v is the i'th variable of V according to the fixed predefined order on Vars.

An easy consequence of existing literature [1, 11] is that every RGXlog program can be evaluated in polynomial time (as usual, under data complexity). Indeed, the evaluation of a RGXlog program P can be done in two steps: *(1)* materialize the regex atoms on the input string $\mathbf{s}$ and get relations over spans, *and (2)* evaluate P as an ordinary Datalog program over an ordinary relational database, treating the regex formulas as the names of the corresponding materialized relations. The first step can be completed in polynomial time [11], and so can the second [1]. Quite remarkably, RGXlog programs capture *precisely* the spanners computable in polynomial time.

▶ **Theorem 10.** *A spanner is definable in* RGXlog *if and only if it is computable in polynomial time.*

In the remainder of this section, we discuss the proof of Theorem 10. The proof of the "only if" direction is described right before the theorem. To prove the "if" direction, we need some definitions and notation.

Definitions. We apply ordinary Datalog programs to databases over arbitrary domains, in contrast to RGXlog programs that we apply to strings, and that involve databases over the domain of spans. Formally, we define a Datalog program as a quadruple $(\mathcal{E}, \mathcal{I}, \Phi, \textsc{Out})$ where $\mathcal{E}$ and $\mathcal{I}$ are disjoint signatures referred to as the *EDB* (input) and *IDB* signatures, respectively, Out is a designated output relation symbol in $\mathcal{I}$, and Φ is a finite set of Datalog rules.[3] As usual, a *Datalog rule* has the form $\varphi \leftarrow \psi_1, \ldots, \psi_m$, where φ is an atomic formula over $\mathcal{I}$ and $\psi_1, \ldots, \psi_m$ are atomic formulas over $\mathcal{E}$ and $\mathcal{I}$. We again require each variable in the head φ to occur in the body $\psi_1, \ldots, \psi_m$. In this paper, we restrict Datalog programs to ones *without constants*; that is, an atomic formula ψ_i is of the form $R(x_1, \ldots, x_k)$ where R is a k-ary relation symbol and the x_i are (not necessarily distinct) variables. An input for a Datalog program Q is an instance D over $\mathcal{E}$ that instantiates every relation symbol of $\mathcal{E}$ with values from an arbitrary domain. The *active domain* of an instance D, denoted $adom(D)$, is the set of constants that occur in D.

An *ordered signature* $\mathcal{E}$ is a signature that includes three distinguished relation symbols: a binary relation symbol Succ, and two unary relation symbols First and Last. An *ordered instance* D is an instance over an ordered signature $\mathcal{E}$ such that Succ is interpreted as a

[3] Note that unlike RGXlog, here there is no need to specify variables for Out. This is because a spanner evaluates to assignments of spans to variables, which we need to relate to Out, whereas a Datalog program evaluates to an entire relation, which is Out itself.

successor relation of some linear (total) order over $adom(D)$, and First and Last determine the first and last elements in this linear order, respectively.

A *semipositive* Datalog program P, or Datalog$^{\perp}$ program in notation, is a Datalog program in which the EDB atoms (i.e., atoms over EDB relation symbols) can be negated. We make the safety assumption that in each rule ρ, every variable that appears in the head of ρ is either (1) a variable appearing in a positive (i.e., non-negated) atom of the body of the rule, or (2) in $\mathsf{Vars}(\gamma)$ for a regex formula γ that appears in the body of the rule. For an instance D over $\mathcal{E}$, we denote by $P(D)$ the database with the signature $\mathcal{I}$ that results from applying P on D.

A *query* Q over a signature $\mathcal{E}$ is associated with a fixed arity $\mathsf{arity}(Q) = k$, and it maps an input database D over $\mathcal{E}$ into a relation $Q(D) \subseteq (adom(D))^k$. As usual, Q is *Boolean* if $k = 0$. We say that Q *respects isomorphism* if for all isomorphic databases D_1 and D_2 over $\mathcal{E}$, and for all isomorphisms $\varphi : adom(D_1) \to adom(D_2)$ between D_1 and D_2, it is the case that $\varphi(Q(D_1)) = Q(D_2)$.

