Constant delay algorithms for regular document spanners

Fernando Florenzano
PUC Chile
faflorenzano@uc.cl

Cristian Riveros
PUC Chile
cristian.riveros@uc.cl

Martín Ugarte
Université Libre de Bruxelles
mugartec@ulb.ac.be

Stijn Vansummeren
Université Libre de Bruxelles
stijn.vansummeren@ulb.ac.be

Domagoj Vrgoč
PUC Chile
dvrgoc@ing.puc.cl

ABSTRACT

Regular expressions and automata models with capture variables are core tools in rule-based information extraction. These formalisms, also called regular document spanners, use regular languages in order to locate the data that a user wants to extract from a text document, and then store this data into variables. Since document spanners can easily generate large outputs, it is important to have good evaluation algorithms that can generate the extracted data in a quick succession, and with relatively little precomputation time. Towards this goal, we present a practical evaluation algorithm that allows constant delay enumeration of a spanner's output after a precomputation phase that is linear in the document. While the algorithm assumes that the spanner is specified in a syntactic variant of variable set automata, we also study how it can be applied when the spanner is specified by general variable set automata, regex formulas, or spanner algebras. Finally, we study the related problem of counting the number of outputs of a document spanner, providing a fine grained analysis of the classes of document spanners that support efficient enumeration of their results.

1. INTRODUCTION

Rule-based information extraction (IE for short) [7, 10, 15] has received a fair amount of attention from the database community recently, revealing interesting connections with logic [11, 12], automata [10, 17], datalog programs [3, 22], and relational languages [6, 16, 13]. In rule-based IE, documents from which we extract the information are modelled as strings. This is a natural assumption for many formats in use today (e.g. JSON and XML files, CSV documents, or plain text). The extracted data are represented by *spans*. These are intervals inside the document string that record the start and end position of the extracted data, plus the substring (the data) that this interval spans. The process of information extraction can then be abstracted by the notion of *document spanners* [10]: operators that map strings to tuples containing spans.

The most basic way of defining document spanners is to use some form or regular expressions or automata with capture variables. The idea is that a regular language is used in order to locate the data to be extracted, and variables to store this data. This approach to IE has been widely adopted in the database literature [10, 9, 3, 11, 17], and also forms the core extraction mechanism of commercial IE tools such as IBM's SystemT [16]. The two classes of expressions and automata for extracting information most commonly used in the literature are *regex formulas* (RGX) and *variable-set automata* (VA), both formally introduced in [10].

A crucial problem when working with RGX and VA in practice is how to evaluate them efficiently. One issue here is that the output can easily become huge. For the sake of illustration, consider the regex formula $\gamma = \Sigma^* \cdot x_1\{\Sigma^* \cdot x_2\{\Sigma^*\} \cdot \Sigma^*\} \cdot \Sigma^*$, where Σ denotes a finite alphabet. Intuitively, γ extracts any span of a document d into x_1, and any sub-span of this span into x_2. Therefore, on a document d over Σ it will produce an output of size $\Omega(|d|^2)$. If we keep nesting the variables (i.e., x_3 inside x_2, etc.), the output size will be $\Omega(|d|^\ell)$, with ℓ the number of variables in γ. Since an evaluation algorithm must at least write down this output, and since the latter is exponential (in γ and d), alternative complexity measures need to be used in order to answer when this problem is efficiently solvable.

A natural option here is to use *enumeration algorithms* [19], which work by first running a precomputation phase, after which they can start producing elements of the output (tuples of spans in our case) with pre-defined regularity and without repetitions. The time taken by an enumeration algorithm that has an input I and an output O is then measured by a function that depends both on the size of I and the size of O. Ideally, we would like an algorithm that runs in total time $O(f(|I|) + |O|)$, where f is a function not depending on the size of the output, so that the output is returned without taking much time between generating two of its consecutive elements. This is achieved by the class of *constant delay enumeration algorithms* [19], that do a pre-computation phase that depends only on the size of the input (γ and d in our case), followed by an enumeration of the output without repetitions where the time between two outputs is constant.

Constant delay algorithms have been studied in various contexts, ranging from MSO queries over trees [4, 8], to relational conjunctive queries [5]. These studies, however, have been mostly theoretical in nature, and did not consider practical applicability of the proposed algorithms. To quote several recent surveys of the area: "We stress that our study is from the theoretical point of view. If most of the algorithms we will mention here are linear in the size of the database, the constant factors are often very big, making any practical implementation difficult." [19, 20, 21]. These surveys also leave open the question of whether practical algorithms could be designed in specific contexts, where the language being processed is restricted in its expressive power. This was already shown to be true in [3], where a constant delay enumeration algorithm for a restricted class of document spanners known as navigation expressions was implemented and tested in practice. Since navigation expressions are a very restricted subclass of RGX and VA [17], and since the latter have been established in the literature as the two most important classes of rule-based IE languages, in this paper we study practical constant delay algorithms for RGX and VA.

Contributions. The principal contribution of our work is an intuitive constant delay algorithm for evaluating a syntactic variant of VA that we call extended VA. Extended VA are designed to streamline the way VA process a string, and the algorithm we present can evaluate an extended VA $\mathcal{A}$ that is both sequential [17] and deterministic over a document d with preprocessing time $O(|\mathcal{A}| \times |d|)$, and with constant delay output enumeration. We then study how this algorithm can be applied to arbitrary RGX and VA, and their most studied restrictions such as functional and sequential RGX and VA. Both sequential and functional VA and RGX are important subclasses of regular spanners: as shown in [10, 17, 13], they have both good algorithmic properties and prohibit unintuitive behaviour. Next, we proceed by extending our findings to the setting where spanners are specified by means of an algebra that allows to combine VA or RGX using unions, joins and projections. As such, we identify upper bounds on the preprocessing times when evaluating the class of regular spanners [10] with constant delay.

In an effort to get some idea of potential lower-bounds on preprocessing times, we study the problem of counting the number of tuples output by a spanner. This problem is strongly connected to the enumeration problem [19], and gives evidence on whether a constant delay algorithm with faster pre-computation time exists. Here, we extend our main constant delay algorithm to count the number of outputs of a deterministic and sequential extended VA $\mathcal{A}$ in time $O(|\mathcal{A}| \times |d|)$. We also show that counting the number of outputs of a functional but not necessarily deterministic nor extended VA is complete for the counting class SpanL [2], thus making it unlikely to compute this number efficiently unless the polynomial hierarchy equals Ptime.

Related work. Constant delay enumeration algorithms (from now on CDAs) for MSO queries have been proposed in [4, 8, 14]. Since any regular spanner can be encoded by an MSO query (where capture variables are encoded by pairs of first-order variables), this implies that CDAs for MSO queries also apply to document spanners. In [8], a CDA was given with preprocessing time $O(|t| \times \log(|t|))$ in data complexity where $|t|$ is the size of the input structure (e.g. document). In [14], a CDA was given based on the deterministic factorization forest decomposition theorem, a combinatorial result for automata. Our CDA has linear precomputation time over the input document and does not rely on any previous results, making it incomparable with [8, 14].

The CDA given by Bagan in [4] requires a more detailed comparison. The core algorithm of [4] is for a deterministic automaton model which has some resemblance with deterministic VA, but there are several differences. First of all, Bagan's algorithm is for tree automata and the output are tuples of MSO variables, while our algorithm works only for VA, whose output are first order variables. Second, Bagan's algorithm has preprocessing time $O(|\mathcal{A}|^3 \times |t|)$, where $\mathcal{A}$ is a tree automaton and t is a tree structure. In contrast, our algorithm has preprocessing time $O(|\mathcal{A}| \times |d|)$, namely, linear in $|\mathcal{A}|$. Although Bagan's algorithm is for tree-automata and this can explain a possible quadratic blow-up in terms of $|\mathcal{A}|$, it is not directly clear how to improve its preprocessing time to be linear in $|\mathcal{A}|$. Finally, Bagan's algorithm is described as a composition of high-level operations over automata and trees, while our algorithm can be described using a few lines of pseudo-code.

There is also recent work [13, 17] tackling the enumeration problem for document spanners directly, but focusing on polynomial delay rather than constant delay. In [17], a complexity theoretic treatise of polynomial delay (with polynomial pre-processing) is given for various classes of spanners. And while [17] focuses on decision problems that guarantee an existence of a polynomial delay algorithm, in the present paper we focus on practical algorithms that furthermore allow for constant delay enumeration. On the other hand, [13] gives an algorithm for enumerating the results of a functional VA automaton $\mathcal{A}$ over a document d with a delay of roughly $O(|\mathcal{A}|^2 \times |d|)$, and pre-processing of the order $O(|\mathcal{A}|^2 \times |d|)$. The main difference of [13] and the present paper is that our algorithm can guarantee constant delay, albeit for a slightly better behaved class of automata. When applying our algorithm directly to functional VA as in [13], we can still obtain constant delay enumeration, but now with a pre-processing time of $O(2^{|\mathcal{A}|} \times |d|)$ (see Section 4). Therefore, if considering only functional VA, the algorithm of [13] would be the preferred option when the automaton is large, and when the number of outputs is relatively small, while for spanners that capture a lot of information, or are executed on very big documents, one would be better off using the constant delay algorithm presented here. Another difference is that the algorithm of [13] is presented in terms of automata theoretic constructions, while we aim to give a concise pseudo-code description.

Organization. We formally define all the notions used throughout the paper in Section 2. The algorithm for evaluating a deterministic and sequential extended VA with linear preprocessing and constant delay enumeration is presented in Section 3, and its application to regular spanners in Section 4. We study the counting problem in Section 5, and conclude in Section 6. Due to space reasons, most proofs are deferred to the appendix.

2. BASIC DEFINITIONS

Documents and spans. We use a fixed finite alphabet Σ throughout the paper. A *document*, from which we will extract information, is a finite string $d = a_1 \dots a_n$ in Σ^*. We denote the length n of document d by $|d|$. A *span* s is a pair $[i, j\rangle$ of natural numbers i and j with $1 \leqslant i \leqslant j$. Such a span is said to be *of document* d if $j \leqslant |d| + 1$. In that case, s is associated with a continuous region of the document d (also called a span of d), whose content is the substring of d from positions i to $j - 1$. We denote this substring by $d(s)$ or $d(i, j)$. To illustrate, Figure 1 shows a document d as well as several spans of d. There, for example, $d(1, 5) =$ `John`. Notice that if $i = j$, then $d(s) = d(i, j) = \varepsilon$. Given two spans $s_1 = [i_1, j_1\rangle$ and $s_2 = [i_2, j_2\rangle$, if $j_1 = i_2$ then their concatenation is equal to $[i_1, j_2\rangle$ and is denoted $s_1 \cdot s_2$. The set of all spans of d is denoted by $\mathrm{span}(d)$.

Mappings. Following [17], we will use mappings to model the information extracted from a document. Mappings differ from tuples (as used by e.g., Fagin et al. [10] and Freydenberger et al. [11, 12]) in that not all variables need to be assigned a span. Formally, let $\mathcal{V}$ be a fixed infinite set of variables, disjoint from Σ. A *mapping* is a function μ from a finite set of variables $\mathrm{dom}(\mu) \subseteq \mathcal{V}$ to spans. Two mappings μ_1 and μ_2 are said to be *compatible* (denoted $\mu_1 \sim \mu_2$) if $\mu_1(x) = \mu_2(x)$ for every x in $\mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$. If $\mu_1 \sim \mu_2$, we define $\mu_1 \cup \mu_2$ as the mapping that results from extending μ_1 with the values from μ_2 on all the variables in $\mathrm{dom}(\mu_2) \backslash \mathrm{dom}(\mu_1)$. The *empty mapping*, denoted by $\varnothing$, is the only mapping such that $\mathrm{dom}(\varnothing) = \varnothing$. Similarly, $[x \to s]$ denotes the mapping whose domain only contains the variable x, which it assigns to be the span s. The *join* of two set of mappings M_1 and M_2 is defined as follows:

$$M_1 \bowtie M_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1,\ \mu_2 \in M_2 \text{ and } \mu_1 \sim \mu_2\}.$$

Document spanners. A *document spanner* is a function that maps every input document d to a set of mappings M such that the range of each $\mu \in M$ are spans of d—thus modeling the process of extracting the information (in form of mappings) from d. Fagin et al. [10] have proposed different languages for defining spanners: by means of regex formulas, by means of automata, and by means of algebra. We next recall the definition of these languages, and define their semantics in the context of mappings rather than tuples.

Document d

J	o	h	n	␣	⟨	j	@	g	.	b	e	⟩	,	␣	J	a	n	e	␣	⟨	5	5	5	−	1	2	⟩
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28

$[\![\gamma]\!]_d$

	name	email	phone
μ_1	$[1, 5\rangle$	$[7, 13\rangle$	
μ_2	$[16, 20\rangle$		$[22, 28\rangle$

Figure 1: A document d and the evaluation $[\![\gamma]\!]_d$, with γ as defined in Equation (1).

$$\begin{aligned}
[\![\gamma]\!]_d &= \{\mu \mid ((1, |d| + 1), \mu) \in [\gamma]_d\} \\
[\varepsilon]_d &= \{(s, \varnothing) \mid s \in \mathrm{span}(d) \text{ and } d(s) = \varepsilon\} \\
[a]_d &= \{(s, \varnothing) \mid s \in \mathrm{span}(d) \text{ and } d(s) = a\} \\
[x\{\gamma\}]_d &= \{(s, \mu) \mid \exists (s, \mu') \in [\gamma]_d : \\
&\qquad x \notin \mathrm{dom}(\mu') \text{ and } \mu = [x \to s] \cup \mu'\} \\
[\gamma_1 \cdot \gamma_2]_d &= \{(s, \mu) \mid \exists (s_1, \mu_1) \in [R_1]_d, \\
&\qquad \exists (s_2, \mu_2) \in [\gamma_2]_d : s = s_1 \cdot s_2, \\
&\qquad \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2) = \varnothing, \text{ and} \\
&\qquad \mu = \mu_1 \cup \mu_2\} \\
[\gamma_1 \vee \gamma_2]_d &= [\gamma_1]_d \cup [\gamma_2]_d \\
[\gamma^*]_d &= [\varepsilon]_d \cup [\gamma]_d \cup [\gamma^2]_d \cup [\gamma^3]_d \cup \cdots
\end{aligned}$$

Table 1: The semantics $[\![\gamma]\!]_d$ of a RGX γ over a document d. Here γ^2 is a shorthand for $\gamma \cdot \gamma$, similarly γ^3 for $\gamma \cdot \gamma \cdot \gamma$, etc.

Regex formulas. Regex formulas extend the syntax of classic regular expressions with variable capture expressions of the form $x\{\gamma\}$. Intuitively, and similar to classical regular expressions, regex formulas specify a search through an input document. However, when, during this search, a variable capture subformula $x\{\gamma\}$ is matched against a substring, the span s that delimits this substring is recorded in a mapping $[x \to s]$ as a side-effect. Formally, the syntax of *regex formulas* is defined by the following grammar [10]:

$$\gamma := \varepsilon \mid a \mid x\{\gamma\} \mid \gamma \cdot \gamma \mid \gamma \vee \gamma \mid \gamma^*.$$

Here, a ranges over letters in Σ and x over variables in $\mathcal{V}$. We will write $\mathrm{var}(\gamma)$ to denote the set of all variables occurring in regex formula γ. We write RGX for the class of all regex formulas.

The mapping-based spanner semantics of RGX is given in Table 1 (cf. [17]). The semantics is defined by structural induction on γ and has two layers. The first layer, $[\gamma]_d$, defines the set of all pairs (s, μ) with $s \in \mathrm{span}(d)$ and μ a mapping such that (1) γ successfully matches the substring $d(s)$ and (2) μ results as a consequence of this successful match. For example, the regex formula $\gamma = a$ matches all substrings of input document d equal to a, but results in only the empty mapping. On the other hand, $\gamma = x\{\gamma_1\}$ matches all substrings that are matched by γ_1, but assigns x the span s that delimits the substring being matched, while preserving the previous variable assignments. Similarly, in the case of concatenation $\gamma_1 \cdot \gamma_2$ we join the mapping defined on the left with the one defined on the right, while imposing that the same variable is not used in both parts (as this would lead to inconsistencies). The second layer, $[\![\gamma]\!]_d$ then simply gives us the mappings that γ defines when matching the entire document. Note that when γ is an ordinary regular expression ($\mathrm{var}(\gamma) = \varnothing$), then the empty mapping is output if the entire document matches γ, and no mapping is output otherwise.

