Spanners: A Formal Framework for Information Extraction

Ronald Fagin
IBM Research – Almaden
San Jose, CA, USA

Benny Kimelfeld
IBM Research – Almaden
San Jose, CA, USA

Frederick Reiss
IBM Research – Almaden
San Jose, CA, USA

Stijn Vansummeren
Université Libre de Bruxelles (ULB)
Bruxelles, Belgium

ABSTRACT

An intrinsic part of information extraction is the creation and manipulation of relations extracted from text. In this paper, we develop a foundational framework where the central construct is what we call a *spanner*. A spanner maps an input string into relations over the spans (intervals specified by bounding indices) of the string. The focus of this paper is on the representation of spanners. Conceptually, there are two kinds of such representations. Spanners defined in a *primitive* representation extract relations directly from the input string; those defined in an *algebra* apply algebraic operations to the primitively represented spanners. This framework is driven by SystemT, an IBM commercial product for text analysis, where the primitive representation is that of regular expressions with capture variables.

We define additional types of primitive spanner representations by means of two kinds of automata that assign spans to variables. We prove that the first kind has the same expressive power as regular expressions with capture variables; the second kind expresses precisely the algebra of the *regular* spanners—the closure of the first kind under standard relational operators. The *core* spanners extend the regular ones by string-equality selection (an extension used in SystemT). We give some fundamental results on the expressiveness of regular and core spanners. As an example, we prove that regular spanners are closed under difference (and complement), but core spanners are not. Finally, we establish connections with related notions in the literature.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*Data models*; H.2.4 [**Database Management**]: Systems—*Textual databases, Relational databases, Rule-based databases*; I.5.4 [**Pattern Recognition**]: Applications—*Text processing*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Algebraic language theory, Classes defined by grammars or automata, Operations on languages*; [[**F**]: .1.1]Computation by Abstract DevicesModels of Computation[Automata, Relations between models]

PODS'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2066-5/13/06 ...$15.00.

General Terms

Theory

Keywords

Information extraction, spanners, regular expressions, finite-state automata

1. INTRODUCTION

Automatically extracting structured information from text is a task that has been pursued for decades. As a discipline, Information Extraction (IE) had its start with the DARPA Message Understanding Conference in 1987 [27]. While early work in the area focused largely on military applications, recent changes have made information extraction increasingly important to an increasingly broad audience. Trends such as the rise of social media have produced huge amounts of text data, while analytics platforms like Hadoop have at the same time made the analysis of this data more accessible to a broad range of users. Since most analytics over text involves the extraction of information items (at least as a first step), IE is nowadays an important part of data analysis in the enterprise.

Broadly speaking, there are two main schools of thought on the realization of IE: the *statistical* (machine-learning) methodology and the *rule-based* approach. The first started with simple models such as AutoSlog [41], CRYSTAL [42] and SRV [22], then progressed to approaches based on probabilistic graph models [32, 33, 36]. Within the rule-based approach, most of the solutions (e.g., GATE/JAPE [18]) build upon *cascaded finite-state transducers* [3]. Most systems in both categories were built for academic settings, where most users are highly-trained computational linguists, where workloads cover only a small number of very well-defined tasks and data sets, and where extraction throughput is far less important than the accuracy of results.

When IBM researchers, driven by the increasing importance of text data in the enterprise, attempted to use these existing tools to solve customers' analytics problems, they encountered a number of practical challenges. Users needed to have an intuitive understanding of machine learning or the ability to build and understand complex and highly interdependent rules. Determining why an extractor produced a given incorrect result was extremely difficult, which made impractical the reuse of extractors across different data sets and applications. Moreover, high CPU and memory requirements made extractors cost-prohibitive in deployment over large-scale data sets.

In 2005, researchers at the IBM Almaden Research Center began the design and development of a new system, specifically geared for practical information extraction in the enterprise. This effort led to SystemT, a rule-based IE system with an SQL-like declarative language named *AQL* (Annotation Query Language) [15, 31, 40]. The declarative nature of AQL enables new kinds of tools for extractor development [35], and a cost-based optimizer for performance [40]. In 2010, SystemT was released as a commercial IBM product.[1] An intensive study by Chiticariu et al. [15] shows the value of SystemT, in particular the high extent to which it overcomes the difficulties mentioned earlier.

Conceptually, AQL can be viewed as built upon two main operations that were supported already in the original research prototype of SystemT [40]. The first operation (expressed as "extract" statements) is the extraction of relations from the underlying text through simple mechanisms. The most commonly used of these mechanisms is that of regular expressions with capture variables. An important special case of that mechanism is the extraction of dictionary (*gazetteer*) matches that are distinguished from general regular expressions by their syntax and underlying implementation. The second operation (expressed as "select" statements) is the manipulation of the relations (from the first operation) through algebraic operators. There are three types of algebraic operators: standard relational operators (e.g., union, projection, join), text-centric operators (e.g., string equality and containment), and conflict resolution (mainly, resolving cases of overlapping spans when those are undesired). In the actual (productized) AQL syntax, these operators are expressed as clauses of a Select-From-Where flavor.[2] In time, SystemT evolved to support additional facilities, like part-of-speech tagging, shallow parsing of XML tags, sorting and additional aggregate functions.

In this paper, we embark on an investigation of the principles underlying AQL. Our ultimate goal is to establish a formal model that is robust enough to capture the principal capabilities of systems featuring AQL's principles, and yet, that is abstract enough to yield useful insights, and solutions with provable guarantees. Towards that, we develop here a framework that captures the core functionality of SystemT, and establish some fundamental results on expressiveness and on the relationship with existing literature. We believe that this work will be the basis of further investigation of tools for text analytics. We further believe that this work and its followups will shed light on the interplay between the textual and the relational querying models (in contrast to their traditional separation as distinct steps). In the remainder of this section, we give a more technical and detailed description of our framework and results.

A *span* of a string $\mathbf{s}$ (where $\mathbf{s}$ represents the text) represents the range of a substring of $\mathbf{s}$, and is given by two indices that specify where the range begins and ends within $\mathbf{s}$. For example, if $\mathbf{s}$ is `ACM_PODS_2013`, then the span $[5, 9\rangle$ refers to the part of $\mathbf{s}$ from the fifth to the eighth symbols inclusive, spanning the substring `PODS`. In this paper we introduce *spanners*, the central concept in our framework. Intuitively, a spanner extracts from a string $\mathbf{s}$ a relation over the spans of $\mathbf{s}$. It is formally defined as follows. An $\mathbf{s}$-*tuple* is associated with a finite domain V of *span variables*, and assigns a span of $\mathbf{s}$ to each variable in V. A *span relation* (*over* $\mathbf{s}$) is a set of $\mathbf{s}$-tuples, all over the same domain V of span variables. That set is naturally viewed as a relation, with the span variables playing the roles of the attribute names and the spans themselves used as attribute values. A *spanner* is a function that maps each string $\mathbf{s}$ into a span relation over $\mathbf{s}$.

For illustration, consider Figure 1, that is used for our running example in this paper. The figure shows two strings $\mathbf{s}$ and $\mathbf{t}$, and considers two spanners P_1 and P_2. The tables in the figure show the four span relations obtained by applying P_1 and P_2 to $\mathbf{s}$ and $\mathbf{t}$. For instance, the top row in the table of $P_1(\mathbf{s})$ shows the $\mathbf{s}$-tuple that assigns the spans $[1, 4\rangle$, $[5, 8\rangle$ and $[1, 8\rangle$ to the variables x, y and z, respectively.

This paper focuses on the representation of spanners. Conceptually, we distinguish between two types of spanner representations. The first type is that of a *primitive* representation, which is a mechanism that extracts the relation directly from the input string $\mathbf{s}$. An example is a regular expression with span variables embedded as capture variables, as in AQL; here, we call such an expression a *regex formula*. The second type of a spanner representation is that of an *algebra*, which is the closure of primitive representations (of some specific class) under some algebraic operators.

Aside from regex formulas, we define two additional primitive spanner representations that are based on two corresponding types of automata. An automaton of each type is an ordinary nondeterministic finite automaton (NFA), except that it is associated with a finite set V of variables, and along a run on a string it can decide to open (i.e., begin the assigned span for) or close (i.e., end the assigned span for) a variable. In an accepting run, each variable in V must be opened and closed exactly once. The difference between the two types is in the data structures that maintain the variables. In a *variable-stack automaton* (*vstk-automaton* for short), that data structure is a stack, and hence, the closed variable is always the most recently opened one. In a *variable-set automaton* (*vset-automaton* for short), that data structure is a set, and the automaton specifies the specific (previously opened) variable to close.

We begin by showing that regex formulas, vstk-automata and vset-automata are tightly related to each other. In particular, regex formulas and vstk-automata have the same expressive power. The vset-automata can express spanners that are not expressible by vstk-automata, since a spanner representable by the latter is necessarily *hierarchical*—the spans of every output $\mathbf{s}$-tuple are nested like balanced parentheses. We prove that the spanners expressible by regex formulas are precisely the hierarchical spanners representable by vset-automata. Moreover, we prove that the expressive power of vset-automata is precisely that of the algebra that closes regex formulas under union, projection and natural join on spans. Finally, we prove that these algebraic operators *do not* increase the expressive power of vset-automata. We call the spanners expressible by vset-automata *regular spanners*. The name arises from the fact that, in the Boolean case, the languages recognizable by vset-automata are the regular ones.

An algebraic operator of AQL that was not mentioned in the previous paragraph is *string-equality* selection, which selects the $\mathbf{s}$-tuples such that the spans for two specified variables x and y correspond to equal substrings of $\mathbf{s}$ (although x and y need not be the same span). The *core* spanners, which we view as capturing the core of AQL, are the ones expressible by regex formulas along with the operators union, projection, natural join on spans, and string-equality selection. In this language, one can also simulate selection operators for other common string relationships such as containment, prefix and suffix. Standard inexpressiveness results for regular expressions easily imply that core spanners are more expressive than regular spanners. We prove a key lemma for core spanners, the "core-simplification lemma," which states that every core spanner can be represented as a *single* vset-automaton, followed by string selections and then by a projection. This lemma is a crucial ingredient for our later proofs of inexpressiveness results.

Focusing on regular and core spanners, we also look at the ability to *simulate* selection operators based on string relations (relations whose entries are strings, not spans). More formally, for a

[1] SystemT is included in IBM InfoSphere BigInsights. http://www.ibm.com/software/data/infosphere/biginsights/.

[2] See http://publib.boulder.ibm.com/infocenter/bigins/v2r0/ for the complete reference of the AQL syntax.

String $\mathbf{s}$

A	a	a	_	A	b	b	_	a	a	_	B	a	a	_	B	b	b	_	b	_	A	b	a	a	_	A	b	b
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

$P_1(\mathbf{s})$

	x	y	z
μ_1	$[1,4\rangle$	$[5,8\rangle$	$[1,8\rangle$
μ_2	$[12,15\rangle$	$[16,19\rangle$	$[12,19\rangle$
μ_3	$[22,26\rangle$	$[27,30\rangle$	$[22,30\rangle$

$P_2(\mathbf{s})$

	x_1	x_2
μ_4	$[1,4\rangle$	$[22,26\rangle$

String $\mathbf{t}$

A	a	a	_	A	b	b	_	A	b	b
1	2	3	4	5	6	7	8	9	10	11

$P_1(\mathbf{t})$

	x	y	z
μ_5	$[1,4\rangle$	$[5,8\rangle$	$[1,8\rangle$
μ_6	$[5,8\rangle$	$[9,12\rangle$	$[5,12\rangle$

$P_2(\mathbf{t})$

	x_1	x_2
μ_7	$[1,4\rangle$	$[5,8\rangle$

Figure 1: Running example: strings $\mathbf{s}$ and $\mathbf{t}$, and the string relations obtained by applying two spanners P_1 and P_2

string relation R, the corresponding selection operator selects all the $\mathbf{s}$-tuples such that the substrings corresponding to a specified sequence of variables (of the same arity as R) is in R. We say that R is *selectable* by a class of spanners (e.g., the regular or core) if that class is closed under the selection operator for R. Like Barceló et al. [5], we look at three classes of string relations: the *recognizable relations* [8,20], which are contained in the *regular relations* [7,20], which are contained in the *rational relations* [8, 39]. We show that every recognizable relation is selectable by the core spanners. We also show the existence of a regular (hence rational) relation that is not selectable by the core spanners, and the existence of a relation that is selectable by the core spanners but is not rational (hence not regular). As for regular spanners, it turns out that their selectable string relations are *precisely* the recognizable ones.