Proof idea for Theorem 10. We now discuss the proof of the "if" direction. The proof is based on Papadimitriou's theorem [22], stating a close connection between semipositive Datalog and polynomial time:

▶ **Theorem 11.** [6, 22] *Let $\mathcal{E}$ be an ordered signature and let Q be a query over $\mathcal{E}$ such that Q respects isomorphism. Then Q is computable in polynomial time if and only if Q is computable by a Datalog$^{\perp}$ program.*

Our proof continues as follows. Let S be a spanner that is computable in polynomial time. We translate S into a RGXlog program P in two main steps. In the first step, we translate S into a Datalog$^{\perp}$ program P_S by an application of Theorem 11. In the second step, we translate P_S into P. To realize the first step of the construction, we need to encode our input string by a database, since P_S operates over databases (and not over strings). To use Theorem 11, we need to make sure that this encoding is computable in polynomial time, and that it is invariant under isomorphism, that is, the encoding allows to restore the string even if replaced by an isomorphic database. To realize the second step of the construction, we need to bridge several differences between RGXlog and Datalog$^{\perp}$. First, the former takes as input a string, and the latter a database. Second, the latter assumes an ordered signature while the former does not involve any order. Third, the former does not allow negation while in the latter EDB atoms can be negated.

For the first step of our translation, we use a standard representation (which we shall explain shortly) of a string as a logical structure and extend it with a total order on its active domain. Note that we have to make sure that the active domain contains the output domain (i.e., all spans of the input string). We define $\mathcal{R}^{\mathsf{ord}}$ to be an ordered signature with the unary relation symbols R_σ for each $\sigma \in \Sigma$, in addition to the required Succ, First and Last. Let $\mathbf{s} = \sigma_1 \cdots \sigma_n$ be an input string. We define an instance $D_{\mathbf{s}}$ over $\mathcal{R}^{\mathsf{ord}}$ by materializing the relations as follows.

- Each relation R_σ consists of all tuples $([i, i+1\rangle)$ such that $\sigma_i = \sigma$.
- Succ consists of the pairs $([i, i'\rangle, [i, i'+1\rangle)$ and all pairs $([i, n+1\rangle, [i+1, i+1\rangle)$ whenever the involved spans are legal spans of $\mathbf{s}$.
- First and Last consist of $[1, 1\rangle$, and $[n+1, n+1\rangle$, respectively.

▶ **Comment 12.** Observe that we view the linear order as the lexicographic order over the spans. The only difference from the usual lexicographic order on ordered pairs (i, j) in that for spans, we must have $i \leq j$. The successor relation Succ is inferred from this order. ◀

An *encoding instance* (or just *encoding*) D is an instance over $\mathcal{R}^{\mathsf{ord}}$ that is isomorphic to $D_{\mathbf{s}}$ for some string $\mathbf{s}$. In this case, we say that D *encodes* $\mathbf{s}$. Note that the entries of an encoding are not necessarily spans. Nevertheless, every encoding encodes a unique string. The following lemma is straightforward.

▶ **Lemma 13.** *Let D be an instance over $\mathcal{R}^{\mathsf{ord}}$. The following hold:*
1. *Whether D is an encoding can be determined in polynomial time.*
2. *If D is an encoding, then there are unique string $\mathbf{s}$ and isomorphism ι such that D encodes $\mathbf{s}$ and $\iota(D_{\mathbf{s}}) = D$; moreover, both $\mathbf{s}$ and ι are computable in polynomial time.*

Let S be a spanner. We define a query Q_S over $\mathcal{R}^{\mathsf{ord}}$ as follows. If the input database D is an encoding and $\mathbf{s}$ and ι are as in Lemma 13, then $Q_S(D) = \iota([\![S]\!](\mathbf{s}))$; otherwise, $Q_S(D)$ is empty. To apply Theorem 11, we make an observation.