Example 2.1. *Consider the task of extracting names, email addresses and phone numbers from documents. To do this we could use the regex formula* γ *defined as*

$$\Sigma^* \cdot name\{\gamma_n\} \cdot \_ \cdot \langle \cdot (email\{\gamma_e\} \vee phone\{\gamma_p\}) \cdot \rangle \cdot \Sigma^* \quad (1)$$

where _ *represents a space; name, email, and phone are variables; and* γ_n*,* γ_e*, and* γ_p *are regex formulas that recognize person names, email addresses, and phone numbers, respectively. We omit the particular definition of these formulas as this is irrelevant for our purpose. The result* $[\![\gamma]\!]_d$ *of evaluating* γ *over the document* d *shown in Figure 1 is shown at the bottom of Figure 1.*

It is worth noting that the syntax of regex formula used here is slightly more liberal than that used by Fagin et al. [10]. In particular Fagin et al. require regex formulas to adhere to certain syntactic restrictions that ensure that the formula is *functional*: every mapping in $[\![\gamma]\!]_d$ is defined on all variables appearing in γ, for every d. For regex formulas that satisfy this syntactic restriction, the semantics given here coincides with that of Fagin et al [10] (see [17] for a detailed discussion).

Variable-set automata. A *variable-set automaton* (VA) [10] is an finite-state automaton extended with captures variables in a way analogous to RGX; that is, it behaves as a usual finite state automaton, except that it can also open and close variables. Formally, a VA automaton $\mathcal{A}$ is a tuple (Q, q_0, F, δ), where Q is a finite set of *states*; $q_0 \in Q$ is the initial state; $F \subseteq Q$ is the set of final states; and δ is a *transition relation* consisting of *letter transitions* of the form (q, a, q') and *variable transitions* of the form $(q, x\vdash, q')$ or $(q, \dashv x, q')$, where $q, q' \in Q$, $a \in \Sigma$ and $x \in \mathcal{V}$. The $\vdash$ and $\dashv$ are special symbols to denote the opening or closing of a variable x. We refer to $x\vdash$ and $\dashv x$ collectively as *variable markers*. We define the set $\text{var}(\mathcal{A})$ as the set of all variables x that are mentioned in some transition of $\mathcal{A}$.

A configuration of a VA automaton over a document d is a tuple (q, i) where $q \in Q$ is the current state and $i \in [1, |d| + 1]$ is the *current position* in d. A run ρ over a document $d = a_1 a_2 \cdots a_n$ is a sequence of the form:

$$\rho \;=\; (q_0, i_0) \xrightarrow{o_1} (q_1, i_1) \xrightarrow{o_2} \cdots \xrightarrow{o_m} (q_m, i_m)$$

where $o_j \in \Sigma \cup \{x\vdash, \dashv x \mid x \in \mathcal{V}\}$ and $(q_j, o_{j+1}, q_{j+1}) \in \delta$. Moreover, $i_0, \ldots, i_n$ is a non-decreasing sequence such that $i_0 = 1$, $i_m = |d| + 1$, and $i_{j+1} = i_j + 1$ if $o_{j+1} \in \Sigma$ (i.e. the automata moves one position in the document only when reading a letter) and $i_{j+1} = i_j$ otherwise. Furthermore, we say that a run ρ is *accepting* if $q_m \in F$ and that it is *valid* if variables are opened and closed in a correct manner (that is, each x is opened or closed at most once, and x is opened at some position i if and only if it is closed at some position j with $i \leqslant j$). Note that not every accepting run is valid. In case that ρ is both accepting and valid, we define μ^ρ to be the mapping that maps x to $[i_j, i_k\rangle \in \text{span}(d)$ if, and only if, $o_{i_j} = x\vdash$ and $o_{i_k} = \dashv x$ in ρ. Finally, the semantics of $\mathcal{A}$ over d, denoted by $[\![\mathcal{A}]\!]_d$ is defined as the set of all μ^ρ where ρ is a valid and accepting run of $\mathcal{A}$ over d.

Note that validity requires only that variables are opened and closed in a correct manner; it does not require that all variables in $\text{var}(\mathcal{A})$ actually appear in the run. Valid runs that do mention all variables in $\text{var}(\mathcal{A})$ are called *functional*. In a functional run, all variables are hence opened and closed exactly once (and in the correct manner) whereas in a valid run they are opened and closed at most once.

A VA $\mathcal{A}$ is *sequential* (sVA) if every accepting run of $\mathcal{A}$ is valid. It is *functional* (fVA) if every accepting run is functional. In particular, every fVA is also sequential. Intuitively, during a run a sVA does not need to check whether variables are opened and closed in a correct manner; the run is guaranteed to be valid whenever a final state is reached.

It was shown in [17, 13] that constant delay enumeration (after polynomial-time preprocessing) is not possible for variable-set automata in general. However, the authors in [17] also show that for the class of fVA or sVA, polynomial delay enumeration is possible, thus leaving open the question of constant delay in this case. As we will see, the sequential property is important in order to have constant-delay algorithms.

Spanner algebras. In addition to defining basic document spanners through RGX or VA, practical information extraction systems also allow spanners to be defined by applying basic algebraic operators on already existing spanners. This is formalized as follows. Let $\mathcal{L}$ be a language for defining document spanners (such as RGX or VA). Then we denote by $\mathcal{L}^{\{\pi,\cup,\bowtie\}}$ the set of all expressions generated by the following grammar:

$$e := \alpha \mid \pi_Y(e) \mid e \cup e \mid e \bowtie e.$$

Here, α ranges over expressions of $\mathcal{L}$, and Y is a finite subset of $\mathcal{V}$. Assume that $[\![\alpha]\!]$ denotes the spanner defined by $\alpha \in \mathcal{L}$. Then the semantics $[\![e]\!]$ of expression e is the spanner inductively defined as follows:

$$[\![\pi_Y(e)]\!]_d = \{\mu|_Y : \mu \in [\![e]\!]_d\}$$
$$[\![e_1 \cup e_2]\!]_d = [\![e_1]\!]_d \cup [\![e_2]\!]_d$$
$$[\![e_1 \bowtie e_2]\!]_d = [\![e_1]\!]_d \bowtie [\![e_2]\!]_d$$

Here, $\mu|_Y$ is the restriction of μ to variables Y and $[\![e_1]\!]_d \bowtie [\![e_2]\!]_d$ is the join of two sets of mappings.

It was shown by Fagin et al. [10] that VA, $\text{RGX}^{\{\pi,\cup,\bowtie\}}$, and $\text{VA}^{\{\pi,\cup,\bowtie\}}$ all express the same class of spanners, called *Regular Spanners*. In particular, every expression in $\text{RGX}^{\{\pi,\cup,\bowtie\}}$, and $\text{VA}^{\{\pi,\cup,\bowtie\}}$ is equivalent to a VA. This will be used later in Section 4.

The enumeration problem. In this paper, we study the problem of enumerating all mappings in $[\![\gamma]\!]_d$, given a document spanner γ (e.g. by means of a VA) and a document d. Given a language $\mathcal{L}$ for document spanners we define the main enumeration problem of evaluating expressions from $\mathcal{L}$ formally as follows:

Problem:	Enumerate[$\mathcal{L}$]
Input:	Expression $\gamma \in \mathcal{L}$ and document d.
Output:	All mappings in $[\![\gamma]\!]_d$ without repetitions.

As usual, we assume that the size $|R|$ of a RGX expression R is the number of alphabet symbols and operations, and the size $|\mathcal{A}|$ of a VA $\mathcal{A}$ is given by the number of transitions plus the number of states. Furthermore, the size $|e|$ of an expression e in $\mathcal{L}^{\{\pi,\cup,\bowtie\}}$ (e.g RGX$^{\{\pi,\cup,\bowtie\}}$) is given by $\sum_i |\alpha_i|$ where α_i are the expressions in $\mathcal{L}$ plus the number of operators (i.e. $\{\pi,\cup,\bowtie\}$) used in e.

Enumeration with constant delay. We use the definition of constant delay enumeration presented in [19, 20, 21] adapted to ENUMERATE[$\mathcal{L}$]. As it is standard in the literature [19], we consider enumeration algorithms over Random Access Machines (RAM) with addition and uniform cost measure [1]. Given a language $\mathcal{L}$ for document spanners, we say that an enumeration algorithm $\mathcal{E}$ for ENUMERATE[$\mathcal{L}$] has constant delay if $\mathcal{E}$ runs in two phases over the input $\gamma \in \mathcal{L}$ and d.
- The first phase (*precomputation*) which does not produce output.
- The second phase (*enumeration*) which occurs immediately after the precomputation phase and enumerates all mappings in $[\![\gamma]\!]_d$ without repetitions. We require that the delay between the start of enumeration, between any two consecutive outputs, and between the last output and the end of this phase depend only on $|\gamma|$. A such, it is constant in $|d|$.

We say that $\mathcal{E}$ is a *constant delay algorithm* for ENUMERATE[$\mathcal{L}$] with precomputation phase $f(|\gamma|,|d|)$, if $\mathcal{E}$ has constant delay and the precomputation phase takes time $O(f(|\gamma|,|d|))$. We say that $\mathcal{E}$ features constant delay enumeration after linear time pre-processing if $f(|\gamma|,|d|) = g(|\gamma|)\cdot|d|$ for some function g. It is important to stress that the delay between consecutive outputs has to be constant, so we seek to reduce the precomputation time $f(|\gamma|,|d|)$ as much as possible.

3. CONSTANT DELAY EVALUATION OF EXTENDED VSET AUTOMATA

In this section we present an algorithm featuring constant delay enumeration after linear pre-processing for a syntactic variant of VA that we call extended variable-set automata (eVA for short). This variant avoids several problems that VA have in terms of evaluation. Later, in Section 4, we show how this algorithm can be applied to ordinary VA, RGX formulas, and spanner algebras. We start by introducing extended VA.

3.1 Extended variable-set automata

VA can open or close variables in arbitrary ways, which can lead to multiple runs that define the same output. An example of this is given in Figure 2, where we have a functional VA (fVA) that has two runs resulting in the same output (i.e. they produce a mapping that assigns the entire document both to x and y). This is of course problematic for constant delay enumeration, as outputs must be enumerated without repetitions[1].

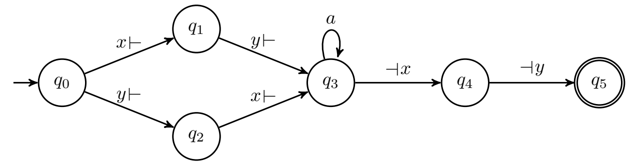


Figure 2: A functional VA with multiple runs defining the same output mapping.

Ideally, when running a VA one would like to start by declaring which variable operations take place before reading the first letter of the input word, then process the letter itself, followed by another step declaring which variable operations take place after this, read the next letter, etc. Extended variable-set automata achieve this by allowing multiple variable operations to take place during a single transition, and by forcing each transition that manipulates variables to be followed by a transition processing a letter from the input word.

Formally, let $\text{Markers}_{\mathcal{V}} = \{x\vdash, \dashv x \mid x \in \mathcal{V}\}$ be the set of open and close markers for all the variables in $\mathcal{V}$. An *extended variable-set automaton* (extended VA, or eVA) is a tuple $\mathcal{A} = (Q, q_0, F, \delta)$, where Q, q_0, and F are the same as for variable-set automata, and δ is the transition relation consisting of letter transitions (q, a, q'), or *extended variable transitions* (q, S, q'), where $S \subseteq \text{Markers}_{\mathcal{V}}$ and $S \neq \varnothing$. A run ρ over a document $d = a_1 a_2 \cdots a_n$ is a sequence of the form:

$$\rho \;=\; q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} q_1 \xrightarrow{S_2} p_1 \xrightarrow{a_2} \;\ldots\; \xrightarrow{a_n} q_n \xrightarrow{S_{n+1}} p_n \quad (2)$$

where every S_i is a (possibly empty) set of markers, $(p_i, a_{i+1}, q_{i+1}) \in \delta$, and $(q_i, S_{i+1}, p_i) \in \delta$ whenever $S_{i+1} \neq \varnothing$, and $q_i = p_i$ otherwise. Notice that extended variable transitions and letter transitions must alternate in a run of an eVA, and that a transition with the $\varnothing$ of variable markers is only allowed when it stays in the same position.

As in the setting of ordinary VA, we say that a run ρ is *valid* if variables are opened and closed in a correct manner: the sets S_i are pairwise disjoint; for every i and every $x\vdash \in S_i$ there exists $j \geqslant i$ with $\dashv x \in S_j$; and, conversely, for every j and every $\dashv x \in S_j$ there exists $i \leqslant j$ with $x\vdash \in S_i$. For a valid run ρ we define the mapping μ^ρ that maps x to $[i,j\rangle \in \text{span}(d)$ if, and only if, $x\vdash \in S_i$, $\dashv x \in S_j$ and $i \leqslant j$. Also, we say that ρ is *accepting* if $p_n \in F$. Finally, the semantics of $\mathcal{A}$ over d, denoted by $[\![\mathcal{A}]\!]_d$ is defined as the set of all mappings μ^ρ where ρ is a valid and accepting run of $\mathcal{A}$ over D. We transfer the notion of being *sequential* (seVA) and *functional* (feVA) from normal VA to extended VA in the obvious way.

An extended VA $\mathcal{A}$ is *deterministic* if the transition relation δ of A is a partial function $\delta : Q \times (\Sigma \cup 2^{\text{Markers}_{\mathcal{V}}} \backslash \{\varnothing\}) \to Q$. If $\mathcal{A}$ is deterministic, then we define $\text{Markers}_\delta(q)$ as the set $\{S \subseteq \text{Markers}_{\mathcal{V}} \mid (q, S) \in \text{dom}(\delta)\}$. Note that, in contrast to determinism for classical NFAs, determinism as defined here does not imply

[1]As shown in [10], such behaviour also leads to a factorial blow-up when defining the join of two VA, as all possible orders between variables need to be considered. See Section 4 for further discussion.

that there is at most one run for each input document d. Instead, it implies that for every document d and every $\mu \in [\![\mathcal{A}]\!]_d$, there is exactly one valid and accepting run ρ with $\mu = \mu^\rho$. In other words: there may still be many valid accepting runs on a document d, but each such run defines a unique mapping. For instance, we could convert the VA $\mathcal{A}$ from Figure 2 into an equivalent eVA $\mathcal{A}'$ by adding a transition $(q_0, \{x\vdash, y\vdash\}, q_3)$ to δ, and removing the states q_1 and q_2, together with their associated transitions. It is easy to see that $\mathcal{A}'$ is deterministic, so all accepting runs will define an unique mapping, thus avoiding the issues that $\mathcal{A}$ has when considering the enumeration of output mappings.

The following results shows that eVA are indeed a natural variant of normal VA and that all eVA can be determinized.

Theorem 3.1. *For every* VA $\mathcal{A}$ *there exists an* eVA $\mathcal{A}'$ *such that* $\mathcal{A} \equiv \mathcal{A}'$ *and vice versa. Furthermore, if* $\mathcal{A}$ *is sequential (resp. functional), then* $\mathcal{A}'$ *is also sequential (resp. functional).*

Proposition 3.2. *For every* eVA $\mathcal{A}$ *there exists a deterministic* eVA $\mathcal{A}'$ *such that* $\mathcal{A} \equiv \mathcal{A}'$.

In Section 4 we will study in detail the complexity of these translations; to present our algorithm we only require equivalence between the models.

3.2 Constant delay evaluation algorithm

The objective of this section is to describe an algorithm that takes as input a *deterministic and sequential* eVA $\mathcal{A}$ (deterministic seVA for short) and a document d, and enumerates the set $[\![\mathcal{A}]\!]_d$ with a constant delay after pre-processing time $O(|\mathcal{A}| \times |d|)$. We start with an intuitive explanation of the algorithm's underlying idea, and then give the full algorithm.

3.2.1 Intuition

As with the majority of constant delay algorithms, in the pre-processing step we build a compact representation of the output that is used later in the enumeration step. In our case, we build a directed acyclic graph (DAG) that can then be traversed in a depth-first manner to enumerate all the output mappings. This DAG will encode all the runs of $\mathcal{A}$ over d, and its construction can be summarized as follows:

- Convert the input word d into a deterministic extended VA $\mathcal{A}_d$;
- Build the product between $\mathcal{A}$ and $\mathcal{A}_d$, and annotate the variable transitions with the position of d where they take place;
- Replace all the letters in the transitions of $\mathcal{A} \times \mathcal{A}_d$ with ε, and construct the "forward" ε-closure of the resulting graph.