In Section 5 we investigate the incorporation of the *difference* operator in our setting. We prove that core spanners are *not* closed under difference. By analogy to the relational model, this may sound straightforward because all the other operators are monotonic. But this argument is invalid here, because regex formulas have the ability to simulate non-monotonic functionality. As evidence, it turns out that regular spanners *are* closed under difference. Moreover, as further evidence, some relations of a non-monotonic flavor are selectable by the core spanners, like inequality, non-prefix and non-suffix. In contrast, we prove with the core-simplification lemma that non-substring is *not* selectable by the core spanners; with that, non-closure under difference is a simple corollary.

Due to space limitations, this paper does not include proofs; those will appear in the full version of this paper.

Related Work

There is a large body of work on designing query languages for string databases (i.e., databases in which the atomic data values are strings) [7, 9, 25, 26]. There are two important differences of these works with ours. First and foremost, the atomic data values within relations in a string database are strings, whereas the atomic data values within span relations are spans. This distinction is important because it yields a different semantics for natural join: in a string database two tuples will join if they contain the same string in the shared attributes, whereas in span relations two tuples will join if they contain the same span. As we show in Section 5, it is exactly the capability of testing for equality on strings that causes loss of closure under difference. A second important difference is that query languages for string databases not only support pattern-matching for the purpose of extracting relevant information from strings, but also support powerful operations for the purpose of transforming strings. Typically, these transformation operations even make the query language Turing-complete in the class of string-to-string functions that can be expressed. In contrast, we focus on pattern matching that has low complexity.

A database query language that is closely related to regular spanners is the language of *Conjunctive Regular Path Queries* (CRPQs for short) [10, 11, 16, 19, 21]. We analyze in depth the relationship between CRPQs and our spanners in Section 6.

There is also a large body of work in extending finite state automata (or regular expressions) with mechanisms such as variables or registers. For example, Grumbach et al. [28] study variable automata. These are simple extensions to finite state automata in which the finite alphabet consists not only of letters, but also of variables that range over an infinite additional alphabet in order to be able to accept strings formed over an infinite alphabet. In contrast, the automata we consider accept only strings over a finite alphabet, and assign to each variable a span. Neven and Schwentick [38] study the expressive power of *query automata* on strings and trees. These automata define mappings from input strings or trees to sets (i.e., unary relations) of positions in the input. Spanners, in contrast, define mappings from input strings to relations of arbitrary arity over the spans of the input. Barceló et al. [6] study the extension of regular expressions with variables. In this extension, a variable can be substituted for a single alphabet letter only. In contrast, our variables bind to spans. A different extension of regular expressions with variables is given by the so called *extended regular expressions* [1, 12, 14, 23, 24]. Here, variables can not only bind to a substring during matching, but can also be used to repeat a previously matched substring. We analyze in depth the relationship between extended regular expressions and spanners in Section 6.

Classic rule-based information extraction systems build upon the *Common Pattern Specification Language* [3] (or CPSL for short), where information extraction rules are specified based on *cascaded finite-state transducers*. The idea behind these transducers is similar to the notion of *attribute grammars* [29, 30]: rules are used to parse (parts of) the input, and each rule can be assigned an action defining the values of attributes to be associated to the matched part of the input. (These attributes are considered to be the "extracted information".) While Neven and Van den Bussche [37] have investigated the expressive power of attribute grammars in querying derivation trees generated by a fixed context-free grammar, we are not aware of any formal investigation of the expressive power of the cascaded finite-state string transducers employed by CPSL. This is probably due to the fact that CPSL does not have a formal semantics. Instead, it explicitly leaves important details to the discretion of the implementation system designer. In addition, CPSL provides many extensions to standard finite state transducers, most notably a complex disambiguation policy and the ability to write rule actions in a Turing complete language through calls to arbitrary user-defined functions. For these reasons, we do not directly compare our framework against CPSL.

Finally, there is a body of research rooted in Allen's seminal paper on interval algebra [2]. In particular, while spans can be viewed as intervals, and spanners can hence be viewed as defining relations over intervals, Allen's interval algebra focuses on reasoning over relationships between intervals, but is not concerned with strings or string matching.

2. SPANNERS

At its core, our focus system (SystemT) implements a textual query language (AQL) that translates the input string into a collection of relations; in turn, those relations are manipulated in a relational-database manner [15]. The values in those relations are spans of the input string. Here we model the creation of those relations by the notion of a *spanner*, which we formally define in this section. In the following section we discuss the representation of spanners, as well as extensions by relational operators. We begin with some preliminary concepts and terminology.

2.1 String Basics

Strings and spans. We fix a finite alphabet Σ of *symbols*. We denote by Σ^* the set of all finite strings over Σ, and by Σ^+ the set of all finite strings of length at least one over Σ. A *language over* Σ is a subset of Σ^*. Let $\mathbf{s} = \sigma_1 \cdots \sigma_n \in \Sigma^*$ be a string. The length n of $\mathbf{s}$ is denoted by $|\mathbf{s}|$. A *span* identifies a substring of $\mathbf{s}$ by specifying its bounding indices. 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 $\mathbf{s}$. We note that the more standard notation would be $[i, j)$, but we use $[i, j\rangle$ to distinguish spans from intervals. For example, $[1, 1)$ and $[2, 2)$ are both the empty interval, hence equal, but in the case of spans we have $[i, j\rangle = [i', j'\rangle$ if and only if $i = i'$ and $j = j'$ (and in particular, $[1, 1\rangle \neq [2, 2\rangle$). We denote by $\mathsf{Spans}(\mathbf{s})$ the set of all the spans of $\mathbf{s}$. Two spans $[i, j\rangle$ and $[i', j'\rangle$ of $\mathbf{s}$ *overlap* if $i \leq i' < j$ or $i' \leq i < j'$, and are *disjoint* otherwise. Finally, $[i, j\rangle$ *contains* $[i', j'\rangle$ if $i \leq i' \leq j' \leq j$.

EXAMPLE 2.1. In a running example that we will use throughout the paper, we fix the alphabet $\Sigma = \{\mathtt{A}, \mathtt{a}, \mathtt{B}, \mathtt{b}, \_\}$ where we think of _ as representing a space between words. Figure 1 shows two strings $\mathbf{s}$ and $\mathbf{t}$ in Σ^*. Later we discuss the tables in this figure. To clarify the meaning of the spans we mention, we write the index under each character of the strings. The span $[22, 26\rangle$ is a span of $\mathbf{s}$ (but not of $\mathbf{t}$, since $22 > |\mathbf{t}| + 1 = 12$) and we have $\mathbf{s}_{[22,26\rangle} = \mathtt{Abaa}$. Also, $\mathbf{s}_{[1,4\rangle}$ and $\mathbf{t}_{[1,4\rangle}$ are both $\mathtt{Aaa}$. □

Regular expressions. Regular expressions over Σ are defined by the language

$$\gamma := \emptyset \mid \epsilon \mid \sigma \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^*$$

where $\emptyset$ is the empty set, ϵ is the empty string, and $\sigma \in \Sigma$. Note that "$\vee$" is the disjunction operator, "$\cdot$" is the concatenation operator, and "$*$" is the Kleene-star operator. We use γ^+ as an abbreviation of $\gamma \cdot \gamma^*$, and $\gamma?$ as an abbreviation for $\gamma \vee \epsilon$. The language recognized by a regular expression γ (i.e., the set of strings $\mathbf{s} \in \Sigma^*$ that γ matches) is denoted by $\mathcal{L}(\gamma)$. A language L over Σ is *regular* if $L = \mathcal{L}(\gamma)$ for some regular expression γ.

String relations. An *n-ary string relation* is a (possibly infinite) subset of $(\Sigma^*)^n$. We will refer to the following well-known classes of string relations: recognizable relations, regular relations (sometimes also called synchronized relations), and rational relations (see Barceló et al. [5] for concise definitions of these classes, as well as a discussion on the relationships between these classes). We denote by REC the class of all recognizable string relations, and by REC_k the class of all recognizable relations of arity k. Similarly, we denote by REG (REG_k) the class of all (k-ary) regular relations, and by RAT (RAT_k) the class of all (k-ary) rational relations. It is known that $\mathrm{REC}_1 = \mathrm{REG}_1 = \mathrm{RAT}_1$ (they all give the regular languages), and that $\mathrm{REC}_k \subsetneq \mathrm{REG}_k \subsetneq \mathrm{RAT}_k$ for all $k > 1$.

Span relations. We fix an infinite set SVars of *span variables*, which may be assigned spans. The sets Σ^* and SVars are disjoint. For a finite set $V \subseteq \mathsf{SVars}$ of variables and a string $\mathbf{s} \in \Sigma^*$, a *$(V, \mathbf{s})$-tuple* is a mapping $\mu \colon V \to \mathsf{Spans}(\mathbf{s})$ that assigns a span of $\mathbf{s}$ to each variable in V. If V is clear from the context, or V is irrelevant, we may write just "$\mathbf{s}$-tuple" instead of "$(V, \mathbf{s})$-tuple.' A set of $(V, \mathbf{s})$-tuples is called a *$(V, \mathbf{s})$-relation*. A $(V, \mathbf{s})$-relation is also called a *span relation* (*over* $\mathbf{s}$). Note that a span relation is always finite, since there are only finitely many $(V, \mathbf{s})$-tuples (given that V and $\mathbf{s}$ are both finite).

2.2 Spanners

A *spanner* is an operator that transforms a given string into a span relation over that string. More formally, a spanner P is a function that is associated with a finite set V of variables, and that maps every string $\mathbf{s}$ to a $(V, \mathbf{s})$-relation $P(\mathbf{s})$. We denote the set V by $\mathsf{SVars}(P)$. We say that a spanner P is *n-ary* if $|\mathsf{SVars}(P)| = n$.

EXAMPLE 2.2. In our running example (started in Example 2.1) we use two spanners: a ternary spanner P_1 and a binary spanner P_2. Later we will specify what exactly each spanner extracts from a given string. For now, the span relations (tables) in Figure 1 show the results of applying the two spanners to the strings $\mathbf{s}$ and $\mathbf{t}$ (also in the figure). □

Following are some special types of spanners that we use throughout this paper.

Boolean Spanners. A spanner P is *Boolean* if $\mathsf{SVars}(P) = \emptyset$. In that case, $P(\mathbf{s}) = \mathbf{true}$ denotes that $P(\mathbf{s})$ consists of the empty $\mathbf{s}$-tuple, and $P(\mathbf{s}) = \mathbf{false}$ denotes that $P(\mathbf{s}) = \emptyset$. If P is Boolean, then we say that P *recognizes* the language of strings that evaluate to $\mathbf{true}$.