▶ **Observation 14.** *The query Q_S respects isomorphism, and moreover, is computable in polynomial time whenever S is computable in polynomial time.*

We can now apply Theorem 11 on Q_S:

▶ **Lemma 15.** *If S is computable in polynomial time, then there exists a Datalog$^\perp$ program P' over $\mathcal{R}^{\mathsf{ord}}$ such that $P'(D) = Q_S(D)$ for every instance D over $\mathcal{R}^{\mathsf{ord}}$.*

The second step of the translation simulates the Datalog$^\perp$ program P' using a RGXlog program. With RGXlog, we can construct $D_{\mathbf{s}}$ from $\mathbf{s}$ with the following rules:

- $R_\sigma(x) \leftarrow \langle x\{\sigma\}\rangle$
- $\textsc{Succ}(x_1, x_2) \leftarrow \langle x_2\{x_1\{.^*\}\ .\}\rangle \vee [.^* x_2\{\ .\ x_1\{\epsilon\}\ .^*\}]$
- $\textsc{First}(x) \leftarrow [x\{\epsilon\}.^*]$
- $\textsc{Last}(x) \leftarrow [.^* x\{\epsilon\}]$

Indeed, if we evaluate the above RGXlog rules on a string $\mathbf{s}$, we get exactly $D_{\mathbf{s}}$. Note that rules in Datalog$^\perp$ that do not involve negation can be viewed as RGXlog rules. However, since RGXlog do not allow negation, we need to include the negated EDBs as additional EDBs. Nevertheless, we can negate EDBs without explicit negation, because regular spanners are closed under difference and complement [7]. We therefore conclude the following lemma.

▶ **Lemma 16.** *If P' is a Datalog$^\perp$ program over $\mathcal{R}^{\mathsf{ord}}$, then there exists a* RGXlog *program P such that $P(\mathbf{s}) = P'(D_{\mathbf{s}})$ for every string $\mathbf{s}$.*

To summarize the proof of the "if" direction of Theorem 10, let S be a spanner computable in polynomial time. We defined Q_S to be such that $Q_S(D_{\mathbf{s}}) = [\![S]\!](\mathbf{s})$ for all $\mathbf{s}$. Lemma 15 implies that there exists a Datalog$^\perp$ program P' such that $P'(D_{\mathbf{s}}) = Q_S(D_{\mathbf{s}})$ for all $\mathbf{s}$. By Lemma 16, there exists a RGXlog program P such that $P(\mathbf{s}) = P'(D_{\mathbf{s}})$ for all $\mathbf{s}$. Therefore, P is the required RGXlog program such that $P(\mathbf{s}) = S(\mathbf{s})$ for all $\mathbf{s}$.

5.1 RGXlog over Monadic Regex Formulas

Our proof of Theorem 10 showed that RGXlog programs over *binary* regex formulas (i.e., regex formulas with two variables) suffice to capture every spanner that is computable in polynomial time. Next, we show that if we allow only *monadic* regex formulas (i.e., regex formulas with one variable), then we strictly decrease the expressiveness. We call such programs *regex-monadic* programs. We can characterize the class of spanners expressible by regex-monadic programs, as follows.

▶ **Theorem 17.** *Let S be a spanner. The following are equivalent:*
1. *S is definable as a regex-monadic program.*

2. *S is definable as a* RGXlog *program where all the rules have the form*

$$\textsc{Out}(x_1, \ldots, x_k) \leftarrow \gamma_1(x_1), \ldots, \gamma_k(x_k), \gamma()$$

where each $\gamma_i(x_i)$ is a unary regex formula and γ is a Boolean regex formula.

Note that in the second part of Theorem 17, the Boolean $\gamma()$ can be omitted whenever $k > 0$, since $\gamma()$ can be compiled into $\gamma_k(x_k)$. To prove the theorem, we use a result by Levy et al. [18], stating that recursion does not add expressive power when every relation in the EDB is unary. This theorem implies that every spanner definable as a regex-monadic program is regular. We then draw the following direct consequence on Boolean programs.

