Joining Extractions of Regular Expressions

Dominik D. Freydenberger
Loughborough University
Loughborough, United Kingdom
ddfy@ddfy.de

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

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

ABSTRACT

Regular expressions with capture variables, also known as "regex formulas," extract relations of spans (interval positions) from text. These relations can be further manipulated via the relational Algebra as studied in the context of "document spanners," Fagin et al.'s formal framework for information extraction. We investigate the complexity of querying text by Conjunctive Queries (CQs) and Unions of CQs (UCQs) on top of regex formulas. Such queries have been investigated in prior work on document spanners, but little is known about the (combined) complexity of their evaluation.

We show that the lower bounds (NP-completeness and W[1]-hardness) from the relational world also hold in our setting; in particular, hardness hits already single-character text. Yet, the upper bounds from the relational world do not carry over. Unlike the relational world, acyclic CQs, and even gamma-acyclic CQs, are hard to compute. The source of hardness is that it may be intractable to instantiate the relation defined by a regex formula, simply because it has an exponential number of tuples. Yet, we are able to establish general upper bounds. In particular, UCQs can be evaluated with polynomial delay, provided that every CQ has a bounded number of atoms (while unions and projection can be arbitrary). Furthermore, UCQ evaluation is solvable with FPT (Fixed-Parameter Tractable) delay when the parameter is the size of the UCQ.

1. INTRODUCTION

Information Extraction (IE) conventionally refers to the task of automatically extracting structured information from text. While early work in the area focused largely on military applications [19], this task is nowadays pervasive in a plethora of computational challenges (especially those associated with Big Data), including social media analysis [5], healthcare analysis [39], customer relationship management [3], information retrieval [44], machine log analysis [17], and open-domain Knowledge Base construction [21, 35, 37, 43].

One of the major commercial systems for rule-based IE is IBM's SystemT[1] that exposes an SQL-like declarative language named *AQL* (Annotation Query Language), along with a query plan optimizer [31] and development tooling [26]. Conceptually, AQL supports a collection of "primitive" extractors of relations from text (e.g., tokenizer, dictionary lookup, regex matcher and part-of-speech tagger), along with an algebra for relational manipulation (applied to the relations obtained from the primitive extractors). A similar approach is adopted by Xlog [34], where user-defined functions, playing the role of primitive extractors, are manipulated by non-recursive Datalog. DeepDive [32, 35] exposes a declarative language for rules and features, which are eventually translated into the factors of a statistical model where parameters (weights) are set by machine learning. There, the primitive extractors are defined in a scripting language, and the generated relations are again manipulated by relational rules.

Fagin et al. [12] proposed the framework of *document spanners* (or just *spanners* for short) that captures the relational philosophy of the aforementioned systems. Intuitively, a spanner extracts from a document $\mathbf{s}$ (which is a string over a finite alphabet) a relation over the spans of $\mathbf{s}$. A *span* of $\mathbf{s}$ represents a substring of $\mathbf{s}$ that is identified by the start and end indices. An example of a spanner representation is a *regex formula*: a regular expression with embedded capture variables that are viewed as relational attributes. A *regular* spanner is one that can be expressed in the closure of the regex formulas under relational operators projection, union, and join.

As an example, it is common to apply sentence boundary detection by evaluating a regex formula [38], which we denote as $\alpha_{\mathsf{sen}}[x]$. When evaluating $\alpha_{\mathsf{sen}}[x]$ over a string $\mathbf{s}$ (representing text in natural language), the result is a relation with a single attribute, x, where each tuple consists of the left and right boundaries of a sentence. The following regular spanner detects sentences that contain both an address in Belgium and the substring `police` inside them. It assumes a regex formula $\alpha_{\mathsf{adr}}[y, z]$ that extracts (annotates) spans y that represent addresses (e. g., `Place de la Nation 2, 1000 Bruxelles, Belgium`) with the country being z (e.g., the span of `Belgium`), the regex formulas $\alpha_{\mathsf{blg}}[z]$ and $\alpha_{\mathsf{plc}}[w]$ that extract spans that are tokens with the words `Belgium` and `police`, respectively, and the regex formula $\alpha_{\mathsf{sub}}[y, x]$ that extracts all pairs of spans (x, y) such that y is a subspan of (i.e., has boundaries within) x.

$$\pi_x\big(\alpha_{\mathsf{sen}}[x] \bowtie \alpha_{\mathsf{adr}}[y, z] \bowtie \alpha_{\mathsf{sub}}[y, x] \bowtie \alpha_{\mathsf{blg}}[z] \bowtie \alpha_{\mathsf{plc}}[w] \bowtie \alpha_{\mathsf{sub}}[w, x]\big) \tag{1}$$

For example, $\alpha_{\mathsf{sub}}[y, x]$ can be represented as the regex formula $\mathbf{\Sigma}^* \cdot x\{\mathbf{\Sigma}^* \cdot y\{\mathbf{\Sigma}^*\} \cdot \mathbf{\Sigma}^*\} \cdot \mathbf{\Sigma}^*$ where, as we later explain in detail, $\mathbf{\Sigma}^*$ matches every possible string. A key construct in the framework of Fagin et al. [12] is

[1] SystemT [25] is the IE engine of IBM BigInsights, available as trial at `http://www-03.ibm.com/software/products/en/ibm-biginsights-for-apache-hadoop` as of 06/2017.

the *variable-set automaton* that was proved that capture precisely the expressive power of regular spanners. A *core* spanner is defined similarly to a regular spanner, but it also allows the string-equality selection predicate on spans (an example can be found further down in this section). Later developments of the framework include the exploration of the complexity of static-analysis tasks on core spanners [16], incorporating inconsistency repairing in spanners [13], a logical characterization [15], and a uniform model for relational data and IE [29].

The research on spanners has been initiated with the motivation of exploring development architectures such as SystemT [25] and Xlog [34]. The basic queries in these architectures are Conjunctive Queries (CQs), and more generally Unions of CQs (UCQs), over regex formulas, expressed as SQL (SystemT) or acyclic Datalog (Xlog). Such UCQs also constitue the language underlying more recent theoretical extensions [13, 29]. Nevertheless, very little is known about the computational complexity of evaluating CQs and UCQs over regex formulas. In this paper, we explore this complexity.

To be more precise, in the framework of (regular and core) spanners, CQs and UCQs over regex formulas are defined similarly to the relational-database world, with two differences. First, the input is not a relational database, but rather a string. Second, the atoms are not relational symbols, but rather regex formulas. Therefore, the *relations* on which the query is applied are implicitly defined as those obtained by evaluating each regex formula over the input string. We refer to these queries as *regex CQs* and *regex UCQs*, respectively. As an example, the above query in (1) is a regex CQ. Under *data complexity*, where the query is assumed fixed, query evaluation is always doable in polynomial time. Here we focus on *combined complexity* where both the query and the data (string) are given as input. One might be tempted to claim that the literature on UCQ evaluation to date should draw the complete picture on complexity, by what we refer to as the *canonical relational* evaluation: evaluate each indvidual regex formula on the input string, and compute the UCQ as if it were query on an ordinary relational database. There are, however, several problems with this claim.

The first problem is that lower bounds on UCQ evaluation do not carry over immediately to our setting, since the input is not an arbitrary relational database, but rather a very specific one: every relation is obtained by applying a regex to the (same) input string. But, quite expectedly, we show that the standard lower bound of NP-completeness [7] for Boolean CQs (we well as W[1]-hardness for some parameters) remain in our setting. The more surprising finding is that hardness holds even if the input string is a single character! Yet, a more fundamental problem with the canonical relational approach to evaluation is that it may be infeasible to materialize the relation defined by a regex formula, as the number of tuples in that relation may be exponential in the size of the input. This problem provably increases the complexity, as we show that Boolean regex-CQ evaluation is NP-complete even on acyclic CQs, and even on the more restricted *gamma-acyclic* CQs. In contrast, acyclic CQs (and more generally CQs of *bounded hypertree width* [18]) admit polynomial-time evaluation. Finally, even if we were guaranteed a polynomial bound on the result of each atomic regex formula, it would not necessarily mean that we can actually materialize the corresponding relation in polynomial time.

Yet, in spite of the above daunting complexity we are able to establish some substantial upper bounds. Our upper bounds are based on a central algorithm that we devise in this paper for evaluating a variable-set automaton (*vset-automaton* for short) over a string. More formally, recall that a vset-automaton A represents a spanner, which we denote as $[\![A]\!]$. When evaluating A on a string $\mathbf{s}$, the result is a relation $[\![A]\!](\mathbf{s})$ over the spans of $\mathbf{s}$. The number of tuples in $[\![A]\!](\mathbf{s})$ can be exponential in the size of the input ($\mathbf{s}$ and A). Our algorithm takes as input a string $\mathbf{s}$ and a vset-automaton A (or, more precisely, a *functional* vset-automaton [16]) and enumerates the tuples of $[\![A]\!](\mathbf{s})$ with *polynomial delay* [23]. This is done by a nontrivial reduction to the problem of enumerating all the words of a specific length accepted by an NFA [2]. Our central algorithm implies several upper bounds, which we establish in two approaches: (a) via what we call the *canonical relational evaluation*, and (b) via *compilation to automata*.

The first approach utilizes known algorithms for relational UCQ evaluation, by materializing the relations defined by the regex formulas. For that, we devise an efficient compilation of a regex formula into a vset-automaton, and establish that a regex formula can be evaluated in polynomial total time (and even polynomial delay). In particular, we can efficiently materialize the relations of each atom of a regex UCQ, whenever we have a polynomial bound on the cardinality of this relation. Hence, there is no need for a specialized algorithm for each cardinality guarantee—one algorithm fits all. Consequently, under such cardinality guarantees, canonical relational evaluation is efficient whenever the underlying UCQ is tractable (e.g., each CQ is acyclic).

In the second approach, compilation to automata, we compile the entire regex UCQ into a vset-automaton. Combining our polynomial-delay algorithm with known results [12, 15], we conclude that regex UCQs can be evaluated with Fixed-Parameter Tractable (FPT) delay when the size of the UCQ is the parameter. Moreover, we prove that the compilation is efficient *if* every disjunct (regex CQ) has a bounded number atoms. In particular, we show that every join of a bounded number of vset-automata can be compiled in polynomial time into a single vset-automaton, and every projection and union (with no bounds) over vset-automata can be compiled in polynomial time into a single vset-automaton. Hence, we establish that for every fixed k, the evaluation of regex k-UCQs (where each CQ has at most k atoms) can be performed with polynomial delay.

Finally, we generalize our results to allow regex-UCQ to include string-equality predicates, which are expressions of the form "x and y span the same substring, possibly in different locations". For example, the following regex CQ with a string equality finds sentences that have the address as expressed in our previous example spanner (see (1) above), but they are not necessarily the same sentences (and in particular they may exclude the word `police`).

$$\pi_{x'}\zeta^{=}_{y,y'}\big(Q[w,x,y,z] \bowtie \alpha_{\mathsf{sen}}[x'] \bowtie \alpha_{\mathsf{sub}}[y',x']\big)$$

where $Q[w,x,y,z]$ is the join expression inside the projection of (1) above. As shown in [16], adding an unbounded number of string equalities can make a tractable regex UCQ intractable, even if that regex UCQ is a single regex formula. We prove that now we no longer retain the above FPT delay, as the problem is W[1]-hard when the parameter is the size of the query.

Nevertheless, much of the two evaluation approaches generalize to string equalities. While this generalization is immediate for the first approach, the second faces a challenge—it is *impossible* to compile string equality into a vset-automaton [12]. Yet, we show that with our compilation techniques, one can compile in string equality *for the specific input string* at hand (that is, not statically but rather at runtime). Hence, we conclude that regex k-UCQs with a bounded number of string equalities can be evaluated with polynomial delay.

The rest of the paper is organized as follows. In Section 2 we recall the framework of document spanners, and set the basic notation and terminology. In Section 3 we give our complexity results for regex UCQs. We describe our polynomial-delay algorithm for evaluating vset-automata in Section 4. In Section 5 we generalize our complexity results to UCQs with string equalities. Finally, we conclude in Section 6.

2. BASICS OF DOCUMENT SPANNERS

We begin by recalling essentials of *document spanners* [12, 13, 16], a formal framework for Information Extraction (IE) inspired by declarative IE systems such as Xlog [34] and IBM's SystemT [24, 25, 31].

2.1 Strings, Spans and Spanners

Throughout this paper, we fix a finite alphabet $\mathbf{\Sigma}$. Unless explicitly stated, we assume that $|\mathbf{\Sigma}| \geq 2$. A *string* is a sequence $\sigma_1\sigma_2\cdots\sigma_N$ of symbols from $\mathbf{\Sigma}$. The set of all strings is denoted by $\mathbf{\Sigma}^*$. We use bold (non-italic) letters, such as $\mathbf{s}$ and $\mathbf{t}$, to denote strings, and ϵ to denote the empty string. The length N of a string $\mathbf{s} := \sigma_1\sigma_2\cdots\sigma_N$ is denoted by $|\mathbf{s}|$; and $|\mathbf{s}|_\sigma$ denotes the number of occurrences of a symbol σ in $\mathbf{s}$.

Let $\mathbf{s} := \sigma_1\sigma_2\cdots\sigma_N$ be a string. A *span* of $\mathbf{s}$ represents an interval of characters in $\mathbf{s}$ by indicating the bounding indices. Formally, a span of $\mathbf{s}$ is an expression of the form $[i,j\rangle$ with $1 \leq i \leq j \leq N{+}1$. For a span $[i,j\rangle$ of $\mathbf{s}$, we denote by $\mathbf{s}_{[i,j\rangle}$ the string $\sigma_i\cdots\sigma_{j-1}$. Note that two spans $[i_1,j_1\rangle$ and $[i_2,j_2\rangle$ are *equal* if and only if both $i_1 = i_2$ and $j_1 = j_2$ hold. In particular, $\mathbf{s}_{[i_1,j_1\rangle} = \mathbf{s}_{[i_2,j_2\rangle}$ does not necessarily imply that $[i_1,j_1\rangle = [i_2,j_2\rangle$.

Example 2.1. Our examples assume that $\mathbf{\Sigma}$ consists of the Latin characters and some common additional symbols such as punctuation and commercial at (@). In addition, the symbol ␣ stands for whitespace.

Let $\mathbf{s}$ be the string `chocolate␣cookie`. Then $|\mathbf{s}| = 16$. The strings $\mathbf{s}_{[4,6\rangle}$ and $\mathbf{s}_{[11,13\rangle}$ are both equal (to the string `co`), and yet, $[4,6\rangle \neq [11,13\rangle$. Likewise, we have $\mathbf{s}_{[1,1\rangle} = \mathbf{s}_{[2,2\rangle} = \epsilon$, but $[1,1\rangle \neq [2,2\rangle$. Finally, observe that $\mathbf{s}$ is the same as $\mathbf{s}_{[1,17\rangle}$. ◊

We assume an infinite set Vars of variables, disjoint from $\mathbf{\Sigma}$. For a finite $V \subset \mathsf{Vars}$ and $\mathbf{s} \in \mathbf{\Sigma}^*$, a $(V,\mathbf{s})$-*tuple* is a function μ that maps each variable in V to a span of $\mathbf{s}$. If context allows, we write $\mathbf{s}$-tuple instead of $(V,\mathbf{s})$-tuple. A set of $(V,\mathbf{s})$-tuples is called a $(V,\mathbf{s})$-*relation*. A *spanner* is a function P that is associated with a finite variable set V, denoted $\mathsf{Vars}(P)$, and that maps every string $\mathbf{s} \in \mathbf{\Sigma}^*$ to a $(V,\mathbf{s})$-relation $P(\mathbf{s})$.

A spanner P is *Boolean* if $\mathsf{Vars}(P) = \emptyset$. If P is Boolean, then either $P(\mathbf{s}) = \emptyset$ or $P(\mathbf{s})$ contains only the empty $(\emptyset,\mathbf{s})$-tuple; we interpret these two cases as *false* and *true*, respectively.

2.2 Spanner Representations

This paper uses two models as basic building blocks for spanner representations *regex formulas* and *vset-automata*. These can be understood as extensions of regular expressions and NFAs, respectively, with variables. Both models were introduced by Fagin et al. [12], and following Freydenberger [15] we define the semantics of these models using so-called *ref-words* [33] (short for *reference-words*).

2.2.1 Ref-words

For a finite variable set $V \subset \mathsf{Vars}$, ref-words are defined over the extended alphabet $\mathbf{\Sigma} \cup \mathbf{\Gamma}_V$, where $\mathbf{\Gamma}_V$ consists of two symbols, $x\vdash$ and $\dashv x$, for each variable $x \in V$. We assume that $\mathbf{\Sigma}$ and $\mathbf{\Gamma}_V$ are disjoint. Intuitively, the letters $x\vdash$ and $\dashv x$ represent opening or closing a variable x. Hence, ref-words can be understood as terminal strings that are extended with encodings of variable operations. As we shall see, treating these variable operations as letters allows us to adapt techniques from automata theory.

A ref-word $\mathbf{r} \in (\mathbf{\Sigma} \cup \mathbf{\Gamma}_V)^*$ is *valid for* V if each variable of V is opened and then closed exactly once, or more formally, for each $x \in \mathsf{Vars}(A)$ the string $\mathbf{r}$ has precisely one occurrence of $x\vdash$, precisely one occurrence of $\dashv x$, and the former occurrence takes place before (i.e., to the left of) the latter.

Example 2.2. Let $V := \{x\}$, and define the ref-words $\mathbf{r}_1 := \mathtt{c}\,x\vdash\mathtt{oo}\,\dashv x\,\mathtt{ie}$, $\mathbf{r}_2 := x\vdash\dashv x$, $\mathbf{r}_3 := \dashv x\mathtt{a}\,x\vdash$, and $\mathbf{r}_4 := x\vdash\mathtt{a}\,\dashv x\,x\vdash\mathtt{a}\dashv x$. Then $\mathbf{r}_1$ and $\mathbf{r}_2$ are valid for V, but $\mathbf{r}_3$ and $\mathbf{r}_4$ are not. Note that $\mathbf{r}_1$ and $\mathbf{r}_2$ are not

valid for V' with $V' \supset V$, as all variables of V' must be opened and closed. ◊

If V is clear from the context, we simply say that a ref-word is valid. To connect ref-words to terminal strings and later to spanners, we define the clearing morphism $\mathsf{clr}\colon (\mathbf{\Sigma} \cup \mathbf{\Gamma}_V)^* \to \mathbf{\Sigma}^*$ by $\mathsf{clr}(\sigma) := \sigma$ for $\sigma \in \mathbf{\Sigma}$, and $\mathsf{clr}(a) := \epsilon$ for $a \in \mathbf{\Gamma}_V$. For fixed $\mathbf{\Sigma}$ and V, and any $\mathbf{s} \in \mathbf{\Sigma}^*$, let $\mathsf{Ref}(\mathbf{s})$ be the set of all valid ref-words $\mathbf{r} \in (\mathbf{\Sigma} \cup \mathbf{\Gamma}_V)^*$ with $\mathsf{clr}(\mathbf{r}) = \mathbf{s}$. By definition, every $\mathbf{r} \in \mathsf{Ref}(\mathbf{s})$ has a unique factorization $\mathbf{r} = \mathbf{r}'_x \cdot x\vdash \cdot \mathbf{r}_x \cdot \dashv x \cdot \mathbf{r}''_x$ for each $x \in V$. With these factorizations, we interpret $\mathbf{r}$ as a $(V, \mathbf{s})$-tuple $\mu^{\mathbf{r}}$ by defining $\mu^{\mathbf{r}}(x) := [i, j\rangle$, where $i := |\mathsf{clr}(\mathbf{r}'_x)|+1$ and $j := i+|\mathsf{clr}(\mathbf{r}_x)|$. Intuitively, $\mathsf{clr}(\mathbf{r}_x)$ contains $\mathbf{s}_{\mu^{\mathbf{r}}(x)}$, while $\mathsf{clr}(\mathbf{r}'_x)$ contains the prefix of $\mathbf{s}$ to the left.