Hierarchical spanners. Let P be a spanner. Let $\mathbf{s} \in \Sigma^*$ be a string, and let $\mu \in P(\mathbf{s})$ be an $\mathbf{s}$-tuple. We say that μ is *hierarchical* if for all variables $x, y \in \mathsf{SVars}(P)$ one of the following holds: *(1)* the span $\mu(x)$ contains $\mu(y)$, *(2)* the span $\mu(y)$ contains $\mu(x)$, *or (3)* the spans $\mu(x)$ and $\mu(y)$ are disjoint. As an example, the reader can verify that all the tuples in Figure 1 are hierarchical. We say that P is *hierarchical* if μ is hierarchical for all $\mathbf{s} \in \Sigma^*$ and $\mu \in P(\mathbf{s})$. We denote by $\mathbf{HS}$ the class of all hierarchical spanners.

Universal spanners. Let P be a spanner. We say that P is *total on* $\mathbf{s}$ if $P(\mathbf{s})$ consists of all the $\mathbf{s}$-tuples over $\mathsf{SVars}(P)$. (Note that over a finite set of variables, there are only finitely many $\mathbf{s}$-tuples.) We say that P is *hierarchically total on* $\mathbf{s}$ if $P(\mathbf{s})$ consists of (exactly) all the hierarchical $\mathbf{s}$-tuples. Let $Y \subseteq \mathsf{SVars}$ be a finite set of variables. The *universal spanner over* Y, denoted Υ_Y, is the unique spanner P such that $\mathsf{SVars}(P) = Y$ and P is total on every $\mathbf{s} \in \Sigma^*$. The *universal hierarchical spanner over* Y, denoted $\Upsilon^{\mathbf{H}}_Y$, is the unique spanner P such that $\mathsf{SVars}(P) = Y$ and P is hierarchically total on every $\mathbf{s} \in \Sigma^*$.

3. SPANNER REPRESENTATION

In our system of focus (SystemT), querying an input string $\mathbf{s}$ entails two steps (conceptually) [15]. In the first step, span relations over $\mathbf{s}$ are extracted by standard string-oriented tools like regular expressions with capture variables or dictionary matchers. In the second step, the final result is obtained by applying algebraic operators to the relations of the first step. We model these two steps by two corresponding types of representations for spanners. The first type is that of *primitive spanner representations*. The second type extends the first type by including operators of a relational algebra.

3.1 Primitive Spanner Representations

We introduce here three types of primitive spanner representations. The first is that of *regular-expression formulas* that extend regular expressions by including variables. The second and third are special automata that we call *variable-stack* and *variable-set* automata.

3.1.1 Regex Formulas

A *regular-expression formula*, or *regex formula* for short, is a regular expression with capture variables. The syntax of such formulas is almost the same as that of regular expressions:

$$\gamma := \emptyset \mid \epsilon \mid \sigma \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^* \mid x\{\gamma\} \qquad (1)$$

The added alternative is $x\{\gamma\}$, where $x \in \mathsf{SVars}$. We denote by $\mathsf{SVars}(\gamma)$ the set of variables that occur in γ. Before we formally define the spanner represented by a regex formula, we give an example.

EXAMPLE 3.1. We continue with our running example. Consider the formula γ_1 that is defined by

$$(\Sigma^* \cdot \_)^* \cdot z\Big\{x\{\gamma_{\mathsf{1stCap}}\} \cdot \_ \cdot y\{\gamma_{\mathsf{1stCap}}\}\Big\} \cdot (\_ \cdot \Sigma^*)^* \qquad (2)$$

where γ_{1stCap} is the regular expression $(\mathtt{A} \vee \mathtt{B}) \cdot (\mathtt{a} \vee \mathtt{b})^*$. After we define the spanner represented by a regex formula, it will turn out that γ_1 has the result of P_1 in Figure 1 on the strings $\mathbf{s}$ and $\mathbf{t}$. Note that $\mathsf{SVars}(\gamma_1) = \{x, y, z\}$. □

We now formally define the spanner that a regex formula represents. This definition is based on the notion of a *parse tree*. In general, a *tree* is associated with an alphabet Λ of labels, and is recursively defined as follows: if $t_1, \dots, t_n$ are trees (where $n \geq 0$) and $\lambda \in \Lambda$, then $\lambda(t_1 \cdots t_n)$ is a tree.

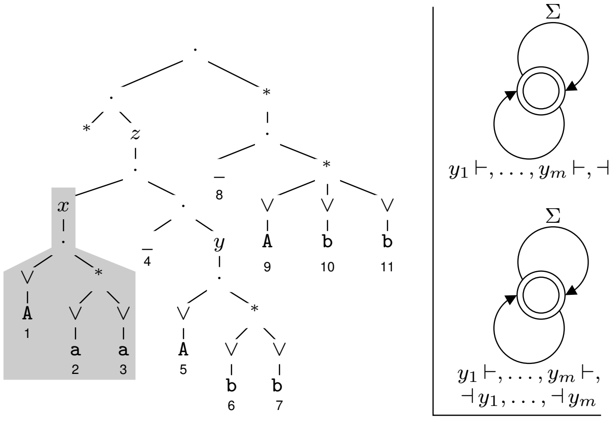
Let Λ be the alphabet $\Sigma \cup \mathsf{SVars} \cup \{\epsilon, \vee, \cdot, {}^*\}$. Let γ be a regex formula, and let $\mathbf{s}$ be a string. We use the following inductive definition. A tree t over the alphabet Λ is a *γ-parse for* $\mathbf{s}$ if one of the following holds.

- $\gamma = \epsilon$, $\mathbf{s} = \epsilon$, and $t = \epsilon()$.
- $\gamma = \sigma \in \Sigma$, $\mathbf{s} = \sigma$, and $t = \sigma()$.
- $\gamma = \gamma_1 \vee \gamma_2$, and $t = \vee(t')$ where t' is either a γ_1-parse or a γ_2-parse for $\mathbf{s}$.
- $\gamma = \gamma_1 \cdot \gamma_2$, and $t = \cdot(t_1 t_2)$ where t_i is a γ_i-parse for $\mathbf{s}_i$ $(i = 1, 2)$ for some strings $\mathbf{s}_1$ and $\mathbf{s}_2$ such that $\mathbf{s} = \mathbf{s}_1\mathbf{s}_2$.
- $\gamma = \delta^*$ and there are strings $\mathbf{s}_1, \dots, \mathbf{s}_n$ such that $\mathbf{s} = \mathbf{s}_1 \cdots \mathbf{s}_n$, $t = {}^*(t_1 \cdots t_n)$, and each t_i is a δ-parse for $\mathbf{s}_i$ $(i = 1, \dots, n)$.
- $\gamma = x\{\delta\}$ and $t = x(t_\delta)$ where t_δ is δ-parse for $\mathbf{s}$.

EXAMPLE 3.2. We continue with our running example. Figure 2(a) shows a γ_1-parse for $\mathbf{t}$ for the regex formula γ_1 of Example 3.1 and the string $\mathbf{t}$ of Figure 1. As we did with Figure 1, we write the index under each character. □

Note that there is no parse tree for the regex formula $\emptyset$. Clearly, a string $\mathbf{s}$ matches the regex formula γ, when variables are ignored, if and only if there exists a γ-parse for $\mathbf{s}$. In principle, a γ-parse t for $\mathbf{s}$ should determine one assignment for $\mathsf{SVars}(\gamma)$, as we later define. But for that, we need t to have *exactly* one occurrence of each variable in $\mathsf{SVars}(\gamma)$. So we restrict our regex formulas to those that guarantee such a behavior of t, a property we call *functional*.

DEFINITION 3.3. A regex formula γ is *functional* if for every string $\mathbf{s} \in \Sigma^*$ and γ-parse t for $\mathbf{s}$, each variable in $\mathsf{SVars}(\gamma)$ has precisely one occurrence in t. □



(a) (b)

Figure 2: (a) A γ_1-parse for $\mathbf{t}$ for the regex formula γ_1 of (2) (Example 3.1) and the string $\mathbf{t}$ of Figure 1 (b) A vstk-automaton A with $\llbracket A \rrbracket = \Upsilon^{\mathbf{H}}_Y$ (top) and a vset-automaton B with $\llbracket B \rrbracket = \Upsilon_Y$ (bottom) for $Y = \{y_1, \dots, y_m\}$

Note that a regex formula can be functional even though it contains multiple occurrences of a variable. An example is the regex formula γ given by $x\{\mathtt{a}\} \vee x\{\mathtt{b}\}$, which has two occurrences of the variable x, although each γ-parse has only one occurrence of x.

EXAMPLE 3.4. Consider again the regex formula γ_1 of Example 3.1. Recall that $\mathsf{SVars}(\gamma_1) = \{x, y, z\}$. Observe that in the γ_1-parse of Figure 2(a), each variable in $\mathsf{SVars}(\gamma_1)$ has indeed exactly one occurrence. In fact, it can be easily verified that this is the case for every γ_1-parse. Consequently, γ_1 is functional. □

Although Definition 3.3 is non-constructive, functionality is a property that can be tested in polynomial time.

PROPOSITION 3.5. *Whether a given formula γ is functional can be tested in polynomial time.*

In the remainder of this paper we implicitly assume that every involved regex formula is functional.

Let γ be a regex formula, and let p be a γ-parse for a string $\mathbf{s}$. If v is a node of p, then the subtree that is rooted at v naturally maps to a span p_v of $\mathbf{s}$. By μ^p we denote the assignment that maps each variable x to the span $\mu^p(x) = p_v$, where v is the unique node of t that is labeled by x. Note that v indeed exists, and is indeed unique, since we assume that γ is functional.

EXAMPLE 3.6. Let p be the γ_1-parse of $\mathbf{t}$ depicted in Figure 2(a), where γ_1 is defined in Example 3.1 and $\mathbf{t}$ is shown in Figure 1. The subtree of p rooted at the node labeled x is shaded grey. We have $\mu^p(x) = [1, 4\rangle$, $\mu^p(y) = [5, 8\rangle$, and $\mu^p(z) = [1, 8\rangle$. Hence, μ^p is the $\mathbf{t}$-tuple μ_5 of Figure 1. □

The spanner $\llbracket \gamma \rrbracket$ that is represented by the regex formula γ is the one where $\mathsf{SVars}(\llbracket \gamma \rrbracket)$ is the set $\mathsf{SVars}(\gamma)$, and where $\llbracket \gamma \rrbracket(\mathbf{s})$ is the span relation $\{\mu^p \mid p \text{ is a } \gamma\text{-parse for } \mathbf{s}\}$.

EXAMPLE 3.7. Consider again the regex formula γ_1 of Example 3.1, the strings $\mathbf{s}$ and $\mathbf{t}$ of Figure 1, and the spanner P_1 mentioned in that figure. The reader can verify that $\llbracket \gamma_1 \rrbracket(\mathbf{s}) = P_1(\mathbf{s})$ and that $\llbracket \gamma_1 \rrbracket(\mathbf{t}) = P_1(\mathbf{t})$. □

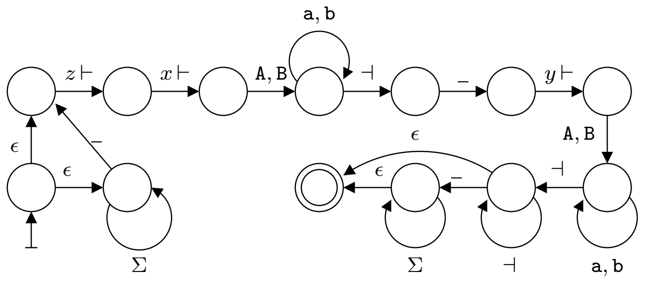


Figure 3: A vstk-automaton A with $[\![A]\!] = [\![\gamma_1]\!]$ for the regex formula γ_1 of (2) (Example 3.1)

3.1.2 *Variable-Stack Automata*

In this section, we define an automaton representation of a spanner. We call this automaton a *variable-stack automaton*, or just *vstk-automaton* for short. Later we will show that vstk-automata capture precisely the expressive power of regex formulas (that is, the two classes of spanner representation can express the same set of spanners).