▶ **Corollary 18.** *A language is accepted by a Boolean regex-monadic program if and only if it is regular.*

For non-Boolean spanners, we can use Theorem 17 to show that regex-monadic programs are *strictly less expressive* than regular spanners. For instance, we can show that the relation "the span x contains the span y" is not expressible as a regex-monadic program, although it is clearly regular. Therefore, we conclude the following.

▶ **Corollary 19.** *The class of regex-monadic programs is strictly less expressive than the class of regular spanners.*

6 Extension to a Combined Relational/Textual Model

In this section, we extend our main Theorem (Theorem 10) to *Spannerlog*—a data and query model introduced by Nahshon et al. [21] that unifies and generalizes relational databases and spanners by considering relations over both strings and spans.

6.1 Spannerlog

The fragment of Spannerlog that we consider is referred to by Nahshon et al. [21] as Spannerlog⟨RGX⟩, and we abbreviate it as simply Spl⟨RGX⟩. A *mixed signature* is a collection of *mixed relation symbols* R that have two types of attributes: *string* attributes and *span* attributes. We denote by $[R]_{\mathsf{str}}$ and $[R]_{\mathsf{spn}}$ the sets of string attributes and span attributes of R, respectively, where an attribute is represented by its corresponding index. Hence, $[R]_{\mathsf{str}}$ and $[R]_{\mathsf{spn}}$ are disjoint and $[R]_{\mathsf{str}} \cup [R]_{\mathsf{spn}} = \{1, \ldots, \mathsf{arity}(R)\}$. A *mixed relation* over R is a set of tuples $(a_1, \ldots, a_m)$ where m is the arity of R and each a_ℓ is a string in Σ^* if $\ell \in [R]_{\mathsf{str}}$ and a span $[i, j\rangle$ if $\ell \in [R]_{\mathsf{spn}}$. A *mixed instance* D over a mixed signature consists of a mixed relation R^D for each mixed relation symbol R. A *query* Q over a mixed signature $\mathcal{E}$ is associated with a mixed relation symbol R_Q, and it maps every mixed instance D over $\mathcal{E}$ into a mixed relation $Q(D)$ over R_Q.

A mixed signature whose attributes are all string attributes (in all of the mixed relation symbols) is called a *span-free signature*. A mixed relation over a relation symbol whose attributes are all string (respectively, span) attributes is called a *string relation* (respectively, *span relation*). To emphasize the difference between mixed signatures (respectively, mixed relation symbols, mixed relations) and the signatures that do not involve types (which we have dealt with up to this section), we often relate to the latter as *standard* signatures (respectively, standard relation symbols, standard relations).

We consider queries defined by Spl⟨RGX⟩ programs, which are defined as follows. We assume two infinite and disjoint sets $\mathsf{Vars}_{\mathsf{str}}$ and $\mathsf{Vars}_{\mathsf{spn}}$ of *string variables* and *span variables*,

GENEO:

Cain␣son␣of␣Adam,␣Abel␣son␣of␣Adam,␣Enoch␣son␣of␣Cain,␣Irad␣...
Obed␣son␣of␣Ruth,␣Obed␣son␣of␣Boaz,␣Jesse␣son␣of␣Obed,␣David␣...

Figure 2 The input for the program in Example 20

respectively. To distinguish between the two, we mark a string variable with an overline (e.g., $\overline{x}$). By a *string term* we refer to an expression of the form $\overline{x}$ or $\overline{x}_y$, where $\overline{x}$ is a string variable and y is a span variable. In $\mathsf{Spl}\langle\mathsf{RGX}\rangle$, an *atom* over an m-ary relation symbol R is an expression of the form $R(\tau_1, \ldots, \tau_m)$ where τ_ℓ is a string term if $\ell \in [R]_{\mathsf{str}}$ or a span variable if $\ell \in [R]_{\mathsf{spn}}$. A *regex atom* is an expression of the form $\langle\tau\rangle[\gamma]$ where τ is a string term and γ is a regex formula. Unlike RGXlog, in which there is a single input string, in $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ a regex atom $\langle\tau\rangle[\gamma]$ indicates that the input for γ is τ. We allow regex formulas to use only span variables. An $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ *program* is a quadruple $\langle\mathcal{E}, \mathcal{I}, \Phi, \text{OUT}\rangle$ where:

- $\mathcal{E}$ is a mixed signature referred to as the *EDB* signature;
- $\mathcal{I}$ is a mixed signature referred to as the *IDB* signature;
- Φ is a finite set of *rules* of the form $\varphi \leftarrow \psi_1, \ldots, \psi_m$ where φ is an atom over $\mathcal{I}$ and each ψ_i is an atom over $\mathcal{I}$, an atom over $\mathcal{E}$, or a regex atom;
- $\text{OUT} \in \mathcal{I}$ is a designated *output* relation symbol.

We require the rules to be *safe* in the following sense: *(a)* every head variable occurs at least once in the body of the rule, *and (b)* every string variable $\overline{x}$ in the rule occurs, *as a string term*, in at least one relational atom (over $\mathcal{E}$ or $\mathcal{I}$) in the rule.

We extend $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ with *stratified negation* in the usual way: the set of relation symbols in $\mathcal{E} \cup \mathcal{I}$ is partitioned into *strata* $\mathcal{I}_0, \mathcal{I}_1, \ldots, \mathcal{I}_m$ such that $\mathcal{I}_0 = \mathcal{E}$, the body of each rule contains only relation symbols from strata that precede or the same as that of the head, and negated atoms in the body are from strata that strictly precede that of the head. In this case, *safe* rules are those for which every head variable occurs at least once in a *positive* atom in the body of the rule and every string variable $\overline{x}$ in the rule occurs, as a string term, in at least one *positive* relational atom (over $\mathcal{E}$ or $\mathcal{I}$) in the rule.

The semantics of an $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ program (with stratified negation) is similar to the semantics of RGXlog programs (with the standard interpretation of stratified negation in Datalog). Given a mixed instance D over $\mathcal{E}$, the $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ program $P = \langle\mathcal{E}, \mathcal{I}, \Phi, \text{OUT}\rangle$ computes the mixed instance $P(D)$ over $\mathcal{I}$ and emits the mixed relation OUT of $P(D)$. A query Q over $\mathcal{E}$ is *definable* in $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ if there exists an $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ program $P = \langle\mathcal{E}, \mathcal{I}, \Phi, \text{OUT}\rangle$ such that $\text{OUT}^{P(D)} = Q(D)$ for all mixed instances D over $\mathcal{E}$.

▶ **Example 20.** Following is an $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ program over the mixed signature of the instance of Figure 2. As usual, OUT is the relation symbol in the head of the last rule, here NONRLTV.

$$\text{ANCSTR}(\bar{x}, y, \bar{x}, z) \leftarrow \text{GENEO}(\bar{x})\,,\ \langle\bar{x}\rangle\gamma_{\mathsf{prnt}}(y, z)$$
$$\text{ANCSTR}(\bar{w}, y, \bar{x}, z) \leftarrow \text{ANCSTR}(\bar{w}, y, \bar{v}, y')\,,\ \text{GENEO}(\bar{x})\,,\ \langle\bar{x}\rangle\gamma_{\mathsf{prnt}}(z', z)\,,\ \text{STREQ}(\bar{x}_{z'}, \bar{v}_{y'})$$
$$\text{RLTV}(\bar{w}, y, \bar{x}, z) \leftarrow \text{ANCSTR}(\bar{v}, y', \bar{w}, y)\,,\ \text{ANCSTR}(\bar{u}, z', \bar{x}, z,)\,,\ \text{STREQ}(\bar{v}_{y'}, \bar{u}_{z'})$$
$$\text{NONRLTV}(\bar{w}, y, \bar{x}, z) \leftarrow \text{GENEO}(\bar{w})\,,\ \langle\bar{w}\rangle\gamma_{\mathsf{prsn}}(y)\,,\ \text{GENEO}(\bar{x})\,,\ \langle\bar{x}\rangle\gamma_{\mathsf{prsn}}(z),\ \neg\text{RLTV}(\bar{w}, y, \bar{x}, z)$$