We first illustrate how this construction works by means of an example. For this, consider the eVA $\mathcal{A}$ from Figure 3. It is straightforward to check that this automaton is functional (hence sequential) and deterministic. To evaluate $\mathcal{A}$ over document $d = ab$ we first

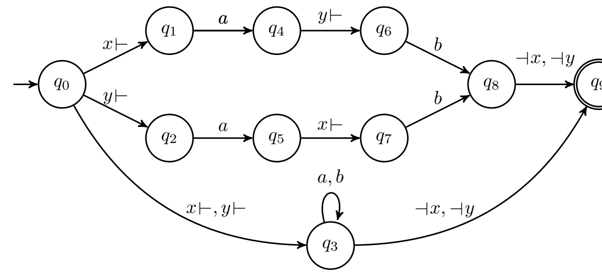
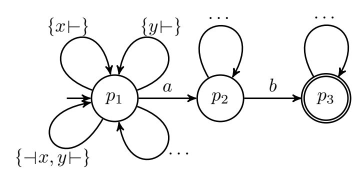


Figure 3: An extended functional VA $\mathcal{A}$.

convert the input document d into an eVA $\mathcal{A}_d$ that represents all possible ways of assigning spans over d to the variables of $\mathcal{A}$. The automaton $\mathcal{A}_d$ is a chain of $|d| + 1$ states linked by the transitions that spell out the word d. That is, $\mathcal{A}_d$ has the states $p_1, \ldots, p_{|d|+1}$, and letter transitions (p_i, d_i, p_{i+1}), with $i = 1 \ldots |d|$, and where d_i is the ith symbol of d. Furthermore, each state p_i has $2^{|\operatorname{var}(\mathcal{A})|-1}$ self loops, each labelled by a different non-empty subset of $\text{Markers}_{\operatorname{var}(A)}$. For instance, in the case of $d = ab$, the automaton $\mathcal{A}_d$ is the following:



Next, we build the product automaton $\mathcal{A} \times \mathcal{A}_d$ in the standard way (i.e. by treating variable transitions as letters and applying the NFA product construction). During construction, we take care to only create product states of the form (q, p) that are reachable from the initial product state (q_0, p_1). In addition, we annotate the variable transitions of this automaton with the position in d where the particular transition is applied. For this, we use the fact that $\mathcal{A}_d$ is a chain of states, so in the product $\mathcal{A} \times \mathcal{A}_d$, each variable transition is of the form $((q, p_i), S, (q', p_i))$. We therefore annotate the set S with the number i. We depict the resulting annotated product automaton for $\mathcal{A}$ and $d = ab$ in Figure 4 (top).

In the final step, we replace all letter transitions with ε-transitions and compute what we call the "forward" ε-closure. This is done by considering each variable transition $((q, p), (S, i), (q', p'))$ of the annotated product automaton, and then computing all the states (r, s) such that one can reach (r, s) from (q', p') using only ε transitions. We then add an annotated variable transition $((q, p), (S, i), (r, s))$ to the automaton. For instance, for the product automaton at the top of Figure 4, we would add a transition $((q_0, p_1), (x\vdash, 1), (q_4, p_2))$, due to the fact that we can reach (q_1, p_1) from (q_0, p_1) using $(x\vdash, 1)$, and we can reach (q_4, p_2) from (q_1, p_1) using ε (which replaced a). We repeat this procedure for all the variable transitions of $\mathcal{A} \times \mathcal{A}_d$, and the newly added transitions, until no new transition can be generated. In the end, we simply erase all the ε transition from the

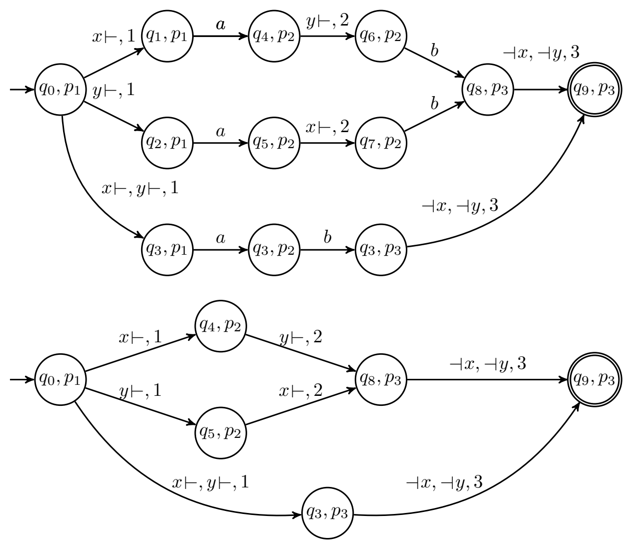


Figure 4: The annotated product automaton (top) and its "forward"ε-closure (bottom).

resulting automaton. An example of this process for the automaton $\mathcal{A}$ of Figure 3 and the document $d = ab$ is given at the bottom of Figure 4.

From the resulting DAG we can now easily enumerate $[\![\mathcal{A}]\!]_d$. For this, we simply start from the final state, and do a depth-first traversal taking all the edges backwards. Every time we reach the initial state, we will have the complete information necessary to construct one of the output mappings. For example, starting from the accepting state and moving backwards to (q_3, p_3), and then again to the initial state. From the labels along this run we can then reconstruct the mapping μ with $\mu(x) = \mu(y) = [1, 3\rangle$.

Since $\mathcal{A}$ and $\mathcal{A}_d$ are deterministic, we will never output the same mapping twice. Also, note that the time for generating each output is bounded by the number of variables in $\mathcal{A}$, and therefore the delay between outputs depends only on $|\mathcal{A}|$ (and is constant in the document).

3.2.2 The algorithm

While the previous construction works correctly, there is no need to perform the three construction phases separately in a practical implementation. In fact, by a clever merge of the three construction steps we can avoid materializing $\mathcal{A}_d$ and $\mathcal{A} \times \mathcal{A}_d$ altogether. The result is a succinct, optimized, and easily-implementable algorithm that we describe next.

There are two main differences with the construction described above and our algorithm. First, the algorithm never materializes $\mathcal{A}_d$, nor the product $\mathcal{A} \times \mathcal{A}_d$. Rather, it *traverses* this product automaton on-the-fly by processing the input document one letter at a time. Second, the algorithm does not construct the ε-closure itself, but its *reverse dual*. That is, the resulting DAG has the edge labels of the ε-closure as nodes and there is an edge from $(T, j) \to (S, i)$ in the reverse dual if we had $(q, p) \xrightarrow{(S,i)} (q', p') \xrightarrow{(T,j)} (q'', p'')$ in the ε-closure for some product states $(q, p), (q', p')$, and (q'', p'').

The algorithm builds the reverse dual DAG incrementally by processing d one letter at a time. In order to do this, it tracks at every position i ($1 \leqslant i \leqslant |d|$) the states of $\mathcal{A}$ that are *live*, i.e., the states $q \in Q$ such that there exists at least one run of $\mathcal{A}$, on the prefix $d(1, i)$ of d that ends in q. For each such state, the algorithm keeps track of the nodes in the reverse dual that represent the last variable transitions taken by runs ending in q. When appropriate, new nodes are added to the reverse dual based on this information.

The different procedures that comprise the evaluation algorithm are given in Algorithms 1 and 2. In particular, the procedure EVALUATE shown in Algorithm 1 takes a deterministic and sequential eVA $\mathcal{A}$ and a document $d = a_1 \dots a_n$ as input, and creates the reverse dual DAG that encodes all the runs of $\mathcal{A}$ over d. The procedure ENUMERATE shown in Algorithm 2 enumerates all the resulting mappings. Before discussing these procedures in detail, we need to elaborate on the data structures used.

Data structures. We store the reverse dual DAG by using the adjacency list representation. Each node n in this DAG is a pair $((S, i), l)$ where $S \subseteq \text{Markers}_{\mathcal{V}}$, $i \in \mathbb{N}$, and a l is the list of nodes to which n has outgoing edges. Given a node n, the method n.`content` retrieves the pair (S, i) while the method n.`list` retrieves the adjacency list l. A special node, denoted by $\perp$ will be used as the sink node (playing the same role as the initial state of $\mathcal{A} \times \mathcal{A}_d$).

The algorithm makes extensive use of list operations. Lists are represented as a pair (s, e) of pointers to the start and end elements in a singly linked list of elements. Elements are created and never modified. The only exception to this is an element whose `next` pointer is `null`. Such an element may have its `next` pointer updated, but only once. Lists are endowed with six methods: `begin`, `next`, `atEnd`, `add`, `lazycopy`, and `append`. The first three methods `begin`, `next`, and `atEnd` are standard methods for iterating through a list. Specifically, `begin` starts the iteration from the beginning (i.e. it locates the position *before* the first node), `next` gives the next node in the list, and `atEnd` tells whether the iteration is at the end or not. The last three methods `add`, `lazycopy` and `append` are methods for modifying or extending a list $l = (s, e)$. `add` receives a node n and inserts n at the beginning of l (i.e., it creates a new element whose payload is n and whose `next` pointer is s, and updates $l := (s', e)$ with s' pointing to this new element). `lazycopy` makes a lazy copy of l by returning a copy of the pair (s, e). This copy is not updated on operations to l (such as, `add`, which would modify s). `append` receives another list $l' = (s', e')$ and appends l' at the end of $l = (s, e)$ by updating the `next` pointer of e to s' and subsequently updating l to (s, e'). Note that all of these operations are clearly $O(1)$ operations.

Evaluation. The procedure EVALUATE maintains a list $list_q$ of nodes, for every state q of $\mathcal{A}$. If $list_q$ is empty, then q is not live for the current letter position. Otherwise, q is live and $list_q$ contains the nodes in the reverse

Algorithm 1 Evaluate $\mathcal{A}$ over the document $a_1 \ldots a_n$

procedure Evaluate($\mathcal{A}$, $a_1 \ldots a_n$)
    for all $q \in Q \backslash \{q_0\}$ **do**
        $list_q \leftarrow \epsilon$
    $list_{q_0} \leftarrow [\perp]$
    for $i := 1$ to n **do**
        Capturing(i)
        Reading(i)
    Capturing($n+1$)
    Enumerate($\{list_q\}_{q \in Q}, F$)

procedure Capturing(i)
    for all $q \in Q$ **do**
        $list_q^{old} \leftarrow list_q$.`lazycopy`
    for all $q \in Q$ **with** $list_q^{old} \neq \epsilon$ **do**
        for all $S \in \text{Markers}_\delta(q)$ **do**
            node $\leftarrow$ `Node`($(S, i), list_q^{old}$)
            $p \leftarrow \delta(q, S)$
            $list_p$.`add`(node)

procedure Reading(i)
    for all $q \in Q$ **do**
        $list_q^{old} \leftarrow list_q$
        $list_q \leftarrow \epsilon$
    for all $q \in Q$ **with** $list_q^{old} \neq \epsilon$ **do**
        $p \leftarrow \delta(q, a_i)$
        $list_p$.`append`($list_q^{old}$)

Algorithm 2 Enumerate all mappings

procedure Enumerate($\{list_q\}_{q \in Q}$, F)
    for all $q \in F$ **with** $list_q \neq \epsilon$ **do**
        `EnumAll`($list_q, \epsilon$)

procedure EnumAll($list$, map)
    $list$.`begin`
    while $list$.`atEnd` = `false` **do**
        node $\leftarrow list$.`next`
        if node $= \perp$ **then**
            `Output`(map)
        else
            $(S, i) \leftarrow$ node.`content`
            EnumAll(node.`list`, $(S, i) \cdot$ map)

dual DAG that represent the last variable transitions taken by runs of $\mathcal{A}$ on the current prefix that end in q. Initially, $list_q$ is empty for every state q except the initial state q_0, which is initialized to the singleton list containing the special sink node $\perp$. Evaluate then alternates between calls to Capturing(i) and Reading(i), where i is a letter position in d (recall that all the runs of an extended automata alternate between variable and letter transitions and start with a variable transition, cf. (2)). Capturing(i) simulates the variable transitions that $\mathcal{A}$ does immediately before reading the letter a_i, and modifies the reverse dual DAG accordingly. Similarly, Reading(i) simulates what $\mathcal{A}$ does when reading the letter a_i of the input. Finally, Capturing($n+1$) simulates the last variable transition of $\mathcal{A}$.

In Capturing(i) we first make a lazy copy of all the lists. We then try to extend the runs of $\mathcal{A}$ from each state q that was live at position $i-1$ (i.e., $list_q^{old} \neq \epsilon$) by executing a variable transition. If we can do this (i.e. there is a transition of the form (q, S, p) in $\mathcal{A}$), we create a new node n labeled by (S, i) that has an edge to each node in $list_q^{old}$. Finally, we add n to the beginning of the list $list_p$, thus recording that $\mathcal{A}$ can be in state p after executing the ith variable transition. Notice that it is possible that two transitions enter the same state p (like the transitions reaching the accepting state in Figure 3). To accommodate for this, our algorithm adds the new node at the beginning of the list, so by traversing the entire list we get the information about all the runs.

It is important to note that in Capturing(i) we do not overwrite the lists $list_q$ that were created in Reading($i-1$) for $i > 1$. This is necessary to correctly keep track of the situation in which no transition using variable markers was triggered in Capturing(i) (i.e. when $S = \varnothing$ in our run). On a run of a sequential extended variable-set automaton this can happen for instance when we have self loops (as in e.g. state q_3 in Figure 3). This way, the list $list_q$ is kept for the next iteration; i.e. Reading(i) can again continue from q since no variable markers were used in between.

In Reading(i) we simulate what happens when $\mathcal{A}$ reads the letter a_i of the input document by updating the lists of the states that $\mathcal{A}$ reaches in this transition. That is, we first mark all lists as "old" lists, and then set $list_q$ to empty. Then for each live state q (i.e., $list_q^{old} \neq \epsilon$, hence $\mathcal{A}$ was in q immediately before reading a_i), and the transitions of the form (q, a_i, p), we append the list $list_q^{old}$ at the end of the list $list_p$. Appending this list at the end is done in order to accommodate the fact that two letter transitions can enter the same state p while reading a_i (see e.g. the state q_8 in the automaton from Figure 3). Note here that each $list_q^{old}$ is appended to at most one $list_p$, since $\mathcal{A}$ is deterministic.

Enumeration. At the end of Evaluate, procedure Enumerate simply traverses the constructed reverse dual DAG in a depth first manner. In this way, Enumerate traces all the accepting runs (since it starts from an accepting state), and outputs a string allowing us to reconstruct the mapping.

Example. Next we give an example detailing the situations that could occur while running Algorithm 1. For this, consider the deterministic seVA $\mathcal{A}$ from Figure 3 and an input document $d = ab$. In this case we have that $[\![\mathcal{A}]\!]_d = \{\mu_1, \mu_2, \mu_3\}$, where:

- $\mu_1(x) = [1, 3\rangle,\ \mu_1(y) = [2, 3\rangle$;
- $\mu_2(x) = [2, 3\rangle,\ \mu_2(y) = [1, 3\rangle$; and
- $\mu_3(x) = [1, 3\rangle,\ \mu_3(y) = [1, 3\rangle$.

To show how Algorithm 1 works, in Figure 5 we provide the state of all the active lists after completion of each phase of the algorithm. To stress that we are

Stage	Non-empty lists
Initial	$list^0_{q_0} = [\bot]$
CAPTURING(1)	$list^0_{q_0} = [\bot]$
	$list^0_{q_1} = [\texttt{node}((\{x\vdash\}, 1), [\bot])]$
	$list^0_{q_2} = [\texttt{node}((\{y\vdash\}, 1), [\bot])]$
	$list^0_{q_3} = [\texttt{node}((\{x\vdash, y\vdash\}, 1), [\bot])]$
READING(1)	$list^1_{q_4} = list^0_{q_1}$
	$list^1_{q_5} = list^0_{q_2}$
	$list^1_{q_3} = list^0_{q_3}$
CAPTURING(2)	$list^1_{q_4} = list^0_{q_1}$
	$list^1_{q_5} = list^0_{q_2}$
	$list^1_{q_3} = list^0_{q_3}$
	$list^1_{q_6} = [\texttt{node}((\{y\vdash\}, 2), list^1_{q_4})]$
	$list^1_{q_7} = [\texttt{node}((\{x\vdash\}, 2), list^1_{q_5})]$
	$list^1_{q_9} = [\texttt{node}((\{\dashv x, \dashv y\}, 2), list^1_{q_3})]$
READING(2)	$list^2_{q_3} = list^1_{q_3}$
	$list^2_{q_8} = [list^1_{q_6}, list^1_{q_7}]$
CAPTURING(3)	$list^2_{q_3} = list^1_{q_3}$
	$list^2_{q_8} = [list^1_{q_6}, list^1_{q_7}]$
	$list^2_{q_9} = [\texttt{node}((\{\dashv x, \dashv y\}, 3), list^2_{q_8}),$ $\texttt{node}((\{\dashv x, \dashv y\}, 3), list^2_{q_3})]$

Figure 5: The state of non-empty lists after executing each stage of the algorithm.

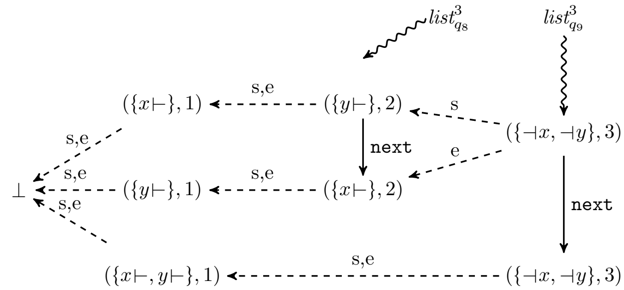


Figure 6: DAG created by Algorithm 1 to record the output mappings.

talking about the state of some list $list_q$ during the iteration i of Algorithm 1, that is, about the state of the list after executing READING(i) or CAPTURING(i), we will use the notation $list^i_q$. To keep the notation simple, we also denote lists using the array notation.