Formally, a vstk-automaton is a tuple (Q, q_0, q_f, δ), where: Q is a finite set of *states*, $q_0 \in Q$ is an *initial state*, $q_f \in Q$ is an *accepting state*, and δ is a finite transition relation consisting of triples, each having one of the forms (q, σ, q'), (q, ϵ, q'), $(q, x\vdash, q')$ or $(q, \dashv, q')$, where $q, q' \in Q$, $\sigma \in \Sigma$, $x \in \mathsf{SVars}$, and $\dashv$ is a special *pop* symbol.

EXAMPLE 3.8. Figure 3 is a representation of a vstk-automaton A. Each circle represents a state, the double circle represents an acceping state, and a label a on an edge from q to q' represents the transition (q, a, q'). Conventionally, as a shorthand notation we write the sequence $a_1, \dots, a_k$ of labels on the edge from q to q' instead of the k edges $(q, a_1, q'), \dots, (q, a_k, q')$. Moreover, if $\Sigma = \{\sigma_1, \dots, \sigma_m\}$ then we write the label Σ instead of $\sigma_1, \dots, \sigma_m$. Later we will link the vstk-automaton A to our running example. $\square$

Let A be a vstk-automaton. We denote by $\mathsf{SVars}(A)$ the set of variables that occur in the transitions of A. A *configuration* of a vstk-automaton A is a tuple $c = (q, \vec{v}, Y, i)$, where $q \in Q$ is the *current state*, $\vec{v}$ is a finite sequence of variables called the *current variable stack*, $Y \subseteq \mathsf{SVars}(A)$ is the set of *available variables*, and i is an index in $\{1, \dots, n+1\}$ (pointing to the next character to be read from $\mathbf{s}$).

Let $\mathbf{s} = \sigma_1 \cdots \sigma_n$ be a string and let A be a vstk-automaton. A *run* ρ of A on $\mathbf{s}$ is a sequence $c_0, \dots, c_m$ of configurations, such that $c_0 = (q_0, \epsilon, \mathsf{SVars}(A), 1)$, and for all $j = 0, \dots, m-1$ one of the following holds for $c_j = (q_j, \vec{v}_j, Y_j, i_j)$ and $c_{j+1} = (q_{j+1}, \vec{v}_{j+1}, Y_{j+1}, i_{j+1})$:

1. $\vec{v}_{j+1} = \vec{v}_j$, $Y_{j+1} = Y_j$, and either (a) $i_{j+1} = i_j + 1$ and $(q_j, s_{i_j}, q_{j+1}) \in \delta$ (ordinary transition), or (b) $i_{j+1} = i_j$ and $(q_j, \epsilon, q_{j+1}) \in \delta$ (epsilon transition).

2. $i_{j+1} = i_j$, and for some $x \in \mathsf{SVars}(A)$, either (a) $\vec{v}_{j+1} = \vec{v}_j \cdot x$, $x \in Y_j$, $Y_{j+1} = Y_j \setminus \{x\}$ and $(q_j, x\vdash, q_{j+1}) \in \delta$ (variable push), or (b) $\vec{v}_j = \vec{v}_{j+1} \cdot x$, $Y_{j+1} = Y_j$ and $(q_j, \dashv, q_{j+1}) \in \delta$ (variable pop).

An easy observation is that every configuration $(q, \vec{v}, Y, i)$ in a run is such that $\vec{v}$ and Y do not share any common variable.

The run $\rho = c_0, \dots, c_m$ is *accepting* if $c_m = (q_f, \epsilon, \emptyset, n+1)$. We let $\mathsf{ARuns}(A, \mathbf{s})$ denote the set of all accepting runs of A on $\mathbf{s}$. If $\rho \in \mathsf{ARuns}(A, \mathbf{s})$, then for each $x \in \mathsf{SVars}(A)$ the run ρ has a unique configuration $c_b = (q_b, \vec{v}_b, Y_b, i_b)$ where x occurs in the current version of $\vec{v}$ (i.e., $\vec{v}_b$) for the first time; and later than that ρ has a unique configuration $c_e = (q_e, \vec{v}_e, Y_e, i_e)$ where x is occurs in the current version of $\vec{v}$ (i.e., $\vec{v}_e$) for the last time; the span $[i_b, i_e\rangle$ is denoted by $\rho(x)$. By μ^ρ we denote the $\mathbf{s}$-tuple that maps each variable $x \in \mathsf{SVars}(A)$ to the span $\rho(x)$. The spanner $[\![A]\!]$ that is represented by A is the one where $\mathsf{SVars}([\![A]\!])$ is the set $\mathsf{SVars}(A)$, and where $[\![A]\!](\mathbf{s})$ is the span relation $\{\mu^\rho \mid \rho \in \mathsf{ARuns}(A, \mathbf{s})\}$.

EXAMPLE 3.9. Consider the vstk-automaton A of Figure 3, described in Example 3.8. Observe that $\mathsf{SVars}(A) = \{x, y, z\}$. Note that in a run ρ, when reaching the final transition $(q, \dashv, q')$ (the leftmost occurrence of $\dashv$ in the bottom row), there is only one variable that is open, namely z. Hence, that transition can take place at most once. Moreover, if ρ is accepting then ρ must take that transition *exactly* once, since otherwise z would not be closed.

Continuing with our running example, now consider again the regex-formula γ_1 of (2), introduced in Example 3.1. The reader can verify that A and γ_1 define the same spanner, that is, $[\![\gamma_1]\!] = [\![A]\!]$. $\square$

EXAMPLE 3.10. The top part of Figure 2(b) depicts a single-state vstk-automaton A where we have $\mathsf{SVars}(A) = Y$, with $Y = \{y_1, \dots, y_m\}$. The reader can verify that $[\![A]\!]$ is the universal hierarchical spanner $\Upsilon^{\mathbf{H}}_Y$. In particular, this example shows that the universal hierarchical spanners are expressible by vstk-automata. $\square$

3.1.3 *Variable-Set Automata*

A *variable-set automaton* (or *vset-automaton*) is defined to be a tuple (Q, q_0, q_f, δ) like a vstk-automaton, except δ does not have triples $(q, \dashv, q')$; instead, δ has triples $(q, \dashv x, q')$ where $x \in \mathsf{SVars}$. We denote by $\mathsf{SVars}(A)$ the set of variables that occur in the transitions of A.

The difference between the two types of automata is also in the definition of a *configuration* and a *run*. In a vset-automaton, a *set* of variables is used rather than a *stack*. More precisely, a configuration of a vset-automaton A is a tuple $c = (q, V, Y, i)$, where $q \in Q$ is the *current state*, $V \subseteq \mathsf{SVars}(A)$ is the *active variable set*, $Y \subseteq \mathsf{SVars}(A)$ is the set of *available variables*, and i is an index in $\{1, \dots, n+1\}$.

For a string $\mathbf{s} = s_1, \dots, s_n$, a *run* ρ of A on $\mathbf{s}$ is a sequence $c_0, \dots, c_m$ of configurations, where $c_0 = (q_0, \emptyset, \mathsf{SVars}(A), 1)$, and for $j = 0, \dots, m-1$ one of the following holds for $c_j = (q_j, V_j, Y_j, i_j)$ and $c_{j+1} = (q_{j+1}, V_{j+1}, Y_{j+1}, i_{j+1})$:

1. $V_{j+1} = V_j$, $Y_{j+1} = Y_j$, and either (a) $i_{j+1} = i_j + 1$ and $(q_j, s_{i_j}, q_{j+1}) \in \delta$ (ordinary transition), or (b) $i_{j+1} = i_j$ and $(q_j, \epsilon, q_{j+1}) \in \delta$ (epsilon transition).

2. $i_{j+1} = i_j$ and for some $x \in \mathsf{SVars}(A)$, either (a) $x \in Y_j$, $V_{j+1} = V_j \cup \{x\}$, $Y_{j+1} = Y_j \setminus \{x\}$, and $(q_j, x\vdash, q_{j+1}) \in \delta$, (variable insert), or (b) $x \in V_j$, $V_{j+1} = V_j \setminus \{x\}$, $Y_{j+1} = Y_j$ and $(q_j, \dashv x, q_{j+1}) \in \delta$ (variable remove).

Note that in a run, each configuration (q, V, Y, i) is such that V and Y are disjoint. The run $\rho = c_0, \dots, c_m$ is *accepting* if $c_m = (q_f, \emptyset, \emptyset, n+1)$. The definitions of $\mathsf{ARuns}(A, \mathbf{s})$ and $[\![A]\!]$ are similar to those for a vstk-automaton (except that we replace the stack $\vec{v}$ with the set V).

EXAMPLE 3.11. Consider again Figure 2(b). The bottom part depicts a single-state vset-automaton B with $\mathsf{SVars}(B) = Y$, where $Y = \{y_1, \dots, y_m\}$. The reader can verify that $[\![B]\!] = \Upsilon_Y$. In particular, this example shows that the universal spanners are expressible by vset-automata. This example also shows that vset-automata can express spanners that regex formulas and vstk-automata cannot. In particular, an easy observation is that the spanner defined by a regex formula, or a vstk-automaton, is necessarily hierarchical. But $\llbracket B \rrbracket$ is certainly not hierarchical. □

3.1.4 Primitive Spanner Representations

We have defined three types of spanner representations. By RGX we denote the class of (functional) regex formulas, by $\mathsf{VA}_{\mathsf{stk}}$ we denote the class of vstk-automata, and by $\mathsf{VA}_{\mathsf{set}}$ we denote the class of vset-automata. We will refer to these three as our *primitive spanner representations* (to contrast with algebraic extensions of these representations).

If SR is any class spanner representations, like the primitive classes RGX, $\mathsf{VA}_{\mathsf{stk}}$ or $\mathsf{VA}_{\mathsf{set}}$, then $\llbracket SR \rrbracket$ represents the set of all the spanners representable by SR; that is, $\llbracket SR \rrbracket = \{\llbracket r \rrbracket \mid r \in SR\}$. For example, $\llbracket \mathsf{RGX} \rrbracket$ is the set of all the spanners $\llbracket \gamma \rrbracket$, where γ is a regex formula.

As mentioned in Example 3.11, every spanner defined by a regex formula or vstk-automaton is hierarchical. In our terminology it is stated as $\llbracket \mathsf{RGX} \rrbracket \subseteq \mathbf{HS}$ and $\llbracket \mathsf{VA}_{\mathsf{stk}} \rrbracket \subseteq \mathbf{HS}$. In Example 3.11 we also mentioned that $\llbracket \mathsf{VA}_{\mathsf{set}} \rrbracket \not\subseteq \mathbf{HS}$. Later, we will show that $\llbracket \mathsf{RGX} \rrbracket = \llbracket \mathsf{VA}_{\mathsf{stk}} \rrbracket$. In fact, we will show that the class of spanners definable by a vstk-automaton is *precisely* the class of hierarchical spanners definable by a vset-automaton, or in our notation, $\llbracket \mathsf{VA}_{\mathsf{stk}} \rrbracket = \llbracket \mathsf{VA}_{\mathsf{set}} \rrbracket \cap \mathbf{HS}$.