An alternative way of understanding $\mu^{\mathbf{r}} = [i, j\rangle$ is that i is chosen such that $x\vdash$ occurs between the positions in $\mathbf{r}$ that are mapped to σ_{i-1} and σ_i, and $\dashv x$ occurs between the positions that are mapped to σ_{j-1} and σ_j (assuming that $\mathbf{s} = \sigma_1 \cdots \sigma_{|\mathbf{s}|}$, and slightly abusing the notation to avoid a special distinction for the non-existing positions σ_0 and $\sigma_{|\mathbf{s}|+1}$).

Example 2.3. Let $\mathbf{s} := \mathtt{cookie}$, and define $\mathbf{r}_1 := \mathtt{c}\, x\vdash \mathtt{oo} \dashv x\, \mathtt{kie}$, and let $\mathbf{r}_2 := \mathtt{cookie}\, x\vdash \dashv x$. Let $V := \{x\}$. Then $\mathbf{r}_1, \mathbf{r}_2 \in \mathsf{Ref}(\mathbf{s})$, with $\mu^{\mathbf{r}_1}(x) := [2, 4\rangle$ and $\mu^{\mathbf{r}_2}(x) := [6, 7\rangle$. ◊

2.2.2 Regex Formulas

A *regex formula (over $\mathbf{\Sigma}$)* is a regular expression that may also include variables (called *capture variables*). Formally, we define the syntax with the recursive rule

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

where $\sigma \in \mathbf{\Sigma}$ and $x \in \mathsf{Vars}$. We add and omit parentheses and concatenation ($\cdot$) symbols freely, as long as the meaning remains clear, and use α^+ as shorthand for $\alpha \cdot \alpha^*$, as well as $\mathbf{\Sigma}$ as shorthand for $\bigvee_{\sigma \in \mathbf{\Sigma}} \sigma$. The set of variables that occur in α is denoted by $\mathsf{Vars}(\alpha)$. The *size* of α, denoted $|\alpha|$, is naturally defined as the number of symbols in α.

We interpret each regex formula α as a generator of a ref-word language $\mathcal{R}(\alpha)$ over the extended alphabet $\mathbf{\Sigma} \cup \mathbf{\Gamma}_{\mathsf{Vars}(\alpha)}$. If α is of the form $x\{\beta\}$, then $\mathcal{R}(\alpha) := x\vdash \mathcal{R}(\beta) \dashv x$. Otherwise, $\mathcal{R}(\alpha)$ is defined like the language $\mathcal{L}(\alpha)$ of a regular expression; for example, $\mathcal{R}(\alpha \cdot \beta) := \mathcal{R}(\alpha) \cdot \mathcal{R}(\beta)$. We let $\mathsf{Ref}(\alpha)$ denote the set of all ref-words in $\mathcal{R}(\alpha)$ that are valid for $\mathsf{Vars}(\alpha)$; and for every string $\mathbf{s} \in \mathbf{\Sigma}^*$, we define $\mathsf{Ref}(\alpha, \mathbf{s}) = \mathcal{R}(\alpha) \cap \mathsf{Ref}(\mathbf{s})$. In other words, $\mathsf{Ref}(\alpha, \mathbf{s})$ contains exactly those valid ref-words from $\mathcal{R}(\alpha)$ that clr maps to $\mathbf{s}$.

Finally, the spanner $[\![\alpha]\!]$ is the one that defines the following $(\mathsf{Vars}(\alpha), \mathbf{s})$-relation for every string $\mathbf{s} \in \mathbf{\Sigma}^*$:

$$[\![\alpha]\!](\mathbf{s}) := \{\mu^{\mathbf{r}} \mid \mathbf{r} \in \mathsf{Ref}(\alpha, \mathbf{s})\}$$

We say that a regex-formula α is *functional* if $\mathcal{R}(\alpha) = \mathsf{Ref}(\alpha)$; that is, every ref-word in $\mathcal{R}(\alpha)$ is valid. Fagin et al. [12] gave an algorithm for testing whether a regex formula is functional. By analyzing the complexity of their construction, we conclude the following.

Theorem 2.4. [12] *Whether a regex formula α with v variables is functional can be tested in $O(|\alpha| \cdot v)$ time.*

Following [12], we assume regex formulas are functional unless explicitly noted (in fact, [12] does not even define $[\![\alpha]\!]$ for non-functional regex formulas α).

Example 2.5. Consider the following regex formula α.

$$\mathbf{\Sigma}^*\big((x\{\mathtt{foo}\}\, \mathbf{\Sigma}^*\, y\{\mathtt{bar}\}) \vee (y\{\mathtt{bar}\}\, \mathbf{\Sigma}^*\, x\{\mathtt{foo}\})\big)\mathbf{\Sigma}^*$$

Then $\mathsf{Vars}(\alpha) = \{x, y\}$. For $\mathbf{s} \in \mathbf{\Sigma}^*$, the $(\mathsf{Vars}(\alpha), \mathbf{s})$-relation $[\![\alpha]\!](\mathbf{s})$ contains every $(\mathsf{Vars}(\alpha), \mathbf{s})$-tuple μ that satisfies $\mathbf{s}_{\mu(x)} = \mathtt{foo}$ and $\mathbf{s}_{\mu(y)} = \mathtt{bar}$. Now, consider the regex formula β defined as follows.

$$\mathbf{\Sigma}^*\, \text{␣}\, x_{\mathsf{mail}}\{x_{\mathsf{user}}\{\gamma\}\, @\, x_{\mathsf{domain}}\{\gamma.\gamma\}\}\, \text{␣}\, \mathbf{\Sigma}^*,$$

where $\gamma := (\mathtt{a} \vee \cdots \vee \mathtt{z})^*$, and, as previously said, ␣ denotes whitespace. This regex formula identifies (simplified) email addresses, where the variable x_{mail} contains the whole address, and x_{user} and x_{domain} the user and domain parts. Both α and β are functional. In contrast, $x\{\mathtt{a}\}\, x\{\mathtt{a}\}$ and $x\{\mathtt{a}\} \vee y\{\mathtt{a}\}$ are not functional. ◊

2.2.3 Variable-Set Automata

While the current paper is mostly motivated by spanners that are constructed using regex formulas, our upper bounds for these are obtained by converting regex formulas to the following automata model.

A *variable-set automaton* (vset-automaton for short) with variables from a finite set $V \subset \mathsf{Vars}$ can be understood as an ϵ-NFA (i.e., an NFA with epsilon transitions allowed) that is extended with edges labeled with variable operations $x\vdash$ or $\dashv x$ for $x \in V$. Formally, a *vset-automaton* is a tuple $A := (V, Q, q_0, q_f, \delta)$, where V is a finite set of variables, Q is the set of *states*, $q_0, q_f \in Q$ are the *initial* and the *final* states, respectively, and $\delta\colon Q \times (\mathbf{\Sigma} \cup \{\epsilon\} \cup \mathbf{\Gamma}_V) \to 2^Q$ is the *transition function*. By $\mathsf{Vars}(A)$ we denote the set V.

The vset-automaton $A := (V, Q, q_0, q_f, \delta)$ can be interpreted as a directed graph, where the nodes are the states, and every $q \in \delta(p, a)$ is represented by an edge from p to q with the label a. To define the semantics of A, we first interpret A as an ϵ-NFA over the terminal alphabet $\mathbf{\Sigma} \cup \mathbf{\Gamma}_V$, and define its *ref-word language* $\mathcal{R}(A)$ as the set of all ref-words $\mathbf{r} \in (\mathbf{\Sigma} \cup \mathbf{\Gamma}_V)^*$ such that some path from q_0 to q_f is labeled with $\mathbf{r}$.

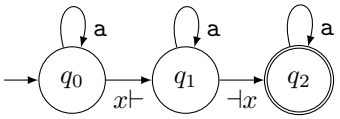
Let A be a vset-automaton. Analogously to regex formulas, we denote by $\mathsf{Ref}(A)$ the set of all ref-words in $\mathcal{R}(A)$ that are valid for V, and define $\mathsf{Ref}(A, \mathbf{s})$ and $[\![A]\!](\mathbf{s})$ accordingly for every $\mathbf{s} \in \mathbf{\Sigma}^*$. Likewise, we say that A is *functional* if $\mathsf{Ref}(A) = \mathcal{R}(A)$; i. e., every accepting run of A generates a valid ref-word. Two vset-automata A_1, A_2 are *equivalent* if $[\![A_1]\!] = [\![A_2]\!]$.

Example 2.6. Let A be the following vset-automaton:

Then $\mathcal{R}(A) = \{x\vdash, \mathsf{a}, \dashv x\}^*$, and

$$\mathsf{Ref}(A) = \{\mathsf{a}^i\, x\vdash \mathsf{a}^j \dashv x\, \mathsf{a}^k \mid i,j,k \geq 0\}.$$

Hence, A is not functional, since $\mathcal{R}(A)$ contains invalid ref-words such as ϵ, $x\vdash$, $\dashv x\mathsf{a}\, x\vdash$, and $x\vdash \mathsf{a} \dashv x \dashv x$. Now consider the vset-automaton A_{fun}:



Then $\mathsf{Ref}(A_{\mathsf{fun}}) = \mathcal{R}(A_{\mathsf{fun}}) = \mathsf{Ref}(A)$. Hence, A_{fun} is functional, and A_{fun} and A are equivalent.

For all $\mathbf{s} \in \mathbf{\Sigma}^* \setminus \{\mathsf{a}\}^*$, $[\![A]\!](\mathbf{s}) = \emptyset$; and for all $\mathbf{s} \in \{\mathsf{a}\}^*$, $[\![A]\!](\mathbf{s})$ contains all possible $(\{x\}, \mathbf{s})$-tuples. ◊

While $\mathsf{Ref}(A_1) = \mathsf{Ref}(A_2)$ is a sufficient criterion for the equivalence of A_1 and A_2, it is only characteristic if the automata have at most one variable. For example, consider $\mathbf{r}_1 := x\vdash y\vdash \dashv x \dashv y$ and $\mathbf{r}_2 := y\vdash \dashv y\, x\vdash \dashv x$. Both are valid ref-words that define the same $(\{x,y\},\epsilon)$-tuple $[1,1\rangle$, but $\mathbf{r}_1 \neq \mathbf{r}_2$.

Example 2.6 suggests that vset-automata can be converted into equivalent functional vset-automata; but Freydenberger [15] showed that although this is possible with standard automata constructions, the resulting blow-up may be exponential in the number of variables.

As shown in Lemma 3.4, every functional regex formula can be converted into an equivalent functional vset-automaton; and we use this connection throughout the paper. We shall see that functional vset-automata are a convenient tool for working with regex formulas. In contrast to this, vset-automata in general can be quite inconvenient (e. g., even deciding emptiness of $[\![A]\!](\epsilon)$ is NP-complete if A is not functional, see [15]).

Freydenberger [15] established the complexity of testing whether a given vset-automaton is functional.

Theorem 2.7. [15] *Whether a given vset-automaton α with n states, m transitions and v variables is functional can be tested in $O(vm + n)$ time.*

A special property of functional vset-automata is that each state implicitly stores which variables have been opened and closed. We discuss it in detail in Section 4.2.

2.2.4 Spanner Algebras

Let P, P_1 and P_2 be spanners. The algebraic operators *union*, *projection*, *natural join*, and *selection* are defined as follows:

Union: If $\mathsf{Vars}(P_1) = \mathsf{Vars}(P_2)$, their *union* $(P_1 \cup P_2)$ is defined by $\mathsf{Vars}(P_1 \cup P_2) := \mathsf{Vars}(P_1)$ and $(P_1 \cup P_2)(\mathbf{s}) := P_1(\mathbf{s}) \cup P_2(\mathbf{s})$ for all $\mathbf{s} \in \mathbf{\Sigma}^*$.

Projection: Let $Y \subseteq \mathsf{Vars}(P)$. The *projection* $\pi_Y P$ is defined by $\mathsf{Vars}(\pi_Y P) := Y$ and $\pi_Y P(\mathbf{s}) := P|_Y(\mathbf{s})$ for all $\mathbf{s} \in \mathbf{\Sigma}^*$, where $P|_Y(\mathbf{s})$ is the restriction of all $\mu \in P(\mathbf{s})$ to Y.

Natural join: Let $V_i := \mathsf{Vars}(P_i)$ for $i \in \{1,2\}$. The *(natural) join* $(P_1 \bowtie P_2)$ of P_1 and P_2 is defined by $\mathsf{Vars}(P_1 \bowtie P_2) := \mathsf{Vars}(P_1) \cup \mathsf{Vars}(P_2)$ and, for all $\mathbf{s} \in \mathbf{\Sigma}^*$, $(P_1 \bowtie P_2)(\mathbf{s})$ is the set of all $(V_1 \cup V_2, \mathbf{s})$-tuples μ for which there exist $\mu_1 \in P_1(\mathbf{s})$ and $\mu_2 \in P_2(\mathbf{s})$ with $\mu|_{V_1}(\mathbf{s}) = \mu_1(\mathbf{s})$ and $\mu|_{V_2}(\mathbf{s}) = \mu_2(\mathbf{s})$.

Selection: Let $x_1, \dots, x_k \in \mathsf{Vars}(P)$. We define the *string equality selection* $\zeta^=_{x_1,\dots,x_k} P$ by $\mathsf{Vars}(\zeta^=_{x_1,\dots,x_k} P) := \mathsf{Vars}(P)$ and, for all $\mathbf{s} \in \mathbf{\Sigma}^*$, $\zeta^=_{x_1,\dots,x_k} P(\mathbf{s})$ is the set of all $\mu \in P(\mathbf{s})$ for which $\mathbf{s}_{\mu(x_1)} = \dots = \mathbf{s}_{\mu(x_k)}$.

Note that the join operator joins tuples that have identical spans in their shared variables. In contrast, the selection operator compares the substrings of $\mathbf{s}$ that are described by the spans, and does not distinguish between different spans that span the same substrings.

Following Fagin et al. [12], we refer to regex formulas and vset-automata as *primitive spanner representations*, and use $\mathsf{VA}_{\mathsf{set}}$ and RGX to denote the sets of all vset-automata and all functional regex formulas, respectively. A *spanner algebra* is a finite set of spanner operators. If O is a spanner algebra and C is a class of primitive spanner representations, then C^{O} denotes the set of all *spanner representations* that can be constructed by (repeated) combination of the symbols for the operators from O with spanners from C. For each spanner representation of the form $o\varrho$ or $\varrho_1\, o\, \varrho_2$, where $o \in \mathsf{O}$ and $\varrho, \varrho_1, \varrho_2 \in C$, we define $[\![o\varrho]\!] = o[\![\varrho]\!]$ and $[\![\varrho_1\, o\, \varrho_2]\!] = [\![\varrho_1]\!]\, o\, [\![\varrho_2]\!]$. Furthermore, $[\![C^{\mathsf{O}}]\!]$ is the closure of $[\![C]\!]$ under the spanner operators in O.

Fagin et al. [12] refer to the elements of $\mathsf{RGX}^{\{\pi,\cup,\bowtie\}}$ as *regular spanner representations*, and to the elements of $\mathsf{RGX}^{\{\pi,\zeta^=,\cup,\bowtie\}}$ as *core spanner representations* (as these form the core of the query language AQL [25]). One of the main results of [12] is that $[\![\mathsf{RGX}^{\{\pi,\cup,\bowtie\}}]\!] = [\![\mathsf{VA}_{\mathsf{set}}{}^{\{\pi,\cup,\bowtie\}}]\!] = [\![\mathsf{VA}_{\mathsf{set}}]\!]$. Hence, $\mathsf{VA}_{\mathsf{set}}{}^{\{\pi,\zeta^=,\cup,\bowtie\}}$ has the same expressive power as core spanner representations. A *regular spanner* is a spanner that can be represented in a regular spanner representation, and a *core spanner* is a spanner that can be represented in a core spanner representation. Fagin et al. [12] also showed the class of regular spanners is closed under the *difference* operator (i.e., $[\![\mathsf{RGX}^{\{\pi,\cup,\bowtie\}}]\!] = [\![\mathsf{RGX}^{\{\pi,\cup,\bowtie,\setminus\}}]\!]$), but the class of core spanners is not.

2.3 (Unions of) Conjunctive Queries

In this paper, we consider Conjunctive Queries (CQs) over regex formulas. Such queries are defined as the class of regular spanner representations that can be composed out of natural join and projection. In addition, we explore the extension obtained by adding string-equality selections.

More formally, a *regex CQ* is an regular spanner representation of the form $q := \pi_Y\, (\alpha_1 \bowtie \dots \bowtie \alpha_k)$ where each α_i is a regex formula. A *regex CQ with string*

equalities is a core spanner representation of the form

$$q := \pi_Y \left(\zeta^{=}_{x_1,x_2} \cdots \zeta^{=}_{x_l,x_{l+1}} (\alpha_1 \bowtie \cdots \bowtie \alpha_k) \right)$$

for some $k \geq 1$ and $l \geq 0$. For clarity of notation, if $Y = \{y_1, \cdots, y_m\}$, we sometimes write $q(y_1, \ldots, y_m)$ when using q to make $\mathsf{Vars}(q)$ more explicit. Each regex formula α_i and each selection $\zeta^{=}_{x_j,y_j}$ is called an *atom*, where the former is a *regex atom* and the latter an *equality atom*. We denote by $\mathsf{atoms}(q)$ the set of atoms of q. For each equality atom $\zeta^{=}_{x_j,y_j}$, we define $\mathsf{Vars}(\zeta^{=}_{x_j,y_j}) := \{x_j, y_j\}$. Note that our definitions imply that every variable that occurs in an equality atom also appears in at least one regex atom.

In the traditional relational model (see, e.g., Abiteboul et al. [1]), a CQ is phrased over a collection of relation symbols (called *signature*), each having a predefined arity. Formally, a *relational CQ* is an expression of the form $Q(y_1, \ldots, y_m) :- \varphi_1, \ldots, \varphi_k$ where each y_i is variable in Vars and each φ_i is an *atomic relational formula* (or simply *atom*), that is, an expression of the form $R(x_1, \ldots, x_m)$ where R is an m-ary relation symbol and each x_j is a variable. Similarly to regex CQs, we denote by $\mathsf{atoms}(Q)$ the set of atoms of Q, and by $\mathsf{Vars}(\gamma)$ the set of variables that occur in an atom γ.

Let $q(y_1, \ldots, y_m)$ be a regex CQ (with string equalities), and let $Q(y_1, \ldots, y_m)$ be a relational CQ. We say that q *maps to* Q if all of the following hold.

- No relation symbol occurs more than once in Q (that is, Q has no self joins).
- There is a bijection $\mu : \mathsf{atoms}(q) \to \mathsf{atoms}(Q)$ that preserves the sets of variables, that is, for each $\gamma \in \mathsf{atoms}(q)$ we have $\mathsf{Vars}(\gamma) = \mathsf{Vars}(\mu(\gamma))$.

Let q be a regex CQ with string equalities. We say that q is *acyclic* if it maps to an acyclic (or *alpha-acyclic*) relational CQ, and *gamma-acyclic* is it maps to a gamma-acyclic relational CQ. (See e.g. [1, 11] for the definitions of acyclicity.) Recall that gamma-acyclicity is strictly more restricted than acyclicity (that is, every gamma-acyclic CQ is acyclic, and there are acyclic CQs that are not gamma-acyclic).