The relation GENEO in Figure 2 contains strings that describe (partial) family trees. We assume for simplicity that every name that occurs in such a string is a unique identifier. The regex formulas $\gamma_{\mathsf{prsn}}(x)$ and $\gamma_{\mathsf{prnt}}(y, z)$ are the same as $\gamma_{\mathsf{cap}}(x)$ and $\gamma_{\mathsf{prnt}}(y, z)$ defined in Example 2, respectively. The first two rules of the program extract the relation ANCSTR that has four attributes: the first and third are string attributes and the second and fourth are span attributes. The first (respectively, third) attribute is the "context" string of the

second (respectively, fourth) span attribute. Observe the similarity to the corresponding definition in Example 5. Here, unlike Example 5, we need also to save the context string of each of the spans, and hence, we need two additional attributes. The second and third rules use the relation StrEq that holds pairs $(\bar{w}_y, \bar{x}_z)$ such that $\bar{w}_y$ and $\bar{x}_z$ are identical. This relation can be expressed in $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ similarly to RGXlog, as described in Section 4.

After evaluating the program, the relation Ancstr holds tuples $(\bar{w}, y, \bar{x}, z)$ such that $\bar{w}_y$ is an ancestor of $\bar{x}_z$. The relation Rltv holds tuples $(\bar{w}, y, \bar{x}, z)$ such that according to the information stored in Geneo, $\bar{w}_y$ is a relative of $\bar{x}_z$ (i.e., they share a common ancestor). The relation NonRltv holds tuples $(\bar{w}, y, \bar{x}, z)$ such that $\bar{w}_y$ is *not* a relative of $\bar{x}_z$. ◀

6.2 Equivalence to Polynomial Time

Let $\mathcal{E}$ be a span-free signature, and D an instance over $\mathcal{E}$. We define the *extended active domain* of D, in notation $adom^+(D)$, to be the union of the following two sets: *(a)* the set of all strings that appear in D, as well as all of their substrings; *and (b)* the set of all spans of strings of D.

Note that for every query Q definable as an $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ program $P = \langle\mathcal{E}, \mathcal{I}, \Phi, \text{Out}\rangle$, and every input database D over $\mathcal{E}$, we have $adom(Q(D)) \subseteq adom^+(D)$, that is, every output string is a substring of some string in D, and every output span is a span of some string in D. Our result in this section states that, under this condition, we can express in $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ with stratified negation every query Q computable in polynomial time.

▶ **Theorem 21.** *Let Q be a query over a span-free signature $\mathcal{E}$, with the property that $adom(Q(D)) \subseteq adom^+(D)$ for all instances D over $\mathcal{E}$. The following are equivalent:*

1. *Q is computable in polynomial time.*
2. *Q is computable in $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ with stratified negation.*

We remark that Theorem 21 can be extended to general mixed signatures $\mathcal{E}$ if we assume that every span mentioned in the input database D is within the boundary of some string in D. We also remark that Theorem 21 is incorrect without negation, and this can be shown using standard arguments of monotonicity. In addition, since we use negation, in order to prevent ambiguity we use the stratified semantics.

Proof idea. We now discuss the proof idea of Theorem 21. The direction $2 \to 1$ is straightforward, so we discuss only the direction $1 \to 2$. Let Q be a query over a span-free signature $\mathcal{E}$, with the property that $adom(Q(D)) \subseteq adom^+(D)$ for all instances D over $\mathcal{E}$. Assume that Q is computable in polynomial time. We need to construct an $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ program P with stratified negation for computing Q. We do so in two steps. In the first step, we apply Theorem 10 to get a (standard) Datalog$^\perp$ program P' that simulates Q. Yet, P' does not necessarily respect the *typing* conditions of $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ with respect to the two types *string* and *span*. So, in the second step, we transform P' to an $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ program P as desired. Next, we discuss each step in more detail.