3.2 Spanner Algebras

Consider a class SR of spanner representations (e.g., one of our primitive representations). We extend SR with *algebraic operator symbols* to form a *spanner algebra*. More formally, each operator symbol corresponds to a *spanner operator*, which is a function that takes as input a fixed-length sequence of spanners (usually one or two, depending on whether the operator is unary or binary), and outputs a single spanner. We now define the spanner operators we focus on in this paper. Let P, P_1 and P_2 be spanners, and let $\mathbf{s}$ be a string.

- **Union.** The union $P_1 \cup P_2$ is defined when P_1 and P_2 are *union compatible*, that is, $\mathsf{SVars}(P_1) = \mathsf{SVars}(P_2)$. In that case, $\mathsf{SVars}(P_1 \cup P_2) = \mathsf{SVars}(P_1)$ and $(P_1 \cup P_2)(\mathbf{s}) = P_1(\mathbf{s}) \cup P_2(\mathbf{s})$.
- **Projection.** If $Y \subseteq \mathsf{SVars}(P)$, then $\pi_Y P$ is the spanner with $\mathsf{SVars}(\pi_Y P) = Y$, where $\pi_Y P(\mathbf{s})$ is obtained from $P(\mathbf{s})$ by restricting the domain of each $\mathbf{s}$-tuple to Y.
- **Natural join.** The spanner $P_1 \bowtie P_2$ is defined as follows. We have $\mathsf{SVars}(P_1 \bowtie P_2) = \mathsf{SVars}(P_1) \cup \mathsf{SVars}(P_2)$, and $(P_1 \bowtie P_2)(\mathbf{s})$ consists of all $\mathbf{s}$-tuples μ that agree with some $\mu_1 \in P_1(\mathbf{s})$ and $\mu_2 \in P_2(\mathbf{s})$; note that the existence of μ implies that μ_1 and μ_2 agree on the common variables of P_1 and P_2, that is, $\mu_1(x) = \mu_2(x)$ for all $x \in \mathsf{SVars}(P_1) \cap \mathsf{SVars}(P_2)$.
- **String selection.** Let R be a k-ary string relation. The string-selection operator ς^R is parameterized by k variables $x_1, \dots, x_k$ in $\mathsf{SVars}(P)$, and may then be written as $\varsigma^R_{x_1,\dots,x_k}$. If P' is $\varsigma^R_{x_1,\dots,x_k} P$, then the span relation $P'(\mathbf{s})$ is taken to be the restriction of $P(\mathbf{s})$ to those $\mathbf{s}$-tuples μ such that $(\mathbf{s}_{\mu(x_1)}, \dots, \mathbf{s}_{\mu(x_k)}) \in R$.

Regarding the natural join, observe that here pairs of tuples are joined based on having equal *spans* in shared variables. This is distinct from the natural join in query languages for string databases [7, 9, 25, 26], where tuples are joined if they have the equal *substrings* in shared attributes. Also observe that in the special case where P_1 and P_2 are union compatible, the spanner $P_1 \bowtie P_2$ produces the intersection $P_1(\mathbf{s}) \cap P_2(\mathbf{s})$ for the given string $\mathbf{s}$; in that case, we denote $P_1 \bowtie P_2$ also as $P_1 \cap P_2$. As another special case, if $\mathsf{SVars}(P_1)$ and $\mathsf{SVars}(P_2)$ are disjoint, then $P_1 \bowtie P_2$ produces the Cartesian product of $P_1(\mathbf{s})$ and $P_2(\mathbf{s})$; in that case, we denote $P_1 \bowtie P_2$ also as $P_1 \times P_2$.

In this work we focus mainly on one particular string-selection operator, namely the binary $\varsigma^{=}$. As defined above, $\varsigma^{=}_{x,y} P(\mathbf{s})$ restricts $P(\mathbf{s})$ to those $\mathbf{s}$-tuples μ with $\mathbf{s}_{\mu(x)} = \mathbf{s}_{\mu(y)}$. Later, we also consider other string selections (featuring other binary string relations). We do not include the *difference* operator yet, but rather dedicate to it a separate discussion in Section 5.

For clarity of presentation, we will abuse notation by using the operator symbol itself to represent the spanner operator. As an example, if γ_1 and γ_2 are regex formulas, then the expression $\gamma_1 \bowtie \gamma_2$ is well formed, and it represents the spanner $\llbracket \gamma_1 \rrbracket \bowtie \llbracket \gamma_2 \rrbracket$. Similarly, if A_1 and A_2 are vstk-automata then $A_1 \cup A_2$ is well formed assuming union compatibility, that is, $\mathsf{SVars}(A_1) = \mathsf{SVars}(A_2)$. Similarly, if A is a vset-automaton then $\pi_Y A$ is well formed assuming $Y \subseteq \mathsf{SVars}(A)$, and similarly $\varsigma^{=}_{x,y} A$ is well formed assuming $x, y \in \mathsf{SVars}(A)$.

EXAMPLE 3.12. We continue with our running example. Let γ_{12} be the regex formula that captures all spans x_1 and x_2 such that x_1 ends before x_2 begins; that is:

$$\gamma_{12}(x_1, x_2) \stackrel{\mathsf{def}}{=} \Sigma^* \cdot x_1\{\Sigma^*\} \cdot \Sigma^* \cdot x_2\{\Sigma^*\} \cdot \Sigma^*$$

The following algebraic expression is denoted as γ_2.

$$\pi_{x_1,x_2}\Big(\varsigma^{=}_{y_1,y_2}\,(\gamma_1(x_1,y_1,z_1) \bowtie \\ \gamma_1(x_2,y_2,z_2) \bowtie \gamma_{12}(x_1,x_2))\Big),$$

where we use $\gamma_1(x_i, y_i, z_i)$ as the regex-formula that is obtained from γ_1 of (2) (Example 3.1) by replacing x, y and z with x_i, y_i and z_i, respectively. Observe that γ_2 selects all the spans x_1 and x_2 that occur in tuples of γ_1, such that the corresponding y_1 and y_2 span the same substrings (though y_1 and y_2 themselves are not required to be equal as spans), and moreover, x_1 ends before x_2 begins. Consider the strings $\mathbf{s}$ and $\mathbf{t}$ in Figure 1. The reader can verify that $\llbracket \gamma_2 \rrbracket$ has the output of P_2 (also shown in the figure) for these two strings. □

A *spanner algebra* is a finite set of spanner operators. If SR is a class of spanner representations and O is a spanner algebra, then SR^O denotes the class of all the spanner representations defined by applying (compositions of) the operators in O to the representations in SR. In other words, S^O is the closure of SR under O (when O is taken as a set of operator symbols); consequently, $\llbracket SR^O \rrbracket$ is the closure of $\llbracket SR \rrbracket$ under O (when O is now taken as a set of spanner operators). For example, one of the algebras we later explore is $\mathsf{VA}_{\mathsf{set}}^{\{\cup,\pi,\bowtie,\varsigma^{=}\}}$. As another example, the expression γ_2 of Example 3.12 is in $\mathsf{RGX}^{\{\pi,\bowtie,\varsigma^{=}\}}$.

4. REGULAR AND CORE SPANNERS

In this section we define the classes of regular and core spanners, and study their relative expressive power.

4.1 Regular Spanners

We call a spanner *hierarchical regular* if it is definable by a vstk-automaton. We call a spanner *regular* if it is definable by a vset-automaton. In this section, we explore expressiveness aspects of hierarchical regular and regular spanners.

Observe that vstk-automata, vset-automata and NFAs are basically the same objects in the Boolean case. In particular, a language $L \subseteq \Sigma^*$ is recognized by some Boolean hierarchical regular spanner if and only if L is recognized by some Boolean regular spanner if and only if L is regular. Hence, the results of this section are of interest only in the non-Boolean case.

Key constructs that we later utilize for establishing our results here are those of a *transition graph* and the special case of a *path union*, both introduced in the next section.

4.1.1 Transition Graphs and Path Unions

We define two types of transition graphs, which function similarly to vstk-automata and vset-automata, respectively, except that in a single transition a whole substring (matching a specified regular expression) can be read, and moreover, every transition to a non-accepting state involves a single operation of opening or closing a variable. Those graphs are similar to the extended automata obtained by the known *state-removal* technique, that is used to convert an automaton into a regular expression [34]. Recall that throughout this paper we fix the alphabet Σ for the input string language.

A *variable-stack transition graph*, or *vstk-graph* for short, is a tuple $G = (Q, q_0, q_f, \delta)$ defined similarly to a vstk-automaton, except that now δ consists of edges of three forms: $(q, \gamma, x\vdash, q')$, $(q, \gamma, \dashv, q')$ and (q, γ, q_f); here, $q, q' \in Q$, γ is a regular expression over Σ, and $x \in \mathsf{SVars}$. We require the accepting state q_f to have only incoming transitions. As usual, $\mathsf{SVars}(G)$ denotes the set of variables that occur in G. A *configuration* $c = (q, \vec{v}, Y, i)$ is defined exactly as in the case of a vstk-automaton, but the definition of a run changes: a run ρ of G on a string $\mathbf{s}$ is a sequence $c_0, \ldots, c_m$ of configurations, such that for all $j = 0, \ldots, m-1$, the configurations $c_j = (q_j, \vec{v}_j, Y_j, i_j)$ and $c_{j+1} = (q_{j+1}, \vec{v}_{j+1}, Y_{j+1}, i_{j+1})$ satisfy the following. First, $i_j \leq i_{j+1}$. Second, δ contains a tuple $(q, \gamma, x\vdash, q')$ or a tuple $(q, \gamma, \dashv, q')$, such that $q = q_j$, the string $\mathbf{s}_{[i_j, i_{j+1}\rangle}$ is in $\mathcal{L}(\gamma)$, and $q' = q_{j+1}$; moreover, in the case of $x\vdash$ we have $x \in Y_j$, $\vec{v}_{j+1} = \vec{v}_j \cdot x$ and $Y_{j+1} = Y_j \setminus \{x\}$; and in the case of $\dashv$ we have $\vec{v}_j = \vec{v}_{j+1} \cdot x$ and $Y_{j+1} = Y_j$. The definition of an *accepting configuration* is similar to that for vstk-automata. Moreover, the definitions of $\mathsf{ARuns}(G, \mathbf{s})$ and $[\![G]\!]$ are similar to those of $\mathsf{ARuns}(A, \mathbf{s})$ and $[\![A]\!]$ in the case of a vstk-automaton A.

A vstk-graph $G = (Q, q_0, q_f, \delta)$ is a *vstk-path* if we can write Q as $\{q_0, q_1, \ldots, q_k = q_f\}$ where δ contains exactly k edges: from q_0 to q_1, from q_1 to q_2, and so on, until q_k. A vstk-path is *consistent* if the variables open and close in a balanced manner (which we define in the natural way like grammatical parentheses). We say that G is a *vstk-path union* if G is the union of consistent vstk-paths, such that: *(1)* every two vstk-paths have the same set of variables, namely $\mathsf{SVars}(G)$, and *(2)* every two vstk-paths share precisely the states q_0 and q_f, as illustrated in Figure 4 (where we omit the opening and closing of variables).

Similarly to the vstk case, we define a *vset-graph* to be a variation of a vset-automaton. In particular, $\mathsf{ARuns}(G, \mathbf{s})$ and $[\![G]\!]$ are now defined when G is a vset-graph. Also similarly we define a *vset-path*, a *consistent vset-path* (where parenthetical balance is not required, but every variable needs to be opened and later closed exactly once), and a *vset-path union*.

We use $\mathsf{PU}_{\mathsf{stk}}$ and $\mathsf{PU}_{\mathsf{set}}$ to denote the class of vstk-path unions and the class of vset-path unions, respectively.