A *Union of regex CQs*, or *regex UCQ* for short, is a regular spanner q of the form $q := \bigcup_{i=1}^{k} q_i$ where each q_i is a regex CQ (recall that, by definition, $\mathsf{Vars}(q_i) = \mathsf{Vars}(q_j)$ must hold). In a regex UCQ *with string equalities*, each q_i can be a UCQ with string equalities. The following theorem follows quite easily from the results of Fagin et al. [12].

Theorem 2.8. *The following hold.*

- *The class of spanners expressible as regex UCQs is that of the regular spanners.*
- *The class of spanners expressible as regex UCQs with string equalities is that of the core spanners.*

In the relational world, a *relational UCQ* is a query of the form $\bigcup_{i=1}^{l} Q_i$, where each Q_i is a relational CQ (and $\mathsf{Vars}(Q_i) = \mathsf{Vars}(Q_j)$ holds). Given a regex UCQ (with or without string equalities) $q := \bigcup_{i=1}^{k} q_i(y_1, \ldots, y_m)$ and a relational UCQ $Q := \bigcup_{i=1}^{l} Q_i(y_1, \ldots, y_m)$, we say that q *maps to* Q if $k = l$ and each q_i maps to Q_i.

A family $\mathbf{P}$ of regex UCQs (with string equalities) *maps to* a family $\mathbf{Q}$ of relational UCQs if each UCQ in $\mathbf{P}$ maps to one or more CQ in $\mathbf{P}$.

3. COMPLEXITY OF UCQ EVALUATION

In this section we give our main complexity results for the evaluation of regex UCQs. We remark that in this section we do not allow string equalities; these will be discussed in Section 5. We begin with a description of the complexity measures we adopt.

3.1 Complexity Measures

Under the measure of *data complexity*, where the UCQ q at hand is assumed fixed (and the string $\mathbf{s}$ is given as input), it is a straightforward observation that query evaluation can be done in polynomial time (see Freydenberger and Holldack [16]). Hence, our main measure of complexity is that of *combined complexity* where both q and $\mathbf{s}$ are given as input.

The task of evaluating a regex query q over an input $\mathbf{s}$ requires the solver algorithm to produce all tuples in $[\![\gamma(\mathbf{s})]\!]$ for each regex formula γ in q. In the worst case there could be exponentially many tuples, and so *polynomial time* is not a proper yardstick of efficiency. For such problems, Johnson, Papadimitriou and Yannakakis [23] introduced several complexity guarantees, which we recall here. An *enumeration problem* $\mathbf{E}$ is a collection of pairs (x, Y) where x is an *input* and Y is a finite set of *answers* for x, denoted by $\mathbf{E}(x)$. In our case, x has the form $(q, \mathbf{s})$ and $\mathbf{E}(x)$ is $[\![q]\!](\mathbf{s})$. A *solver* for an enumeration problem $\mathbf{E}$ is an algorithm that, when given an input x, produces a sequence of answers such that every answer in $\mathbf{E}(x)$ is printed precisely once. We say that a solver S for an enumeration problem $\mathbf{E}$ runs in *polynomial total time* if the total execution time of S is polynomial in $(|x| + |\mathbf{E}(x)|)$; and in *polynomial delay* if the time between every two consecutive answers produced is polynomial in $|x|$.

We also consider *parameterized complexity* [10, 20] for various parameters determined by q. Formally, a *parameterized problem* is a decision problem where the input consists of a pair (x, k), where x is an ordinary input and k is a parameter (typically small, relates to a property of x). Such a problem is *Fixed-Parameter Tractable* (*FPT*) if there is a polynomial p, a computable function f and a solver S, such that S terminates in time $f(k) \cdot p(|x|)$ on input (x, k). We similarly define *FPT-delay* for a parameterized enumeration algorithm: the delay between every two consecutive answers is bounded by $f(k) \cdot p(|x|)$. A standard lower bound is W[1]-*hardness*, and the standard complexity assumption is that a W[1]-hard problem is not FPT [20].

Whenever we give an upper bound, it applies to a general UCQ, and whenever we give a lower bound, it applies to Boolean CQs. When we give asymptotic

running times, we assume the unit-cost RAM-model, where the size of each machine word is logarithmic in the size of the input. Regarding $\mathbf{\Sigma}$, our lower bounds and asymptotic upper bounds assume that it is fixed with at least two characters; our "polynomial" upper bounds hold even if $\mathbf{\Sigma}$ is given as part of the input.

3.2 Lower Bounds

We begin with lower bounds. Recall that Boolean CQ evaluation is NP-complete [7]. This result does not extend directly to regex UCQs, since relations are not given directly as input (but rather extracted from an input string using regex formulas). But quite expectedly, the evaluation of Boolean regex CQs indeed remains NP-complete. What is less expected is that this holds even for strings that consist of a single fixed character.

Theorem 3.1. *Evaluation of Boolean regex CQs is* NP*-complete, and remains* NP*-hard even under both of the following assumptions.*

1. *Each regex formula is of bounded size*
2. *The input string is of length one.*

Proof. The upper bound is obvious (even for core spanners, see [16]). For the lower bound, we construct a reduction from 3CNF-satisfiability to the evaluation problem of Boolean regex CQs. The input to 3CNF is a formula ψ with the free variables $x_1, \ldots, x_n$ such that ψ has the form $C_0 \wedge \cdots \wedge C_m$ where each C_j is a clause. Each clause is a conjunction of three literals from the set $\{x_i, \neg x_i \mid 1 \leq i \leq n\}$. The goal is to determine whether there is an assignment $\tau\colon \{x_1, \ldots, x_n\} \to \{0, 1\}$ that satisfies ψ. Given a 3CNF-formula ψ, we construct a regex CQ q and an input string $\mathbf{s}$ such that there is a satisfying assignment for ψ if and only if $[\![q]\!](\mathbf{s}) \neq \emptyset$. We define $\mathbf{s} := \mathtt{a}$. To construct q, we associate each variable x with a corresponding capture variable x and each assignment τ with a regex formula that assigns x the span $[1,1\rangle$ if $\tau(x) = 0$ and $[2,2\rangle$ if $\tau(x) = 1$. For instance, given the assignment τ such that $\tau(x) = \tau(y) = 0$ and $\tau(z) = 1$ its corresponding regex is $x\{y\{\epsilon\}\} \cdot \mathtt{a} \cdot z\{\epsilon\}$. Note that since each clause C_i contains three variables, it has exactly seven satisfying assignments (out ot all possible eight assignments to its variables). We denote these assignments by $\tau_i^1, \ldots, \tau_i^7$ and their corresponding regex formulas by $\gamma_i^1, \ldots, \gamma_i^7$. We then define $\gamma_i = \bigvee_{j=1}^{7} \gamma_i^j$ and $q := \pi_\emptyset \bowtie_{i=1}^{m} \gamma_i$.

It is straightforward to show that if there exists a satisfying assignment τ for ψ then there is at least one $\mu \in [\![q]\!](\mathbf{s})$. This μ is obtained from τ by defining $\mu(x) := [1,1\rangle$ if $\tau(x) = 0$ and $\mu(x) := [2,2\rangle$ if $\tau(x) = 1$. The other direction is shown analogously: If there is at least one $\mu \in [\![q]\!](\mathbf{s})$, then a satisfying assignment τ for ψ is obtained by defining $\tau(x) := 0$ if $\mu(x) = [1,1\rangle$ and $\tau(x) := 1$ if $\mu(x) = [2,2\rangle$. □

One might be tempted to think that the evaluation of regex CQs over a string $\mathbf{s}$ is tractable as long as the regex CQ $q := \pi_Y(\alpha_1 \bowtie \cdots \bowtie \alpha_k)$ maps to a relational CQ of a tractable class (e.g., acyclic CQs where evaluation is in polynomial total time [42]), by applying what we refer to as the *canonical relational* evaluation:

(a) Evaluate each regex formula: $r_i := [\![\alpha_i]\!](\mathbf{s})$.

(b) Evaluate $\pi_Y(r_1 \bowtie \cdots \bowtie r_k)$ (as a relational CQ).

There are, though, two problems with the canonical relational evaluation. The first (and main) problem is that r_i may already be too large (e.g., exponential number of tuples). The second problem is that, even if r_i is of manageable size, it is not clear that it can be efficiently constructed. In the next section we will show that the second problem is solvable: we can evaluate α_i over $\mathbf{s}$ in polynomial total time. However, the first problem remains. In fact, the following theorem states that the evaluation of regex CQs is intractable, even if we restrict to ones that map to acyclic CQs, and even the more restricted gamma-acyclic CQs! In addition, we can show W[1]-hardness with respect to the number of variables or regex formulas.

Theorem 3.2. *Evaluation of gamma-acyclic Boolean regex CQs is* NP*-complete. The problem is also* W[1]*-hard with respect to the number of* (a) *variables,* and (b) *atoms.*

Proof. The NP-upper bound was already discussed in the proof of Theorem 3.1. We prove both lower bounds at the same time by defining a polynomial time FPT-reduction from the k-clique problem. Given an undirected graph $G := (V, E)$ and a $k \geq 2$, this problem asks whether G contains a clique with k nodes. Let $\mathbf{\Sigma} := \{\mathtt{a}, \mathtt{b}, \vdash, \#, \dashv\}$ (the proof can be adapted to a binary $\mathbf{\Sigma}$ with standard techniques). We assume $V = \{v_1, \ldots, v_n\}$, $n = |V|$, and associate each $v_i \in V$ with a unique string $\mathbf{v}_i \in \{\mathtt{a}, \mathtt{b}\}^*$ such that $|\mathbf{v}_i|$ is $O(\log n)$.

For all $1 \leq i < j \leq n$, let $\mathbf{e}_{i,j} = \epsilon$ if $\{v_i, v_j\} \notin E$, and $\mathbf{e}_{i,j} := \vdash \mathbf{v}_i \# \mathbf{v}_j \dashv$ if $\{v_i, v_j\} \in E$. Finally, we define $\mathbf{s} := (\mathbf{e}_{1,2} \cdots \mathbf{e}_{1,n}) \cdot (\mathbf{e}_{2,3} \cdots \mathbf{e}_{2,n}) \cdots (\mathbf{e}_{n-1,n})$. Thus, $\mathbf{s}$ encodes E, such that an edge $\{v_i, v_j \mid i < j\}$ precedes an edge $\{v_{i'}, v_{j'} \mid i' < j'\}$ if $i < i'$, or $i = i'$ and $j < j'$.

Next, we construct q such that $[\![q]\!](\mathbf{s}) \neq \emptyset$ if and only if there is a k-clique in G. Note that a k-clique has k nodes $v_{c(1)}, \ldots, v_{c(k)}$, and we assume that $i < j$ implies $c(i) < c(j)$. For each $v_{c(l)}$, q shall contain the $(k-1)$ variables $y_{1,l}$ to $y_{l-1,l}$ and $x_{l,l+1}$ to $x_{l,k}$. The intuition is that each pair $x_{i,j}$ and $y_{i,j}$ of variables corresponds to an edge $\{v_{c(i)}, v_{c(j)}\}$. In particular, $x_{i,j}$ and $y_{i,j}$ shall represent the node with the smaller and larger index, respectively. To this end, we define $\gamma := \gamma_1 \cdots \gamma_{k-1}$ with $\gamma_i := \gamma_{i,i+1} \cdots \gamma_{i,k}$ and

$$\gamma_{i,j} := \mathbf{\Sigma}^* \vdash x_{i,j}\{(\mathtt{a} \vee \mathtt{b})^*\} \# y_{i,j}\{(\mathtt{a} \vee \mathtt{b})^*\} \dashv \mathbf{\Sigma}^*$$

for all $1 \leq i < j \leq k$. The idea is that $x_{i,j}$ and $y_{i,j}$ respectively match $\mathbf{v}_{c(i)}$ and $\mathbf{v}_{c(j)}$ in $\mathbf{e}$ in $\mathbf{s}$, which uses the same order. We now want to ensure that for each $1 \leq l \leq k$, all $y_{i,l}$ and all $x_{l,j}$ with $1 \leq i < l < j \leq k$ have to be matched to various occurrences of the same

substring. To ensure this, for each $1 \leq l \leq k$, we define the regex formula $\delta_l := \bigvee_{i=1}^{n} \delta_{l,\mathbf{v}_i}$ where

$$\delta_{l,\mathbf{v}} := \mathbf{\Sigma}^* \# y_{1,l}\{\mathbf{v}\} \dashv \mathbf{\Sigma}^* \cdots \mathbf{\Sigma}^* \# y_{l-1,l}\{\mathbf{v}\} \dashv \mathbf{\Sigma}^* \cdot \mathbf{\Sigma}^* \vdash x_{l,l+1}\{\mathbf{v}\} \# \mathbf{\Sigma}^* \cdots \mathbf{\Sigma}^* \vdash x_{l,k}\{\mathbf{v}\} \# \mathbf{\Sigma}^*.$$

Finally we define q to be the query

$$q := \pi_\emptyset \left(\gamma \bowtie \bigbowtie_{1 \leq i \leq k-1} \delta_i \right)$$

Note that q contains $O(k)$ atoms and $O(k^2)$ variables. Additionally, q contains no gamma-cycles since each two different δ_l have no common variables. Moreover, as $|\gamma|$ is $O(k^2)$, and each $|\delta_l|$ is $O(kn \log n)$, $|q|$ is $O(k^2 + k^2 n \log n) = O(k^2 n \log n)$. Furthermore, $|\mathbf{s}|$ is $O(|E|)$. Hence, q and $\mathbf{s}$ can be constructed in polynomial time, and the construction is FPT with respect to the number of variables and atoms.

All that is left to show that the reduction is correct; that is $[\![q]\!](\mathbf{s}) \neq \emptyset$ iff. G contains a k-clique. Assume that G contains a k-clique $\{v_{c(1)}, \ldots, v_{c(k)}\}$ where $c(i) < c(j)$ whenever $i < j$. Let μ be an $\mathbf{s}$-tuple that is defined as follows: For all $1 \leq i < j \leq k$, find the substring $\vdash \mathbf{v}_{c(i)} \# \mathbf{v}_{c(j)} \dashv$. Then map $x_{i,j}$ to the span that corresponds to $\mathbf{v}_{c(i)}$ in this substring, and $y_{i,j}$ to the span that corresponds to $\mathbf{v}_{c(j)}$. Then $\mu \in [\![\gamma]\!](\mathbf{s})$, since each two nodes in the clique are connected and since the encoding of E in $\mathbf{s}$ is ordered. Moreover, for each l, the restriction of μ to $\mathsf{Vars}(\delta_l)$ is in $[\![\delta_l]\!](\mathbf{s})$, since the strings spanned by the $y_{i,l}$ and the $x_{l,j}$ are equal, and μ respects the order of the variables in δ_l.

Now assume $\mu \in [\![q]\!](\mathbf{s})$. We can now derive the nodes $v_{c(1)}, \ldots, v_{c(k)}$ of the clique directly from the variables $x_{i,j}$ and $y_{i,j}$, as for each l, $\mu \in [\![\delta_l]\!](\mathbf{s})$ ensures that there is a unique $c(l)$ such that $\mathbf{s}_{\mu(y_{i,l})} = \mathbf{s}_{\mu(x_{l,j})} = \mathbf{v}_{c(l)}$ for all $1 \leq i < l < j \leq k$. Furthermore, $\mu \in [\![\gamma]\!](\mathbf{s})$ ensures that for all $i < j$, $\mu(x_{i,j})$ and $\mu(y_{i,j})$ map to the encoding of the edge $\{v_{c(i)}, v_{c(j)}\}$ in $\mathbf{s}$. Hence, $\{v_{c(1)}, \ldots, v_{c(k)}\}$ is a k-clique in G. □

Note that the we did not show W[1]-hardness with respect to the length of the query.

3.3 Upper Bounds

3.3.1 Evaluation of Variable-Set-Automata

Our central complexity result states that functional vset-automata can be evaluated with polynomial delay.

Theorem 3.3. *Given a functional vset-automaton A with n states and m transitions, and a string $\mathbf{s}$, one can enumerate $[\![A]\!](\mathbf{s})$ with polynomial delay of $O(n^2|\mathbf{s}|)$, following a polynomial preprocessing of $O(n^2|\mathbf{s}| + mn)$.*

Note that Losemann [27] introduced an algorithm that evaluates vset-automata with logarithmic delay, but only *if the number of variables is bounded*[2]. We discuss our algorithm and other details of the proof in Section 4.

[2]Otherwise, two steps of the algorithm become exponential, namely the encoding of vset-automata to another automata model and the actual enumeration. Using functional automata might make the first step polyonimal, but not the second.

In the remainder of this section, we explain the implications of this theorem for both evaluation approaches.

3.3.2 Canonical Relational Approach

It was already shown by Fagin et al. [12] that every regex formula can be converted into an equivalent vset-automaton (where a vset-automaton A and a regex formula α are said to be *equivalent* if $[\![A]\!] = [\![\alpha]\!]$). It is probably not at all surprising that this is possible in a way that is efficient and results in functional vset-automata (recall that we assume regex formulas to be functional by convention).

Lemma 3.4. *Given a regex formula α, one can construct in time $O(|\alpha|)$ a functional vset-automaton A with $[\![A]\!] = [\![\alpha]\!]$.*

The proof uses the Thompson construction for converting regular expressions into NFAs; but as the proof operates via ref-words, other method for converting regular expressions to NFAs could be chosen as well. The same result was shown independently in [28].

We note here that the complexity of the preprocessing stated in Theorem 3.3 holds for a general functional vset-automaton. If, however, the automaton is derived from a regex formula α by the construction we use for proving Lemma 3.4, then the time of this preprocessing drops to $O(n^2|\mathbf{s}|)$, where n is $O(|\alpha|)$.

As a consequence, we conclude that the canonical relational approach to evaluation is actually efficient for UCQs, under two conditions. The first condition is that the regex CQs map to a tractable class of relational CQs. More formally, by *tractable class* of relational CQs we refer to a class $\mathbf{Q}$ of CQs that can be evaluated in polynomial total time,[3] such as acyclic CQs, or more generally CQs with *bounded hypertree width [18]*. But, as shown in Theorem 3.2, this condition is not enough. The second condition is that there is a polynomial bound on the number of tuples of each regex formula. More formally, we say that a class $\mathbf{A}$ of regex formulas is *polynomially bounded* if there exists a positive integer d such that for every regex formula $\alpha \in \mathbf{A}$ and string $\mathbf{s}$ we have $|[\![\alpha]\!](\mathbf{s})|$ is $O(|\mathbf{s}|^d)$.

Clearly, if every regex formula in $\mathbf{A}$ can be evaluated in polynomial time, then $\mathbf{A}$ is polynomially bounded. From Theorem 3.3 we conclude that the other direction also holds. Hence, from Theorem 3.3 and Lemma 3.4 we establish the following theorem.

Theorem 3.5. *Let $\mathbf{P}$ be a class of regex UCQs. If the regex formulas in the UCQs of $\mathbf{P}$ belong to a polynomially bounded class, and $\mathbf{P}$ maps to a tractable class of relational UCQs, then UCQs in $\mathbf{P}$ can be evaluated in polynomial total time.*

[3]We could also use other yardsticks of enumeration efficiency, such as polynomial delay and *incremental polynomial time* [23]. Then, every occurrence of "polynomial total time" in this part would be replaced with the other yardstick.

Examples of polynomially bounded classes of regex formulas are the following.

- The class of regex formulas with at most k variables, for some fixed k.
- The class of regex formulas α with a *key attribute*, that is, a variable $x \in \mathsf{Vars}(\alpha)$ with the property that for all strings $\mathbf{s}$ and tuples μ and μ' in $[\![\alpha]\!](\mathbf{s})$, if $\mu(x) = \mu'(x)$ then $\mu = \mu'$.