At the beginning only the list $list_{q_0}$ corresponding to the initial state of $\mathcal{A}$ is non-empty. When CAPTURING(1) is triggered, we create three new nodes, each corresponding to the variable transitions leaving the state q_0. These nodes are then added to the appropriate lists. In READING(1) we "move" the non-empty lists by renaming their state. For instance, since A can go from q_1 to q_4 while reading $a_1 = a$, the list $list^0_{q_1}$ now becomes $list^1_{q_4}$, signalling that q_4 is one of the states where $\mathcal{A}$ can be at this point. The same is done by the other two transition reading the letter a. Notice that the list $list_{q_0}$ becomes empty at this point.

Next, CAPTURING(2) is executed. Here, the lists that were non-empty after READING(1) will remain unchanged after CAPTURING(2), simulating the situation when no variable bindings were used in the run of $\mathcal{A}$ over d after processing the first letter. Other variable transitions that can be triggered create new nodes and add them at the beginning of the appropriate lists.

READING(2) again "moves" the lists according to what $\mathcal{A}$ does when reading $a_2 = b$. The lists $list^1_{q_3}$ gets propagated (simulating a self loop). A more interesting situation occurs when the transitions $\delta(q_6, b) = q_8$ and $\delta(q_7, b) = q_8$ are processed. Since they both reach q_8, we first append the list $list^1_{q_6}$ at the end of (the empty list) $list^2_{q_8}$, and then to keep track that one can also get here from q_7, also append the list $list^1_{q_7}$ at the end of (now non empty list) $list^2_{q_8}$. Since these are the only way that $\mathcal{A}$ can move while reading b, we forget about all the other lists.

Finally, CAPTURING(3) keeps track of what happens during the last variable transition of $\mathcal{A}$. There are two transitions that can reach the accepting state q_9, and they get added to the list $list^3_{q_9}$. Note that the two lists from READING(2) also remain non-empty at this stage.

The DAG created by Algorithm 1 is given in Figure 6. Here the dashed edges point to the list corresponding to the node with this label (i.e. the list representation (s, e)). For instance, $\texttt{node}((\{x\vdash\}, 0), \bot)$ corresponds to the edge between the node with the label $(\{x\vdash\}, 0)$ and its associated list $\bot = list^0_{q_0}$. Full edges link the nodes that belong to the same list, and curvy edges to the start of a list generated after CAPTURING(3).

To enumerate the answers, we now call the procedure ENUMERATE, passing it as a parameter all the lists corresponding to the final states of $\mathcal{A}$. Since $\mathcal{A}$ has only one final state, the procedure will trigger only ENUMALL($list_{q_9}, \varepsilon$). This procedure now recursively traverses the structure of connected lists created by Algorithm 1 in a depth-first manner generating the output mappings. For instance, the mapping μ_1, with $\mu_1(x) = [1, 3\rangle$ and $\mu_2(y) = [2, 3\rangle$ is generated by traversing the upper most path from $(\{\dashv x, \dashv y\}, 2)$ until reaching $\bot$, and similarly for other mappings.

Correctness. To prove the correctness of the above algorithm, we first introduce some notation. For encoding mappings in the enumeration procedure, we assume that mappings are sequences of the form $(S_1, i_1) \ldots (S_m, i_m)$ where $S_j \subseteq \text{Markers}_{\mathcal{V}}$, $i_1 < \ldots < i_m$ and variables in $S_1 \ldots S_m$ are open and closed in a correct manner, i.e. like in the definition of a run of an extended variable set automata. Clearly, from a sequence $M = (S_1, i_1) \ldots (S_m, i_m)$ we can obtain a mapping μ^M and viceversa. For this reason, in the sequel we call M and μ mappings without making any distinction. Furthermore, we say that a sequence $M = (S_1, i_1) \dots (S_k, i_k)$ is a *partial mapping* if it is the prefix sequence of some mapping, i.e., it can be extended to the right to create a mapping. This is useful to represent the output of partial run of $\mathcal{A}$ over d; that is, if ρ is of the form:

$$\rho \;=\; q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} q_1 \xrightarrow{S_2} p_1 \xrightarrow{a_2} \;\dots\; \xrightarrow{a_i} q_i \xrightarrow{S_{i+1}} p_i$$

where $i \leqslant |d|$, the mapping μ^ρ is not necessarily well-defined, or is possibly incomplete. We therefore define a partial mapping M of ρ, denoted by $\text{Out}(\rho)$, as the concatenation of all the pairs (S_j, j) where $S_j \neq \varnothing$, in an increasing order on j. For instance, in the run $\rho = q_0 \xrightarrow{\{x\vdash\}} p_0 \xrightarrow{a_1} q_1 \xrightarrow{\varnothing} p_1 \xrightarrow{a_2} q_2 \xrightarrow{\{y\vdash\}} p_2$ we will have that $\text{Out}(\rho) = (\{x\vdash\}, 0)\,(\{y\vdash\}, 2)$. Note that in the case that ρ is an accepting run of A, it is then clear that $\text{Out}(\rho)$ defines the mapping μ^ρ.

The proof that Algorithm 1 correctly enumerates all the mappings in $[\![\mathcal{A}]\!]_d$ without repetitions follows from the invariant stated in the lemma below.

Lemma 3.3. *Let $d = a_1 \dots a_n$ be a document and $\mathcal{A}$ an extended variable-set automaton that is deterministic and sequential (deterministic* seVA*). Then for every $0 \leqslant i \leqslant n$, the following two statements are equivalent:*

1. *There exists a run of $\mathcal{A}$ over $a_1 \cdots a_i$ of the form $\rho = q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} q_1 \;\dots\; \xrightarrow{a_i} q_i \xrightarrow{S_{i+1}} p_i$.*
2. *After executing* Capturing$(i+1)$ *in Algorithm 1, it holds that $list_{p_i} \neq \epsilon$ and there is partial output M of* EnumAll*$(list_{p_i}, \epsilon)$ with $M = \text{Out}(\rho)$.*

The proof of the lemma is done by a detailed induction on the number of steps of the algorithm and can be found in the appendix. Note that the case when $i = 0$ corresponds to a run over the empty word ε (i.e. processing the part of d "before" a_1), thus simulating the first variable transition of $\mathcal{A}$. With the invariant proved in Lemma 3.3, we can now easily show that running Evaluate$(\mathcal{A}, d)$ will enumerate all of the mappings in $[\![\mathcal{A}]\!]_d$ and only those mappings. Indeed, if $\mu \in [\![\mathcal{A}]\!]_d$, this means that there is an accepting run ρ such that $\mu^\rho = \mu$, so by Lemma 3.3, the algorithm will output M with $M = \text{Out}(\rho)$. On the other hand, if Evaluate$(\mathcal{A}, d)$ produces an output M, we can match this output with a run ρ_M. Furthermore, since the output was produced from an accepting state, and since $\mathcal{A}$ is sequential, this means ρ_M is valid, so $\mu^{\rho_M} = \mu^M \in [\![A]\!]_d$ as desired.

Finally, we need to show that Algorithm 1 does not enumerate any answer twice when executed over a deterministic seVA $\mathcal{A}$ and a document d. For this, observe that if we have two accepting runs ρ and ρ' of $\mathcal{A}$ over d such that $\mu^\rho = \mu^{\rho'}$, then $\rho = \rho'$. This follows from the fact that $\mathcal{A}$ is deterministic. Therefore, it follows from Lemma 3.3 that there is a one to one correspondence between accepting runs of A and outputs of Algorithm 1, which gives us the desired result.

Complexity. It is rather straightforward to see that the pre-processing step takes time $O(|\mathcal{A}| \times |d|)$. Namely, for each letter a_i of d we run the procedures Capturing(i) and Reading(i) once. These two procedures simply scan the transitions of the automaton and manipulate the list pointers as needed, thus taking $O(|\mathcal{A}|)$ time, where $|\mathcal{A}|$ is measured as the number of transitions of the automaton.

As far as the enumeration is concerned, Algorithm 2, traverses the graph generated in the pre-processing step in a depth-first manner. From Lemma 3.3, it follows that all the paths in the constructed graph must reach the initial node $\perp$ and that the length of each path is linear in the number of variables. Thus, we are able to enumerate the output by taking only constant delay (i.e. constant in the size of the document) between two consecutive mappings.

Note that the actual delay is not really dependent on the entire automaton $\mathcal{A}$, as allowed by the definition of constant delay, but depends only on the number of variables. We argue that this is the best delay that can be achieved, because to write down a single output mapping one needs at least the time that is linear in the number of variables.

4. EVALUATING REGULAR SPANNERS

The previous section shows an algorithm that evaluates a deterministic and sequential extended VA (deterministic seVA for short) $\mathcal{A}$ over a document d with constant-delay enumeration after $O(|\mathcal{A}| \times |d|)$ pre-processing. Since the wider objective of this algorithm is to evaluate regular spanners, in this section we present a fine-grained study of the complexity of transforming an arbitrary regular spanner, expressed in $\text{RGX}^{\{\pi,\cup,\bowtie\}}$ or $\text{VA}^{\{\pi,\cup,\bowtie\}}$ to a deterministic seVA. This will illustrate the real cost of our constant delay algorithm for evaluating regular spanners.

Because it is well-known that RGX formulas can be translated into VA in linear time [10], we can focus our study on the setting where spanners are expressed in $\text{VA}^{\{\pi,\cup,\bowtie\}}$. We first study how to translate arbitrary VAs into deterministic seVAs, and then turn to the algebraic constructs. For the sake of simplification, throughout this section we assume the following notation: given a VA $\mathcal{A} = (Q, q_0, F, \delta)$, $n = |Q|$ denotes the number of states, $m = |\delta|$ the number of transitions, and $\ell = |\text{var}(\mathcal{A})|$ the number of variables in $\mathcal{A}$.

To obtain a sequential VA from a VA, we can use a construction similar to the one presented in [11]. This yields a sequential VA with $2^n 3^\ell$ states that can later be extended and determinized (see Theorem 3.1 and Proposition 3.2, respectively). Unfortunately, following these steps would yield an automaton whose size is double exponential in the size of the original VA. The first positive result in this section is that we can actually transform a VA into a deterministic seVA avoiding this double exponential blow-up.

Proposition 4.1. *For any* VA *$\mathcal{A}$ there exists an equivalent deterministic* seVA *$\mathcal{A}'$ with at most $2^n 3^\ell$ states and $2^n 3^\ell (2^\ell + |\Sigma|)$ transitions.*

Therefore, evaluating an arbitrary VA with constant delay can be done with preprocessing that is exponential

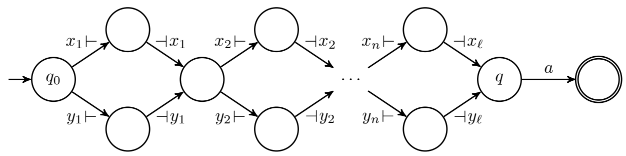


Figure 7: A sequential VA with ℓ variables such that every equivalent eVA has $O(2^\ell)$ transitions.

in the size of the VA and linear in the document. However, note that the resulting deterministic seVA is exponential both in the number of states and in the number of variables of the original VA. While having an automaton that is exponential in the number of states is to be expected due to the deterministic restriction of the resulting VA, it is natural to ask whether there exists a subclass of VA where the blow-up in the number of variables can be avoided.

The two subclasses of VA that were shown to have good algorithmic properties [13, 17] are sequential VA and functional VA, so we will consider if the cost of translation is smaller in these cases. In the more general case of sequential VA we can actually show that the blow-up in the number of variables is inevitable. The main issue here is that preserving the sequentiality of a VA when transforming it to an extended VA can be costly. To illustrate this, consider the automaton in Figure 7. In this automaton any path between q_0 and q_F opens and closes exactly one variable in $\{x_i, y_i\}$, for each $i \in \{1, \ldots, n\}$. Therefore, to simulate this behaviour in an extended VA (which disallows two consecutive variable transitions), we need 2^ℓ transitions between the initial and final states, one for each possible set of variables. More formally, we have the following proposition.

PROPOSITION 4.2. *For every $\ell > 0$ there is a sequential VA $\mathcal{A}$ with $3\ell + 2$ states, $4\ell + 1$ transitions, and 2ℓ variables, such that for every extended* VA *$\mathcal{A}'$ equivalent to $\mathcal{A}$ it is the case that $\mathcal{A}'$ has at least 2^ℓ transitions.*

On the other hand, if we consider functional VA, the exponential factor depending on the number of variables can be eliminated when translating a functional VA into a deterministic seVA.

PROPOSITION 4.3. *For any functional* VA *$\mathcal{A}$ there exists an equivalent deterministic* seVA *$\mathcal{A}'$ with at most 2^n states and $2^n(n^2 + |\Sigma|)$ transitions.*

Due to this, and the fact that functional VA are probably the class of VA most studied in the literature [10, 13, 11], for the remainder of this section we will be working with functional VA.

Now we proceed to study how to apply the algebraic operators to evaluate regular spanners. In [10], it was shown that any regular spanner (i.e. a join-union-projection expression built from RGX or VA as atoms) is in fact equivalent to a single VA, and effective constructions were given. In particular, it is known that for every pair of VA $\mathcal{A}_1$ and $\mathcal{A}_2$, there exists a VA $\mathcal{A}$ of exponential size such that $[\![\mathcal{A}]\!]_d = [\![\mathcal{A}_1]\!]_d \bowtie [\![\mathcal{A}_2]\!]_d$. The exponential blow-up comes from the fact that each transition is equipped with at most one variable, and two variable transitions can occur consecutively. Therefore, one needs to consider all possible orders of consecutive variable transitions when computing a product (see [10]). On the other hand, as shown by a subset of the author's in their previous work [18], and independently in [13], this blow-up can be avoided when working with functional VA. In the next proposition, we generalize this result to extended VA[2].

PROPOSITION 4.4. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two functional* eVA*, and $Y \subset \mathcal{V}$. Furthermore, let $\mathcal{A}_3$ and $\mathcal{A}_4$ be two functional* eVA*s that use the same set of variables. Then there exist functional* eVA*s $\mathcal{A}_\bowtie$, $\mathcal{A}_\cup$, and $\mathcal{A}_\pi$ such that:*

- *$\mathcal{A}_\bowtie \equiv \mathcal{A}_1 \bowtie \mathcal{A}_2$, and $\mathcal{A}_\bowtie$ is of size $|\mathcal{A}_1| \times |\mathcal{A}_2|$.*
- *$\mathcal{A}_\cup \equiv \mathcal{A}_3 \cup \mathcal{A}_4$, and $\mathcal{A}_\cup$ is of size $|\mathcal{A}_3| + |\mathcal{A}_4|$.*
- *$\mathcal{A}_\pi \equiv \pi_Y \mathcal{A}_1$, and $\mathcal{A}_\pi$ is of size $|\mathcal{A}_1|$.*

Combining these results we can now determine the precise cost of compiling a regular spanner γ into a deterministic seVA automaton that can then be used by the algorithm from Section 3 to enumerate $[\![\gamma]\!]_d$ with constant delay, for an arbitrary document d. More precisely, we have the following.

PROPOSITION 4.5. *Let γ be a regular spanner in* $\text{VA}^{\{\pi,\cup,\bowtie\}}$ *using k functional* VA *as input, each of them with at most n states. Then there exists an equivalent deterministic* seVA *$\mathcal{A}_\gamma$ with at most 2^{n^k} states, and at most $2^{n^k} \cdot (n^{2k} + |\Sigma|)$ transitions.*

In this case the 2^n factor from Proposition 4.3 turns to 2^{n^k}, thus making it double-exponential depending on the number of algebraic operations used in γ. Ideally, we would like to isolate a subclass of regular spanners for which this factor can be made single exponential. Unfortunately, in the general case we do not know if the double exponential factor 2^{n^k} can be avoided. The main problem here is dealing with projection, since it does not preserve determinism, thus causing an additional blow-up due to an extra determinization step. However, if we consider $\text{VA}^{\{\cup,\bowtie\}}$, we can obtain the following.

PROPOSITION 4.6. *Let γ be a regular spanner in* $\text{VA}^{\{\cup,\bowtie\}}$ *using k functional* VA *as input, each of them with at most n states. Then, there exists an equivalent deterministic* seVA *$\mathcal{A}_\gamma$ with at most $2^{n \cdot k}$ states, at most $2^{n \cdot k} \cdot (n^{2k} + |\Sigma|)$ transitions.*

Overall, compiling arbitrary VA or expressions in $\text{VA}^{\{\pi,\cup,\bowtie\}}$ into deterministic seVA can be quite costly. However, restricting to the functional setting and disallowing projections yields a class of document spanners where the size of the resulting deterministic seVA is manageable. In terms of practical applicability, it is also interesting to note that all of these translations can be fed to Algorithm 1 on-the-fly, thus rarely needing to materialize the entire deterministic seVA.

[2]Note that since [13] does not consider extended VA, the size of the join automaton is $O(n^4)$, and not quadratic.