4.1.2 Relative Expressive Power

We can now give some results on the (relative) expressive power of the regular spanners. A key lemma is the following.

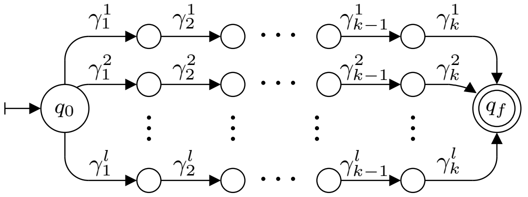


Figure 4: An illustration of a vstk-path union or a vset-path union

LEMMA 4.1. *The following hold.*

1. *Every hierarchical regular spanner is definable by a vstk-path union and vice versa; that is,* $[\![\mathsf{VA}_{\mathsf{stk}}]\!] = [\![\mathsf{PU}_{\mathsf{stk}}]\!]$.
2. *Every regular spanner is definable by a vset-path union and vice versa; that is,* $[\![\mathsf{VA}_{\mathsf{set}}]\!] = [\![\mathsf{PU}_{\mathsf{set}}]\!]$.

In the proof of Lemma 4.1, translating a vstk-path union into a vstk-automaton (resp., a vset-path union into a vset-automaton) is fairly straightforward. The translation of a vstk-automaton to a vstk-path union entails two main steps. (And similar steps are taken in the vset case.) First, the vstk-automaton is converted into a vstk-graph by an adaptation of the well known *state-removal* procedure [34] for translating an automaton into a regular expression. Second, the vstk-graph is converted into a vstk-path union through the observation that only a finite number of paths in the vstk-graph are of relevance.

Our first theorem states that the spanners definable by regex formulas are precisely the hierarchical regular ones.

THEOREM 4.2. *A spanner is hierarchical regular if and only if it is definable by a regex formula; that is,* $[\![\mathsf{VA}_{\mathsf{stk}}]\!] = [\![\mathsf{RGX}]\!]$.

The proof is as follows. We convert a regex formula into a vstk-automaton by an adaptation of the standard construction by Thompson (see, e.g., [34]), namely, incremental construction of an automaton from a regular expression through a bottom-up traversal of the parse of a regular expression. The other direction is an immediate consequence of Lemma 4.1, since the conversion of a vstk-path union into a regex formula is straightforward.

The next theorem states that the hierarchical regular spanners are precisely the spanners that are both regular and hierarchical. Again, the proof uses Lemma 4.1.

THEOREM 4.3. *A spanner is hierarchical regular if and only if it is both regular and hierarchical; that is,* $[\![\mathsf{VA}_{\mathsf{stk}}]\!] = [\![\mathsf{VA}_{\mathsf{set}}]\!] \cap \mathbf{HS}$.

The following theorem states that the union, projection and natural-join operators *do not* increase the expressive power of vset-automata.

THEOREM 4.4. *The class of regular spanners is closed under union, projection and natural join; that is,* $[\![\mathsf{VA}_{\mathsf{set}}^{\{\cup,\pi,\bowtie\}}]\!] = [\![\mathsf{VA}_{\mathsf{set}}]\!]$.

Our proof of Theorem 4.4 is by separately considering each of the operators union, projection, and natural join, and showing closure of $\mathsf{VA}_{\mathsf{set}}$ thereunder. While the first two closures are easy to prove, showing closure under natural join involves subtleties. The expected approach is similar to intersecting two NFAs: a vset-automaton for $A_1 \bowtie A_2$ runs on A_1 and A_2 in parallel; when a variable x is common to both automata, the two parallel runs must open and close x together (as x must be the same span in both runs in taking the join). This approach, however, fails, for a subtle reason. As an example, A_1 and A_2 of Figure 5 are such that

$[\![A_1]\!] = [\![A_2]\!] = [\![A_1 \bowtie A_2]\!]$. However, our construction for A_1 and A_2 will result in the empty spanner, since A_1 requires x to open *before* y (with an epsilon transition in between), and A_2 requires x to open *after* y. We solve this problem by converting A_1 and A_2 into a *normalized form* where common tuples necessarily correspond to "similar" runs (and again we are using Lemma 4.1 for that).

Finally, the next theorem implies that to express all regular spanners, it suffices to enrich the vstk-automata with union, projection and join. Our proof shows how to simulate a given vset-automaton by composing vstk-automata using the three operators.

THEOREM 4.5. $[\![\mathsf{VA}_{\mathsf{stk}}^{\{\cup,\pi,\bowtie\}}]\!] = [\![\mathsf{VA}_{\mathsf{set}}^{\{\cup,\pi,\bowtie\}}]\!] = [\![\mathsf{VA}_{\mathsf{set}}]\!]$.

4.1.3 Simulation of String Relations

Let R be a k-ary string relation, and let $\mathcal{C}$ be a class of spanners. We say that R is *selectable by* $\mathcal{C}$ if for every spanner $P \in \mathcal{C}$ and sequence $\vec{x} = x_1, \dots, x_k$ of variables in $\mathsf{SVars}(P)$, the spanner $\varsigma^{R}_{\vec{x}} P$ is also in $\mathcal{C}$. Let $\vec{x} = x_1, \dots, x_k$ be a sequence of span variables, and let $X = \{x_1, \dots, x_k\}$. The *R-restricted universal spanner over $\vec{x}$*, denoted $\Upsilon^{R}_{\vec{x}}$, is the spanner $\varsigma^{R}_{\vec{x}} \Upsilon_X$. (Recall that Υ_X is the universal spanner over X.) The following (straightforward) proposition states that under some assumptions (that hold in all the spanner classes of our interest), selectability of R is equivalent to the ability to define the R-restricted universal spanners. We will later use this proposition as a tool to decide whether or not a relation R is selectable by a class of spanners at hand.

PROPOSITION 4.6. *Let R be a string relation, and let $\mathcal{C}$ be a class of spanners. Assume that $\mathcal{C}$ contains all the universal spanners, and that $\mathcal{C}$ is closed under natural join. R is selectable by $\mathcal{C}$ if and only if $\Upsilon^{R}_{\vec{x}} \in \mathcal{C}$ for all $\vec{x} \in \mathsf{SVars}^k$.*

Let REC_k be as defined in Section 2.1. It is well known (see [8, 20]) that a k-ary string relation R is in REC_k if and only if it is a finite union of Cartesian products $L_1 \times \cdots \times L_k$, where each L_i is a regular language over Σ. That, combined with Proposition 4.6, easily implies that every recognizable relation is selectable by the regular spanners. Interestingly, the other direction is also true.

THEOREM 4.7. *A string relation is selectable by the regular spanners if and only if it is recognizable. That is,* REC *is precisely the class of string relations selectable by* $[\![\mathsf{VA}_{\mathsf{set}}]\!]$.

4.2 Core Spanners

As the core of AQL we identify the algebra $\mathsf{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}$. Henceforth, we call a spanner in $[\![\mathsf{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}]\!]$ a *core spanner*. A consequence of Theorems 4.2 and 4.5 is that the algebra $\mathsf{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}$ has the same expressive power as $\mathsf{VA}_{\mathsf{stk}}^{\{\cup,\pi,\bowtie,\varsigma^=\}}$ and $\mathsf{VA}_{\mathsf{set}}^{\{\cup,\pi,\bowtie,\varsigma^=\}}$. Therefore, the core spanners are obtained from the regular spanners by extending the algebra with the selection operator $\varsigma^=$.

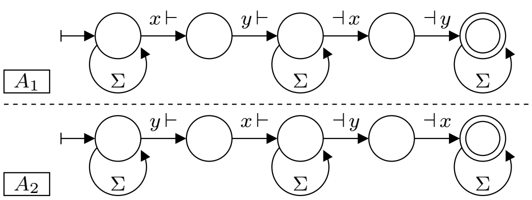


Figure 5: Two vset-automata with equal spanners

The following lemma is a key tool for reasoning about the expressiveness of core spanners. This lemma, which we call the *core-simplification lemma*, states that every core spanner can be defined by a very simple expression: a single vset-automaton, on top of which we apply string-equality selections, and finally a single projection. The proofs of the inexpressibility results we later give for core spanners are inherently based on this result.

LEMMA 4.8 (CORE-SIMPLIFICATION LEMMA). *Every core spanner is definable by an expression of the form $\pi_V SA$, where A is a vset-automaton, $V \subseteq \mathsf{SVars}(A)$, and S is a sequence of selections $\varsigma^{=}_{x,y}$ for $x, y \in \mathsf{SVars}(A)$.*

Next, we discuss selectable relations. Observe that string equality, which is obviously selectable by the core spanners, is not selectable by the regular spanners, because string equality is not in REC (and because of Theorem 4.7). Another way of seeing that is as follows: if string equality were selectable by the regular spanners, then a Boolean regular spanner (which can be represented as an NFA) could recognize the non-regular language $\{\mathbf{s}\cdot\mathbf{s} \mid \mathbf{s} \in \Sigma^*\}$ by $\pi_\emptyset \varsigma^{=}_{x,y}(x\{\Sigma^*\} \cdot y\{\Sigma^*\})$.

Let $\mathbf{s}$ and $\mathbf{t}$ be two strings. By $\mathbf{s} \sqsubseteq \mathbf{t}$ we denote that $\mathbf{s}$ is a (consecutive) substring of $\mathbf{t}$ (i.e, $\mathbf{s}$ is equal to some $\mathbf{t}_{[i,j\rangle}$). By $\mathbf{s} \sqsubseteq_{\mathsf{prf}} \mathbf{t}$ we denote that $\mathbf{s}$ is a prefix of $\mathbf{t}$ (i.e, $\mathbf{s}$ is equal to some $\mathbf{t}_{[1,j\rangle}$). By $\mathbf{s} \sqsubseteq_{\mathsf{sfx}} \mathbf{t}$ we denote that $\mathbf{s}$ is a suffix of $\mathbf{t}$ (i.e, $\mathbf{s}$ is equal to some $\mathbf{t}_{[i,|\mathbf{t}|+1\rangle}$).

Next, we will use Proposition 4.6 to show that the binary substring relation $\sqsubseteq$ is selectable by the core spanners. Due to Proposition 4.6, it suffices to show that the spanner $\Upsilon^{\sqsubseteq}_{x,y}$ is definable in $[\![\mathsf{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}]\!]$. Let $\gamma(x', y)$ be the spanner that captures the property that x' is a sub-span of y. We can define $\gamma(x', y)$ by $\Sigma^* \cdot y\{\Sigma^* \cdot x'\{\Sigma^*\} \cdot \Sigma^*\} \cdot \Sigma^*$. Then $\Upsilon^{\sqsubseteq}_{x,y}$ is defined by

$$\pi_{\{x,y\}} \varsigma^{=}_{x,x'} \left(\Upsilon_{\{x,x',y\}} \bowtie \gamma(x', y)\right) .$$

Similar constructions show that the relations $\sqsubseteq_{\mathsf{prf}}$ and $\sqsubseteq_{\mathsf{sfx}}$ are also selectable by the core spanners. We record this as a proposition, for later use. We also include in the proposition the fact that every relation in REC is also selectable by the core spanners; the proof is by the same argument that precedes Theorem 4.7.

PROPOSITION 4.9. *Every string relation in* REC*, as well as each of the string relations $\sqsubseteq$, $\sqsubseteq_{\mathsf{prf}}$ and $\sqsubseteq_{\mathsf{sfx}}$, is selectable by the core spanners.*

The next theorem shows that the classes of regular and rational relations are incomparable with the class of relations selectable by the core spanners.