First step. In order to produce the Datalog$^\perp$ program P', some adaptation is required to apply Theorem 10. First, we need to deal with the fact that the output of Q may include values that are not in the active domain of the input (namely, spans and substrings). Second, we need to establish a linear order over the active domain. Third, we need to assure that the query that Theorem 10 is applied on respects isomorphism. To solve the first problem, we extend the input database D with relations that contain every substring and every span of every string in D. This can be done using $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ rules with regex atoms. For the second problem, we construct a linear order over the domain of all substrings and spans of strings of

D, again using $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ rules. For this part, stratified negation is needed. For the third problem, we show how our extended input database allows us to restore D even if all values (strings and spans) are replaced with other values by applying an injective mapping.

Second step. In order to transform P' into a "legal" $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ program P that obeys the typing of attributes and variables, we do the following. First, we replace every IDB relation symbol R with every possible *typed* version of R by assigning types to attributes. Semantically, we view the original R as the union of all of its typed versions. Second, we replace every rule with every typed version of the rule by replacing relation symbols with their typed versions. Third, we eliminate rules that treat one or more variable inconsistently, that is, the same variable is treated once as a string variable and once as a span variable. The following example demonstrates the steps described above:

▶ **Example 22.** Let us consider the Datalog$^{\perp}$ program that contains the rule $R(x,y) \leftarrow S(x), T(y,z)$. The relation atom $R(x,y)$ has four different typed versions, such as the following.

- $R_{\mathsf{str},\mathsf{str}}(\overline{x},\overline{y})$ wherein both attributes are string attributes.
- $R_{\mathsf{spn},\mathsf{str}}(x,\overline{y})$ wherein the first attribute is a span attribute and the second is a string attribute.

The rule $R(x,y) \leftarrow S(x), T(y,z)$ has 2^5 different typed versions, one for each "type assignment" for its variables, such as the following.

- $R_{\mathsf{str},\mathsf{str}}(\overline{x},\overline{y}) \leftarrow S_{\mathsf{str}}(\overline{x}), T_{\mathsf{str},\mathsf{str}}(\overline{y},\overline{z})$
- $R_{\mathsf{spn},\mathsf{str}}(x,\overline{y}) \leftarrow S_{\mathsf{str}}(\overline{x}), T_{\mathsf{str},\mathsf{str}}(\overline{y},\overline{z})$

Note that the second rule is type inconsistent due to the variable x that is regarded as a span variable in the head atom and as a string variable in the atom $S_{\mathsf{str}}(\overline{x})$, and thus it is eliminated.

Finally, we prove that this replacement preserves the semantics of the program.

7 Conclusions

We studied RGXlog, namely, Datalog over regex formulas. We proved that this language expresses precisely the spanners computable in polynomial time. RGXlog is more expressive than the previously studied language of core spanners and, as we showed here, more expressive than even the language of generalized core spanners. We also observed that it takes very simple binary regex formulas to capture the entire expressive power. Unary regex formulas, on the other hand, do not suffice: in the Boolean case, they recognize precisely the regular languages, and in the non-Boolean case, they produce a strict subset of the regular spanners. Finally, we extended the equivalence result to $\mathsf{Spl}\langle\mathsf{RGX}\rangle$ with stratified negation over mixed instances, a model that generalizes both the relational model and the document spanners.

The expressive power of RGXlog is somewhat mysterious, since we do not yet have a good understanding of how to phrase some simple polynomial-time programs *naturally* in RGXlog. The constructive proof simulates the corresponding polynomial-time Turing machine, and does not lend itself to program clarity. For instance, is there a natural program for computing the *complement* of the transitive closure of a binary relation encoded by the input? An interesting future work is to investigate this aspect by studying the complexity of translating simple formalisms, such as generalized core spanners, into RGXlog.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

2 Jitendra Ajmera, Hyung-Il Ahn, Meena Nagarajan, Ashish Verma, Danish Contractor, Stephen Dill, and Matthew Denesuk. A CRM system for social media: challenges and experiences. In *WWW*, pages 49–58. ACM, 2013.