A key attribute implies a polynomial bound as the number of spans of a string $\mathbf{s}$ is quadratic in $|\mathbf{s}|$. Interestingly, its existence can be tested in polynomial time.

Proposition 3.6. *Given a functional vset-automaton A with n states, and a variable $x \in \mathsf{Vars}(A)$, it can be decided in time $O(n^4)$ whether x is a key attribute.*

3.3.3 Compilation to Automata

We now discuss the second evaluation approach: compiling the regex UCQ to a functional vset-automaton, and then applying Theorem 3.3. An immediate consequence of a combination with past results is as follows. There are computable (but potentially exponential) conversions of spanners in a regular representation into a vset-automaton (Fagin et al. [12]), and of a vset-automaton into a functional vset-automaton (Freydenberger [15]). Hence, we get the following.

Corollary 3.7. *Spanners q in a regular representation (e.g., regex UCQs) can be evaluated with FPT delay for the parameter $|q|$.*

The corollary should be contrasted with the traditional relational case, where Boolean CQ evaluation is W[1]-hard when the size of the query is the parameter [30]. Hence, in that respect regex UCQ evaluation is substantially more tractable than UCQ evaluation in the relational model.

Our next results are established by applying *efficient* compilations. Such compilations were obtained independently by Morciano et al. [28]; we discuss the differences in the approaches after each result (generally, both here and in [28], the proofs are based on standard constructions, but ref-words allow us to take shortcuts). Furthermore, our proofs also discuss the constructions with respect to Theorem 3.3. We begin with the most straightforward result, the projection operator.

Lemma 3.8. *Given a functional vset-automaton A and $Y \subseteq \mathsf{Vars}(A)$, one can construct in linear time a functional vset-automaton A_Y with $[\![A_Y]\!] = [\![\pi_Y(A)]\!]$.*

This is shown by replacing all transitions for operations on variables that are not in Y with ϵ-transitions. One advantage of our proof is that it showcases a nice use of the ref-word semantics. The situation is similar for the union operator.

Lemma 3.9. *Given functional vset-automata $A_1, \ldots, A_k$ with $\mathsf{Vars}(A_1) = \cdots = \mathsf{Vars}(A_k)$, one can construct in linear time a functional vset-automaton A with $[\![A]\!] = [\![A_1 \cup \cdots \cup A_k]\!]$.*

Like [28], we prove this using the union construction for NFAs. In the proof, we also argue that the upper-bound for the worst case complexity of Theorem 3.3 is lower than the number of states of the constructed automaton suggests. Observe that in Lemma 3.9, the number of automata is not bounded. The situation is different for the join operator, which also uses the only construction that is not completely straightforward.

Lemma 3.10. *Given two functional vset-automata A_1 and A_2, each with $O(n)$ states and $O(v)$ variables, one can construct in time $O(vn^4)$ a functional vset-automaton A with $[\![A_1 \bowtie A_2]\!] = [\![A]\!]$.*

Both this proof and the one from [28] build on the standard construction for automata intersection; the key difference is that our proof takes advantage of variable configurations. Note that joining k automata leads to a time of $O(vn^{2k})$, which is only polynomial if k is bounded. Due to Theorem 3.2, this is unavoidable under standard complexity theoretic assumptions.

This motivates the following definition. Let $k \geq 0$ be fixed. A *regex k-CQ* is a regex CQ with at most k atoms. A *regex k-UCQ* is a UCQ where each CQ is a k-CQ (i. e., a disjunction of k-CQs). From Lemmas 3.4 and 3.10 we conclude that we can convert, in polynomial time, a join of k regex formulas into a single functional vset-automaton. Then, Lemma 3.8 implies that projection can also be efficiently pushed into the functional vset-automaton. Hence, in polynomial time we can translate a k-CQ into a functional vset-automaton. Then, with Lemma 3.9 we conclude that this translation extends to k-UCQs. Finally, by applying Theorem 3.3 we arrive at the following main result.

Theorem 3.11. *For every fixed k, regex k-UCQs can be evaluated with polynomial delay.*

Hence, where Theorem 3.11 applies, it is more powerful than Theorem 3.5: The former holds for all regex k-UCQs, while the latter has additional requirements, even when limited to k-UCQs.

In the next section we describe the algorithm of Theorem 3.3. Then, in Section 5 we extend the main theorems of this section, Theorems 3.5 and 3.11, to incorporate string equalities.

4. EVALUATING VSET-AUTOMATA

In this section, we give a high-level description of the proof of Theorem 3.3 and the resulting algorithm that, given a functional vset-automaton A and a string $\mathbf{s} \in \mathbf{\Sigma}^*$, enumerates $[\![A]\!](\mathbf{s})$ with polynomial delay. Before we discuss the algorithm in Section 4.2, we first introduce the central notion of variable configurations.

4.1 Sequences of Variable Configurations

Before we discuss the actual algorithm, we first examine a property of functional vset-automata. Building on this, we introduce a novel way of working with vset-automata, which is the main part of our algorithm.

In order to introduce the concept, we consider an arbitrary functional vset-automaton $A = (V, Q, q_0, q_f, \delta)$, and ensure that all states are reachable from q_0, and that q_f is reachable from every state.

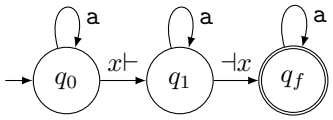
Then, for each state $q \in Q$ and all $x \in V$, each ref-word $\mathbf{r} \in (\mathbf{\Gamma}_V \cup \mathbf{\Sigma})^*$ that takes A from q_0 to q satisfies exactly one of these mutually exclusive conditions:

1. $|\mathbf{r}|_{x\vdash} = |\mathbf{r}|_{\dashv x} = 0$,
2. $|\mathbf{r}|_{x\vdash} = 1$ and $|\mathbf{r}|_{\dashv x} = 0$, or
3. $|\mathbf{r}|_{x\vdash} = |\mathbf{r}|_{\dashv x} = 1$, and $x\vdash$ occurs before $\dashv x$.

If neither of these conditions is met, $\mathbf{r}$ contains $x\vdash$ or $\dashv x$ twice, or the two symbols appear in the wrong order. Then we can choose any ref-word $\mathbf{r}' \in (\mathbf{\Gamma}_V \cup \mathbf{\Sigma})^*$ that takes A from q to q_f, and obtain a contradiction by observing that $\mathbf{r} \cdot \mathbf{r}' \in \mathcal{R}(A)$, although $\mathbf{r} \cdot \mathbf{r}'$ is not valid. Furthermore, if we compare two ref-words $\mathbf{r}_1$ and $\mathbf{r}_2$ that both take A from q_0 to a common state q, we know that both must satisfy the same of these three conditions (otherwise, $\mathbf{r}_1 \cdot \mathbf{r}'$ or $\mathbf{r}_2 \cdot \mathbf{r}'$ is not valid).

In other words, in each run of A, the information which variables have been opened or closed is stored implicitly in the states. To formalize this notion, we define the set $\mathcal{V}$ of *variable states* as $\mathcal{V} := \{\mathtt{w}, \mathtt{o}, \mathtt{c}\}$ (the symbols stand for `w`aiting, `o`pen, and `c`losed). A *variable configuration (for V)* is a function $\vec{c}\colon V \to \mathcal{V}$. For each state $q \in Q$, we define its variable configuration $\vec{c}_q : V \to \mathcal{V}$ as follows: Choose any ref-word $\mathbf{r}$ that takes A from q_0 to q. We define $\vec{c}_q(x) := \mathtt{c}$ if $\mathbf{r}$ contains $\dashv x$, $\vec{c}_q(x) := \mathtt{o}$ if $\mathbf{r}$ contains $x\vdash$ but not $\dashv x$, and $\vec{c}_q(x) := \mathtt{w}$ if $\mathbf{r}$ contains neither of the two symbols (as explained above, $\vec{c}_q$ is well-defined for each $q \in Q$).

Example 4.1. Recall the functional vset-automaton A_{fun} from Example 2.6.



Then $\vec{c}_{q_0}(x) = \mathtt{w}$, $\vec{c}_{q_1}(x) = \mathtt{o}$, and $\vec{c}_{q_f}(x) = \mathtt{c}$. ◊

Before we use this for the enumeration algorithm, we consider a more general view on variable configurations, that is independent of the automaton. As we shall see, the enumeration algorithm relies on the fact that for each $\mathbf{s} \in \mathbf{\Sigma}^*$ ($\mathbf{s} = \sigma_1 \cdots \sigma_N$ with $N \geq 0$), each $(V, \mathbf{s})$-tuple μ can be interpreted as a sequence of $N+1$ variable configurations $\vec{c}_1, \ldots, \vec{c}_{N+1}$ in the following way: For $x \in V$, assume that $\mu(x) = [i, j\rangle$. For $1 \leq l \leq N+1$, we define $\vec{c}_l(x) := \mathtt{w}$ if $l < i$, $\vec{c}_l(x) := \mathtt{o}$ if $i \leq l < j$, and $\vec{c}_l(x) := \mathtt{c}$ if $l \geq j$. The idea is that each $\vec{c}_l$ is the variable configuration immediately before reading σ_l.

To illustrate this, consider a ref-word $\mathbf{r}$ with $\mu = \mu^{\mathbf{r}}$ (see Section 2.2.1). Then $x\vdash$ is read between $\vec{c}_{i-1}$ and $\vec{c}_i$, while $\dashv x$ is read between $\vec{c}_{j-1}$ and $\vec{c}_j$ (again ignoring the technicality that we do not define $\vec{c}_0$).

Example 4.2. Let $V := \{x\}$, and let $\mathbf{s} := \mathtt{aa}$. The following table contains all possible $(V, \mathbf{s})$-tuples, and the corresponding sequence $\vec{c}_1(x), \vec{c}_2(x), \vec{c}_3(x)$:

$\mathbf{r}$	$\mu^{\mathbf{r}}(x)$	$\vec{c}_1(x), \vec{c}_2(x), \vec{c}_3(x)$
$x\vdash \dashv x\mathtt{aa}$	$[1, 1\rangle$	`c, c, c`
$x\vdash \mathtt{a} \dashv x\, \mathtt{a}$	$[1, 2\rangle$	`o, c, c`
$x\vdash \mathtt{aa} \dashv x$	$[1, 3\rangle$	`o, o, c`
$\mathtt{a}\, x\vdash \dashv x\mathtt{a}$	$[2, 2\rangle$	`w, c, c`
$\mathtt{a}\, x\vdash \mathtt{a} \dashv x$	$[2, 3\rangle$	`w, o, c`
$\mathtt{aa}\, x\vdash \dashv x$	$[3, 3\rangle$	`w, w, c`

Note that this is exactly $[\![A_{\mathsf{fun}}]\!](\mathbf{s})$, where A_{fun} is the vset-automaton from Example 4.1. ◊

We say that a sequence of variable configurations for V is *valid* if it respects the order of variables states; i. e., $\vec{c}_i(x) = \mathtt{c}$ implies $\vec{c}_{i+1}(x) = \mathtt{c}$, and $\vec{c}_i(x) = \mathtt{o}$ implies $\vec{c}_{i+1}(x) \in \{\mathtt{o}, \mathtt{c}\}$ for all $x \in V$. Obviously, each valid sequence of $|\mathbf{s}| + 1$ variable configurations for V can be interpreted as a $(V, \mathbf{s})$-tuple; and it is easy to see that this is a one-to-one correspondence.

To connect this point of view to the variable configurations of A, note that each $\mathbf{r} \in \mathsf{Ref}(A, \mathbf{s})$ can be written as $\mathbf{r} = \mathbf{r}_0 \cdot \sigma_1 \cdot \mathbf{r}_1 \cdot \sigma_2 \cdots \mathbf{r}_{N-1} \cdot \sigma_N \cdot \mathbf{r}_N$, where $\mathbf{r}_i \in \mathbf{\Gamma}_V^*$. For $1 \leq i \leq N+1$, we determine $\vec{c}_i$ from $\mathbf{r}_0 \cdot \mathbf{r}_1 \cdots \mathbf{r}_{i-1}$ (as in the definition of the $\vec{c}_{q_i}$ above). This has the same effect as defining $\vec{c}_i := \vec{c}_{q_i}$ for any q_i that can be reached by processing $\mathbf{r}_0 \cdot \sigma_1 \cdot \mathbf{r}_1 \cdot \sigma_2 \cdots \mathbf{r}_{i-1}$. In other words, in each run of A on $\mathbf{r}$, immediately before processing σ_i, A must be in a state q_i with $\vec{c}_i = \vec{c}_{q_i}$. Thus, the sequence $\vec{c}_1, \ldots, \vec{c}_{N+1}$ corresponds to the $(V, \mathbf{s})$-tuple $\mu^{\mathbf{r}}$.

Like ref-words, sequences of variable configurations can be understood as an abstraction of spanner behavior. In fact, both can be seen as successive steps of generalization: Ref-words hide the actual behavior of primitive spanner representations (i. e., the actual sequence of states in the vset-automaton, or which parts of the regex are mapped to which part of the input); they only express in which order variables are opened and closed. Sequences of variable configurations take this one step further, and compress successive variable operations (without terminals in-between) into a single step. Hence, treating $[\![A]\!](\mathbf{s})$ as a language over the alphabet $\mathcal{V}^{|V|}$ is exactly the level of granularity that is needed to distinguish different tuples.

Note that although some proofs in [15] use the concept of variable configurations for states, identifiying $(V, \mathbf{s})$-tuples with sequences of variable configurations is a main conceptual contribution of the present paper.

4.2 The Algorithm

The algorithm enumerates the $(V, \mathbf{s})$-tuples of $[\![A]\!](\mathbf{s})$ by enumerating the corresponding sequences of $|\mathbf{s}| + 1$ variable configurations for V. In order to do so, we interpret each variable configuration as a letter of the alphabet $\mathcal{K} := \{\vec{c}_q \mid q \in Q\}$ (note that while there are $3^{|V|}$ possible letters that might occur in $\mathcal{K}$, its actual size is always bounded by $|Q|$). More specifically, the algorithm has the following two steps:

1. Given A and $\mathbf{s}$, construct an NFA A_G over the alphabet $\mathcal{K}$ such that $\mathcal{L}(A_G)$ contains exactly those

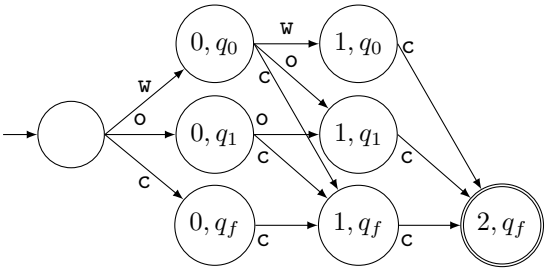


Figure 1: The NFA A_G from Example 4.3. For convenience, we write each variable configuration $\vec{c}_q$ as $\vec{c}_q(x)$.

strings $\kappa_1 \cdots \kappa_{|\mathbf{s}|+1}$, $\kappa_i \in \mathcal{K}$, that correspond to the elements of $[\![A]\!](\mathbf{s})$.

2. Enumerate $\mathcal{L}(A_G)$ with polynomial delay.

The algorithm constructs the NFA A_G by first constructing a graph G whose nodes are tuples (i, q), which encode that A can be in state q after reading $\sigma_1 \cdots \sigma_i$. The edges are drawn accordingly: There is an edge from (i, p) to $(i + 1, q)$ if A can reach q from p by reading σ_{i+1} and then arbitrarily many variable operations. The NFA A_G is then directly obtained from G by interpreting every edge from (i, p) to $(i+1, q)$ as a transition for the letter $\vec{c}_q$.

Finally, to enumerate $\mathcal{L}(A_G)$ without repetitions, we tailor an optimized version of the algorithm by Ackerman and Shallit [2] that, given $l \geq 0$ and an NFA M over some alphabet $\boldsymbol{T}$, enumerates $\mathcal{L}(M) \cap \boldsymbol{T}^l$.

In summary, the graph G represents all runs of A on $\mathbf{s}$. By interpreting $(V, \mathbf{s})$-tuples as strings from $\mathcal{K}^{|\mathbf{s}|+1}$, we can treat G as an NFA A_G, and enumerate $[\![A]\!](\mathbf{s})$ by enumerating $\mathcal{L}(A_G)$. As there is a one-to-one correspondence between these two sets, the fact that $\mathcal{L}(A_G)$ is enumerated without repetitions guarantees that no element of $[\![A]\!](\mathbf{s})$ is repeated.

Example 4.3. Consider A_{fun} (from Example 4.1) and $\mathbf{s} := \mathtt{aa}$. From A_{fun} and $\mathbf{s}$, the algorithm constructs the NFA A_G that is shown in Figure 1. To see that $\mathcal{L}(A_G)$ corresponds to $[\![A_{\mathsf{fun}}]\!](\mathbf{s})$, take note that the table in Example 4.2 lists each element of $[\![A_{\mathsf{fun}}]\!](\mathbf{s})$ together with its corresponding sequence of variable configurations. A more detailed version of this example and a second example can be found in Section A.3 in the Appendix. ◊

Note that the NFA A_G in Example 4.3 is deterministic, which means that enumerating $\mathcal{L}(A_G)$ actually requires less effort than in the general case: As stated in Theorem 3.3, if A has n states, we can enumerate $[\![A]\!](\mathbf{s})$ with a delay of $O(n^2|\mathbf{s}|)$. But if A_G is deterministic, this can be lowered to $O(|\mathbf{s}|)$.

While there are automata where the worst case for the algorithm is reached, the stated upper bounds only applies to automata where the states have many outgoing transitions (or a large number of ϵ-transitions). For examples where we can use it for better upper bounds, see the proofs of Lemma 3.9 and Theorem 5.4.

As a final remark, we observe that the algorithm uses the variable operations on the transitions of A only to compute the variable configurations. Afterwards, these transitions are treated as ϵ-transitions instead. This paper uses functional vset-automata to represent regex formulas. Instead, one could directly convert each regex formula into an ϵ-NFA A and a function that maps each state of A to a variable configuration. The enumeration algorithm and the "compilation lemmas" from Section 3.3.3 would directly work with this model. Whether this actually allows constructions that are more efficient or significantly simpler remains to be seen.

5. STRING EQUALITY

Our results in the previous two sections only apply to regex UCQs. As these are not allowed to use string equality selections, these have the same expressive power as regular spanner representations. In this section, we discuss how our results can be extended to include string equality selections, which allows us reach the full expressive power of core spanners.

5.1 Lower Bound

The main difficulty when dealing with string equality selections is that this operator quickly becomes computationally expensive, even without using joins. More specifically, Freydenberger and Holldack [16] showed that combined with string equalities, a single regex formula and a projection to a Boolean spanner already lead to an intractable evaluation problem.

Theorem 5.1. [16] *Evaluation of Boolean regex CQs with string equalities is* NP*-complete, even if restricted to queries of the form* $\pi_\emptyset \zeta^{=}_{x_1,y_1} \cdots \zeta^{=}_{x_m,y_m} \alpha$.

In other words, even a single regex formula already leads to NP-hardness. The proof from [16] uses a reduction from the membership problem for so-called pattern languages. It was shown by Fernau et al. [14] that this membership problem is W[1]-complete for various parameters. We prove that the situation is analogous for Boolean regex CQs with string equalities.

Theorem 5.2. *Evaluation of Boolean regex CQs q with string equalities is* W[1]*-hard for the parameter $|q|$, even if restricted to queries of the form* $\pi_\emptyset \zeta^{=}_{x_1,y_1} \cdots \zeta^{=}_{x_m,y_m} \alpha$.