THEOREM 4.10. *There is a string relation that is selectable by the core spanners but is non-rational (and hence nonregular), and there is a regular (and hence rational) relation that is not selectable by the core spanners.*

The existence of a regular relation that is not selectable by the core spanners is due to the following theorem.

THEOREM 4.11. *Assume that the alphabet Σ contains at least two symbols. The string relation $\{(\mathbf{s}, \mathbf{t}) \mid |\mathbf{s}| = |\mathbf{t}|\}$ is not selectable by the core spanners.*

Theorem 4.11 is a fairly direct consequence of the following theorem.

THEOREM 4.12. *The language $\{0^m 1^m \mid m \in \mathbb{N}\}$ is not recognizable by any Boolean core spanner.*

5. DIFFERENCE

In this section, we discuss the difference operator. Let P_1 and P_2 be spanners that are union compatible (that is, $\mathsf{SVars}(P_1) = \mathsf{SVars}(P_2)$). The difference $P_1 \setminus P_2$ is defined as follows. First, $\mathsf{SVars}(P_1 \setminus P_2) = \mathsf{SVars}(P_1)$. Second, if $\mathbf{s}$ is a string, then $(P_1 \setminus P_2)(\mathbf{s}) = P_1(\mathbf{s}) \setminus P_2(\mathbf{s})$.

The result with the most involved proof in this section states that core spanners are *not* closed under difference. Recall that the core spanners are those spanners that are expressible in $\mathsf{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}$. One may be tempted to think that non-closure of core spanners under difference should be trivial to prove due to some monotonicity properties, as in the case of ordinary relational algebra. But this is not the case, because our algebra does not involve ordinary relations, but rather spanners; and the primitive representation of spanners (e.g., regex formulas or vset-automata) can simulate non-monotonic behavior (e.g., regular expressions are closed under complement). In fact, we later show that core spanners can simulate string relations of a non-monotonic flavor. Moreover, *regular* (but not *core*) spanners are actually closed under difference.

THEOREM 5.1. *Regular spanners are closed under difference; that is,* $[\![\mathsf{VA}_{\mathsf{set}}^{\{\setminus\}}]\!] = [\![\mathsf{VA}_{\mathsf{set}}]\!]$.

In an attempt to prove that core spanners are not closed under difference (or, equivalently, complement), we tried to prove that the language $\{\mathbf{s}\#\mathbf{t} \mid \mathbf{s} \neq \mathbf{t}\}$, where $\mathbf{s}$ and $\mathbf{t}$ are over the alphabet $\{0,1\}$, and $\#$ is a new symbol, is not recognizable by any Boolean core spanner. After multiple failing attempts, we were surprised to discover that our candidate language L is a wrong candidate, since it actually *is* recognizable by a Boolean core spanner, for the following reason.

PROPOSITION 5.2. *The binary string relation $\neq$ is selectable by the core spanners.*

We remark that a proof similar to that of Proposition 5.2 shows that the string relations $\not\sqsubseteq_{\mathsf{prf}}$ and $\not\sqsubseteq_{\mathsf{sfx}}$ are also selectable by the core spanners. Eventually, we were able to prove non-closure of the core spanners under difference through the (complement of) the substring relation.

THEOREM 5.3. *Assume that the alphabet Σ contains at least two symbols. The string relation $\not\sqsubseteq$ is not selectable by the core spanners.*

Building on the core-simplification lemma (Lemma 4.8) and on Proposition 4.6, our proof of Theorem 5.3 obtains a contradiction by assuming that an expression E given by $\pi_V SA$, as in Lemma 4.8, is such that $[\![E]\!] = \Upsilon^{\not\sqsubseteq}_{x,y}$.

Recall from Proposition 4.9 that the string relation $\sqsubseteq$ is selectable by the core spanners. Theorem 5.3, on the other hand, states that $\not\sqsubseteq$ is not selectable by the core spanners. By combining these two we get the following.

THEOREM 5.4. *Assume that the alphabet Σ contains at least two symbols. Core spanners are* not *closed under difference; that is,* $[\![\mathsf{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}]\!] \subsetneq [\![\mathsf{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=,\setminus\}}]\!]$.

Theorems 5.1 and 5.4 show an interesting contrast between regular and core spanners with respect to difference.

6. SPANNERS VS. OTHER FORMALISMS

We now discuss the relationship between (core and regular) spanners and two related formalisms in the literature.

6.1 Extended Regular Expressions

We first relate core spanners to *extended regular expressions* [1, 12, 14, 23] (xregex for short), which extend the classic regular expressions with *backreferences* (a.k.a. *variable references*) that specify repetitions of a previously matched substring. Their expressive power goes strictly beyond the class of regular languages and, due to their usefulness in practice, most modern regular expression matching engines actually support extended regular expressions [24]. From a theoretical perspective, the extended regular expressions were formalized by Aho [1], and investigated with respect to the complexity of their membership problem [1], their expressiveness and closure properties [12–14], and their conciseness and decidability [23], among other properties.

Syntactically, an xregex can be viewed as a regex formula, but with two major differences. First, there is no restriction on the number of bindings of a variable to a span. Second, in addition to the variable-binding expressions $x\{\gamma\}$ an xregex also allows variable *backreferences* of the form $\&x$. For example, if δ_1 is $x\{(0 \vee 1)^*\} \cdot \&x$, and δ_2 is $x\{(0 \vee 1)^*\} \cdot \&x \cdot x\{(0 \vee 1)^*\} \cdot \&x$, then δ_1 and δ_2 are xregexes. An xregex is interpreted from left to right as follows when parsing an input string $\mathbf{s}$ (cf., e.g., [12, 23]). As before, a binding subexpression $x\{\gamma\}$ matches a substring if γ matches the substring, in which case x is bound to the corresponding span. A backreference $\&x$ matches a substring $\mathbf{s}'$ if $\mathbf{s}' = \mathbf{s}_{[i,j\rangle}$ with $[i,j\rangle$ the span previously bound to x. If x has been bound multiple times, then the last binding prior to the backreference is taken; and if x has not been bound before, $\&x$ matches the empty string. As an example, the above xregex δ_1 matches precisely the strings $\mathbf{ss}$ with $\mathbf{s} \in \{0,1\}^*$, and δ_2 matches precisely the strings $\mathbf{sss}'\mathbf{s}'$ with $\mathbf{s}, \mathbf{s}' \in \{0,1\}^*$. Observe that neither of these languages is regular.

The evaluation of an xregex over a string is not (naturally) mapped to an $\mathbf{s}$-tuple, since a variable can be assigned multiple spans. Therefore, we restrict our discussion to the comparison of xregexes with *Boolean* core spanners. An important part of the expressive power of xregexes stems from the fact that both variable binders and backreferences can occur under the scope of a Kleene star (or plus). For example, $(x\{(0 \vee 1)^*\} \cdot \&x)^+$ matches all strings $\mathbf{s_1 s_1} \cdots \mathbf{s_n s_n}$ with $n \geq 1$ and every $\mathbf{s_i} \in \{0,1\}^*$. Moreover,

$$1^+ \cdot x\{0^*\} \cdot (1^+ \cdot \&x)^* \cdot 1^+$$

matches all strings $\mathbf{s}_1\mathbf{t}\mathbf{s}_2\mathbf{t} \cdots \mathbf{s}_{n-1}\mathbf{t}\mathbf{s}_n$, where $\mathbf{t} \in 0^*$ and every $\mathbf{s}_i$ is in 1^+. In other words, it accepts the language of strings over $\{0,1\}^*$ that start and end with 1, and where all maximal chunks of consecutive 0's are of equal length. We refer to this language as the *uniform*-0-*chunk* language. As the following theorem states, this language is beyond the expressive power of core spanners.

THEOREM 6.1. *The uniform-0-chunk language is recognizable by an xregex but is not recognizable by any Boolean core spanner.*

It is currently still open whether every language recognized by a Boolean core spanner can also be recognized by an xregex. We do note the following. Consider a core spanner represented by $\pi_Y SA$, as in the core-simplification lemma (Lemma 4.8). If the variables of the vset-automaton A cover disjoint spans, then it is easy to prove that such a simulating xregex must exist. To illustrate, consider the regex formula $\gamma := x\,\{\gamma_1\} \cdot \gamma_2 \cdot y\{\gamma_3\}$, where x and y are variables, and γ_1, γ_2, and γ_3 are regular expressions. Then the core spanner $\pi_\emptyset \varsigma^=_{x,y}(\gamma)$ is specified by the xregex $x\{\delta\} \cdot \gamma_2 \cdot \&x$, where δ is the regular expression that recognizes the intersection of the regular expressions γ_1 and γ_3. The problem in finding an xregex that corresponds to a Boolean core spanner arises when the variables in the core spanner have overlapping spans.

6.2 CRPQs on Marked Paths

Regular expressions have been extensively used and studied in database theory as a means to express reachability queries in semistructured and graph databases since the late 1980s. Arguably, the simplest form of such queries are the *regular path query* (*RPQ* for short) on directed graphs with labeled edges [16, 17]. RPQs search for the existence of a path, such that the word formed by the edge labels belongs to a specified regular language. A *conjunctive regular path queries* (*CRPQ* for short) applies conjunction and existential quantification (over nodes) to RPQs; this concept has been the subject of much investigation [10, 11, 16, 19, 21].

Superficially speaking, spanners and CRPQs are inherently different concepts: spanners operate on *strings* while CRPQs operate on *graphs* (directed, edge-labeled graphs); and the variables in the spanner world represent *spans*, while those in the CRPQ world represent *nodes*. However, we can adjust CRPQs to represent spanners, as follows.

In terms of the data model, a string can be viewed as a special case of a graph, namely a simple path. Formally, given a string $\mathbf{s} = \sigma_1 \cdots \sigma_n$, we denote by $p(\mathbf{s})$ the simple path $1 \to 2 \to \cdots \to n+1$ (with the natural numbers $1, \dots, n+1$ as nodes), where for $i = 1, \dots, n$ the label of the edge $i \to i+1$ is σ_i. Now, the span $[i, j\rangle$ of $\mathbf{s}$ can be naturally represented by the pair i, j of nodes from $p(\mathbf{s})$. A CRPQ Q is evaluated over $p(\mathbf{s})$ by means of assignments α from Q's variables to the node set $\{1, \dots, n+1\}$. Restricted to the simple paths $p(\mathbf{s})$, casting a CRPQ as a spanner representation entails the following.

- The node variables of a CRPQ are set to be of two kinds: $x^{\vdash}$, where $x \in \mathsf{SVars}$, represents the left border of a span, and $x^{\dashv}$ represents the right border of the span. Hence, a span variable x is represented by $[x^{\vdash}, x^{\dashv}\rangle$.
- The valid assignments α are now required to be *consistent*: $\alpha(x^{\vdash}) \leq \alpha(x^{\dashv})$ for all relevant $x \in \mathsf{SVars}$.

It is not difficult to see that in our adjustment so far, a CRPQ Q can represent only spanners that are monotonic w.r.t. substrings: if $\mathbf{s} \sqsubseteq \mathbf{t}$, then the assignments for Q on $p(\mathbf{s})$ are, up to needed realignment, among the assignments for Q on $p(\mathbf{t})$. The reason is that CRPQs cannot recognize the endpoints of the input path. To go beyond monotonic spanners, we need to make those endpoints recognizable. Interestingly, it is not clear how to do so without significantly complicating the model. The cleanest way we found is to extend $p(\mathbf{s})$ with the two loops $0 \to 0$ and $(n+1) \to (n+1)$, labeled with new labels $\rhd$ and $\lhd$ (not in the alphabet Σ), respectively. We call the resulting graph a *marked* path. As an example, the marked path for $\mathbf{s} = \texttt{Aba}$ is the following.

$$\overset{\rhd}{\circlearrowleft}\, 1 \xrightarrow{\texttt{A}} 2 \xrightarrow{\texttt{b}} 3 \xrightarrow{\texttt{a}} 4\, \overset{\lhd}{\circlearrowright}$$

With this adjustment, we can show that every regular spanner can be simulated by a union of CRPQs over marked paths. Quite interestingly, we can also show the reverse direction: every union of CRPQs over marked paths simulates some regular spanner. So within our adjustment, unions of CRPQs over marked paths capture precisely the regular spanners. A formal, detailed discussion will appear in the extended version of this paper.

