Efficient enumeration algorithms for regular document spanners

FERNANDO FLORENZANO, Pontificia Universidad Católica de Chile and IMFD Chile
CRISTIAN RIVEROS, Pontificia Universidad Católica de Chile and IMFD Chile
MARTÍN UGARTE, Université Libre de Bruxelles
STIJN VANSUMMEREN, Université Libre de Bruxelles
DOMAGOJ VRGOČ, Pontificia Universidad Católica de Chile and IMFD Chile

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 efficient 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 output-linear delay enumeration of a spanner's result 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 and provide a fine grained analysis of the classes of document spanners that support efficient enumeration of their results.

CCS Concepts: • **Information systems → Information extraction**; • **Theory of computation → Database query processing and optimization (theory)**; *Formal languages and automata theory*;

Additional Key Words and Phrases: Information Extraction, Spanners, Enumeration Delay, Automata, Capture Variables

ACM Reference Format:
Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoč. 2018. Efficient enumeration algorithms for regular document spanners. 1, 1 (August 2018), 42 pages. https://doi.org/10.1145/3196959.3196987

1 INTRODUCTION

Information extraction (IE for short) is the process of retrieving relevant pieces of data from large documents. For instance, in business listings, we are often interested in retrieving only the names and addresses of companies that provide a specific service, and not all of the companies. Similarly, in such a listing we might not necessarily need all the information about the companies that are of interest to us, but only wish to retrieve their name, phone number, and address.

Authors' addresses: Fernando Florenzano, Pontificia Universidad Católica de Chile and IMFD Chile, faflorenzano@uc.cl; Cristian Riveros, Pontificia Universidad Católica de Chile and IMFD 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, Pontificia Universidad Católica de Chile and IMFD Chile, dvrgoc@ing.puc.cl.

XXXX-XXXX/2018/8-ART $15.00
https://doi.org/10.1145/3196959.3196987

Historically, there have been two main approaches to information extraction: the *statistical* approach that uses machine-learning methods to find the data to be extracted, and the *rule-based* approach that utilizes traditional finite-language methods for this purpose. In this paper we focus on the rule-based approach which has been successfully adopted in industry [8] and which has recently received a great deal of attention in the database research community [12, 20]—revealing interesting connections with logic [15, 16], automata [12, 22], datalog programs [4, 28], and relational languages [7, 17, 21].

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* [12]: operators that map strings to tuples containing spans.

The most basic way of defining document spanners is to use regular expressions or automata with capture variables. The idea is that a regular language is used to locate the data to be extracted, and variables are used to store the corresponding spans. This approach to IE has been adopted by many practical information extraction systems such as Xlog [28], Instaread [18], or IBM's SystemT [21], and is prevalent in the database literature [4, 12, 13, 15, 22]. The most important classes of expressions and automata for extracting information are *regex formulas* (RGX) and *variable-set automata* (VA), respectively, which form the logical core of SystemT's extraction language [12].

A crucial problem when working with RGX and VA in practice is implementing them in such a way that all the required spans can be extracted efficiently. One issue that arises here is that commercial tools such as SystemT require the *all-match* semantics, meaning that they do want to capture all possible ways in which a regular expression or automaton can match an input string. To illustrate what this means, consider the regex formula $\gamma_1 = x\{a^*\} \cdot y\{a \vee \varepsilon\}$. Intuitively, the formula γ_1 splits an input document d into two spans, where the first span is stored in x, and matches the regular expression a^*, while the second span is stored in y and matches either the symbol a, or the empty string ε. Furthermore, the concatenation of the two spans must equal the entire document d. Notice that on a concrete input document $d = aaa$, γ_1 has two ways of splitting d into spans: (i) we can store the span corresponding to the first two occurrences of a into x, and the span corresponding to the final a into y, or (ii) we can store the span corresponding to the entire document d into x, and the empty span into y. Under an all-match semantics, both are valid answers and as such both possibilities should be output.

The issue with the all-match semantics is that it is not well supported by standard tools for regular expression matching [10, 11]. Indeed, most of standard regular expression processing tools will apply an approximate way of retrieving the matches such as only trying to find the longest match (e.g., for γ_1 and d in the example above they will only consider the case (ii)) [30]. The question hence is how to efficiently evaluate RGX formula and VA under this particular semantics.