Hence, while a regex CQ that consists of a single regex formula can be evaluated efficiently (due to Lemmas 3.4 and 3.8 and Theorem 3.3), even limited use of string equalities can become expensive. This result can be obtained by combining the reduction from [14] with the reduction from [16], the former proof requires encodings that are needlessly complicated for our purposes (this is caused by the comparatively low expressive power of pattern languages). Instead, we take the basic idea of an FPT-reduction from the k-clique problem, and

give a direct proof that is less technical. In fact, the proof of Theorem 5.2 is similar to that of Theorem 3.2, except that in the former the query q constructed in the reduction is determined solely by the parameter k, and not the size of the graph as in the latter.

The reader should contrast Theorem 5.2 with Corollary 3.7. The combination of the two complexity results shows that, with respect to the parameter $|q|$, string equality considerably increases the parameterized complexity: Boolean regex CQ evaluation used to be FPT, and is now W[1]-hard.

Finally, note that queries as in Theorems 5.1 and 5.2 are always acyclic, as the variables of each equality atom also occur in α. Although they are not gamma acyclic, we can rewrite each query into an equivalent gamma-acyclic query with k-ary string equalities, by merging each pair $\zeta^{=}_{X}$ and $\zeta^{=}_{Y}$ with sets $X \cap Y \neq \emptyset$ into $\zeta^{=}_{X \cup Y}$ until all equalities range over pairwise disjoint sets.

5.2 Upper Bounds

We now examine the upper bounds for our two evaluation strategies on regex UCQs with string equalities. For the canonical relational approach, we observe that each equality atom corresponds to a relation that is of polynomial size. Hence, we can directly use Theorem 3.5 to conclude the following.

Corollary 5.3. *Let* $\mathbf{P}$ *be a class of regex UCQs with string equalities. If the regex formulas in the UCQs of* $\mathbf{P}$ *belong to a polynomially bounded class, and* $\mathbf{P}$ *maps to a tractable class of relational UCQs, then UCQs in* $\mathbf{P}$ *can be evaluated in polynomial total time.*

The upper bound for compilation to automata requires more effort. We first observe the following.

Theorem 5.4. *For every fixed* $m \geq 1$ *there is an algorithm, that given a functional vset-automaton* A*, a sequence* $S := \zeta^{=}_{x_1,y_1} \cdots \zeta^{=}_{x_m,y_m}$ *of string equality selections over* $\mathsf{Vars}(A)$*, and a string* $\mathbf{s}$*, enumerates* $[\![SA]\!](\mathbf{s})$ *with polynomial delay.*

The main idea of the proof is to construct a vset-automaton A_{eq} that defines exactly the string equalities on $\mathbf{s}$. Specifically, $\mu \in [\![A_{\mathsf{eq}}]\!](\mathbf{s})$ holds if and only if $\mathbf{s}_{\mu(x_i)} = \mathbf{s}_{\mu(y_i)}$ for all x_i, y_i where $\zeta^{=}_{x_i,y_i}$ is a selection in S. We then use Lemma 3.10 to construct a vset-automaton A_{join} with $[\![A_{\mathsf{join}}]\!] = [\![A \bowtie A_{\mathsf{eq}}]\!]$, and use Theorem 3.3 to enumerate $[\![A_{\mathsf{join}}]\!](\mathbf{s}) = [\![SA]\!](\mathbf{s})$. Note that the construction of A_{eq} depends on $\mathbf{s}$; in fact, $[\![A_{\mathsf{eq}}]\!](\mathbf{s}') = \emptyset$ for all strings $\mathbf{s}' \neq \mathbf{s}$. This dependency on $\mathbf{s}$ is unavoidable: Recall that as shown by Fagin et al. [12], regular spanners are strictly less expressive than core spanners (i. e., string equality adds expressive power). If we could construct an A_{eq} that worked on every string $\mathbf{s}'$, this would immediately lead to a contradiction.

Analogously to the join in Section 3.3.3, the polynomial upper bound only holds for the construction if m is fixed, and Theorem 5.2 suggests that this cannot be overcome under standard assumptions. Hence, for all fixed $k, m \geq 0$, we define the notion of a *regex k-UCQ with up to m string equalities* analogously to a regex k-UCQ, with the additional requirement that each of the CQs uses at most m binary string equality selections. It follows from Theorem 5.4 that for every constant k, adding a fixed number of string equalities to a regex k-UCQ does not affect its enumeration complexity (in the sense that the complexity stays polynomial).

Corollary 5.5. *For all fixed m and k, regex k-UCQs with up to m string equalities can be evaluated with polynomial delay.*

In other words, Theorem 3.11 can be extended to cover string equalities.

6. CONCLUDING REMARKS

We have studied the combined complexity of evaluating regex CQs and regex UCQs. We showed that the complexity is not determined only by the structure of the CQ as a relational query; rather, complexity can go higher since an atomic regex formula can already define a relation that is exponentially larger than the combined size of the input and output. Our upper bounds are based on an algorithm for evaluating a vset-automaton with polynomial delay. These bounds are based on two altenative evaluation strategies—the canonical relational evaluation and query-to-automaton compilation. We conclude the paper by proposing several directions for future research.

One direction is to generalize the compilation into a vset-automaton into a class of queries that is more general than regex k-UCQs with string equality. In particular, we would like to have a robust definition of a class of algebraic expressions that we can efficiently translate into a vset-automaton.

Fagin et al. [12] have shown that regular spanners, or equivalently regex UCQs (as we established in Theorem 2.8), can be phrased as *Unions of Conjunctive Regular Path Queries* (UCRPQs) [4,6,8,9]. However, their translation entails an exponential blowup. Moreover, even if the translation could be made efficiently, it is not at all clear that tractability properties of the regex UCQ (e.g., bounded number of atoms, or low hypertree width) would translate into tractable properties of UCRPQs. Importantly, in the UCRPQ every atom involves two variables, and so, the problem of intractable materialization of an atom (i.e., the main challenge we faced here) does not occur. So, another future direction is a deeper exploration of the relationship between the complexity results of the two frameworks.

Last but not least, there is the crucial future direction of translating the upper bounds we presented into algorithms that substantially outperform the state of the art, at least when our tractability conditions hold. Beyond optimizing our translation and polynomial-delay algorithm, we would like to incorporate techniques of aggressive filtering for matching regular expressions [40, 41] and parallelizing polynomial-delay enumeration [41].

7. REFERENCES

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

[2] M. Ackerman and J. Shallit. Efficient enumeration of words in regular languages. *Theor. Comput. Sci.*, 410(37):3461–3470, 2009.

[3] J. Ajmera, H.-I. Ahn, M. Nagarajan, A. Verma, D. Contractor, S. Dill, and M. Denesuk. A CRM system for social media: challenges and experiences. In *WWW*, pages 49–58. ACM, 2013.

[4] P. Barceló, M. Romero, and M. Y. Vardi. Semantic acyclicity on graph databases. *SIAM J. Comput.*, 45(4):1339–1376, 2016.

[5] E. Benson, A. Haghighi, and R. Barzilay. Event discovery in social media feeds. In *ACL*, pages 389–398. The Association for Computer Linguistics, 2011.

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

[7] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90. ACM, 1977.

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

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

[10] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[11] R. Fagin. Acyclic database schemes (of various degrees): A painless introduction. In *CAAP*, pages 65–89. Springer, 1983.

[12] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12, 2015.

[13] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Declarative cleaning of inconsistencies in information extraction. *ACM Trans. Database Syst.*, 41(1):6, 2016.

[14] H. Fernau, M. L. Schmid, and Y. Villanger. On the parameterised complexity of string morphism problems. *Theory Comput. Syst.*, 59(1):24–51, 2016.

[15] D. D. Freydenberger. A logic for document spanners. In *ICDT*, pages 13:1–13:18, 2017. Full version available at `http://ddfy.de/publications/F-ALfDS.html`.

[16] D. D. Freydenberger and M. Holldack. Document spanners: From expressive power to decision problems. *Theory Comput. Syst.*, pages 1–45, 2017.

[17] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*, pages 149–158. IEEE Computer Society, 2009.

[18] G. Gottlob, Z. Miklós, and T. Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM*, 56(6), 2009.

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

[20] M. Grohe and J. Flum. *Parameterized Complexity Theory*. Theoretical Computer Science. Springer, 2006.

[21] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, 194:28–61, 2013.

[22] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[23] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988.

[24] 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.

[25] Y. Li, F. Reiss, and L. Chiticariu. SystemT: A declarative information extraction system. In *ACL*, pages 109–114. ACL, 2011.

[26] B. Liu, L. Chiticariu, V. Chu, H. V. Jagadish, and F. Reiss. Automatic rule refinement for information extraction. *PVLDB*, 3(1):588–597, 2010.

[27] K. Losemann. *Foundations of Regular Languages for Processing RDF and XML*. PhD thesis, University of Bayreuth, 2015.

[28] A. Morciano, M. Ugarte, and S. Vansummeren. Automata-based evaluation of AQL queries. Technical report, Université Libre de Bruxelles, 2016.

[29] Y. Nahshon, L. Peterfreund, and S. Vansummeren. Incorporating information extraction in the relational database model. In *WebDB*, page 6. ACM, 2016.

[30] C. H. Papadimitriou and M. Yannakakis. On the complexity of database queries. *J. Comput. Syst. Sci.*, 58(3):407–427, 1999.

[31] 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 Computer Society, 2008.

[32] C. D. Sa, A. Ratner, C. Ré, J. Shin, F. Wang, S. Wu, and C. Zhang. Deepdive: Declarative knowledge base construction. *SIGMOD Record*, 45(1):60–67, 2016.

[33] M. L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Inform. Comput.*, 249:1–17, 2016.

[34] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, pages 1033–1044. ACM, 2007.

[35] J. Shin, S. Wu, F. Wang, C. D. Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using deepdive. *PVLDB*, 8(11):1310–1321, 2015.

[36] S. S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, 2nd edition, 2008.

[37] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A core of semantic knowledge. In *WWW*, pages 697–706. ACM, 2007.

[38] D. J. Walker, D. E. Clements, M. Darwin, and J. W. Amtrup. Sentence boundary detection: A comparison of paradigms for improving MT quality. In *MT Summit VIII*, pages 18–22. EAMT, 2001.

[39] H. Xu, S. P. Stenner, S. Doan, K. B. Johnson, L. R. Waitman, and J. C. Denny. MedEx: a medication information extraction system for clinical narratives. *JAMIA*, 17(1):19–24, 2010.

[40] X. Yang, T. Qiu, B. Wang, B. Zheng, Y. Wang, and C. Li. Negative factor: Improving regular-expression matching in strings. *ACM Trans. Database Syst.*, 40(4):25, 2016.

[41] X. Yang, B. Wang, T. Qiu, Y. Wang, and C. Li. Improving regular-expression matching on strings using negative factors. In *SIGMOD*, pages 361–372. ACM, 2013.

[42] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94. IEEE Computer Society, 1981.

[43] A. Yates, M. Banko, M. Broadhead, M. J. Cafarella, O. Etzioni, and S. Soderland. TextRunner: Open information extraction on the web. In *ACL-HLT*, pages 25–26. ACL, 2007.

[44] H. Zhu, S. Raghavan, S. Vaithyanathan, and A. Löser. Navigating the intranet with high precision. In *WWW*, pages 491–500. ACM, 2007.

APPENDIX

A. PROOFS

A.1 Proof of Theorem 2.8

THEOREM 2.8. *The following hold.*

- *The class of spanners expressible as regex UCQs is that of the regular spanners.*
- *The class of spanners expressible as regex UCQs with string equalities is that of the core spanners.*

Proof. We begin with the first claim, that regex UCQs express exactly $[\![\mathsf{RGX}^{\{\pi,\cup,\bowtie\}}]\!]$, the class of regular spanners. First, note that regex UCQs are a special case of regular spanner representations. For the other direction, the proof of Lemma 4.13 by Fagin et al. [12] shows that every vset-automaton can be converted into an equivalent a union of projections over joins of regex formulas – or, in our terminology, a UCQ.

For the second claim, we also only need to proof that every core spanner can be expressed by a regex UCQ with string equalities. For this, we use the core-simplification lemma (Lemma 4.19 in [12]), which states that every core spanner P can be expressed as $\pi_Y SA$, where A is a vset-automaton, S is a sequence of string equality selections over $V := \mathsf{Vars}(A)$, and $Y \subseteq V$. We then use the previous result to convert A into an equivalent regex UCQ $q := \bigcup_{i=1} q_i$, and define the regex UCQ with string equalites $q' := \bigcup_{i=1} q'_i$, where each q'_i is obtained by adding π_Y and S to q_i. Obviously, q' and P are equivalent. □

A.2 Proof of Theorem 3.3

THEOREM 3.3. *Given a functional vset-automaton A with n states and m transitions, and a string $\mathbf{s}$, one can enumerate $[\![A]\!](\mathbf{s})$ with polynomial delay of $O(n^2|\mathbf{s}|)$, following a polynomial preprocessing of $O(n^2|\mathbf{s}| + mn)$.*

Proof. Let $\mathbf{s} = \sigma_1 \cdots \sigma_N$ with $N := |\mathbf{s}|$ and $\sigma_i \in \mathbf{\Sigma}$. Let $A = (X, Q, \delta, q_0, q_f)$ be a functional vset-automaton. Without loss of generality, we assume that each state is reachable from q_0, and q_f can be reached from every state in Q. Let $n := |Q|$, $v := |X|$, and let m denote the number of transitions of A.

This proof is structured as follows: First, we consider some preliminary considerations. In particular, we discuss how every functional vset-automaton can be simulated using an NFA together with a function that represents how the vset-automaton operates on variables. Building on this, we then give the actual enumeration algorithm.

Prelimary Considerations: As A is functional, we know that for all $q \in Q$ and all ref-words $\mathbf{r}_1, \mathbf{r}_2$ with $q \in \delta(q_0, r_1)$ and $q \in \delta(q_0, r_2)$, $|\mathbf{r}_1|_a = |\mathbf{r}_2|_a$ holds for all $a \in \mathbf{\Gamma}_X = \{x\vdash, \dashv x \mid x \in X\}$. Furthermore, we know that each ref-word that is accepted by A is also valid (i.e., for each $x \in X$, the ref-word contains each of $x\vdash$ and each $\dashv x$ exactly once, and in the correct order). In other words, in each state $q \in Q$ and for each variable $x \in X$, it is well-defined whether x has not been opened yet, or has been opened, but not closed, or has been opened and closed. We formalize this as follows: First, define the set $\mathcal{V}$ of *variable states* as $\mathcal{V} := \{\mathtt{w}, \mathtt{o}, \mathtt{c}\}$, where $\mathtt{w}$ stands for *waiting*, $\mathtt{o}$ for *open*, and $\mathtt{c}$ for *closed*. A *variable configuration* is a function $\vec{c}\colon X \to \mathcal{V}$, which maps variables to variable states. For each $q \in Q$, we choose any ref-word $\mathbf{r}_q$ sucht that $q \in \delta(q_0, \mathbf{r}_q)$, and use this to define $\vec{c}_q\colon X \to \mathcal{V}$, the *variable configuration of the state q*, as follows:

$$\vec{c}_q(x) := \begin{cases} \mathtt{w} & \text{if } \mathbf{r}_q \text{ contains neither } x\vdash, \text{ nor } \dashv x, \\ \mathtt{o} & \text{if } \mathbf{r}_q \text{ contains } x\vdash, \text{ but not } \dashv x, \\ \mathtt{c} & \text{if } \mathbf{r}_q \text{ contains } x\vdash \text{ and } \dashv x. \end{cases}$$

Due to A being functional (and due to our initial assumption that no state of A is redundant), no other cases need to be considered, and $\vec{c}_q$ is well-defined for every $q \in Q$. We observe that $\vec{c}_{q_0}(x) = \mathtt{w}$ and $\vec{c}_{q_f}(x) = \mathtt{c}$ holds for all $x \in X$.

The *variable configuration of the automaton A* is the function $\vec{C}$ that maps each $q \in Q$ to its variable configuration $\vec{c}_q$, which allows us to interpret $\vec{C}$ as a function $\vec{C}\colon Q \times X \to \mathcal{V}$ by $\vec{C}(q, x) = \vec{c}_q(x)$. To see how A operates on variables when changing from a state p to a state q, it suffices to compare $\vec{c}_p$ and $\vec{c}_q$. We now use this to define an ϵ-NFA A_N over $\mathbf{\Sigma}$ that, when combined with $\vec{C}$, simulates A. Let $A_N := (Q, \delta_N, q_0, q_f)$, where δ_N is obtained from δ by defining, for all $q \in Q$, $\delta_N(q, \sigma) := \delta(q, \sigma)$ for all $\sigma \in \mathbf{\Sigma}$, as well as

$$\delta_N(q, \epsilon) := \delta(q, \epsilon) \cup \bigcup_{a \in \mathbf{\Gamma}_X} \delta(q, a).$$

In other words, A_N is obtained from A by replacing each transition that has a variable operation with an ϵ-transition. Now, there is a one-to-one-correspondence between paths in A_N and A. This also follows from the fact that A is

functional: In a general vset-automaton, it would be possible to have two states p and q such that there are two transitions with different variable operations from p to q (e. g. $x\vdash$ and $\dashv x$, recall Example 2.6). But in functional automata, this is not possible (we already used this insight to define variable configurations).

Furthermore, for every accepting run of A_N on $\mathbf{s}$, we can use $\vec{C}$ to derive a $\mu \in [\![A]\!](\mathbf{s})$ that corresponds to this run. Due to the ϵ-transitions, this requires some additional effort. We begin with a definition: For each $q \in Q$, define $\mathcal{E}(q)$, the ϵ-closure of q, as the set of all $p \in Q$ that can be reached from q by using only ϵ-transitions (including q itself). Note that $\mathcal{E}$ can also be derived directly from A, by defining $\mathcal{E}(q)$ as the set of all p that can be reached from q by using only ϵ-transitions and variable transitions.

Now, assume $\mathbf{s} \in \mathcal{L}(A_N)$, and recall that $\mathbf{s} = \sigma_1 \cdots \sigma_N$ with $N \geq 0$. Then there exists a sequence of states $q_0, \hat{q}_0, \ldots, q_N, \hat{q}_N \in Q$ such that

1. $\hat{q}_i \in \mathcal{E}(q_i)$ for $0 \leq i \leq N$,
2. $q_{i+1} \in \delta_N(\hat{q}_i, \sigma_{i+1})$ for $0 \leq i < N$,
3. $\hat{q}_l = q_f$.

In other words, each q_i is the state that is entered immediately after reading $\mathbf{s}_i$, and $\hat{q}_i$ is the state immediately before $\mathbf{s}_{i+1}$ is read. Hence, $\hat{q}_{i+1}$ is reached from $\hat{q}_i$ solely using ϵ-transitions (which correspond to variable operations or to ϵ-transitions in A). Hence, in this point of view on runs, we only focus on the parts of an accepting run immediately before and after processing terminals. Take note that it is possible that $q_i = \hat{q}_i$, which corresponds to not operating on variables between reading σ_i and σ_{i+1}.

We now discuss how this can be used to derive a $(X, \mathbf{s})$-tuple μ. When introducing variable configurations, we already remarked that $\vec{c}_{q_0}(x) = \mathtt{w}$ and $\vec{c}_{q_f}(x) = \mathtt{c}$ must hold for all $x \in X$. The latter immediately implies $\vec{c}_{\hat{q}_n} = \mathtt{c}$. Furthermore, as terminal transitions cannot change variable configurations, $\vec{c}_{\hat{q}_i} = \vec{c}_{q_{i+1}}$ must hold as well. This allows us to define μ solely by considering the $\vec{c}_{\hat{q}_i}$. As A is functional, for each $x \in X$, there exist i and j with $0 \leq i \leq j \leq N$ such that $\vec{c}_{\hat{q}_{i'}} = \mathtt{w}$ for all $i' < i$ and $\vec{c}_{\hat{q}_{j'}} = \mathtt{c}$ for all $j' \geq j$. Less formally: In the variable configurations of the states $\hat{q}_0, \hat{q}_1, \ldots, \hat{q}_n$, there is first a (possibly empty) block of states where x is waiting, followed by a (possibly empty) block where x is open, and finally a non-empty block where x is closed. (The block of open configurations is empty if and only if x is opened and closed without reading a terminal letter). Hence, for every $x \in X$, there exist

1. a minimal i, such that $\vec{c}_{\hat{q}_i} \neq \mathtt{w}$,
2. a minimal j, such that $\vec{c}_{\hat{q}_j} = \mathtt{c}$.