7. SUMMARY AND DISCUSSION

We introduced the concept of a spanner, and investigated three primitive spanner representations: regex formulas, vstk-automata and vset-automata. As we showed, the classes of regex formulas and vstk-automata have the same expressive power, and vset-automata (defining the regular spanners) have the same expressive power as the closure of regex formulas under the relational operators union, natural join and projection. By adding the string-equality operator, one gets the core spanners. We gave some basic results on core spanners, like the core-simplification lemma. We discussed selectable string relations, and showed, among other things, that REC is precisely the class of relations selectable by the regular spanners. We showed that regular spanners are closed under difference, but core spanners are not (which we proved using the core-simplification lemma). Finally, we discussed the connection between core spanners and xregexes, and showed a tight connection between regular spanners and CRPQs.

This work is our first step in embarking on the investigation of spanners. Indeed, many aspects remain to be considered, and many problems remain to be solved. One major aspect is that of complexity. For example, what is the complexity of the translations among spanner representations that were applied in this paper? What is the (data and combined) complexity that query evaluation entails in each representation? Regarding the difference operator, an intriguing question is whether we can find a simple form, in the spirit of the core-simplification lemma, when adding difference to the representation of core spanners (i.e., the class $\mathsf{VA}_{\mathsf{set}}^{\{\cup,\pi,\bowtie,\varsigma^{=},\setminus\}}$); as illustrated here, such a result would be highly useful for reasoning about the expressive power of that class. As another open problem, we repeat the one we mentioned in Section 6: can extended regular expressions express every Boolean core spanner? We conclude by discussing the major issue of *conflict resolvers*.

Conflict Resolvers

Resolution of conflicting tuples has an important role in the practice of rule-based information extraction [15]. As a simple example, on the string `John_Fitzgerald_Kennedy`, one component of an extraction program may identify the span of `John_Fitzgerald` as that of a person name, another may do so for `Fitzgerald_Kennedy`, and a third may do so for `John_Fitzgerald_Kennedy`. As only one of these is the mentioning of a person name, a cleanup resolution filters out two of the three annotations. In CPSL [3], for instance, this resolution takes place implicitly at every stage (cascade). A significant differentiator of AQL is that it exposes conflict resolution as an explicit relational operator, similarly to selection, and moreover, supports multiple resolution semantics. Yet, this operator is different from a standard selection, as it is not applied in a tuple-by-tuple basis, but rather in an aggregate manner. In this section, we discuss the semantics of such an operator, which we shall investigate more deeply in a future paper.

How should a conflict resolver be defined? At the high level, it is a unary spanner operator cr_x parameterized by a variable $x \in \mathsf{SVars}$. This operator takes as input a spanner P with $x \in \mathsf{SVars}(P)$ and outputs a spanner P', such that $\mathsf{SVars}(P') = \mathsf{SVars}(P)$ and $P' \subseteq P$ (i.e., for all strings $\mathbf{s} \in \Sigma^*$ we have $P'(\mathbf{s}) \subseteq P(\mathbf{s})$). The operator cr_x filters out $\mathbf{s}$-tuples whenever conflicts are involved in the spans assigned to x by different $\mathbf{s}$-tuples. The output is a conflict-free subset of $P(\mathbf{s})$.

For concreteness, let us focus on the simple (yet practical) case where cr_x is specified by a *conflict condition* stating when two spans $\mu_1(x)$ and $\mu_2(x)$ are in conflict, and a *resolution rule* stating which of $\mu_1(x)$ and $\mu_2(x)$ prevails. Still, how should resolution be defined for a given conflict condition and resolution rule? Eliminating tuples sequentially does not seem to be the right way, since the result may be sensitive to the order in which conflicts are considered. For example, a standard conflict condition says that $\mu_1(x)$ and $\mu_2(x)$ overlap but are not equal, and the resolution rule is *left-to-right winner*: $\mu_1(x)$ prevails over $\mu_2(x)$ if $\mu_1(x)$ starts before $\mu_2(x)$; and if they start at the same position, then $\mu_1(x)$ is longer.

Now, take the three spans $\mu_1(x) = [1,3\rangle$, $\mu_2(x) = [2,4\rangle$, and $\mu_3(x) = [3,5\rangle$. Resolving conflicts from left to right gives a different result than the right-to-left resolution. Indeed, from left to right, we take $[1,3\rangle$, discard $[2,4\rangle$, then keep $[3,5\rangle$; and from right to left we take $[3,5\rangle$, then discard $[3,5\rangle$ in favor of the span $[2,4\rangle$, then discard $[2,4\rangle$ in favor of the span $[1,3\rangle$.

In accommodating the above, an approach we are exploring is adopting the concept of *inconsistent databases* [4] to our setting. Specifically, we can think of our span relation as an inconsistent relation, and every maximal non-conflicting subset of tuples as a *possible world*. But the traditional theory of inconsistent databases does not allow for different priority among tuples, and we treat such priorities as first-class citizens (and specify them with our resolution rule). Nevertheless, recent work of Staworko et al. [43] proposes and studies various concepts of inconsistent databases with *prioritized repairing*, and we are currently studying the application of prioritized repairing to conflict resolution within spanners.

Acknowledgments

We are grateful to Pablo Barceló, Kenneth Clarkson, and Leonid Libkin for helpful discussions. We also thank the SystemT group their intensive work in establishing the system, and for useful input.

8. REFERENCES

[1] A. V. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 255–300. North Holland, 1990.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, Nov. 1983.

[3] D. E. Appelt and B. Onyshkevych. The common pattern specification language. In *Proceedings of the TIPSTER Text Program: Phase III*, pages 23–30, Baltimore, Maryland, USA, 1998. Association for Computational Linguistics.

[4] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79. ACM, 1999.

[5] P. Barceló, D. Figueira, and L. Libkin. Graph logics with rational relations and the generalized intersection problem. In *LICS*, pages 115–124. IEEE, 2012.

[6] P. Barceló, J. L. Reutter, and L. Libkin. Parameterized regular expressions and their languages. *Theor. Comput. Sci.*, 474:21–45, 2013.

[7] M. Benedikt, L. Libkin, T. Schwentick, and L. Segoufin. Definable relations and first-order query languages over strings. *J. ACM*, 50(5):694–751, 2003.

[8] J. Berstel. *Transductions and Context-Free Languages*. Teubner Studienbücher, Stuttgart, 1979.

[9] A. J. Bonner and G. Mecca. Sequences, datalog, and transducers. *J. Comput. Syst. Sci.*, 57(3):234–259, 1998.

[10] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR 2000*, pages 176–185, 2000.

[11] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. View-based query processing and constraint satisfaction. In *LICS*, pages 361–371, 2000.

[12] C. Câmpeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14(6):1007–1018, 2003.

[13] C. Câmpeanu and N. Santean. On the intersection of regex languages with regular languages. *Theor. Comput. Sci.*, 410(24-25):2336–2344, 2009.

[14] B. Carle and P. Narendran. On extended regular expressions. In *LATA 2009*, volume 5457 of *Lecture Notes in Computer Science*, pages 279–289, 2009.

[15] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, and S. Vaithyanathan. SystemT: An algebraic approach to declarative information extraction. In *ACL*, pages 128–137. The Association for Computer Linguistics, 2010.

[16] M. P. Consens and A. O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *PODS*, pages 404–416. ACM, 1990.

[17] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *SIGMOD Conference*, pages 323–330. ACM, 1987.

[18] H. Cunningham. Gate, a general architecture for text engineering. *Computers and the Humanities*, 36(2):223–254, 2002.

[19] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *DBPL*, pages 21–39, 2001.

[20] C. C. Elgot and J. E. Mezei. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, 9:47–68, 1965.

[21] D. Florescu, A. Y. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *PODS*, pages 139–148, 1998.

[22] D. Freitag. Toward general-purpose learning for information extraction. In *COLING-ACL*, pages 404–408, 1998.

[23] D. D. Freydenberger. Extended regular expressions: Succinctness and decidability. In *STACS 2011*, volume 9 of *LIPIcs*, pages 507–518. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[24] J. Friedl. *Mastering Regular Expressions*. O'Reilly Media, 2006.

[25] S. Ginsburg and X. S. Wang. Regular sequence operations and their use in database queries. *J. Comput. Syst. Sci.*, 56(1):1–26, 1998.

[26] G. Grahne, M. Nykänen, and E. Ukkonen. Reasoning about strings in databases. *J. Comput. Syst. Sci.*, 59(1):116–162, 1999.

[27] R. Grishman and B. Sundheim. Message understanding conference-6: A brief history. In *COLING*, pages 466–471, 1996.

[28] O. Grumberg, O. Kupferman, and S. Sheinvald. Variable automata over infinite alphabets. In A. H. Dediu, H. Fernau, and C. Martín-Vide, editors, *LATA*, volume 6031 of *Lecture Notes in Computer Science*, pages 561–572. Springer, 2010.

[29] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[30] D. E. Knuth. Correction: Semantics of context-free languages. *Mathematical Systems Theory*, 5(1):95–96, 1971.

[31] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. SystemT: a system for declarative information extraction. *SIGMOD Record*, 37(4):7–13, 2008.

[32] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, pages 282–289. Morgan Kaufmann, 2001.

[33] T. R. Leek. Information extraction using hidden markov models. Master's thesis, UC San Diego, 1997.

[34] P. Linz. *An introduction to formal languages and automata*. Jones and Bartlett Publishers, Inc., Sudbury, Mass. [N.A.], third edition, 2001.

[35] B. Liu, L. Chiticariu, V. Chu, H. Jagadish, and F. Reiss. Automatic rule refinement for information extraction. *Proceedings of the VLDB Endowment*, 3(1-2):588–597, 2010.

[36] A. McCallum, D. Freitag, and F. C. N. Pereira. Maximum entropy markov models for information extraction and segmentation. In *ICML*, pages 591–598. Morgan Kaufmann, 2000.

[37] F. Neven and J. V. den Bussche. Expressiveness of structured document query languages based on attribute grammars. *J. ACM*, 49(1):56–100, 2002.

[38] F. Neven and T. Schwentick. Query automata over finite trees. *Theoretical Computer Science*, 275(2):633 – 674, 2002.

[39] M. Nivat. Transduction des langages de Chomsky. *Ann. Inst. Fourier*, 18:339–455, 1968.

[40] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE*, pages 933–942. IEEE, 2008.

[41] E. Riloff. Automatically constructing a dictionary for information extraction tasks. In *AAAI*, pages 811–816. AAAI Press / The MIT Press, 1993.

[42] S. Soderland, D. Fisher, J. Aseltine, and W. G. Lehnert. CRYSTAL: Inducing a conceptual dictionary. In *IJCAI*, pages 1314–1321. Morgan Kaufmann, 1995.

[43] S. Staworko, J. Chomicki, and J. Marcinkowski. Prioritized repairing and consistent query answering in relational databases. *Ann. Math. Artif. Intell.*, 64(2-3):209–246, 2012.