To compare the efficiency of different evaluation algorithms under the all-match semantics we need to move from the traditional worst-case complexity analysis to a more detailed output-sensitive analysis. The reason for this is that the output of the extraction process can easily become huge, and a complexity bound based on the worst-case output size may vastly overestimate the needed running time. To understand this, first consider the regex formula $\gamma_2 = \Sigma^* \cdot x_1\{\Sigma^* \cdot x_2\{\Sigma^*\} \cdot \Sigma^*\} \cdot \Sigma^*$, where Σ denotes a finite alphabet. Intuitively, γ_2 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 this formula will produce an output of size $\Omega(\|d\|^2)$, where $\|d\|$ denotes the size of d. Clearly, any evaluation algorithm must at

least produce this large output, and therefore must run in time $\Omega(\|d\|^2)$ in the worst case.[1] Next, consider the regex formula $\gamma_3 = a \cdot \gamma_1 \vee b \cdot \gamma_2$, which extracts information using γ_1 on inputs that start with letter a, and using γ_2 on inputs that start with the letter b. An algorithm for evaluating γ_3 that runs in time $\Theta(\|d\|^2)$ is worst-case optimal because of the $\Omega(\|d\|^2)$ worst-case output size (on inputs that start with letter b), but is far from ideal on inputs that start with letter a where the output is of size $O(1)$ and a complexity of $\Omega(\|d\|^2)$ on such inputs is hence unreasonable. In conclusion, to compare the relative efficiency of possible evaluation algorithms for document spanners we need to move to an output-sensitive analysis instead of a worst-case analysis.

In output-sensitive analysis, we bound the complexity of an evaluation algorithm on input I is a function of both $\|I\|$ and $\|O\|$, where O is the output of the algorithm on I. For example, for regex formula γ_3 above, an algorithm that runs in time $O(\|d\| + \|O\|)$ is clearly asymptotically preferable over an algorithm that runs in time $O(\|d\|^2 + \|O\|)$.

In this work, we are interested in designing efficient output-sensitive evaluation algorithms for documents spanners under combined complexity, i.e., the efficiency of the evaluation algorithm is measured in terms of both the spanner and the document over which it is evaluated.

Enumeration algorithms, and their complexity. A strategy to design efficient output-sensitive evaluation algorithms that has been recently successful is the use of *enumeration algorithms* [5, 6, 9, 25]. Given an input (γ and d in our case), an enumeration algorithm first performs a *preprocessing phase* during which it does not produce output but may compute certain data structures. Then, it performs an *enumeration phase* during which it uses these data structures to produces all the elements of the output (each element being a tuple of spans in our case) without duplicates and in rapid succession. The efficiency of an enumeration algorithm is measured by determining both the complexity of the preprocessing phase and the maximum *delay* incurred between output elements during the enumeration phase. If this delay is polynomial in $\|\gamma\|$ and $\|d\|$, then the enumeration algorithm is said to have *polynomial delay*. For an enumeration algorithm with polynomial delay, the total runtime can always be expressed in the output-sensitive form

$$O\left(f(\|\gamma\|, \|d\|) + p(\|\gamma\|, \|d\|) \times |O| \right),$$

where $f\colon \mathbb{N}^2 \to \mathbb{N}$ is some function that bounds the complexity of the preprocessing phase, $p\colon \mathbb{N}^2 \to \mathbb{N}$ is a polynomial, and $|O|$ denotes the cardinality of the set of tuples O. Actually, an enumeration algorithm is more efficient when the delay does not depend on γ nor on d. In that case, the enumeration algorithm is said to have *constant delay*. For an enumeration algorithm with constant delay, the total runtime can always be expressed in the output-sensitive form $O\left(f(\|\gamma\|, \|d\|) + |O|\right)$.

Other notions of bounded delay also exist. For example, if the delay is always linear *in the size of the element that is being output* (and independent of γ and d), then the algorithm is said to have *output-linear delay*. It yields a total runtime of the form $O(f(\|\gamma\|, \|d\|) + \|O\|)$, where $\|O\|$ denotes the representation size of O (e.g. the number of bits to encode O). Output-Linear delay is actually the best we can hope for in an enumeration algorithm, since obviously one cannot expect the delay to be less than the size of the element to be output. After all, we require this many instructions to actually write the output element. Note in particular that there is a small distinction between constant delay and output-linear delay: an enumeration algorithm has constant delay if and only if it has output-linear delay and, in addition, every output element is of constant size.