We then define $\mu(x) := [i+1, j+1\rangle$. The intuition behind this is as follows: The choice of i means that in A, the operation $x\vdash$ occurs somewhere between q_i and $\hat{q}_i$. Hence, the first position in $\mu(x)$ must belong to the next terminal, which is $\mathbf{s}_{i+1}$. Likewise, the choice of j means that in A, the operation $\dashv x$ occurs somewhere between q_j and $\hat{q}_j$, which means that $\mathbf{s}_{j+1}$ must be the first letter after $\mu(x)$.

Thus, together with $\vec{C}$, every accepting run of A_N for a string $\mathbf{s}$ can be interpreted as a $\mu \in [\![A]\!](\mathbf{s})$. As every accepting run in A corresponds to an accepting run of A_N, in order to enumerate $[\![A]\!](\mathbf{s})$, it suffices to enumerate all accepting runs of A_N. Take particular note that, instead of considering the states $\hat{q}_i$, it suffices to consider their variable configurations $\vec{c}_{\hat{q}_i}$ in order to define μ. Hence, a $(X, \mathbf{s})$-tuple μ can be derived from a sequence of $|\mathbf{s}| + 1$ variable configurations. In fact, this approach also works in the reverse direction: Every $(X, \mathbf{s})$-tuple μ can be used to derive such a sequence, simply be examining how the variables are opened and closed with respect to the positions of $\mathbf{s}$.

Enumeration Algorithm: The algorithm for the enumeration of $[\![A]\!](\mathbf{s})$ has two steps: First, we use the NFA A_N to construct a directed acyclic graph G that can be interpreted as an NFA A_G such that $\mathcal{L}(A_G)$ encodes exactly the accepting runs of A on $\mathbf{s}$. Second, we use the algorithm enumerate (Algorithm 1 below) to enumerate all elements of $\mathcal{L}(A_G)$, and thereby all of $[\![A]\!](\mathbf{s})$. Take particular note that the alphabet of A_G is the set of all $\vec{c}_q$ with $q \in Q$, and not $\mathbf{\Sigma}$.

We begin with the first step, computing the function $\vec{C}$, which maps each $q \in Q$ to its variable configuration $\vec{c}_q$. As A is functional, all ref-words that label paths which lead from q_0 to a state q contain exactly the same symbols from $\mathbf{\Gamma}_X$. Hence, $\vec{C}$ can be computed by executing a breadth-first-search that, whenever it proceeds from a state p to a state q, derives $\vec{c}_q$ from $\vec{c}_p$ (if the transition opens or closes a variable x, $\vec{c}_p(x)$ is set to $\mathtt{o}$ or $\mathtt{c}$, respectively; otherwise, both variable configurations are identical). As breadth-first-search is possible in $O(m+n)$, this step runs in time $O(vm + vn)$. Note that this process can also be used to check whether A is functional (this is the idea of the proof of Theorem 2.7, cf. [15]).

Next, we compute the ϵ-closure as a function $\mathcal{E} \colon Q \to 2^Q$ directly from A. Using a standard transitive closure algorithm (see e. g. Skiena [36]), $\mathcal{E}$ can be computed in time $O(n(n+m))$, which we can assume to be $O(mn)$. As $v \leq n \leq m$, this dominates the complexity of computing the variable configurations.

Instead of directly computing G, we first construct a graph $G' := (V', E')$ from A_N, where $V' := \bigcup_{i=0}^{N} V'_i$ with

$$V'_0 := \{(0, q) \mid q \in \mathcal{E}(q_0)\},$$
$$V'_{i+1} := \{(i+1, q) \mid q \in \mathcal{E}(\delta_N(p, \sigma_{i+1})) \text{ for some } (i, p) \in V'_i\}$$

for all $0 \leq i < n$. In a slight abuse of notation, we restrict V'_N to $V'_N \cap \{(N, q_f)\}$. Furthermore, without loss of generality, we can assume that $V'_N = \{(N, q_f)\}$, as otherwise, $[\![A]\!](\mathbf{s}) = \emptyset$. Less formally explained, V'_i contains all (i, q) such that q can act as $\hat{q}_i$ when using A_N and $\vec{C}$ to simulate A as explained above.

Following this idea, we define E' to simulate these transitions; in other words, we define $E' := \bigcup_{0 \leq i < N} E_i$, where

$$E'_i := \{\big((i, p), (i+1, q)\big) \mid (i, p) \in V'_i \text{ and } q \in \mathcal{E}(\delta(p, \sigma_{i+1}))\}.$$

Observe that $|V'| \leq Nn + 1$, as $|V'_i| \leq N$ for $0 \leq i < N$. Also, as each node of some V'_i with $i < N$ has at most n outgoing edges, $|E'| \leq Nn^2$. Next, we obtain $G := (V, E)$ by removing from G' all nodes (and associated edges) from which (n, q_f) cannot be reached. Using standard reachability algorithms, this is possible in time $O(|V'| + |E'|) = O(Nn^2)$. For each V'_i, we define $V_i := V'_i \cap V$.

Now, every path π from a node $(0, q)$ to (N, q_f) in G corresponds to an accepting run of A_N (and for every accepting run of A_N, there is a corresponding path in G). Hence, if we consider a path $\pi = ((0, \hat{q}_0), (1, \hat{q}_1), \ldots, (N, \hat{q}_N))$ in G (which implies $\hat{q}_N = q_N$), we can use the sequence of variable configurations $\vec{c}_{\hat{q}_0}, \vec{c}_{\hat{q}_1}, \ldots, \vec{c}_{\hat{q}_N}$ to define a $\mu^\pi \in [\![A]\!](\mathbf{s})$.

While this allows us to enumerate all elements of $[\![A]\!](\mathbf{s})$, two different paths π_1 and π_2 in G might lead to the same sequence of variable configurations (which would imply $\mu^{\pi_1} = \mu^{\pi_2}$), which means that G alone cannot be used to compute $[\![A]\!](\mathbf{s})$ with polynomial delay.

The crucial part of the next step is interpreting the set of variable configurations of states as an alphabet $\mathcal{K} = \{\vec{c}_q \mid q \in Q\}$, and treating sequences of variable configurations as strings over $\mathcal{K}$. Then G defines the language of all variable configurations that correspond to an element of $[\![A]\!](\mathbf{s})$. By adding a starting state q_0, we turn G into an NFA A_G over $\mathcal{K}$ that accepts exactly this language. More specifically, we define $A_G := (Q_G, \delta_G, q_0, F_G)$, where

- $Q_G := V \cup \{q_0\}$,
- $F_G := V_N = \{(N, q_f)\}$,
- δ_G is defined in the following way:
  - for each $(0, q) \in V_0$, δ_G has a transition from q_0 to q with label $\vec{c}_q$,
  - for each transition $((i, p), (i+1, q)) \in E$, δ_G has a transition from p to q with label $\vec{c}_q$.

We observe that all incoming transitions of a state (i, q) are labeled with the same terminal letter $\vec{c}_q$. Furthermore, although $|X|$ variables allow for $3^{|X|}$ distinct variable configurations, only at most $|Q|$ of these appear in A_G.

As there is a one-to-one correspondence between strings of $\mathcal{L}(A_G)$ and elements of $[\![A]\!](\mathbf{s})$, we now want to enumerate all elements of $\mathcal{L}(A_G)$. To do so, we introduce a variant of the general framework by Ackerman and Shallit [2] for algorithms that, given an NFA, enumerate all strings in its language that are of length N. Note that the restricted form of A_G allows us to ignore the intricacies of the algorithms that are compared in [2], which also leads to a lower complexity.

Before we proceed to the actual enumeration algorithm, we introduce another definition. We fix a total order $<_\mathcal{K}$ on $\mathcal{K}$, and extend this to the *radix order* $<_\mathsf{r}$ on $\mathcal{K}^*$ as follows: Given $\mathbf{u}, \mathbf{v} \in \mathcal{K}^*$, we define $\mathbf{u} <_\mathsf{r} \mathbf{v}$ if $|\mathbf{u}| < |\mathbf{v}|$, or if $|\mathbf{u}| = |\mathbf{v}|$, and there exist $\mathbf{p}, \mathbf{s_u}, \mathbf{s_v} \in \mathcal{K}^*$ and $\kappa_\mathbf{u}, \kappa_\mathbf{v} \in \mathcal{K}$ with $\mathbf{u} = \mathbf{p} \cdot \kappa_\mathbf{u} \cdot \mathbf{s_u}$, $\mathbf{v} = \mathbf{p} \cdot \kappa_\mathbf{v} \cdot \mathbf{s_v}$, and $\kappa_\mathbf{u} <_\mathcal{K} \kappa_\mathbf{v}$.

As all strings in $\mathcal{L}(A_G)$ have length $N + 1$, we write them as $\kappa_0 \kappa_1 \cdots \kappa_N$, with $\kappa_i \in \mathcal{K}$. These indices correspond to the level numbers that are encoded in the states of Q_G, i. e., a letter κ_i takes a state $(i-1, p)$ to a state (i, q). Consequently, $\kappa_N = \vec{c}_{q_f}$ for all strings of $\mathcal{L}(A_G)$.

The enumeration algorithm uses a global stack that stores sets $S_i \subset Q_G$, which we call the *state stack*. In fact, we shall see that $S_{i+1} \subseteq \{(i, q) \mid q \in Q\}$ shall hold. Intuitively, if the algorithm has constructed a string $\kappa_0 \cdots \kappa_i$, the state stack shall store sets $S_0, \ldots, S_{i+1}$ such that each S_{j+1} contains the states that can be reached from the states of S_j by processing the letter κ_j. As every state of A_G is connected to the final state, and as $\kappa_N = \vec{c}_{q_f}$ must hold, the algorithm does not put S_{N-1} or S_N on the state stack.

Let $\bot \notin \mathcal{K}$ be a new letter, and define $\kappa <_\mathcal{K} \bot$ for all $\kappa \in \mathcal{K}$. The enumeration algorithm uses the functions $\mathsf{minLetter}\colon Q_G \to \mathcal{K}$ and $\mathsf{nextLetter}\colon Q_G \times \mathcal{K} \to \mathcal{K} \cup \{\bot\}$ as subroutines, which we define as follows for all $q \in Q_G$ and $c \in \mathcal{K}$:

- $\mathsf{minLetter}(q)$ is the $<_\mathcal{K}$-smallest $\kappa \in \mathcal{K}$ with $\delta_G(q, c) \neq \emptyset$,
- $\mathsf{nextLetter}(q, \kappa)$ is the smallest $\kappa' \in \mathcal{K}$ with $\kappa <_\mathcal{K} \kappa'$, and $\delta_G(q, \kappa') \neq \emptyset$; or $\bot$, if no such κ' exists.

It is easily seen that these functions can be precomputed in time $O(n^2 N)$, ideally when computing A_G. The actual enumeration algorithm, $\mathsf{enumerate}$, is given as Algorithm 1 below.

Algorithm 1: enumerate

```
S_0 := {q_0};
k = minString(0);
while k ≠ ⊥ do
    output k;
    k = nextString(k);
```

The main idea is very simple: enumerate calls minString (Algorithm 2) to construct the $<_{\mathsf{r}}$-smallest string of $\mathcal{L}(A_G)$, using the minLetter functions. Starting at $S_0 = \{q_0\}$, minString determines $\kappa_0 \cdots \kappa_N$ by first chosing κ_0 as the $<_{\mathcal{K}}$-smallest letter that can be read when in q_0, and then computing S_1 as all states that can be reached from there by reading κ_0. For each S_i, the algorithm then applies nextLetter to each state of S_i, chooses κ_i as the $<_{\mathcal{K}}$-smallest of these letters, and puts all states that can be reached by reading κ_i when in a state in S_i into the set S_{i+1}. In other words, constructing the sets S_i can be understood as an on-the-fly simulation of the power-set construction on A_G.

As all strings of $\mathcal{L}(A_G)$ have length $N+1$, and the final state of A_G is reachable from every state, this is sufficient to determine the $<_{\mathsf{r}}$-minimal string of the language.

To enumerate all further strings, enumerate uses the subroutine nextString (Algorithm 3 below) repeatedly. Given a string $\mathbf{k} = \kappa_0 \cdots \kappa_N$, nextString finds the rightmost letter of $\mathbf{k}$ that can be increased (according to $<_{\mathcal{K}}$), and changes this κ_i accordingly. It then calls minString to complete the string $<_{\mathsf{r}}$-minimally, by finding letters $\kappa_{i+1}, \ldots, \kappa_N$ that are $<_{\mathcal{K}}$-minimal. While doing so, minString updates the state stack, according to the computed letters.

Algorithm 2: minString(l)

Input : an integer l with $0 \leq l \leq N$
Assumptions: state stack contains $S_0, \ldots, S_l$ for some $\mathbf{p} \in \mathcal{K}^l$
Output : the $<_{\mathsf{r}}$-smallest $\mathbf{k} \in \mathcal{K}^{N-l}$ that is accepted by A_G when starting in a state of S_l
Side effects : updates the state stack to $S_0, \ldots, S_{N-1}$ for $\mathbf{p} \cdot \mathbf{k}$

```
for i := l to N − 1 do
    find q ∈ S_i such that minLetter(q) is <_K-minimal;
    κ_i := minLetter(q);
    if i < N − 1 then
        S_{i+1} := ⋃_{p∈S_i} δ_G(p, κ_i);
        push S_{i+1} on state stack;
κ_N = c⃗_{q_f};
return κ_l ··· κ_{N−1} · κ_N;
```

Algorithm 3: nextString(w)

Input : a string $\mathbf{k} = \kappa_0 \cdots \kappa_N$, $\kappa_i \in \mathcal{K}$
Assumptions: state stack contains $S_0, \ldots, S_{N-1}$
for **k Output** : the $<_{\mathsf{r}}$-smallest word $\mathbf{k}' \in \mathcal{L}(A_G)$ with $\mathbf{k} <_{\mathsf{r}} \mathbf{k}'$, or $\perp$ if no such $\mathbf{k}'$ exists
Side effects : updates the state stack to $S'_0, \ldots, S'_{N-1}$ for $\mathbf{k}'$

```
for i := N − 1 to 0 do
    let κ_i be the <_K-minimal element of {nextLetter(q, κ_i) | q ∈ S_i};
    if κ_i = ⊥ then pop S_i from the state stack;
    else
        if i < N − 1 then
            S_{i+1} := ⋃_{p∈S_i} δ_G(p, κ_i);
            push S_{i+1} on state stack;
        return κ_0 ··· κ_i · minString(i + 1);
return ⊥;
```

We now determine the complexity of the sub-routines. We begin with minString: The for-loop is executed $O(N)$

times. In each iteration of the loop, q can be determined in time $O(n)$ by using the precomputed minLetter-function, and κ_i can then be computed in $O(1)$. To build the set S_{i+1}, we need time $O(n^2)$. Hence, a call of minString takes at most time $O(Nn^2)$.

In each call of nextString, the for-loop is executed $O(N)$ times. In each iteration, κ_i can be found in time $O(n)$, as nextLetter was precomputed. Furthermore, if $\kappa_i \neq \bot$, the subroutine updates the state stack in $O(n^2)$ and terminates after a single call of minString, which takes $O(Nn^2)$. Hence, the total complexity of nextString is $O(Nn + n^2 + Nn^2)$, or simply $O(Nn^2)$.

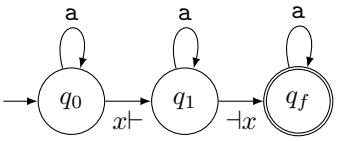
The total complexity of the preprocessing is obtained by adding up the complexity of computing $\mathcal{E}$ (and $\vec{C}$) of $O(mn)$, constructing G and A_G in $O(Nn^2)$, computing minLetter and nextLetter in $O(Nn^2)$, and a call of minString in $O(Nn^2)$. This adds up to $O(Nn^2 + mn)$ for the preprocessing and finding the first answer. Furthermore, as each call of nextString takes $O(Nn^2)$, the delay is clearly polynomial.

Recall that by Lemma 3.4, m is in $O(n)$ if we derived A from a regex formula. Hence, for regex formulas, the preprocessing is in $O(Nn^2)$ as well.

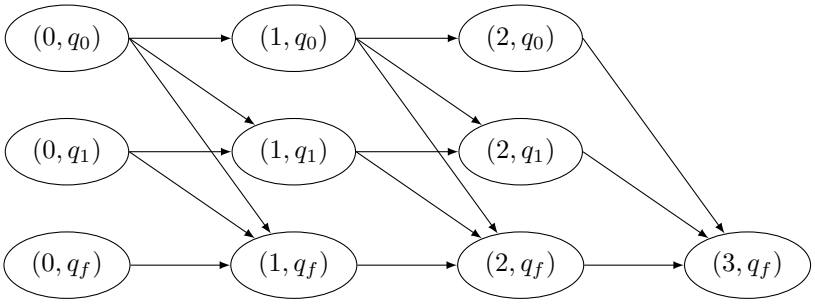
Finally, note that if A_G is deterministic, the complexity of the delay drops to $O(N)$: As each S_i can contain at most state (which can have at most one letter), both minString and nextString can be computed in $O(N)$. □

A.3 Examples for Theorem 3.3

Example A.1. Consider the following vset-automaton A for the functional regex formula $\mathtt{a}^* \, x\{\mathtt{a}^*\} \, \mathtt{a}^*$, where $q_0 = 0$ and $q_f = 2$:



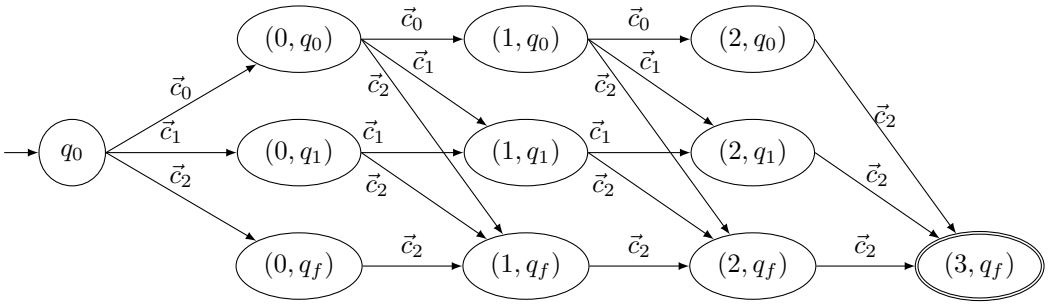
Then A is functional, and the variable configurations are defined as $\vec{c}_0(x) = \mathtt{w}$, $\vec{c}_1(x) = \mathtt{o}$, and $\vec{c}_2(x) = \mathtt{c}$. The NFA A_N is obtained by replacing the each of the labels $x\vdash$ and $\dashv x$ with ϵ. We now consider the input word $\mathtt{s} = \mathtt{aaa}$, and construct the corresponding graph G:



This allows us to obtain the following list of all $\mu \in [\![A]\!](\mathbf{s})$:

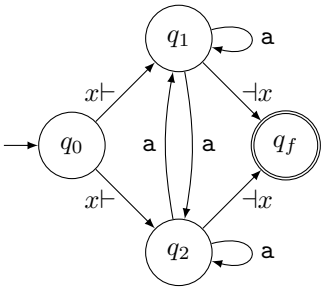
ref-word	$(\hat{q}_0, \hat{q}_1, \hat{q}_2, \hat{q}_3)$	$(\vec{c}_{\hat{q}_0}(x), \ldots, \vec{c}_{\hat{q}_3}(x))$	$\mu(x)$
$x\vdash \dashv x\mathtt{aaa}$	(q_f, q_f, q_f, q_f)	$(\mathtt{c}, \mathtt{c}, \mathtt{c}, \mathtt{c})$	$[1, 1\rangle$
$x\vdash \mathtt{a} \dashv x \, \mathtt{aa}$	(q_1, q_f, q_f, q_f)	$(\mathtt{o}, \mathtt{c}, \mathtt{c}, \mathtt{c})$	$[1, 2\rangle$
$x\vdash \mathtt{aa} \dashv x \, \mathtt{a}$	(q_1, q_1, q_f, q_f)	$(\mathtt{o}, \mathtt{o}, \mathtt{c}, \mathtt{c})$	$[1, 3\rangle$
$x\vdash \mathtt{aaa} \dashv x$	(q_1, q_1, q_1, q_f)	$(\mathtt{o}, \mathtt{o}, \mathtt{o}, \mathtt{c})$	$[1, 4\rangle$
$\mathtt{a}\, x\vdash \dashv x\mathtt{aa}$	(q_0, q_f, q_f, q_f)	$(\mathtt{w}, \mathtt{c}, \mathtt{c}, \mathtt{c})$	$[2, 2\rangle$
$\mathtt{a}\, x\vdash \mathtt{a} \dashv x \, \mathtt{a}$	(q_0, q_1, q_f, q_f)	$(\mathtt{w}, \mathtt{o}, \mathtt{c}, \mathtt{c})$	$[2, 3\rangle$
$\mathtt{a}\, x\vdash \mathtt{aa} \dashv x$	(q_0, q_1, q_1, q_f)	$(\mathtt{w}, \mathtt{o}, \mathtt{o}, \mathtt{c})$	$[2, 4\rangle$
$\mathtt{aa}\, x\vdash \dashv x\mathtt{a}$	(q_0, q_0, q_f, q_f)	$(\mathtt{w}, \mathtt{w}, \mathtt{c}, \mathtt{c})$	$[3, 3\rangle$
$\mathtt{aa}\, x\vdash \mathtt{a} \dashv x$	(q_0, q_0, q_1, q_f)	$(\mathtt{w}, \mathtt{w}, \mathtt{o}, \mathtt{c})$	$[3, 4\rangle$
$\mathtt{aaa}\, x\vdash \dashv x$	(q_0, q_0, q_0, q_f)	$(\mathtt{w}, \mathtt{w}, \mathtt{w}, \mathtt{c})$	$[4, 4\rangle$

The NFA A_G looks as follows:

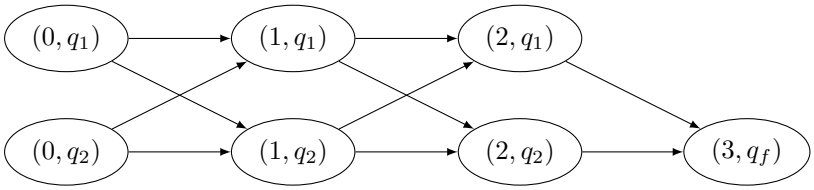


As $\vec{c}_i \neq \vec{c}_j$ if $i \neq j$, this NFA could actually be interpreted as a DFA. As a result, the running time of the enumeration algorithm is lower than in the worst case. ◊

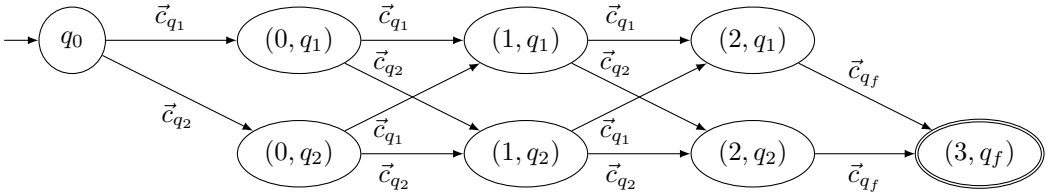
Example A.2. In order to examine a case where constructing A_D is necessary, consider the following functional vset-automaton A:



It is easily seen that A is functional, and the variable configurations are defined by $\vec{c}_{q_0}(x) = \mathtt{w}$, $\vec{c}_{q_1}(x) = \vec{c}_{q_2}(x) = \mathtt{o}$, and $\vec{c}_{q_f}(x) = \mathtt{c}$. Furthermore, for every $\mathbf{s} \in \mathtt{a}^*$, $[\![A]\!](\mathbf{s})$ contains only a single μ, with $\mu(x) = [1, |\mathbf{s}| + 1\rangle$, which corresponds to the ref-word $x\vdash \mathbf{s} \dashv x$. Nonetheless, for each such $\mathbf{s}$, there are $2^{|\mathbf{s}|}$ paths from q_0 to q_f in A. Now consider the case of $\mathbf{s} = \mathtt{a}^3$. The corresponding graph G looks as follows:



By adding a new starting state q_0, we construct the NFA A_G:



Now, note that A_G is not deterministic, as $\vec{c}_{q_1} = \vec{c}_{q_2}$. In fact, $\mathcal{L}(A_G)$ consists only of the word $(\vec{c}_{q_1})^3\vec{c}_{q_f}$, which corresponds to μ with $\mu(x) = [1, 4\rangle$, the only element of $[\![A]\!](\mathbf{s})$. ◊

A.4 Proof of Lemma 3.4

LEMMA 3.4. *Given a regex formula α, one can construct in time $O(|\alpha|)$ a functional vset-automaton A with $[\![A]\!] = [\![\alpha]\!]$.*

Proof. Let α be a regex formula, and define $V := \mathsf{Vars}(\alpha)$. Assume that α is represented as its syntax tree. We first rewrite α into a regular expression $\hat{\alpha}$ over the alphabet $\mathbf{\Sigma} \cup \mathbf{\Gamma}_V$ with $\mathcal{L}(\hat{\alpha}) = \mathcal{R}(\alpha)$. This is done by recursively

replacing every node that represents a variable binding $x\{\beta\}$ with the concatenation $x\vdash \cdot \beta \cdot \dashv x$. Then the length of $\hat{\alpha}$ is linear in the length of α; and the rewriting is possible in linear time as well.

Next, we convert $\hat{\alpha}$ into an ϵ-NFA A with $\mathcal{L}(A) = \mathcal{L}(\hat{\alpha})$. Using the Thompson construction (cf. e. g. [22]), this is possible in linear time. Furthermore, both the number of states and the number of transitions of A are linear in the length of $\hat{\alpha}$ (and, hence, also in the length of α). Finally, note that the construction ensures that A has only one accepting state (recall that vset-automata are required to have a single accepting state).

This allows us to interpret A as a vset-automaton with variable set V, using $\mathcal{R}(A) = \mathcal{L}(A)$. As $\mathcal{L}(A) = \mathcal{L}(\hat{\alpha}) = \mathcal{R}(\alpha)$ holds by definition, we know that $\mathcal{R}(A) = \mathcal{R}(\alpha)$. Furthermore, as α is functional, $\mathcal{R}(\alpha) = \mathsf{Ref}(\alpha)$ holds, which implies $\mathsf{Ref}(A) = \mathcal{R}(A) = \mathsf{Ref}(\alpha)$. This allows us to make two conclusions: Firstly, that $\mathsf{Ref}(A) = \mathcal{R}(A)$, hence A is functional. And, secondly, that $[\![A]\!] = [\![\alpha]\!]$, as $\mathsf{Ref}(A) = \mathsf{Ref}(\alpha)$ implies $\mathsf{Ref}(A, \mathbf{s}) = \mathsf{Ref}(\alpha, \mathbf{s})$ for all $\mathbf{s} \in \mathbf{\Sigma}^*$. □

A.5 Proof of Proposition 3.6

PROPOSITION 3.6. *Given a functional vset-automaton A with n states, and a variable $x \in \mathsf{Vars}(A)$, it can be decided in time $O(n^4)$ whether x is a key attribute.*

Proof. The proof uses a modification of the intersection construction for NFAs. Given a functional vset-automaton $A = (V, Q, q_0, q_f, \delta)$, our goal is to construct an NFA A_x such that $\mathcal{L}(A_x)$ is empty if and only if x is not a key attribute. More specifically, the language consists of all $\mathbf{s}$ for which there exist $\mu, \mu' \in [\![A]\!](\mathbf{s})$ such that $\mu(x) = \mu'(x)$ and $\mu(y) \neq \mu'(y)$ for some $y \in V$. Without loss of generality, we can assume that A has at least two variables, and that we consider only non-empty strings (as $[\![A]\!](\epsilon)$ contains at most one element). For our complexity analysis, let $n := |Q|$ and $v := |V|$.

Before we discuss the actual construction, we bring A into a more convenient form. As in the proof of Theorem 3.3, we use the fact that every state of A has a uniquely defined variable configuration $\vec{c}_q \colon X \to \mathcal{V}$ (also see Section 4). Hence, the first step of the construction is computing the variable configurations of A. Recall that, if A has n states and m transitions, this can be done in time $O(m + n)$. We can also assume that every state is reachable from q_0, and that q_f is reachable from each state. We now replace all variable transitions with ϵ-transitions, and obtain a new transition function δ' from δ by removing all ϵ-transitions. Using the standard procedure, this takes time $O(n^3)$ (in contrast to previous proofs, we do not need to use the tighter bound $O(mn)$).

The main idea of the construction is that A_x is an NFA over $\mathbf{\Sigma}$ that simulates two copies of A in parallel. Both copies have the same behavior on x, but different behavior on (at least) one witness variable y. In addition to this, A_x uses its states to keep track whether such a y has been found. Hence, we define $A_x := (Q_x, q_{0,x}, q_{f,x}, \delta_x)$, where

$$\begin{aligned} Q_x := &\{(0, q_1, q_2) \mid q_1, q_2 \in Q, \vec{c}_{q_1} = \vec{c}_{q_2}\} \\ &\cup \{(1, q_1, q_2) \mid q_1, q_2 \in Q, \vec{c}_{q_1}(x) = \vec{c}_{q_2}(x)\}. \end{aligned}$$

Intuitively, each state triple contains the state of each of the two copies of A, and a bit that encodes with 1 or 0 whether a witness y has been found or not (respectively). Hence, we require for all state pairs that their variable configurations on x are identical; but as long as no witness y has been found, the variable configurations of all other variables also have to be identical. Following this intuition, we define $q_{0,x} := (0, q_0, q_0)$ and $(1, q_f, q_f)$. Finally, we construct δ_x is as follows: For each pair of transitions $q_1 \in \delta'(p_1, \sigma)$ and $q_2 \in \delta'(p_2, \sigma)$ that satisfies $\vec{c}_{q_1}(x) = \vec{c}_{q_2}(x)$, we first define $(1, q_1, q_2) \in \delta_x((1, p_1, p_2), \sigma)$ (as the bit signifies that a witness y has been found, we simply continue simulating the two copies of A). In addition to this, if $\vec{c}_{q_1} = \vec{c}_{q_2}$, we also define

- $(0, q_1, q_2) \in \delta_x((0, p_1, p_2), \sigma)$ if $\vec{c}_{q_1} = \vec{c}_{q_2}$, or
- $(1, q_1, q_2) \in \delta_x((0, p_1, p_2), \sigma)$ if $\vec{c}_{q_1} \neq \vec{c}_{q_2}$, i. e., there is a variable y with $\vec{c}_{q_1}(y) \neq \vec{c}_{q_2}(y)$.

In the first case, all variables have identical behavior, so we have not found a witness y, and do not change the bit. In the second case, there is a witness y, which allows us to change the bit. Note that we can precompute the sets of all $(q_1, q_2) \in Q$ with $\vec{c}_{q_1} = \vec{c}_{q_2}$ or $\vec{c}_{q_1} \neq \vec{c}_{q_2}$ in time $O(vn^2)$.

Now A_x describes exactly the language of strings $\mathbf{s}$ for which there exist demonstrate that x does not have the key property. As A' has $O(n^2)$ transitions, we can construct A_x in time $O(vn^2 + n^4) = O(n^4)$, and emptiness of $\mathcal{L}(A_x)$ can also be decided in time $O(n^4)$, by checking whether A_x contains a path from $(0, q_0, q_0)$ to $(1, q_f, q_f)$. Note that such a path also provides a witness input string $\mathbf{s}$, and, by decoding the variable configurations of the states along the path, the corresponding $\mu, \mu' \in [\![A]\!](\mathbf{s})$ with $\mu(x) = \mu'(x)$ and $\mu \neq \mu'$. As a side-effect, this construction also shows that the shortest witness that x does not have the key property is of length $O(n^2)$. □

A.6 Proof of Lemma 3.8

LEMMA 3.8. *Given a functional vset-automaton A and $Y \subseteq \mathsf{Vars}(A)$, one can construct in linear time a functional vset-automaton A_Y with $[\![A_Y]\!] = [\![\pi_Y(A)]\!]$.*

Proof. Let A be a functional vset-automaton, $V := \mathsf{Vars}(A)$, and $Y \subseteq V$. We define the morphism $h_Y : (\mathbf{\Gamma}_V \cup \mathbf{\Sigma})^* \to (\mathbf{\Gamma}_Y \cup \mathbf{\Sigma})^*$ for every $g \in (\mathbf{\Gamma}_V \cup \mathbf{\Sigma})$ by $h_Y(g) = \epsilon$ if $g \in (\mathbf{\Gamma}_V \setminus \mathbf{\Gamma}_Y)$, and $h_Y(g) := g$ for all $g \in (\mathbf{\Gamma}_Y \cup \mathbf{\Sigma})$. In other words, h_Y erases all $x\vdash$ and $\dashv x$ with $x \notin Y$, and leaves all other symbols unchanged.

We obtain A_Y from A by replacing the label g of each transition with $h(g)$. In other words, for each $x \notin Y$, each transition with $x\vdash$ or $\dashv x$ is replaced with an ϵ-transition. Clearly, this is possible in linear time, and $\mathcal{R}(A_Y) = h_Y(\mathcal{R}(A))$. As A is functional, this implies $\mathcal{R}(A_Y) = h_Y(\mathsf{Ref}(A))$.

Now assume that A_Y is not functional, i. e., that there exists an $\mathbf{r} \in (\mathsf{Ref}(A_Y) \setminus \mathcal{R}(A_Y))$. Then $\mathbf{r}$ is not valid, which means that there is an $y \in Y$ such that $y\vdash$ or $\dashv y$ does not occur exactly once, or both symbols occur in the wrong order. By definition of A_Y, there is an $\hat{\mathbf{r}} \in \mathcal{R}(A)$ with $h_Y(\hat{\mathbf{r}}) = \mathbf{r}$; and as h_Y erases only symbols of $\mathbf{\Gamma}_V \setminus \mathbf{\Gamma}_Y$ and leaves all other symbols unchanged, this means that $\hat{\mathbf{r}}$ is also not valid, which contradicts our assumption that A is functional. Hence, $\mathsf{Ref}(A_Y) = \mathcal{R}(A_Y) = h_Y(\mathsf{Ref}(A))$ holds. This also implies $[\![A_Y]\!] = [\![\pi_Y(A)]\!]$. □

A.7 Proof of Lemma 3.9

LEMMA 3.9. *Given functional vset-automata $A_1, \ldots, A_k$ with $\mathsf{Vars}(A_1) = \cdots = \mathsf{Vars}(A_k)$, one can construct in linear time a functional vset-automaton A with $[\![A]\!] = [\![A_1 \cup \cdots \cup A_k]\!]$.*

Proof. For $1 \leq i \leq k$, let n_i denote the number of states and m_i denote the number of transitions of A_i. We can construct A by using the standard construction for union: We add a new initial and a new final state, and connect these with ϵ-transitions to the "old" inital and final states of the A_i. Clearly, $\mathcal{R}(A) = \bigcup_{i=1}^{k} \mathcal{R}(A_i)$, and as each A_i is functional, $\mathsf{Ref}(A) = \bigcup_{i=1}^{k} \mathsf{Ref}(A_i)$ follows. Hence, A is functional, and $[\![A]\!] = \bigcup_{i=1}^{k} [\![A_i]\!] = [\![A_1 \cup \cdots \cup A_k]\!]$. Adding the new states takes time $O(k)$, and outputting A takes time $O(\sum_{i=1}^{k}(n_i + m_i))$. As this is the same size as the input of the algorithm, the construction runs in linear time.

Now, let $\mathbf{s} \in \mathbf{\Sigma}^*$, and define $N := |\mathbf{s}|$. If we now construct the graph G for A and $\mathbf{s}$ as in the proof of Theorem 3.3, each A_i leads to its own subgraph of G; and for distinct A_i, A_j, these subgraphs are disjoint. Hence, G has $O(N \sum_{i=1}^{k} n_i)$ nodes, and as each node of the subgraph for A_i can have at most n_i successors, G has $O(N \sum_{i=1}^{k} n_i^2)$ edges. This allows us to compute A_G in $O(\sum_{i=1}^{k}(Nn_i^2 + m_i n_i))$.

Following the same reasoning, we can conclude that for A_G, $\mathsf{minString}$ runs in time $O(N \sum_{i=1}^{k} n_i^2)$, as in each of the $O(N)$ iterations of the for-loop, S_i contains $O(\sum_{i=1}^{k} n_i)$ states, and each state that was derived from A_i has $O(n_i)$ successors. As this determines the complexity of $\mathsf{enumerate}$, we conclude that $[\![A]\!](\mathbf{s})$ can be enumerated with delay $O(N \sum_{i=1}^{k} n_i^2)$ after $O(\sum_{i=1}^{k}(Nn_i^2 + m_i n_i))$ preprocessing. As $n_i \leq n$ and $m_i \leq m$ for all $1 \leq i \leq k$, this can be simplified to a delay of $O(kNn^2)$ after a preprocessing of $O(kNn^2 + kmn)$. □

A.8 Proof of Lemma 3.10

LEMMA 3.10. *Given two functional vset-automata A_1 and A_2, each with $O(n)$ states and $O(v)$ variables, one can construct in time $O(vn^4)$ a functional vset-automaton A with $[\![A_1 \bowtie A_2]\!] = [\![A]\!]$.*

Proof. Assume we are given two functional vset-automata A_1 and A_2, with $A_i = (V_i, Q_i, q_{0,i}, q_{f,i}, \delta_i)$. Let $v_i := |V_i|$, $n_i := |Q_i|$, and let m_i denote the number of transitions of A_i. Furthermore, let $V := V_1 \cup V_2$ and define $v := |V|$. We assume that in each A_i, all states are reachable from $q_{0,i}$, and $q_{f,i}$ can be reached from every state. In order to construct a functional vset-automaton A with $[\![A]\!] = [\![A_1 \bowtie A_2]\!]$, we modify the product construction for the intersection of two NFAs.

Note that construction has to deal with the fact that the ref-words from $\mathcal{R}(A_1)$ and $\mathcal{R}(A_2)$ can use the variables in different orders. As a simple example, consider $\mathbf{r}_1 := x\vdash y\vdash \mathtt{a} \dashv y \dashv x$ and $\mathbf{r}_2 := y\vdash x\vdash \mathtt{a} \dashv x \dashv y$. Then $\mu^{\mathbf{r}_1} = \mu^{\mathbf{r}_2}$, although $\mathbf{r}_1 \neq \mathbf{r}_2$. Hence, we cannot directly apply the standard construction for NFA-intersection; and for complexity reasons, we do not rewrite the automata to impose an ordering on successive variable operations. Instead, we use the variable configurations of A_1 and A_2.