3 Marcelo Arenas, Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. A framework for annotating csv-like data. *PVLDB*, 9:876–887, 2016.

4 Edward Benson, Aria Haghighi, and Regina Barzilay. Event discovery in social media feeds. In *ACL*, pages 389–398. The Association for Computer Linguistics, 2011.

5 Pierre Boullier. From contextual grammars to range concatenation grammars. *Electr. Notes Theor. Comput. Sci.*, 53:41–52, 2001.

6 Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

7 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12, 2015.

8 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Declarative cleaning of inconsistencies in information extraction. *ACM Trans. Database Syst.*, 41(1):6:1–6:44, 2016.

9 Dominik D. Freydenberger. A logic for document spanners. In *ICDT*, volume 68 of *LIPIcs*, pages 13:1–13:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

10 Dominik D. Freydenberger and Mario Holldack. Document spanners: From expressive power to decision problems. In *ICDT*, volume 48 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

11 Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. Joining extractions of regular expressions. *CoRR*, abs/1703.10350, 2017.

12 Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*, pages 149–158. IEEE Computer Society, 2009.

13 Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113, January 2004.

14 Alon Y. Halevy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Static analysis in Datalog extensions. *J. ACM*, 48(5):971–1012, 2001.

15 Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artif. Intell.*, 194:28–61, 2013.

16 K. N. King. Alternating multihead finite automata. *Theor. Comput. Sci.*, 61:149–174, 1988.

17 Christopher C. Leary. *A Friendly Introduction to Mathematical Logic*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1999.

18 Alon Y. Levy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Equivalence, query-reachability, and satisfiability in datalog extensions. In Catriel Beeri, editor, *PODS*, pages 109–122. ACM Press, 1993.

19 Roger Levy and Christopher D. Manning. Deep dependencies from context-free statistical parsers: Correcting the surface dependency approximation. In *ACL*, pages 327–334. ACL, 2004.

20 Yunyao Li, Frederick Reiss, and Laura Chiticariu. SystemT: A declarative information extraction system. In *ACL*, pages 109–114. ACL, 2011.

21 Yoav Nahshon, Liat Peterfreund, and Stijn Vansummeren. Incorporating information extraction in the relational database model. In *WebDB*, page 6. ACM, 2016.

22 Christos H. Papadimitriou. A note on the expressive power of Prolog. *Bulletin of the EATCS*, 26:21–22, 1985.

23 M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves*, pages 92–101, Warszawa, 1929.

24 Christopher De Sa, Alexander Ratner, Christopher Ré, Jaeho Shin, Feiran Wang, Sen Wu, and Ce Zhang. DeepDive: Declarative knowledge base construction. *SIGMOD Record*, 45(1):60–67, 2016.

25 Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative information extraction using Datalog with embedded extraction predicates. In *VLDB*, pages 1033–1044, 2007.

26 Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. Incremental knowledge base construction using DeepDive. *PVLDB*, 8(11):1310–1321, 2015.

27 Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. YAGO: A core of semantic knowledge. In *WWW*, pages 697–706. ACM, 2007.

28 Hua Xu, Shane P. Stenner, Son Doan, Kevin B. Johnson, Lemuel R. Waitman, and Joshua C. Denny. MedEx: a medication information extraction system for clinical narratives. *JAMIA*, 17(1):19–24, 2010.

29 Alexander Yates, Michele Banko, Matthew Broadhead, Michael J. Cafarella, Oren Etzioni, and Stephen Soderland. TextRunner: Open information extraction on the web. In *ACL-HLT*, pages 25–26. ACL, 2007.

30 Huaiyu Zhu, Sriram Raghavan, Shivakumar Vaithyanathan, and Alexander Löser. Navigating the intranet with high precision. In *WWW*, pages 491–500. ACM, 2007.