Confusingly, the terms "constant" and "output-linear delay" are also sometimes used in data-complexity settings where the query is considered fixed and its size is hence constant. To avoid

[1] In general, we can keep nesting the variables (i.e., x_3 inside x_2, etc.), and the output size will be $\Omega(\|d\|^\ell)$, with ℓ the maximum nesting level of variables in γ.

confusion, we say that an enumeration algorithm has *data-constant delay* if there exists some function $g\colon \mathbb{N} \to \mathbb{N}$ such that, for every d the algorithm's delay between output elements is bounded by $g(\|\gamma\|)$ (which is constant if we consider γ constant). The total runtime in the case of data-constant delay is of the form $O(f(\|\gamma\|, \|d\|) + g(\|\gamma\|) \times |O|)$. Similarly, we say the algorithm has *data-output-linear* delay if there exists g such that, for every d and every output element x, the algorithm's delay for producing x is bounded by $g(\|\gamma\|) \times \|x\|$. It has total runtime $O(f(\|\gamma\|, \|d\|) + g(\|\gamma\|) \times \|O\|)$. Obviously, constant delay implies data-constant delay and output-linear delay implies data-output-linear delay. We will use the term "bounded delay" to refer to any enumeration algorithm that has constant, output-linear, data-constant, or data-output-linear delay. As is evident from the total runtime bounds given, every bounded-delay enumeration algorithm can be thought of as being efficient in the output-sensitive context, provided that we can make f (and g, where applicable), small. The ideal algorithm, then, is one with output-linear delay and small f.

As already mentioned, enumeration algorithms with bounded delay have been studied in various contexts, ranging from monadic second order (MSO) queries over trees [5, 9], to relational conjunctive queries [6]. 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 in the data-complexity setting [25–27]: "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." 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 [4], where an output-linear 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, 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 output-linear delay algorithms for RGX and VA.

Contributions. The principal contribution of our work is an intuitive and efficient output-linear delay algorithm for evaluating a syntactic variant of VA that we call extended VA. Extended VA are designed to streamline the way that VA process documents. Specifically, the algorithm we present can evaluate extended VA that are both *deterministic* and *sequential*. Here, deterministic means that for every document d and every span assignment resulting from evaluating the VA $\mathcal{A}$ on d, there is only one accepting run of $\mathcal{A}$ on d generating that assignment. Sequential means that the matching algorithm encoded by $\mathcal{A}$ is well-defined in the following sense: while matching, if $\mathcal{A}$ assigns a start position to a span variable, it will also assign an end position. Furthermore, within a single run it will never overwrite a previously assigned start position; and it will never assign a stop position before assigning a start position. Given an extended VA $\mathcal{A}$ that is both deterministic and sequential, the algorithm we present can evaluate $\mathcal{A}$ over a document d with preprocessing time $O(\|\mathcal{A}\| \times \|d\|)$, and with output-linear delay. This leads to the main result of our paper:

Theorem 1.1. *There is an enumeration algorithm that, given a deterministic and sequential extended* VA $\mathcal{A}$ *and a document* d*, enumerates the result* O *of evaluating* $\mathcal{A}$ *on* d *with preprocessing time* $O(\|\mathcal{A}\| \times \|d\|)$ *and output-linear delay, yielding a total evaluation complexity of* $O(\|\mathcal{A}\| \times \|d\| + \|O\|)$.

We then study how our evaluation algorithm can be applied to arbitrary RGX and VA. We do so by showing that arbitrary RGX and VA can always be transformed into deterministic and sequential extended VA, and by subsequently studying the complexity of this transformation. This works for general RGX and VA, but also for important subclasses such as (non-extended) functional and sequential RGX and VA. As shown in [12, 17, 22], these subclasses have both good algorithmic

Class of regular spanners	Preprocessing phase
deterministic sequential extended VA	$m \cdot \|d\|$
sequential extended VA	$2^n \cdot m \cdot \|d\|$
functional VA	$2^n \cdot (n^2 + \|\Sigma\|) \cdot \|d\|$
VA / RGX	$(2^n 6^\ell + 2^n 3^\ell \|\Sigma\|) \cdot \|d\|$
$\text{VA}^{\{\cup,\bowtie\}}$ using k operations	$2^{n\cdot(k+1)} \cdot (n^{2(k+1)} + \|\Sigma\|) \cdot \|d\|$
$\text{VA}^{\{\pi,\cup,\bowtie\}}$ using k operations	$2^{n^{k+1}} \cdot (n^{2(k+1)} + \|\Sigma\|) \cdot \|d\|$