5. COUNTING DOCUMENT SPANNERS

In this section we study the problem of counting the number of output mappings in $[\![\gamma]\!]_d$, where γ is a document spanner. Counting the number of outputs is strongly related to the enumeration problem [19] and can give some evidence on the limitations of finding constant delay algorithms with better precomputation phases. Formally, given a language $\mathcal{L}$ for specifying document spanners, we consider the following problem:

Problem:	COUNT[$\mathcal{L}$]
Input:	An expression $\gamma \in \mathcal{L}$, a document d.
Output:	$\lvert[\![\gamma]\!]_d\rvert$

It is common that constant delay enumeration algorithms can be extended to count the number of outputs efficiently [19]. We show that this is the case for our algorithm over deterministic seVA.

THEOREM 5.1. *Given a deterministic sequential extended* VA $\mathcal{A}$ *and a document* d*,* $\lvert[\![\gamma]\!]_d\rvert$ *can be computed in time* $O(|\mathcal{A}| \times |d|)$.

Therefore, COUNT[$\mathcal{L}_1$], where $\mathcal{L}_1$ is the class of deterministic seVA, can be computed in polynomial time in combined complexity. The algorithm for COUNT[$\mathcal{L}_1$] can be found in the appendix. This algorithm is a direct extension of Algorithm 1, modified to keep the number of (partial) output mappings in each state instead of a compact representation of the mappings (i.e. $list_q$).

Unfortunately, the efficient algorithm of Theorem 5.1 cannot be extend beyond the class of sequential deterministic VA, that is, we show that COUNT[fVA] is a hard counting problem, where fVA is the class of functional VA (that are not necessarily extended). First, we note that COUNT[fVA] is not a #P-hard problem – a property that most of the hard counting problems usually have in the literature [23]. We instead show that COUNT[fVA] is complete for the class SPANL [2], a counting complexity class that is included in #P and is incomparable with FP, the class of functions computable in polynomial time.

Intuitively, SPANL is the class of all functions f for which we can find a non-deterministic Turing machine M with an output tape, such that $f(x)$ equals the number of different outputs (i.e. without repetitions) that M produces in its accepting runs on an input x, and M runs in logarithmic space. We say that a function f is SPANL-complete if $f \in$ SPANL and every function in SPANL can be reduced to f by log-space parsimonious reductions (see [2] for details). It is known [2] that SPANL functions can be computed in polynomial time if, and only if, all the polynomial hierarchy is included in P (in particular NP = P). By well-accepted complexity assumptions the SPANL-hardness of COUNT[fVA] hence implies that counting the number of outputs of a fVA over a document cannot be done in polynomial time.

THEOREM 5.2. COUNT[fVA] *is* SPANL*-complete.*

It is not hard to see that any functional VA can be converted in polynomial time into an functional extended VA (see [18]). Therefore, the above theorem also implies intractability in counting the number of output mappings of a functional extended VA. Given that all other classes of regular spanners studied in this paper (i.e. sequential, non-sequential, etc) include either the class of functional VA or functional extended VA, this implies that COUNT[$\mathcal{L}$] is intractable for every $\mathcal{L}$ different from $\mathcal{L}_1$, the class of deterministic seVA.

In Section 4 we have shown that enumerating the answers of a functional VA with constant delay can be done after a pre-computation phase that takes the time linear in the document but exponential in the document spanner. The big question that is left to answer is whether enumerating the answers of a functional VA can be done with a lower pre-computing time, ideally $O(|\mathcal{A}| \times |d|)$. Given that constant delay algorithms with efficient pre-computation phases usually imply the existence of efficient counting algorithms [19], Theorem 5.2 sheds some light that it may be impossible to find a constant delay algorithm that has pre-computation time better than $O(2^{|\mathcal{A}|} \times |d|)$, that is obtained by determinizing a fVA and running the algorithm from Section 3. Of course, this does not establish that a constant delay algorithm with precomputation phase sub-exponential in $\mathcal{A}$ (i.e. $o(2^{|\mathcal{A}|} \times |d|)$) for fVA cannot exist, since we are relying on the conjuncture that constant delay algorithms with efficient precomputation phase implies efficient counting algorithms. We leave it as an open problem whether this is indeed true.

6. CONCLUSIONS

We believe that the algorithm described in Section 3 is a good candidate algorithm to evaluate regular document spanners in practice. Throughout the paper we have provided a plethora of evidence for this claim. First, the proposed algorithm is intuitive and can be described in a few lines of code, lending itself to easy implementations. Second, its running time is very efficient for the class of deterministic sequential extended VA, and the latter in fact subsumes the class of all regular spanners. Third, we have shown the cost of executing our algorithm on arbitrary regular spanners, obtaining bounds that, although not ideal, are reasonable for a wide range of spanners usually encountered in practice. Finally, we have shown that better precomputation times for arbitrary regular spanners are not very likely, as one would expect to be able to compute the number of their outputs more efficiently.

In terms of future directions, we are working on implementing the algorithm from Section 3 and testing it in practice. We are also looking into the fine points of optimizing its performance, especially with respect to the different translations given in Section 4. As far as theoretical aspects of this work are concerned, we are also interested in establishing hard lower bounds for constant delay algorithms, that do not relay on conjectured claims.

7. REFERENCES

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Carme Álvarez and Birgit Jenner. A very hard log-space counting class. *Theoretical Computer Science*, 107(1):3–30, 1993.

[3] Marcelo Arenas, Francisco Maturana, Cristian Riveros, and Domagoj Vrgoč. A framework for annotating csv-like data. *Proc. VLDB Endow.*, 9(11):876–887, July 2016.

[4] Guillaume Bagan. Mso queries on tree decomposable structures are computable with linear delay. In *CSL*, volume 4207, pages 167–181. Springer, 2006.

[5] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proc. of CSL*, pages 208–222, 2007.

[6] Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, and Shivakumar Vaithyanathan. Systemt: An algebraic approach to declarative information extraction. In *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden*, pages 128–137, 2010.

[7] Laura Chiticariu, Yunyao Li, and Frederick R. Reiss. Rule-based information extraction is dead! long live rule-based information extraction systems! In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013*, pages 827–832, 2013.

[8] Bruno Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009.

[9] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Cleaning inconsistencies in information extraction via prioritized repairs. In *PODS*, pages 164–175, 2014.

[10] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *Journal of the ACM*, 62(2), 2015.

[11] Dominik D. Freydenberger. A logic for document spanners. In *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*, pages 13:1–13:18, 2017.

[12] Dominik D Freydenberger and Mario Holldack. Document spanners: From expressive power to decision problems. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 48. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[13] Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. Joining Extractions of Regular Expressions. *PODS 2018 (to appear)*, abs/1703.10350, 2017.

[14] Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *ACM Transactions on Computational Logic (TOCL)*, 14(4):25, 2013.

[15] Benny Kimelfeld. Database principles in information extraction. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 156–163, 2014.

[16] Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, Shivakumar Vaithyanathan, and Huaiyu Zhu. Systemt: a system for declarative information extraction. *SIGMOD Record*, 37(4):7–13, 2008.

[17] Francisco Maturana, Cristian Riveros, and Domagoj Vrgoč. Document Spanners for Extracting Incomplete Information: Expressiveness and Complexity. *PODS 2018 (to appear)*, abs/1707.00827, 2017.

[18] Andrea Morciano, Martín Ugarte, and Stijn Vansummeren. Automata-based evaluation of AQL queries. *Technical report, Université Libre de Bruxelles*, 2016.

[19] Luc Segoufin. Enumerating with constant delay the answers to a query. In *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 10–20, 2013.

[20] Luc Segoufin. A glimpse on constant delay enumeration. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, pages 13–27, 2014.

[21] Luc Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015.

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

[23] Leslie G Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

APPENDIX

A. PROOFS FROM SECTION 3

Proof of Theorem 3.1

We will show that given a VA $\mathcal{A}$, one can construct an equivalent extended VA (eVA) $\mathcal{A}'$ and vice versa. Both of these constructions have the property that, if the input automaton is sequential or functional, then the output automaton preserves this property.

Let $\mathcal{A} = (Q, q_0, F, \delta)$ be a VA. The resulting EVA $\mathcal{A}'$ should produce valid runs that alternate between letter transitions and extended variable transitions. To this end, we say that a variable-path between two states p and q is a sequence $\pi : p = p_0 \xrightarrow{v_1} p_1 \xrightarrow{v_2} \ldots \xrightarrow{v_n} p_n = q$ such that $(p_i, v_{i+1}, p_{i+1}) \in \delta$ are variable transitions and $v_i \neq v_j$ for every $i \neq j$. Since all transitions in π are variable transitions, we define $\text{Markers}(\pi) = \{v_1, \ldots, v_n\}$ as the union of all variable markers appearing in π.

Consider now the following extended VA $\mathcal{A}' = (Q, q_0, F, \delta')$ where $\delta' = \{(p, a, q) \in \delta \mid a \in \Sigma\} \cup \delta_{\text{ext}}$ and $(p, S, q) \in \delta_{\text{ext}}$ if, and only if, there exists a variable-path π between p and q such that $\text{Markers}(\pi) = S$. Intuitively, this construction condenses variable transitions into a single extended transition. It does so in a way that it can be assured that two consecutive extended transitions are not needed, but also, preserving all possible valid runs. The equivalence $[\![\mathcal{A}]\!]_d = [\![\mathcal{A}']\!]_d$ for every document d follows directly from the construction and definition of a variable-path.

The opposite direction follows a similar idea, namely, a run in $\mathcal{A}'$ can be separated into single variable marker transitions in $\mathcal{A}$ since each extended transition can be separated into a variable-path in $\mathcal{A}$. Formally, consider a EVA $\mathcal{A}' = (Q', q_0', F', \delta')$. The equivalent VA $\mathcal{A}$ construction is straightforward: for every extended transition between two states, a single path must be created between those two states such that they have the same effect as the single extended transition. The only issue to consider is that one must preserve an order between variable markers in such a way that $\mathcal{A}$ does not open and close a variable in the wrong order. To this end, given an arbitrary order $\preceq$ of variables $\mathcal{V}$, we can expand this order over $\text{Markers}_{\mathcal{V}}$ such that $x{\vdash} \preceq {\dashv}y$, and $x \preceq y$ implies $x{\vdash} \preceq y{\vdash}$ and ${\dashv}x \preceq {\dashv}y$. Namely, two different variable markers follow the original order but all opening markers precede closing markers. From this, in every extended transition set S we can find a first and last marker in the set, following the mentioned order. Also, we can find for each marker, a successor marker in S, as the one that goes after, following the induced order.

Consider now the VA $\mathcal{A} = (Q' \cup Q_{\text{ext}}, q_0, F, \delta)$ where $Q_{\text{ext}} = \{\, q^v_{(p,S,p')} \mid (p, S, p') \in \delta' \text{ and } v \in S\}$, $\delta = \{(p, a, q) \in \delta' \mid a \in \Sigma\} \cup \delta_{\text{first}} \cup \delta_{\text{succ}} \cup \delta_{\text{last}} \cup \delta_{\text{one}}$ and:

$$\delta_{\text{first}} = \{(p, v, q^v_{(p,S,p')}) \mid v \text{ is the } \preceq\text{-minimum element in } S\ \}$$
$$\delta_{\text{suc}} = \{(q^v_{(p,S,p')}, v', q^{v'}_{(p,S,p')}) \mid v, v' \in S \text{ and } v' \text{ is the } \preceq\text{-successor of } v \text{ in } S\}$$
$$\delta_{\text{last}} = \{(q^v_{(p,S,p')}, v', p') \mid v, v' \in S,\ v' \text{ is the } \preceq\text{-successor of } v \text{ in } S, \text{ and } v' \text{ is the } \preceq\text{-maximum of } S\}$$
$$\delta_{\text{one}} = \{(p, v, p') \mid (p, \{v'\}, p') \in \delta'\}$$

The previous construction maintains the shape of $\mathcal{A}'$ but adds the needed intermediate states to form a whole extended marker transition. For every extended transition (p, S, p'), $|S| - 1$ states are added, labeled with the incoming marker that will arrive to that state. δ_{first} defines how to get to the first state of the path, using the first marker of S, δ_{succ} defines how to get to the next marker in S and δ_{last} how to get back to the $\mathcal{A}'$ state p', having finished the extended transition. δ_{one} defines the case when $|S| = 1$ and no intermediate states are needed and just use the only marker to do the transition. Note that a different set of intermediate states are added for each extended transition (p, S, p'), so states do not get reused or transitions do not get mixed. As each transition (p, S, p') of $\mathcal{A}'$ has a corresponding variable-path in $\mathcal{A}$, it is obvious that a run in either $\mathcal{A}$ or $\mathcal{A}'$ has a corresponding run in the opposite automaton with the same properties, thanks to the order preservation established in the created variable-paths. Finally, it is straightforward to show that $[\![\mathcal{A}]\!]_d = [\![\mathcal{A}']\!]_d$ for every document d.

Let us show that for both constructions, if the input automaton is sequential or functional, then the output automaton preserves this property. In the first case, if $\mathcal{A}$ is sequential, it is easy to see that all accepting runs of $\mathcal{A}'$ must be valid, since all extended marker transitions are performed in the same order as in the original automaton $\mathcal{A}$, and therefore, are also valid. If $\mathcal{A}$ uses all the variables for all accepting runs, this must also hold for $\mathcal{A}'$, preserving functionality.

Proof of Proposition 3.2

This result follows from the classical NFA determinization construction. In this case, let $\mathcal{A} = (Q, q_0, F, \delta)$ be an eVA, then the following is an equivalent deterministic eVA for $\mathcal{A}$: $\mathcal{A}' = (2^Q, \{q_0\}, F', \delta')$, where $F' = \{B \in 2^Q \mid B \cap F \neq \varnothing\}$ and $\delta'(B, o) = \{q \in Q \mid \exists p \in B.\ (p, o, q) \in \delta\}$. One can easily check that δ' is a function and therefore $\mathcal{A}'$ is deterministic. The fact that $[\![\mathcal{A}]\!]_d \equiv [\![\mathcal{A}']\!]_d$ for every document d follows, as well, from NFA determinization: namely, a valid and accepting run in $\mathcal{A}$ can be translated using the same transitions onto a valid and accepting run in $\mathcal{A}'$

where the set-states hold the states from the original run. On the other hand, a valid and accepting run in $\mathcal{A}'$ can only exists if there exists a sequence of states using the same transitions in the original automaton $\mathcal{A}$.

Finally, since the construction works with sets of n state, then in the worst case it may use 2^n states. As for transitions, if each state has all m transitions defined, then the determinization, at most, has $2^n \cdot m$ transitions.

Proof of Lemma 3.3

The proof is done by induction on i. For the sake of simplification, we denote every object in the i-th iteration, that is, while running $\textsc{Reading}(i)$, or $\textsc{Capturing}(i)$, with a superscript i. For example, the value of the $list_q$ in the i-th iteration is denoted by $list_q^i$.

For the base case assume that $i = 0$. At the beginning we have that $list_{q_0} = \bot$. If it holds that $\delta(q_0, S) = p_0$ for some $S \neq \emptyset$, then while running $\textsc{Capturing}(1)$, the algorithm will create a new node n with $n.\texttt{content} = (S, 1)$ and a $n.\texttt{list} = list^1_{q_0} = \bot$, and add it at the beginning of the list $list^1_{p_0}$. Note that $list^1_{p_0}$ can also contain other elements coming from other transitions of the form $\delta(q_0, S') = p_0$ with $S' \neq S$. Running then $\textsc{EnumAll}(list^1_{p_0}, \varepsilon)$, will eventually reach the node n in $list^1_{p_0}$, resulting in the output $M = (S, 1)$. Since $n.\texttt{list} = \bot$, then the output will be $(S, 1)$ which is the output of the corresponding run $\rho = q_0 \xrightarrow{S} p_0$ and clearly $\textsc{Out}(\rho) = M$. The other direction is analogous.

For the inductive step, assume that the claim holds for some $0 \leqslant i < n$. To show that the claim holds for $i + 1$ assume first that there is a run:

$$\rho_{i+1} \;=\; q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} \;\ldots\; \xrightarrow{a_i} q_i \xrightarrow{S_{i+1}} p_i \xrightarrow{a_{i+1}} q_{i+1} \xrightarrow{S_{i+2}} p_{i+1} \tag{3}$$

that defines an output $\textsc{Out}(\rho_{i+1})$. By the induction hypothesis, we know that after running $\textsc{Capturing}(i + 1)$ we have that $list^{i+1}_{p_i} \neq \epsilon$, and that running $\textsc{EnumAll}(list^{i+1}_{p_i}, \epsilon)$ results in an output M_i with $M_i = \textsc{Out}(\rho_i)$, where $\rho_i = q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} q_1 \xrightarrow{S_2} p_1 \xrightarrow{a_2} \ldots \xrightarrow{a_i} q_i \xrightarrow{S_{i+1}} p_i$. The algorithm now proceeds by executing the procedures $\textsc{Reading}(i + 1)$ and $\textsc{Capturing}(i + 2)$ one after the other.