Construction: In order to simplify the construction, we slightly abuse the definition, and label the variable transitions of A with sets of variable transitions, instead of single transitions. We discuss the technical aspects of this of this at the end of the proof (in particular its effect on the complexity). This decision allows us to restrict the number of states in A, which in turn simplifies the construction: Like the standard construction of the intersection, the main idea of the proof is that A simulates each of A_1 and A_2 in parallel. Hence, its state set is a subset of $Q_1 \times Q_2$. But as we shall see, this construction uses the variable configurations $\vec{C}_i$ of A_i to determine how A should act on the variables (instead of the variable transitions). In particular, we shall require that all states of A are consistent, where $(q_1, q_2) \in Q_1 \times Q_2$ is *consistent* if $\vec{C}_1(q_1)(x) = \vec{C}_2(q_2)(x)$ for all $x \in V_1 \cap V_2$. In other words, the variable configurations of q_1 and q_2 agree on the common variables of A_1 and A_2.

By using sets of variable operations, we can disregard the order in which A_1 and A_2 process the common variables; and as variable transitions connect only consistent states of A that encode consistent pairs of states, the fact that A_1 and A_2 are functional shall ensure that A is also functional.

Before the algorithm constructs A, it computes the following sets and functions:

1. the variable configurations function $\vec{C}_i$ for A_i,
2. the set Q of all consistent $(q_1, q_2) \in Q_1 \times Q_2$,
3. the sets $Q_i^{\equiv}$ of all $(p, q) \in Q_i \times Q_i$ with $\vec{C}_i(p) = \vec{C}_i(q)$,
4. the ϵ-closures $\mathcal{E}_i \colon Q_i \to 2^{Q_i}$, where for each $p \in Q_i$, $\mathcal{E}_i(p)$ contains all $q \in Q_i$ that can be reached from p by using only ϵ-transitions,
5. the *variable-ϵ-closures* $\mathcal{VE}_i \colon Q_i \to 2^{Q_i}$, where for each $p \in Q_i$, $\mathcal{VE}_i(p)$ contains all $q \in Q_i$ that can be reached from p by using only transitions from $\{\epsilon\} \cup \mathbf{\Gamma}_{V_i}$,
6. for each $\sigma \in \mathbf{\Sigma}$, a function $\mathcal{T}_i^\sigma \colon Q_i \to 2^{Q_i}$, that is defined by $\mathcal{T}_i^\sigma(p) := \bigcup_{q \in \delta_i(p,\sigma)} \mathcal{E}_i(q)$ for each $p \in Q_i$.

In other words, $\mathcal{VE}_i$ can be understood as replacing all variable transitions in A_i with ϵ-transitions, and then computing the ϵ-closure; while $\mathcal{T}_i^\sigma(q)$ contains all states that can be reached by processing exactly one terminal transition, and then arbitrarily many ϵ-transitions.

As mentioned in the proof of Theorem 3.3, the variable configurations $\vec{C}_i$ can be computed in $O(v_i m_i)$. These can then be used to compute Q in time $O(|V_1 \cap V_2| n_1 n_2)$, and each $Q_i^{\equiv}$ in time $O(v_i n_i^2)$. By using the standard algorithm for transitive closures in directed graphs (see e. g. Skiena [36]), each $\mathcal{E}_i$ and $\mathcal{VE}_i$ can be computed in time $O(n_i m_i)$. Finally, each of the $\mathcal{T}_i^\sigma$ can the be computed in time $O(m n_i)$, by processing each transition of A_i and then using $\mathcal{E}_i$.

Now, $O(v_i n_i^2)$ dominates $O(v_i m_i)$. Furthermore, for at least one i, $O(v_i n_i^2)$ also dominates $O(|V_1 \cap V_2| n_1 n_2)$, as $|V_1 \cap V_2| \leq \min\{v_1, v_2\}$. Hence, the total running time of this precomputations adds up to $O(v_1 n_1^2 + m_1 n_1 + v_2 n_2^2 + m_2 n_2)$.

We are now ready to define $A := (V, Q, q_0, q_f, \delta)$, where Q is the set we defined above, $q_0 := (q_{0,1}, q_{0,2})$, and $q_f := (q_{f,1}, q_{f,2})$. Before we define δ, note that q_0 and q_f are always consistent—as A_1 and A_2 are functional, the variable configurations of the initial and final states map all variables to o and c, respectively. For the same reason, for all $p \in Q_i$, all $\sigma \in \mathbf{\Sigma}$, $\vec{C}_i(q) = \vec{C}_i(p)$ must hold for all $q \in \mathcal{T}_i^\sigma(p)$ or $q \in \mathcal{E}_i(p)$.

We now define δ as follows:

1. For every pair of states (q_1, q_2) with $q_i \in \mathcal{E}_i(q_{0,i})$, we add an ϵ-transition from $(q_{0,1}, q_{0,2})$ to (q_1, q_2).
2. For every pair of states $(p_1, p_2) \in Q$ and every $\sigma \in \mathbf{\Sigma}$, we compute all $(q_1, q_2) \in \mathcal{T}_1^\sigma(p_1) \times \mathcal{T}_2^\sigma(p_2)$. To each such (q_1, q_2), we add a transition with label σ from (p_1, p_2).
3. For every pair of states $(p_1, p_2) \in Q$, we compute all $(q_1, q_2) \in \mathcal{VE}_1(p_1) \times \mathcal{VE}_2(p_2)$. If $(q_1, q_2) \in Q$, and $\vec{C}_i(q_i) \neq \vec{C}_i(p_i)$, we add a variable transition from (p_1, p_2) to (q_1, q_2) that contains exactly the operations that map each $\vec{C}_i(p_i)$ to $\vec{C}_i(q_i)$.

Each of these three rules guarantees that the destination (q_1, q_2) of each transition is indeed a consistent pair of states (the third rule requires this explicitly, while the other two use the properties of $\mathcal{E}_i$ and $\mathcal{T}_i^\sigma$ mentioned above together with the fact that (p_1, p_2) is consistent).

Correctness and complexity: To see why this construction is correct, consider the following: The basic idea is that A simulates A_1 and A_2 in parallel. The actual work of the simulation is performed by the transitions that were derived from the second or the third rule. The former simulate the behavior of A_1 and A_2 on terminal letters, while the letter simulate sequences of variable actions. In particular, as we use the variable-ϵ-closures, a transition of A can simulate all sequences of variable actions that lead to a consistent state pair. Hence, it does not matter in which order A_1 or A_2 would process variables, the fact that the state of A is a consistent pair means that both automata agree on their common variables.

In order to keep the second and third rule simple, we also include the first rule: The effect of these transitions is that A can simulate that A_1 or A_2 changes states via ϵ-transitions before it starts processing terminals or variables. Finally, to see that A is functional, assume that it is not. Then there is a ref-word $\mathbf{r} \in \mathcal{R}(A) \setminus \mathsf{Ref}(A)$ that is not valid. Let $\mathbf{r}_i$ be the result of projecting $\mathbf{r}$ to $\mathbf{\Sigma} \cup \mathbf{\Gamma}_{V_i}$ (i. e., removing all elements of $\mathbf{\Gamma}_{V_j}$ with $j \neq i$). Then at least one of $\mathbf{r}_1$ or $\mathbf{r}_2$ is also not valid; we assume $\mathbf{r}_1$. But according to the definition of A, this means that $\mathbf{r}_1 \in \mathcal{R}(A_1)$, which implies $\mathcal{R}(A_1) \neq \mathsf{Ref}(A_1)$, and contradicts our assumption that A_1 is functional. Hence, A must be functional.

Regarding the complexity of the construction, note that A has $O(n_1 n_2)$ states and $O(n_1^2 n_2^2)$ transitions, and can be constructed in time $O(n_1^2 n_2^2)$. All transitions that follow from the first rule can obviously be computed in $O(n_1 n_2)$. For the second rule, for each of $O(n_1 n_2)$ many pairs $(p_1, p_2) \in Q$, we enumerate $O(n_1 n_2)$ many pairs $(q_1, q_2) \in \mathcal{T}(p_1)_1^\sigma \times \mathcal{T}(p_2)_2^\sigma$, and check whether $(q_1, q_2) \in Q$. As we precomputed Q, this check is possible in $O(1)$,

which means that these transitions can be constructed in time $O(n_1^2 n_2^2)$. Likewise, for the third rule, we use that we precomputed the sets $Q_i^{\equiv}$, which allows us to check $\vec{C}_i(p) \neq \vec{C}_i(q)$ in time $O(1)$ as well.

For the third rule, we need to take into consideration that the sets of variable operations have to be computed. Doing this naïvely would bring the complexity to $O((v_1 + v_2)n_1^2 n_2^2)$, but using the right representation, these sets can be precomputed independently for each A_i in time $O(v_i m_i)$, which allows us to leave the time for the variable transitions unchanged.

Up to this point, the complexity of the construction with precomputations is $O(v_1 n_1^2 + m_1 n_1 + v_2 n_2^2 + m_2 n_2 + n_1^2 n_2^2)$. For v, m, n with $v_i \in O(v)$, $m_i \in O(m)$, $n \in O(n)$, this becomes $O(vn^2 + mn + n^4)$, which we can simplify to $O(n^4)$, as $v \leq n$, and $m \leq n^2$. Note that this is the same complexity as constructing the product automaton for the intersection of two NFAs.

If we want to strictly adhere to the definition of vset-automata, we can rewrite A into an equivalent vset-automaton A_{strict}, by taking each variable transition that is labeled with a set S, and replacing it with a sequence of states and transitions that enumerates the elements of S. This would add $O((v_1 + v_2)n_1^2 n_2^2)$ states and the same number of transitions, and increase the complexity of the construction by $O((v_1 + v_2)n_1^2 n_2^2)$, netting a total complexity of $O(m_1 n_1 + m_2 n_2 + (v_1 + v_2)n_1^2 n_2^2)$. Under the assumptions from the last paragraph, this can be simplified to $O(vn^4)$.

While a more detailed analysis or a more refined construction might avoid this increase, it might be more advantageous to generalize the definition of vset-automata to allow sets of variable operations on transitions, as the complexities of computing the variable configurations and of Theorem 3.3 generalize to this model (as long as the number of variables is linear in the number of states).

In particular, if our actual goal is evaluating $A_1 \bowtie A_2$ by using Theorem 3.3, we can skip the step of rewriting A into A_{strict}, and directly construct G from A (or, without explicitly constructing A, from the $\mathcal{E}_i$ and $\mathcal{T}_i^{\sigma}$ functions). The total time for preprocessing is then $O(v_1 n_1^2 + m_1 n_1 + v_2 n_2^2 + m_2 n_2 + N n_1^2 n_2^2)$, or, if simplified, $O(Nn^4)$. This is also the complexity of the delay.

Naturally, all these constructions can be extended to the join of k vset-automata, where the complexities become $O(n^{2k})$ for the construction of the vset-automaton A with sets of variable operations, as well as $O(vn^{2k})$ for the construction of A_{strict} or $O(Nn^{2k})$ for the preprocessing and the delay of the enumeration algorithm. □

A.9 Proof of Theorem 5.2

THEOREM 5.2. *Evaluation of Boolean regex CQs q with string equalities is* W[1]*-hard for the parameter $|q|$, even if restricted to queries of the form $\pi_\emptyset \zeta^{=}_{x_1,y_1} \cdots \zeta^{=}_{x_m,y_m} \alpha$.*

Proof. We prove the claim by modifying the proof of Theorem 3.2: Let $\mathbf{s}$ and γ be defined as in that proof. However, we do not use regex formulas δ_l to ensure that all variables $y_{i,l}$ and $x_{l,j}$ with $1 \leq i < l < j \leq k$ map to the same substring $\mathbf{v}_l$. Instead, for each $1 \leq l \leq k$, we define a sequence S_l of $k-2$ binary string equality selections that is equivalent to the $(k-1)$-ary string equality selection on the variables $y_{1,l}$ to $y_{l-1,l}$ and $x_{l,l+1}$ to $x_{l,k}$. We then define our query q as

$$q := \pi_\emptyset S_1 \cdots S_k \gamma.$$

Note that being able to use string equality predicates, we do not need to iterate through all of the possible $\mathbf{v}$ as in the δ_l. Therefore the proof of correctness of the reduction used in the proof of Theorem 3.2 can be simply adapted to show correctness of this reduction. Observe that this is an FPT reduction with parameter $|q|$ since $|q|$ is of size $O(k^2)$ (i.e., the number of string equality predicates we use depends only on the clique size). Finally, we remark that like the proof of Theorem 3.2, this proof can be adapted to a binary alphabet by using the standard coding techniques. □

A.10 Proof of Theorem 5.4

THEOREM 5.4. *For every fixed $m \geq 1$ there is an algorithm, that given a functional vset-automaton A, a sequence $S := \zeta^{=}_{x_1,y_1} \cdots \zeta^{=}_{x_m,y_m}$ of string equality selections over $\mathsf{Vars}(A)$, and a string $\mathbf{s}$, enumerates $[\![SA]\!](\mathbf{s})$ with polynomial delay.*

Proof. Let A be a functional vset-automaton with n states and m transitions, $\mathbf{s}$ an input string, k a fixed integer, and $x_i, y_i \subseteq \mathsf{Vars}(A)$ for $1 \leq i \leq k$. Let $N := |\mathbf{s}|$, and let $v := \mathsf{Vars}(A)$. We describe an algorithm for enumerating the tuples in $[\![\zeta^{=}_{x_1,y_1} \cdots \zeta^{=}_{x_k,y_k} A]\!](\mathbf{s})$ in polynomial delay.

Algorithm: Iterating over all of the possible couples of spans of $\mathbf{s}$, we can obtain all pairs s_1, s_2 of spans of $\mathbf{s}$ that refer to equal substrings; i. e., $\mathbf{s}_{s_1} = \mathbf{s}_{s_2}$. This can be done naïvely in time $O(N^4)$. Using these pairs, we construct a functional vset-automata A_{eq} with $\mathsf{Vars}(A^{=}) = \{x_i, y_i \mid 1 \leq i \leq k\}$ such that $\mu \in [\![A_{\mathsf{eq}}]\!]$ if and only if $\mathbf{s}_{\mu(x_i)} = \mathbf{s}_{\mu(y_i)}$

holds for all $1 \leq i \leq k$. In other words, A_{eq} describes exactly those spans of $\mathbf{s}$ that satisfy the string equality selections.

We ensure that A_{eq} has $O(N^{3k+1})$ states, and each state except the initial state and the final state has exactly one outgoing transition. More specifically, A_{eq} encodes each possible $\mu \in [\![A]\!](\mathbf{s})$ in a path of states. Each of these μ is characterized by selecting for each $1 \leq i \leq k$ the length of $\mathbf{s}_{\mu(x_i)}$ (and, hence, $\mathbf{s}_{\mu(y_i)}$), as well as the starting positions of $\mu(x_i)$ and $\mu_(y_i)$. Hence, there are $O(N^{3k})$ possible μ, and each can be encoded in a path of $O(N+k)$ successive states if we take variable operations into account. If we allow multiple variable actions on a single transition (see the Proof of Lemma 3.10), this is possible in $O(N)$ states; but even if we do not, we can safely assume that $k \leq N$, which also gives $O(N)$ states.

The initial state of A_{eq} non-deterministically choses with an ϵ-transition which of these paths to take, which means that A_{eq} can be constructed with $O(N^{3k+1})$ states and transitions. Moreover, we can build A_{eq} while enumerating the pairs s_1, s_2 as described above, which means that the construction takes time $O(N^{3k+1})$. Note that if two selections have a common variable, we can lower the exponent by two, as we only need to account once for the length and the starting position. For example, A_{eq} for $\zeta^{=}_{x,y}\zeta^{=}_{y,z}A$ only needs $O(N^5)$ states, instead of $O(N^7)$.

We can now use Lemma 3.10 to construct a functional vset-automaton A_{join} with $[\![A_{\mathsf{join}}]\!] = [\![A \bowtie A_{\mathsf{eq}}]\!]$. As $[\![A_{\mathsf{join}}]\!](\mathbf{s}) = [\![\zeta^{=}_{x_1,y_1} \cdots \zeta^{=}_{x_k,y_k} A]\!](\mathbf{s})$ holds, this solves our problem.

Complexity analysis: If we directly use Lemma 3.10 together with Theorem 3.3 naïvely, preprocessing and delay each have a complexity of $O(N(nN^{3k+1})^2) = O(N^{6k+3}n^2)$. While this is polynomial in n and N, we can lower the degree by exploiting the structure of A_{eq}.

Take particular note of the structural similarity between A_{eq} and the graphs that are used in the proof of Theorem 3.3 when matching on the string $\mathbf{s}$: Both have N levels, where a level i represents that i letters of N have been processed. Hence, if we wanted to enumerate $[\![A_{\mathsf{eq}}]\!](\mathbf{s})$ as described in the proof of Theorem 3.3, the constructed graph would only need to have $O(N^{3k+1})$ nodes, with exactly one outgoing edge for all nodes (except the initial and the final node). Consequently, the resulting NFA that is used for $\mathsf{enumerate}$ would only have $O(N^{3k+1})$ nodes and $O(N^{3k+1})$ transitions.

We now examine A_{join}. Each of its states (q, q_{eq}) encodes a state q of A and a state q_{eq} of A_{eq}. The state q has $O(n)$ successor states in A, and q_{eq} has at most one successor state in A_{eq} (unless it is the initial state; but as this state cannot be reached after it has been left, the number of transitions from it are dominated by the number of the other transitions). Hence, (q, q_{eq}) has at most $O(n)$ successors in A_{join}. In total, A_{join} has $O(N^{3k+1}n)$ states, and $O(N^{3k+1}n^2)$ transitions. By combining these observations with the construction from the proof of Lemma 3.10, we see that A_{join} can be constructed in time $O(vn^2 + mn + N^{3k+1}n^2)$.

Moreover, A_{join} exhibits the same level structure as A_{eq}. Hence, in order to enumerate $[\![A_{\mathsf{join}}]\!](\mathbf{s})$, we can directly derive G and A_G from A_{join}, without needing the extra factor of $O(N)$ states to store how much of $\mathbf{s}$ has been processed.

Thus, A_G has $O(N^{3k+1}n)$ states and $O(N^{3k+1}n^2)$ transitions. In time $O(N^{3k+1}n^2)$, we can construct A_G from A_{join}, precompute its functions $\mathsf{minLetter}$ and $\mathsf{nextLetter}$, and ensure that the final state is reachable from every state, and that each state is reachable. Finally, note that the alphabet $\mathcal{K}$ of A_G is the set of all variable configurations of A, as $\mathsf{Vars}(A_{\mathsf{eq}}) \subseteq \mathsf{Vars}(A)$.

All that remains is calling $\mathsf{enumerate}$ on A_G. As in the proof of Theorem 3.3, its complexity is determined by the complexity of $\mathsf{minString}$: The for-loop is executed $O(N)$ times. As each level of A_G contains $O(N^{3k}n)$ nodes, the same bound holds for each S_i. Furthermore, as each of these nodes has $O(n)$ successors, each S_{i+1} can be computed in time $O(N^{3k}n^2)$. Hence, executing $\mathsf{minString}$ requires time $O(N^{3k+1}n^2)$.

Hence, we can enumerate $[\![A_{\mathsf{join}}]\!](\mathbf{s})$ with a delay of $O(N^{3k+1}n^2)$. Including the construction of A_{join}, the preprocessing takes $O(vn^2 + mn + N^{3k+1}n^2)$, which we can simplify to $O(n^3 + N^{3k+1}n^2)$. Except for rather pathological cases, we can assume this to be $O(N^{3k+1}n^2)$. □