Table 1. A summary of the precomputation time taken by our output-linear delay enumeration algorithm, given for each class of regular spanners. Here, n is the number of states, ℓ the number of variables and m the number of transitions of the input VA. All preprocessing times are measured using O-notation.

properties and prohibit unintuitive behaviour. Finally, we extend our findings to the setting where spanners are specified by means of an algebra that allows to combine RGX or VA using unions, joins, and projections. Again, we analyze the complexity of transforming such algebra expressions into deterministic and sequential extended VA.

By combining the complexity of the studied transformations with the runtime of our evaluation algorithm we hence obtain upper bounds on the preprocessing times when evaluating the class of regular spanners [12] with output-linear delay. The specific upper bounds that we obtain are summarized in Table 1.

In an effort to also get an idea of potential lower bounds on preprocessing times, we study the problem of counting the number of tuples generated by a spanner. This problem is strongly connected to the enumeration problem [25], and gives evidence on whether an output-linear delay algorithm with faster preprocessing 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.

Additional material. Some of the material presented in this article was first published as a conference paper [14]. The main contributions added to this manuscript not present in the conference version can be summarized as follows:

- *A strengthened version of the main result.* In [14] we only showed that our algorithm has data-constant delay. Here, we make a more detailed analysis, showing that it has output-linear delay, which is a stronger notion than data-constant delay. In particular, in our setting output-linear delay implies that the algorithm also has data-constant delay, because the size of each output element is $O(\|\gamma\|)$. However, the converse implication does not hold in general.
- *Correctness proof for the main algorithm and auxiliary results.* The conference version of this paper only sketched the reasons explaining why the proposed algorithm works correctly. In contrast, here we formally prove the correctness of our enumeration algorithm. Similarly, [14] contained many auxiliary theorems that were only stated without formal proof. Here we provide all the missing proofs.
- *Connections with new related work.* In recent work, Amarilli et al. [3] have presented a new enumeration algorithm for arbitrary sequential VA that refuses a conjecture made in the conference version of this paper [14]. This conjecture stated that data-constant delay enumeration is not possible for arbitrary sequential VA. In Section 6 we discuss the precise relationship between [3] and our work, and emphasize that the failure of the conjecture is

based on the fact that data-constant delay is not a fine-grained enough notion to capture efficient enumeration algorithms. Based on this, we introduce a stronger variant of the conjecture that states that output-linear delay enumeration is not possible for arbitrary sequential VA.

Related work. Output-linear and data-output-linear delay enumeration algorithms for monadic second order (MSO) queries have been proposed in [5, 9, 19]. 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 enumeration for MSO queries also apply to document spanners. In [9], a data-output-linear delay enumeration algorithm was given with preprocessing time $O(\|t\| \times \log(\|t\|))$ in data complexity where $\|t\|$ is the size of the input structure. In [19], an algorithm with data-output-linear delay was given based on the deterministic factorization forest decomposition theorem, a combinatorial result for automata. Our algorithm has linear precomputation time over the input document and does not rely on any previous results, making it incomparable with [9, 19].

The output-linear delay enumeration algorithm given by Bagan in [5] requires a more detailed comparison. This algorithm works on 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 cubic 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 [17, 22] tackling the enumeration problem for document spanners directly, but focusing on polynomial delay rather than output-linear delay. In [22], a complexity theoretic treatise of polynomial delay (with polynomial preprocessing) is given for various classes of spanners. And while [22] 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 output-linear delay enumeration. On the other hand, [17] 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 preprocessing of the order $O(\|\mathcal{A}\|^2 \times \|d\|)$. The main difference of [17] and the present paper is that our algorithm can guarantee output-linear delay, but requires automata that are extended, deterministic and sequential.[2] By first converting the functional VA of [17] into a deterministic, sequential, extended VA and then using our evaluation algorithm, we can still obtain output-linear delay enumeration, but now with a preprocessing time of $O(2^{\|\mathcal{A}\|} \times \|d\|)$ (see Section 5). Therefore, if considering only functional VA, the algorithm of [17] would be the preferred option when the automaton is large and the number of outputs is relatively small, while for spanners that capture a lot of information, or are executed on very large documents, one would be better off using the output-linear delay algorithm presented here. Another difference is that the algorithm of [17] is presented in terms of automata theoretic constructions, while we aim to give a practical algorithm that is simple to implement.