Consider what happens when the procedure $\textsc{Reading}(i + 1)$ is executed. First, we know that the list $list^{i+1}_{p_i}$ gets copied to $list^{old}_{p_i}$ and $list^{i+1}_{p_i}$ is reset to the empty list ϵ. Then, since $list^{old}_{p_i} \neq \epsilon$, and since $q_{i+1} = \delta(p_i, a_{i+1})$, the procedure $\textsc{Reading}(i + 1)$ will append the entire list $list^{i+1}_{p_i}$ somewhere in the list $list^{i+1}_{q_{i+1}}$. Therefore, we know that after executing $\textsc{Reading}(i + 1)$, the entire list $list^{i+1}_{p_i}$ will appear in the list $list^{i+1}_{q_{i+1}}$ before the procedure $\textsc{Capturing}(i + 2)$ is executed.

In $\textsc{Capturing}(i + 2)$ we will first guard a copy of $list^{i+1}_{q_{i+1}}$ in $list^{old}_{q_{i+1}}$. What follows depends on whether $S_{i+2} = \emptyset$ or not. In the case that $S_{i+2} = \emptyset$, we know that $p_{i+1} = q_{i+1}$ and that the nodes in $list^{old}_{q_{i+1}}$ remain on the list $list^{i+2}_{p_{i+1}} = list^{i+1}_{q_{i+1}}$. The latter follows since any other transition such that $\delta(q, S) = p_{i+1}$ will simply add a new node at the beginning of $list^{i+2}_{p_{i+1}}$. Because of this we also have $list^{i+2}_{p_{i+1}} \neq \epsilon$. Therefore, after $\textsc{Capturing}(i + 2)$ has executed, running $\textsc{EnumAll}(list^{i+2}_{p_{i+1}}, \varepsilon)$ will have M_i with $M_i = \textsc{Out}(\rho_i)$ as one of its outputs, since it will traverse the part of the list $list^{i+1}_{q_{i+1}}$ which was already present after $\textsc{Capturing}(i + 1)$. Since $\textsc{Out}(\rho_{i+1}) = \textsc{Out}(\rho_i)$, the result follows.

On the other hand, if $S_{i+2} \neq \emptyset$, since $\delta(q_{i+1}, S_{i+2}) = p_{i+1}$, and $list^{old}_{q_{i+1}} \neq \epsilon$, the procedure $\textsc{Capturing}(i + 2)$ will create a new node n to be added to the list $list^{i+2}_{p_{i+1}}$. The node n will have the values $n.\texttt{content} = (S_{i+2}, i + 2)$ and $n.\texttt{list} = list^{old}_{q_{i+1}} = list^{i+1}_{q_{i+1}}$. In particular, after $\textsc{Capturing}(i + 2)$, we have that $list^{i+2}_{p_{i+1}} \neq \epsilon$ and that running $\textsc{EnumAll}(list^{i+2}_{p_{i+1}}, \varepsilon)$ will eventually do a call to the procedure $\textsc{EnumAll}(n.\texttt{list}, (S_{i+2}, i + 2) \cdot \varepsilon)$. Therefore one of the outputs of the original call $\textsc{EnumAll}(list^{i+2}_{p_{i+1}}, \varepsilon)$ will simply append the pair $(S_{i+2}, i + 2)$ to M_i resulting in $M_{i+1} = M_i \cdot (S_{i+2}, i + 2)$ as output. From (3) it is clear that $M_{i+1} = \textsc{Out}(\rho_{i+1})$.

For the other direction, assume now that we have executed the procedure $\textsc{Capturing}(i + 2)$ in Algorithm 1 and that $list^{i+2}_{p_{i+1}} \neq \epsilon$. Furthermore, assume that $n \neq \bot$ is a node in $list^{i+2}_{p_{i+1}}$ and $(S, j) = n.\texttt{content}$. The first observation we make is that for any node $n' \neq \bot$ that is inside the list $n.\texttt{list}$ with $(S', j') = n'.\texttt{content}$, it holds that $j' < j$. This is evident from the algorithm since the only way that the node n' can enter the list $n.\texttt{list}$ is when the node n is being created in $\textsc{Capturing}(j)$. However, in this case, the node n' must have already been defined in some previous iteration of the algorithm (as $n.\texttt{list}$ guards the "old" pointers $list^{old}_p$ for some p), and since new nodes are being created only in the procedure $\textsc{Capturing}$, this means that n' was created in $\textsc{Capturing}(j')$. Because of this we have that $j' < j$. Moreover, given that each iterative call of $\textsc{EnumAll}$ uses elements from the list $n.\texttt{list}$ we have that for any output $M = (S_1, i_1) \ldots (S_k, i_k)$ of $\textsc{EnumAll}(list^{i+1}_{p_{i+1}}, \epsilon)$, it holds that $i_k > i_{k-1} > \cdots > i_1$.

Let $M_{i+1} = (S_1, i_1) \ldots (S_{k-1}, i_{k-1})(S_k, i_k)$, where $k \geqslant 0$, be one output of $\textsc{EnumAll}(list^{i+2}_{p_{i+1}}, \varepsilon)$. There are two possible cases: either $i_k = i + 2$, or $i_k \neq i + 2$. Consider first the case when $i_k \neq i + 2$. In this case, the procedure $\textsc{EnumAll}(list^{i+2}_{p_{i+1}}, \varepsilon)$ will not access a node created in $\textsc{Capturing}(i+2)$ when generating the output M_{i+1}. Therefore, it will have to start with some node n that got in the list $list^{i+2}_{p_{i+1}}$ during $\textsc{Reading}(i+1)$. This can only happen if $\delta(p_i, a_{i+1}) = q_{i+1} = p_{i+1}$, for some state p_i such that $list^{old}_{p_i} \neq \epsilon$ at the beginning of $\textsc{Reading}(i+1)$, and n belongs to $list^{i+1}_{p_i}$, since the only thing $\textsc{Reading}(i+1)$ does is to copy and merge the lists $list^{i+1}_{p_i}$, for different states p. However, this means that $list^{i+1}_{p_i} \neq \epsilon$ after executing $\textsc{Capturing}(i+1)$. This means that M_{i+1} is one of the outputs of $\textsc{EnumAll}(list^{i+1}_{p_i}, \varepsilon)$ after executing $\textsc{Capturing}(i+1)$. Using the induction hypothesis, there is a run $\rho_i = q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} q_1 \dots \xrightarrow{a_i} q_i \xrightarrow{S_{i+1}} p_i$ such that $\textsc{Out}(\rho_i) = M_{i+1}$. Consider now the run $\rho_{i+1} = q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} q_1 \dots \xrightarrow{a_i} q_i \xrightarrow{S_{i+1}} p_i \xrightarrow{a_{i+1}} q_{i+1} \xrightarrow{\varnothing} p_{i+1}$. Since clearly $\textsc{Out}(\rho_{i+1}) = \textsc{Out}(\rho_i)$, we get that the claim holds true for $i+1$ when $i_k \neq i+2$.

Consider now the case when $i_k = i+2$. To produce M_{i+1} as output, the procedure $\textsc{EnumAll}(list^{i+2}_{p_{i+1}}, \varepsilon)$ had to do a recursive call to $\textsc{EnumAll}(n.\texttt{list}, (S_k, i+2)\cdot\varepsilon)$, for some node n on $list^{i+2}_{p_{i+1}}$. Since $i_k = i+2$ we know that node n was created in $\textsc{Capturing}(i+2)$. Therefore, there must exist a state q_{i+1} such that $\delta(q_{i+1}, S_k) = p_{i+1}$ and $list^{old}_{q_{i+1}} \neq \epsilon$ at the beginning of $\textsc{Capturing}(i+2)$. As $n.\texttt{list} = list^{old}_{q_{i+1}}$, we know that running $\textsc{EnumAll}(list^{old}_{q_{i+1}}, \varepsilon)$ must have $M_i = (S_1, i_1)\dots(S_{k-1}, i_{k-1})$ as one of its outputs. However, since all the nodes in $list^{old}_{q_{i+1}}$ must already be in $list^{i+1}_{q_{i+1}}$ after running $\textsc{Reading}(i+1)$, they must enter this list in $\textsc{Reading}(i+1)$ because there is some transition $\delta(p_i, a_{i+1}) = q_{i+1}$, for some state $p_i \in Q$. In particular, the recursive call of $\textsc{EnumAll}(n.\texttt{list}, \varepsilon)$ that produced M_i as its output used a node on $list^{i+1}_{q_{i+1}}$ that was already present in $list^{i+1}_{p_i}$, for state p_i such that $\delta(p_i, a_{i+1}) = q_{i+1}$. By the induction hypothesis, there is a run $\rho_i = q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} q_1 \dots \xrightarrow{a_i} q_i \xrightarrow{S_{i+1}} p_i$ such that $\textsc{Out}(\rho_i) = M_i$. Because of this, the run $\rho_{i+1} = q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} q_1 \dots \xrightarrow{a_i} q_i \xrightarrow{S_{i+1}} p_i \xrightarrow{a_{i+1}} q_{i+1} \xrightarrow{S_k} p_{i+1}$ clearly has $\textsc{Out}(\rho_{i+1}) = M_{i+1}$. This concludes the proof.

B. PROOFS FROM SECTION 4

Proof of Proposition 4.1

Let $\mathcal{A} = (Q, q_0, F, \delta)$ be a VA with $|Q| = n$, $|\delta| = m$ and ℓ variables. We show how to construct a deterministic seVA $\mathcal{A}' = (Q', q_0', F', \delta')$ that is equivalent to $\mathcal{A}$ and has $2^n \times 3^\ell$ states, $2^n 3^\ell (2^\ell + |\Sigma|)$ transitions and ℓ variables. Let us first describe the set Q' of states of $\mathcal{A}$. Intuitively, every state will correspond to a tuple $(\{q_1, \dots, q_k\}, S)$, where $q_1, \dots, q_k \in Q$ are the states reached by $\mathcal{A}$ by reading the set of variable markers S. Since there are n states, the first component (the set of reached states) can be chosen of 2^n different sets. Now for each state in the chosen set, we have a set of variable markers. Note that we need to exclude the sets of variable markers that contain a variable that is closed but not opened. Therefore, if we have ℓ variables the number of such sets of variable markers is $\sum_{i=1}^{\ell} \binom{n}{i} 2^i$, where i represents the number of opened variables, $\binom{n}{i}$ the different ways of choosing those i variables, and 2^i possible ways of closing those variables. From this we obtain

$$\sum_{i=0}^{\ell} \binom{n}{i} 2^i = \sum_{i=0}^{\ell} \binom{n}{i} 2^i 1^{n-i} = (1+2)^\ell = 3^\ell$$

Therefore, it is clear that we have $2^n 3^\ell$ states. The only initial state is $q_0' = (\{q_0\}, \varnothing)$.

Let us now define the set of transitions δ'. Given a character $c \in \Sigma$, the transition $\delta((P, S), c)$ is simply defined as $(\delta(P, c), S)$, where $\delta(P, c) = \{q \in Q \mid \exists q' \in P \text{ s.t. } (q', c, q) \in \delta\}$. Let us now describe the variable transitions. Intuitively, $\delta'((P, S), S')$ will contain the set of states that can be reached from a state of P by following a path in $\mathcal{A}$ of variable transitions in which each variable marker in S' is mentioned exactly once. Formally, we define a variable path in $\mathcal{A}$ as a sequence of transitions $p = (q_{i_1}, m_1, q_{i_2})(q_{i_2}, m_2, q_{i_3})\dots(q_{i_{h-1}}, m_{h-1}, q_{i_h})$ in δ^*, where each m_j is a variable marker and for all $j \neq k \in \{0, \dots, m\}$ we have $m_j \neq m_k$. If $S = \{m_1, \dots, m_h\}$ we say that p is an S-path from q_{i_1} to q_{i_h}. Then, for every $P \subset Q$ and every pair (S, S') of variable markers such that S and S' are compatible (in the sense that every closed variable in $S' \cup S$ is also opened), $\delta'((P, S), S')$ is defined as (P', S'') where:

1. $S'' = S \cup S'$ and
2. for every $q' \in P'$ there exists $q \in P$ such that there is an S'-path between q and q'.

If S and S' are not compatible, $\delta'((\{q_0, \dots, q_k\}, S), S')$ is undefined (note that this makes the automaton sequential).

Let us analyze the number of transitions in δ'. To do a fine-grained analysis of the variable transitions, for each $i \in \{0, \dots, \ell\}$ we consider the number of states (P, S) where S has exactly i open variables (i.e. $2^n \binom{\ell}{i} 2^i$), multiplied by the number of variable transitions that can originate in such a state. This number is again analyzed for the $\binom{\ell-i}{j}$ sets of size j of opened variables (out of the $\ell - i$ remaining variables), and for each of these sets which variables are closed (2^j possibilities). The variable transitions are

$$\sum_{i=0}^{\ell}\left[2^n\binom{\ell}{i}2^i\sum_{j=0}^{\ell-i}\binom{\ell-i}{j}2^j\right] \;=\; 2^n\sum_{i=0}^{\ell}\binom{\ell}{i}2^i3^{\ell-i} \;=\; 2^n(2+3)^\ell \;=\; 2^n5^\ell$$

This is the number of variable transitions in $\mathcal{A}'$. To this number, we must add the number of character transitions, which is at most one transition per state per character, i.e. $2^n3^\ell|\Sigma|$. Then, the total number of transitions is $2^n5^\ell + 2^n3^\ell|\Sigma|$ as expected. Finally, we define the set F' of final states as those states (P, S) in which $P \cap F \neq \varnothing$ and all variables opened in S are also closed.

It is trivial to see that $\mathcal{A}'$ is sequential. Since the only way to reach a state (P, S) using a variable transition is from a previous state (P', S') and a set of markers S'' such that $S' \cup S'' = S$, it is clear that if a run ρ ends in state (P, S) then S is the union of all variable markers seen in ρ. Sequentiality then follows since we require at all times that every variable is opened and closed at most once, variables are opened before they are closed, and in final states all opened variables are closed. The fact that $\mathcal{A}'$ is deterministic can be immediately seen from the construction; for every state there is at most one transition for each character, and at most one transition for each set of variable markers. Since $\mathcal{A}'$ is an extended VA and must alternate between variable and character transitions, this implies that two different runs cannot generate the same mapping.

We now show that $\mathcal{A}$ is equivalent to $\mathcal{A}'$. Let d be a document and assume the mapping μ is produced by a valid accepting run $\rho = (q_0, i_0) \xrightarrow{o_1} (q_1, i_1) \xrightarrow{o_2} \cdots \xrightarrow{o_m} (q_m, i_m)$ of $\mathcal{A}$ over d. Define a function f with domain $i \in \{1, \ldots, m\}$ as follows:

$$f(i) = \begin{cases} k & \text{if } \forall j \in \{1,\ldots,k\}\ o_i \text{ is a variable marker, and either } k = m \text{ or } o_{k+1} \text{ is not a variable marker.} \\ (o_i, \varnothing) & \text{if } o_i \in \Sigma \text{ and } o_{i+1} \in \Sigma \\ o_i & \text{otherwise.} \end{cases}$$

With this definition, we construct a run for $\mathcal{A}'$ generating μ starting with ρ' as the run that only contains q_0' and $i = 1$ as follows:

1. If $f(i) = k$, define the set of variable markers S as $\bigcup_{j=i}^{k} o_j$, update ρ' to $\rho' \xrightarrow{S} \delta'((P', S'), S)$, where (P', S') was the last state of ρ' before this update. Finally, update i to $k + 1$.

2. If $f(i) = (o_i, \varnothing)$, update ρ' to $\rho' \xrightarrow{o_i} \delta'(P', S') \xrightarrow{\varnothing} \delta'((P', S'), \varnothing)$, where (P', S') was the previous final state of ρ'. Finally update i to $i + 1$.

3. If $f(i) = o_i$, update ρ' to $\rho' \xrightarrow{o_i} \delta'((P', S'), o_i)$, where (P', S') was the previous final state of ρ'. Finally update i to $i + 1$.

We need to show that this is actually a valid and accepting run of $\mathcal{A}'$ over d. To show that it is a run over $\mathcal{A}'$ is simple: since ρ is a run over $\mathcal{A}$, the construction of f implies that for every step of the form $(P', S') \xrightarrow{S} (P, S \cup S')$ in ρ' there is an S-path from a state in P to a state in P' (assuming $S \neq \varnothing$). The $\varnothing$ and character transitions immediately yield valid transitions for ρ'. The fact that ρ' follows from the construction, as we can see that it will open and close variables in the same order and in the same positions as ρ, which was already valid. This also shows that ρ' generates μ. The fact that ρ' is valid follows because $q_m \in F$ is final and belongs to the last state of ρ'.

The opposite direction is similar: considering a mapping μ generated by a valid accepting run ρ' of $\mathcal{A}'$ over d, we need to show a valid accepting run ρ of $\mathcal{A}$ over d generating μ. We omit this direction as ρ can be generated by doing essentially the same process as before but in reverse: We know that ρ' ends in a state that mentions a final state $q_f \in F$. Then, for each step $(P, S) \xrightarrow{o} (P', S')$ of ρ and the selected state in P' (at the beginning, q_f), there is a transition or an $(S' \backslash S)$-path going from a state $q \in P$ to q'. This way we can construct ρ' backwards; proving it is valid, accepting and it generates μ follows again by the construction.

Proof of Proposition 4.2

Unfortunately, a sequential VA has an exponential blow-up in terms of the number of transitions the resulting eVA may have. For every ℓ consider the VA $\mathcal{A}$, with $3\ell + 2$ states and $4\ell + 1$ transitions depicted in Figure 8 with 2ℓ variables: $x_1, \ldots, x_\ell, y_1, \ldots, y_\ell$. $\mathcal{A}$ only produces valid runs for the document $d = a$, the resulting mapping is always valid but never total, as it properly opens and closes variables, but never all of them. At each intermediate state, the run has the option to choose opening and closing either x_i or y_i, for every $1 \leqslant i \leqslant \ell$, generating 2^ℓ different runs. Therefore, if we only consider the equivalent eVA that extends transitions from q_0 to q and no other pair in between, we obtain the extended VA $\mathcal{A}'$ in Figure 9. This is the smallest eVA equivalent to $\mathcal{A}$, since each of the mentioned transitions group the greatest amount of variables in a different run. Specifically, each of this transitions has a corresponding and different ϵ mapping, the one where the contained variables is defined. Therefore, it has 2^ℓ transitions, as well as any other equivalent eVA.

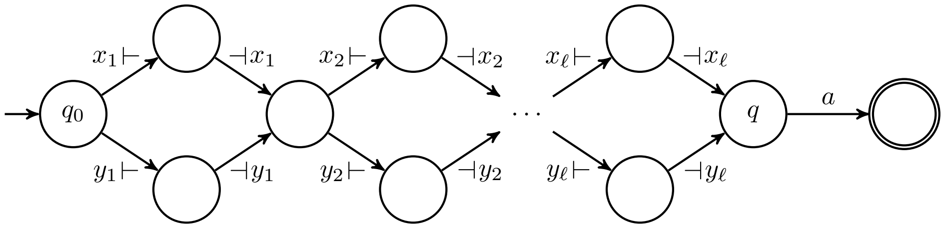


Figure 8: A sequential VA **with 2ℓ variables such that every equivalent** eVA **has $O(2^\ell)$ transitions.**

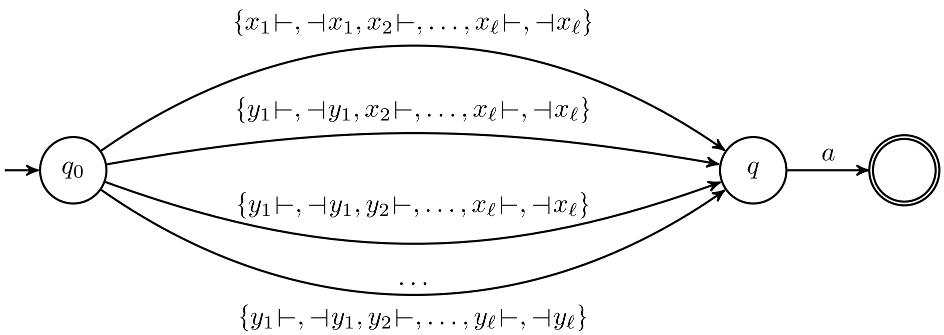


Figure 9: The smallest eVA **$\mathcal{A}'$ equivalent to $\mathcal{A}$ with 2^ℓ transitions.**

Proof of Proposition 4.3

We showed in the proof of Theorem 3.1 that given a VA $\mathcal{A}$ we can construct an equivalent eVA $\mathcal{A}_{\text{ext}}$, and the functional property also holds for $\mathcal{A}_{\text{ext}}$. We show here that if $\mathcal{A}$ has n states and m transitions, then $\mathcal{A}_{\text{ext}}$ has at most n states and $m + n^2$ transitions.

The bound n over the number of states in $\mathcal{A}_{\text{ext}}$ directly follows from the construction in Theorem 3.1. The bound $m+n^2$ over the number of transitions in $\mathcal{A}_{\text{ext}}$ follows from the fact that $\mathcal{A}$ is functional, given that in a functional VA the number of extended marker transitions that can be established between two states is at most one. Specifically, we prove the following lemma[3] (for a formal definition of variable path see the Proof of Theorem 3.1).

Lemma B.1. *If $\mathcal{A}$ is functional, then for every two states q and q' in $\mathcal{A}$ that can produce valid runs, it holds that* $\text{Markers}(\pi) = \text{Markers}(\pi')$ *for every pair of variable paths π and π' between q and q'.*

Proof. If not, then there are two states q and q' in $\mathcal{A}$ such that the are at least two variable paths π and π' between q and q', with different sets of markers appearing in them. Since q and q' can produce a valid run, then they are both reachable from q_0 and can reach a final state. Specifically, let π_i be the path from q_0 to q, and π_f be the path from q' to a final state. Then the concatenated paths $\pi_i\pi\pi_f$ and $\pi_i\pi'\pi_f$ are both accepting. Both also must be valid, because $\mathcal{A}$ is functional. But, the set of markers in π and π' are different, yet, the rest of the paths are the same and they open and close all variables in $\mathcal{A}$. This is a contradiction: either $\pi_i\pi\pi_f$ or $\pi_i\pi'\pi_f$ cannot open and close all variables. Therefore, all paths between q and q' must contain the same set of markers appearing in them. $\square$

Thanks to the previous lemma, we can bound the number of possible extended marker transitions between every pair of states to just one: the set of markers appearing in paths connecting these two states. Therefore, using our construction for $\mathcal{A}_{\text{ext}}$, at most one extended marker transition may be added between two states. Then, additionally to the m transitions in $\mathcal{A}$, at most n^2 extended marker transitions can be added (for every pair of states in $\mathcal{A}$). We conclude that $\mathcal{A}_{\text{ext}}$ has at most $m + n^2$ transitions.

Finally, as showed in Proposition 3.2, deterministic seVA $\mathcal{A}'$ can be constructed such that $\mathcal{A}_{\text{ext}} \equiv \mathcal{A}'$, where, $\mathcal{A}'$ has at most 2^n states. Since $\mathcal{A}'$ is deterministic, every state can have, at most, the number of extended transitions added or all the possible symbols in Σ. Therefore, the number of transitions for $\mathcal{A}'$ is at most $2^n(n^2 + |\Sigma|)$.

Proof of Proposition 4.4

Join of functional extended VA

Let $\mathcal{A}_1 = (Q_1, q_0^1, F_1, \delta_1)$ and $\mathcal{A}_2 = (Q_2, q_0^2, F_2, \delta_2)$ be two feVA. Let $\mathcal{V}_1 = \text{var}(\mathcal{A}_1)$, $\mathcal{V}_2 = \text{var}(\mathcal{A}_2)$ and $\mathcal{V}_\bowtie = \mathcal{V}_1 \cap \mathcal{V}_2$. The intuition behind the following construction is similar to the standard construction for intersection of NFAs:

[3] A similar lemma appears in [13].

we run both automaton in parallel, limiting the possibility to use simultaneously markers on both automata only on shared variables, and let free use of markers that are exclusive to $\mathcal{V}_1$ or $\mathcal{V}_2$. Formally, we define $\mathcal{A}_{\bowtie} = (Q_1 \times Q_2, (q_0^1, q_0^2), F_1 \times F_2, \delta)$ where δ is defined as follows:

- $\big((p_1, p_2), a, (q_1, q_2)\big) \in \delta$ if $a \in \Sigma$, $(p_1, a, q_1) \in \delta_1$ and $(p_2, a, q_2) \in \delta_2$.
- $\big((p_1, p_2), S_1, (q_1, p_2)\big) \in \delta$ if $p_2 \in Q_2$, $(p_1, S_1, q_1) \in \delta_1$, and $S_1 \cap \text{Markers}_{\mathcal{V}_{\bowtie}} = \emptyset$.
- $\big((p_1, p_2), S_2, (p_1, q_2)\big) \in \delta$ if $p_1 \in Q_1$, $(p_2, S_2, q_2) \in \delta_2$ and $S_2 \cap \text{Markers}_{\mathcal{V}_{\bowtie}} = \emptyset$.
- $\big((p_1, p_2), S_1 \cup S_2, (q_1, q_2)\big) \in \delta$ if $(p_1, S_1, q_1) \in \delta_1$, $(p_2, S_2, q_2) \in \delta_2$, and $S_1 \cap \text{Markers}_{\mathcal{V}_{\bowtie}} = S_2 \cap \text{Markers}_{\mathcal{V}_{\bowtie}}$.

To show that $[\![\mathcal{A}_{\bowtie}]\!]_d \subseteq [\![\mathcal{A}_1]\!]_d \bowtie [\![\mathcal{A}_2]\!]_d$, let μ be a mapping in $[\![\mathcal{A}_{\bowtie}]\!]_d$ for the document d, and ρ_μ the corresponding valid and accepting run of $\mathcal{A}_{\bowtie}$ over d. By construction, from ρ_μ we can get a sequence of states in $\mathcal{A}_1$ and $\mathcal{A}_2$ that define runs ρ_1 and ρ_2 in their respective automaton. This preserves both order and positions of markers. Since $\mathcal{A}_1$ and $\mathcal{A}_2$ are functional and ρ_μ is accepting, then ρ_1 and ρ_2 are accepting and valid runs of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. This implies that $\mu^{\rho_1} \in [\![\mathcal{A}_1]\!]_d$ and $\mu^{\rho_2} \in [\![\mathcal{A}_2]\!]_d$. Finally, since all common marker transitions are performed by both automata at the same union transitions, then $\mu^{\rho_1} \sim \mu^{\rho_2}$ and therefore $\mu = \mu^{\rho_1} \cup \mu^{\rho_2} \in [\![\mathcal{A}_1]\!]_d \bowtie [\![\mathcal{A}_2]\!]_d$.

To show that $[\![\mathcal{A}_1]\!]_d \bowtie [\![\mathcal{A}_2]\!]_d \subseteq [\![\mathcal{A}_{\bowtie}]\!]_d$, let $\mu_1 \in [\![\mathcal{A}_1]\!]_d$, $\mu_2 \in [\![\mathcal{A}_2]\!]_d$ such that $\mu_1 \sim \mu_2$ and ρ^{μ_1} and ρ^{μ_2} be their corresponding runs. Since they are compatible mappings, then both runs use each marker in Markers$(\mathcal{V}_{\bowtie})$ in the same positions of d. Therefore, by merging the marker transitions made in each run, the corresponding union transitions must exists in $\mathcal{A}_{\bowtie}$ and used to construct a run ρ in $\mathcal{A}_{\bowtie}$. Finally, since ρ_1 and ρ_2 are accepting, valid, and total, then ρ is also accepting, valid and total for $\text{var}(\mathcal{A}_1) \cup \text{var}(\mathcal{A}_2)$, that is, $\mu^\rho \in [\![\mathcal{A}_{\bowtie}]\!]_d$. It is easy to see that $\mu^\rho = \mu_1 \cup \mu_2$, and therefore $\mu_1 \cup \mu_2 \in [\![\mathcal{A}_{\bowtie}]\!]_d$.

To show that $\mathcal{A}_{\bowtie}$ is also functional, let ρ be an accepting run in $\mathcal{A}_{\bowtie}$ for d. Thanks to the construction, and as shown before, corresponding runs in $\mathcal{A}_1$ and $\mathcal{A}_2$ can be produced from ρ that are also accepting, and therefore valid and total since they are functional. Since all common markers are used in the same positions and precisely once in the corresponding runs, this is also true for ρ. Also, all variables are used in runs of $\mathcal{A}_1$ and $\mathcal{A}_2$, therefore ρ is valid and total for $\text{var}(\mathcal{A}_1) \cup \text{var}(\mathcal{A}_2)$. Regarding the size of $\mathcal{A}_{\bowtie}$, one can verify that $\mathcal{A}_{\bowtie}$ has $|Q_1| \times |Q_2|$ states and at most $O(|\delta_1| \times |\delta_2|)$ transitions. Therefore, $\mathcal{A}_{\bowtie}$ is quadratic in size.

Projection of functional extended VA

To prove this, for the sake of simplification we use the notion of ϵ-transitions in eVA, as the usual notion for regular NFA, namely, transition of the form (q, ϵ, p). As it is standard in automata theory, if a run uses an ϵ-transition, this produces no effect on the document read or variables that are opened or closed, and only the current state of the automaton changes from q to p. Furthermore, in the semantics of ϵ-transitions we assume that no two consecutive ϵ-transitions can be used. Clearly, ϵ-transitions do not add expressivity to the model and only help to simplify the construction of the projection.

Let $\mathcal{A} = (Q, q_0, F, \delta)$ be a feVA and $Y \subset \mathcal{V}$. Let $U = \text{Markers}_{\mathcal{V}} \setminus \text{Markers}_Y$ be markers for unprojected variables, then $\mathcal{A}_\pi = (Q, q_0, F, \delta')$ where $(q, a, p) \in \delta'$ whenever $(q, a, p) \in \delta$ for every $a \in \Sigma$, $(q, S \setminus U, p) \in \delta'$ whenever $(q, S, p) \in \delta$ and $S \setminus U \neq \emptyset$, and $(q, \epsilon, p) \in \delta'$ whenever $(q, S, p) \in \delta$ and $S \setminus U = \emptyset$.

The equivalence between $\mathcal{A}$ and $\mathcal{A}_\pi$ is straightforward. For every $\mu \in [\![\mathcal{A}]\!]_d$, there exists an accepting and valid run ρ in $\mathcal{A}$ over d. For ρ there exists a run ρ' in $\mathcal{A}_\pi$ formed by the same sequence of states, but extended marker or ϵ-transitions are used that only contain markers from Y. Moreover, ρ' must also be valid since it maintains the order of Y-variables used in ρ. This shows that $\pi_Y [\![\mathcal{A}]\!]_d \subseteq [\![\mathcal{A}_\pi]\!]_d$. The other direction, $[\![\mathcal{A}_\pi]\!]_d \subseteq \pi_Y [\![\mathcal{A}]\!]_d$, follows from the fact that $\mathcal{A}'$ has no additional accepting paths in comparison to $\mathcal{A}$. It is also easy to see that $\mathcal{A}_\pi$ must be functional.

Finally, it is important to note that, as for classical NFAs, from $\mathcal{A}'$ an equivalent ϵ-transition free eVA can constructed using ϵ-closure over states.

Union of functional extended VA

This construction is the standard disjoint union of automaton, with ϵ-transitions to each corresponding initial state. Let $\mathcal{A}_1 = (Q_1, q_0^1, F_1, \delta_1)$ and $\mathcal{A}_2 = (Q_2, q_0^2, F_2, \delta_2)$ be two feVA such $Q_1 \cap Q_2 = \emptyset$. Then, $\mathcal{A}_\cup = (Q_1 \cup Q_2, q_0, F_1 \cup F_2, \delta_1 \cup \delta_2 \cup \{(q_0, \epsilon, q_0^1), (q_0, \epsilon, q_0^2)\})$ where q_0 is a fresh new state. This simply adds a new initial state connected with ϵ-transitions to the initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Therefore every run in $\mathcal{A}_\cup$ must produce a run from $\mathcal{A}_1 \cup \mathcal{A}_2$ and vice versa. An equivalent ϵ-transition free automaton can be constructed as in the projection case.

Proof of Proposition 4.5

Let γ be a regular spanner in $\text{VA}^{\{\pi, \cup, \bowtie\}}$ that uses k functional VA as input, each of them with at most n states. By Proposition 4.4, we know that we can construct the product between two automata with n and m states, and the resulting automaton will have at most nm states and nm transitions. Moreover, projections and unions remain linear in the size of the input automata. Therefore, if we apply the transformations of Proposition 4.4 in a bottom-up fashion to γ, each algebraic operation will multiply the number of states and transitions of the resulting automaton by n. It is trivial to prove then by induction that the final automaton will have at most n^k states and n^k transitions. This automaton needs to be determinized at the end. By Proposition 4.3, the result will have 2^{n^k} states and $n^{2k} + |\Sigma|$ transitions, concluding the proof.

Proof of Proposition 4.6

Contrary to the previous proposition, the idea here is to first determinize each automaton and then apply the join and union construction of functional eVA. Given that each automaton will have size $O(2^n)$ after determinization, then the product (e.g. join) of two automata of size $O(2^n)$ will have size $O(2^{2n})$. Therefore, the number of states of the whole construction will be $O(2^{kn})$ where k is the number of functional eVAs in the expression.

The only subtle point here is that each operation (i.e. join or union) must preserve the functional and deterministic property of the input automata in order to avoid a determinization after the join and union operations are computed. Indeed, one can easily check in the Proof of Proposition 4.4 that this is the case for the join of two deterministic feVA. Unfortunately, the linear construction of the union of two feVA does not preserve the deterministic property of the input automaton. For this reason, we need an alternative construction of the union that preserves determinism. This is shown in the next lemma concluding the proof of the proposition.

LEMMA B.2. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two deterministic* feVA*. Then there exists a deterministic* feVA *$\mathcal{A}_\cup$ such that $\mathcal{A}_\cup \equiv \mathcal{A}_1 \cup \mathcal{A}_2$. Moreover, $\mathcal{A}_\cup$ is of size $|\mathcal{A}_1| \times |\mathcal{A}_2|$, i.e. quadratic with respect to $\mathcal{A}_1$ and $\mathcal{A}_2$.*

PROOF. Let $\mathcal{A}_1 = (Q_1, q_0^1, F_1, \delta_1)$ and $\mathcal{A}_2 = (Q_2, q_0^2, F_2, \delta_2)$ be two feVA such $Q_1 \cap Q_2 = \varnothing$. The intuition behind the construction is to start running both automata in parallel, but add the possibility to branch off and continue the run in just one automaton, only when both cannot simultaneously execute a transition. Formally, let $\mathcal{A}_\cup = (Q, (q_0^1, q_0^2), F, \delta)$ such that $Q = Q_1 \times Q_2 \cup Q_1 \cup Q_2$, $F = F_1 \times Q_2 \cup Q_1 \times F_2 \cup F_1 \cup F_2$, and δ satisfies that:

- $\delta_1 \subseteq \delta$ and $\delta_2 \subseteq \delta$,
- $\big((p_1, p_2), o, (q_1, q_2)\big) \in \delta$ whenever $(p_1, o, q_1) \in \delta_1$, and $(p_2, o, q_2) \in \delta_2$,
- $\big((p_1, p_2), o, q_1\big) \in \delta$ whenever $(p_1, o, q_1) \in \delta_1$, and $(p_2, o, q_2) \notin \delta_2$ for every $q_2 \in Q_2$, and
- $\big((p_1, p_2), o, q_2\big) \in \delta$ whenever $(p_2, o, q_2) \in \delta_2$, and $(p_1, o, q_1) \notin \delta_1$ for every $q_1 \in Q_1$.

To show $[\![\mathcal{A}_\cup]\!]_d \subseteq [\![\mathcal{A}_1]\!]_d \cup [\![\mathcal{A}_2]\!]_d$, let $\mu \in [\![\mathcal{A}_\cup]\!]_d$ be an arbitrary mapping and ρ be the corresponding run in $\mathcal{A}_\cup$. Since ρ is accepting, the last state in the run can either be from $Q_1 \times Q_2$, Q_1 or Q_2. It is easy to see that either case, a run for μ exists in the original automaton. More specifically, if it is from $Q_1 \times Q_2$, then both automata have complete runs defined for μ, if it is from Q_1 or Q_2, then $\mathcal{A}_1$ or $\mathcal{A}_2$, respectively, has a defined run for μ. Then, we conclude that $\mu \in [\![\mathcal{A}_1]\!]_d \cup [\![\mathcal{A}_2]\!]_d$.

To show $[\![\mathcal{A}_1]\!]_d \cup [\![\mathcal{A}_2]\!]_d \subseteq [\![\mathcal{A}_\cup]\!]_d$, consider μ in either $[\![\mathcal{A}_1]\!]_d$ or $[\![\mathcal{A}_2]\!]_d$. Without loss of generality, assume that $\mu \in [\![\mathcal{A}_1]\!]_d$. Then, a run ρ_1 in $\mathcal{A}_1$ exists that produces μ. If we can also define a run ρ_2 of $\mathcal{A}_2$ over d that outputs μ, then the run ρ in $\mathcal{A}_\cup$ can be constructed by coupling up states from ρ_1 and ρ_2. Since both use the same transitions, then the last state in ρ must be in $F_1 \times Q_2$ and is accepting. Otherwise, ρ_2 cannot be defined, then some transition in ρ_1 is not defined in $\mathcal{A}_2$. This means that ρ in $\mathcal{A}_\cup$ can be constructed by following first the transitions defined in both $\mathcal{A}_1$ and $\mathcal{A}_2$, but, at the first undefined transition for $\mathcal{A}_2$, ρ can branch off and continue on states from Q_1. That transition exists since it does not exist for $\mathcal{A}_2$, but it does for $\mathcal{A}_1$. After that, ρ continues as ρ_1, making ρ also accepting. In both cases, we conclude that $\mu \in [\![\mathcal{A}_\cup]\!]_d$.

One can easily check that if $\mathcal{A}_1$ and $\mathcal{A}_2$ are functional, then $\mathcal{A}_\cup$ is also functional, since every accepting run ρ in $\mathcal{A}_\cup$ has a corresponding accepting run either in $\mathcal{A}_1$, $\mathcal{A}_2$, or both. These runs are valid and total and, thus, ρ must also be valid. From the construction, one can also check that if $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic, then $\mathcal{A}_\cup$ is also deterministic. Finally, the size of $\mathcal{A}_\cup$ is quadratic since it uses $O(|Q_1| \times |Q_2|)$ states and at most $O(|\delta_1| \times |\delta_2|)$ transitions. This was to be shown. □

C. PROOFS FROM SECTION 5

Proof of Theorem 5.1

The COUNT function in Algorithm 3 calculates $|[\![\mathcal{A}]\!]_d|$ given a deterministic seVA $\mathcal{A} = (Q, q_0, F, \delta)$ and a document $d = a_1 \dots a_n$. This algorithm is a natural extension of Algorithm 1 in Section 3. Instead of keeping the set of list $\{list_q\}_{q \in Q}$ where each list $list_q$ succinctly encodes all mappings of runs which end in state q, we keep an array N where $N[q]$ stores the number of runs that end in state q. Since $\mathcal{A}$ is sequential (i.e. every partial run encodes a valid partial mapping) and deterministic (i.e. each partial run encodes a different partial mapping), we know that the number of runs ending in state q is equal to the number of valid partial mappings in state q. Therefore, if $N[q]$ stores the number of runs at state q, then the sum of all values $N[q]$ for every state $q \in F$ is equal to the number of mappings that are output at the final states.

Algorithm 3 Count the number of mappings in $[\![\mathcal{A}]\!]_d$ over the document $d = a_1 \ldots a_n$

1: **function** COUNT($\mathcal{A}$, $a_1 \ldots a_n$)
2:     **for all** $q \in Q \backslash \{q_0\}$ **do**
3:         $N[q] \leftarrow 0$
4:     $N[q_0] \leftarrow 1$
5:     **for** $i := 1$ to n **do**
6:         CAPTURING(i)
7:         READING(i)
8:     CAPTURING($n+1$)
9:     **return** $\sum_{q \in F} N[q]$

10: **procedure** CAPTURING(i)
11:     $N' \leftarrow N$
12:     **for all** $q \in Q$ **with** $N'[q] > 0$ **do**
13:         **for all** $S \in \text{Markers}_\delta(q)$ **do**
14:             $p \leftarrow \delta(q, S)$
15:             $N[p] \leftarrow N[p] + N'[q]$

16: **procedure** READING(i)
17:     $N' \leftarrow N$
18:     $N \leftarrow 0$
19:     **for all** $q \in Q$ **with** $N'[q] > 0$ **do**
20:         $p \leftarrow \delta(q, a_i)$
21:         $N[p] \leftarrow N[p] + N'[q]$

As we said, Algorithm 3 is very similar to the constant delay algorithm. At the beginning (i.e. lines 2-4), the array N is initialize with $N[q] = 0$ for every $q \neq q_0$ and $N[q_0] = 1$, namely, the only partial run before reading or capturing any variable is the run q_0. Next, the algorithm iterates over all letters in the document, alternating between CAPTURING and READING procedures (lines 5-8). The purpose of the CAPTURING(i) procedure is to extend runs by using extended variable transitions between letters a_{i-1} and a_i. This procedure first makes a copy of N into N' (i.e. N' will store the number of runs in each state before capturing) and then adds to $N[p]$ the number of runs that reach q before capturing (i.e. $N'[q]$) whenever there exists a transition $(p, S, q) \in \delta$ for some $S \in \text{Markers}_\delta(q)$. On the other side, the procedure READING(i) is coded to extend runs by using a letter transition when reading a_i. Similar to CAPTURING, READING starts by making a copy of N into N' (line 17) and N to 0 (line 18). Intuitively, N' will store the number of valid runs before reading a_i and N will store the number of valid runs after reading a_i. Then, READING procedure iterates over all states q that are reached by at least one partial run and adds $N'[q]$ to $N[p]$ whenever there exists a letter transition $(q, a_i, p) \in \delta$. Clearly, if there exists $(q, a_i, p) \in \delta$, then all runs that reach q after reading $a_1 \ldots a_{i-1}$ can be extended to reach p after reading $a_1 \ldots a_i$. After reading the whole document and alternating between CAPTURING(i) and READING(i), we extend runs by doing the last extended variable transition after reading the whole word, by calling CAPTURING($n+1$) in line 8. Finally, the output is the sum of all values $N[q]$ for every state $q \in F$, as explained before.

The correctness of Algorithm 3 follows by a straightforward induction over i. Indeed, the inductive hypothesis states that after the i-iteration, $N[q]$ has the number of partial runs of $\mathcal{A}$ over $a_1 \ldots a_i$. Then, by following the same arguments as in Lemma 3.3, one can show that $N[q]$ store the number of partial runs of $\mathcal{A}$ after capturing and reading the $(i+1)$-th letter.

Proof of Theorem 5.2

Let us first define the class SPANL. Formally, let M be a non-deterministic Turing machine with output tape, where each accepting run of M over an input produces an output. Given an input x, we define $span_M(x)$ as the number of *different* outputs when running M on x. Then, SPANL is the counting class of all functions f for which there exists a non-deterministic logarithmic-space Turing machine with output such that $f(x) = span_M(x)$ for every input x. We say that a function f is SPANL-complete if $f \in$ SPANL and every function in SPANL can be reduced into f by log-space parsimonious reductions [2].

For the inclusion of COUNT[fVA] in SPANL, let M be a non-deterministic TM that receives $\mathcal{A}$ and d as input. The work of M is more or less straightforward: it must simulate a run of $\mathcal{A}$ over d to generate a mapping $\mu \in [\![\mathcal{A}]\!]_d$, and it does so by alternating between extended variable transitions and letter transitions reading d and writing the corresponding run on the output tape. At all times, M keeps a pointer (i.e. with log space) for the current state and a pointer to the current letter. Furthermore, it starts and ends with a variable transition as defined in Section 3.

Whenever a variable transition is up, the machine must choose non-deterministically from all its outgoing variable transitions from the current state. Recall that M can also choose to not take any variable transition, in which case it stays in the same state without writing on the output tape. Instead, if (q, S, p) is chosen then M writes the set of variables in S on the output tape and updates the current state. It does so maintaining a fixed order between variables (either lexicographic or the order presented in the input). On the other hand, when a letter transition is up, if a transition with the corresponding letter from d exists (defined by the current letter), then the current letter is printed in the output tape and, the current state and letter are updated, changing to a capturing phase. If no transition exists from the current state, then M stops and rejects. Once the last letter is read (the pointer to the current letter is equal to $|d|$), then the last variable transition is chosen. Finally, if the final state is accepting, then M accepts and outputs what is on the output tape. The correctness of M (i.e. $|[\![\mathcal{A}]\!]_d| = span_M(\mathcal{A}, d)$) follows directly from the functional properties of $\mathcal{A}$. More precisely, we know that each accepting run is valid, and will therefore produce an output. Finally, in case that $\mathcal{A}$ has two runs on x that produce the same output, by the definition of SpanL this output will be counted only once, as required to compute $|[\![\mathcal{A}]\!]_d|$ correctly.

For the lower-bound, we show that the Census problem [2], which is SpanL-hard, can be reduced into Count[fVA] via a parsimonious reduction in logarithmic-space. Formally, given a NFA $\mathcal{B}$ and length n, the Census problem asks to count the number of words of length n that are accepted by $\mathcal{B}$. We reduce an input of the Census problem $(\mathcal{B}, n)$ into Count[fVA] by computing a functional VA $\mathcal{A}_{\mathcal{B},n}$ and a document $d_{\mathcal{B},n}$ such that the number of words of length n that $\mathcal{B}$ accepts, is equivalent to count how many mappings does $\mathcal{A}_{\mathcal{B},n}$ generate over $d_{\mathcal{B},n}$. Let $\mathcal{B} = (Q, \Sigma, \Delta, q_0, F)$ be an NFA with $\Sigma = \{a, b\}$. Define $d_{\mathcal{B},n} = (\#cc)^n$ and $\mathcal{A}_{\mathcal{B},n} = (Q', q_0', F', \delta')$ over the alphabet $\{c, \#\}$ such that $Q' = Q \times \{0, \ldots, n\}$, $q_0' = (q_0, 0)$, $F' = F \times \{n\}$. Furthermore, for the sake of simplification we define δ' by using extended transitions as follows:

$$\begin{array}{lll} (q,a,p) \in \Delta & \text{then} & \Big((q,i-1),\ \# \cdot x_i\!\vdash \cdot c \cdot \dashv\!x_i \cdot c,\ (p,i)\Big) \in \delta' \ \text{ for all } i \in \{1,\ldots,n\} \\ (q,b,p) \in \Delta & \text{then} & \Big((q,i-1),\ \# \cdot c \cdot x_i\!\vdash \cdot c \cdot \dashv\!x_i,\ (p,i)\Big) \in \delta' \ \text{ for all } i \in \{1,\ldots,n\} \end{array}$$

In the previous definition, a transition of the form $((q, i-1), w, (p, i))$ means that the VA will go from state $(q, i-1)$ to the state (p, i) by following the sequence of operations in w. For example the sequence $\# \cdot x_i\!\vdash \cdot c \cdot \dashv\!x_i \cdot c$ means that an #-symbol will be read, followed by open x_i, read c, close x_i, and read c. Clearly, extended transitions like above can be encoded in any standard VA by just adding more states.

Note that to get to a state (p, i) the only option is to start from the state $(q, i-1)$. Since all runs start at $(q_0, 0)$ and final states are of the form (p, n), an accepting run of $\mathcal{A}_{\mathcal{B},n}$ over $d_{\mathcal{B},n}$ must traverse $n+1$ states of the form (q, i), one for each $i \in \{0, \ldots, n\}$, and therefore assign all n variables x_i. Also, between two consecutive states the transition always captures a span of length 1 (i.e. $x_i\!\vdash \cdot c \cdot \dashv\!x_i$) and read three characters, starting with an #-symbol which is never captured. Therefore, all accepting runs assign all n variables, and x_i is either assigned to $[3i-1, 3i\rangle$ or $[3i, 3i+1\rangle$. Since all the variables are opened and closed correctly between each $(q, i-1)$ and (p, i), we can conclude that $\mathcal{A}_{\mathcal{B},n}$ is functional.

One can easily check that the reduction of $(\mathcal{B}, n)$ to $(\mathcal{A}_{\mathcal{B},n}, d_{\mathcal{B},n})$ can be done with logarithmic space. To prove that the reduction is indeed parsimonious (i.e. $|\{w \in \Sigma^n \mid w \in \mathcal{L}(\mathcal{B})\}| = |[\![\mathcal{A}_{\mathcal{B},n}]\!]_{d_{\mathcal{B},n}}|$), we show that there exists a bijection between words of length n accepted by $\mathcal{B}$ and mappings in $[\![\mathcal{A}_{\mathcal{B},n}]\!]_{d_{\mathcal{B},n}}$. Specifically, consider the function $f : \{w \in \Sigma^n \mid w \in \mathcal{L}(\mathcal{B})\} \to [\![\mathcal{A}_{\mathcal{B},n}]\!]_{d_{\mathcal{B},n}}$ such that $f(w)$ is equivalent to the mapping $\mu_w : \{x_1, \ldots, x_n\} \to \text{span}(d_{\mathcal{A},n})$:

$$\mu_w(x_i) \quad = \quad \begin{cases} [3i-1, 3i\rangle, & \text{if } w_i = a \\ [3i, 3i+1\rangle, & \text{if } w_i = b \end{cases}$$

for every word $w = w_1 \ldots w_n \in \mathcal{L}(\mathcal{B})$. To see that f is indeed a bijection, note that for every word $w \in \mathcal{L}(\mathcal{B})$ of length n we have an accepting run of length n in $\mathcal{A}$ and we can build a mapping in $[\![\mathcal{A}_{\mathcal{B},n}]\!]_{d_{\mathcal{B},n}}$. Note that all accepting runs for w give the same mapping. Moreover, note that for two different words, different mapping are defined and then f is an injective function. In the other direction, for every mapping in $[\![\mathcal{A}_{\mathcal{B},n}]\!]_{d_{\mathcal{B},n}}$ we can build some word of length n that is accepted by $\mathcal{B}$ and, thus, f is surjective. Therefore, f is a bijection and the reduction from the Census problem into Count[fVA] is a parsimonius reduction. This completes the proof.