Organization. We formally define the basic notions used throughout the paper in Section 2. Extended VA are introduced in Section 3. The algorithm for evaluating a deterministic and sequential

[2]Every functional VA is also sequential, therefore, the only extra requirements are that the automaton is extended and deterministic.

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	$[16, 20\rangle$		$[22, 28\rangle$

Fig. 1. A document d and the evaluation $[\![\gamma]\!]_d$, with γ as defined in Example 2.1.

extended VA with linear preprocessing and output-linear delay enumeration is presented in Section 4, and its application to regular spanners in Section 5. We study the counting problem in Section 6, and conclude in Section 7.

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 $1 \leq i \leq j$. Such a span is said to be *of document* d if $j \leq |d| + 1$. 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 position i to position $j - 1$. 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, $d(1, 5) = \texttt{John}$. Notice that if $i = j$, then $d(s) = d(i, j) = \varepsilon$. Given two spans $s_1 = [i_1, j_1\rangle$ and $s_2 = [i_2, j_2\rangle$ such that $j_1 = i_2$, we define their *concatenation* as $s_1 \cdot s_2 = [i_1, j_2\rangle$. The set of all spans of d is denoted by $\text{span}(d)$.

Mappings. Following [22], we will use mappings to model the information extracted from a document. Mappings differ from tuples (as used by e.g., Fagin et al. [12] and Freydenberger et al. [15, 16]) in that not all variables need to be assigned a span. Formally, let $\mathcal{V}$ be a fixed (and not necessarily finite) set of variables, disjoint from Σ. A *mapping* is a function μ from a finite set of variables $\text{dom}(\mu) \subseteq \mathcal{V}$ 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 $\text{dom}(\mu_1) \cap \text{dom}(\mu_2)$. 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 $\text{dom}(\mu_2) \setminus \text{dom}(\mu_1)$. The *empty mapping*, denoted by $\emptyset$, is the only mapping such that $\text{dom}(\emptyset) = \emptyset$. 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:

$$M_1 \bowtie M_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1, \mu_2 \in M_2 \text{ and } \mu_1 \sim \mu_2\}. \tag{1}$$

Document spanners. A *document spanner* is a function that maps every 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. [12] 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.

Regex formulas. Regex formulas (RGX) 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, if during this search a subformula of the form $x\{\gamma\}$ is matched against a substring, the span s that delimits this substring

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

Table 2. 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.

is recorded in a mapping $[x \to s]$ as a side-effect. Formally, the syntax of *regex formulas* is defined by the following grammar [12]:

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

Here, a ranges over letters in Σ and x over variables in $\mathcal{V}$. We will write $\text{var}(\gamma)$ 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 2 (cf. [22]). 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 $s \in \text{span}(d)$ 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 a matches all substrings of input document d equal to a, but results in only the empty mapping. On the other hand, $x\{\gamma_1\}$ 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 ($\text{var}(\gamma) = \emptyset$), 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*

$$\Sigma^* \cdot \mathit{name}\{\gamma_n\} \cdot \_ \cdot \langle \cdot (\mathit{email}\{\gamma_e\} \vee \mathit{phone}\{\gamma_p\})\cdot \rangle \cdot \Sigma^* \tag{2}$$

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 formulas here is more relaxed than the one used by Fagin et al. [12]. In particular, Fagin et al. require regex formulas to adhere to syntactic restrictions that ensure that the formula is *functional*: every mapping in $[\![\gamma]\!]_d$ is defined on all variables of γ, for every d. For regex formulas that satisfy this restriction, the semantics given here coincides with that of Fagin et al [12] (see [22] for further discussion).

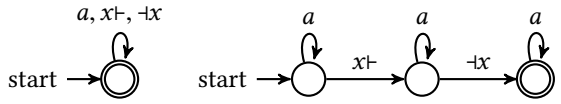

Fig. 2. A non-sequential VA (left) and an equivalent functional VA (right).

Variable-set automata. A *variable-set automaton* (VA) [12] is a finite-state automaton extended with capture 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 $\mathcal{A}$ 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 $\mathrm{var}(\mathcal{A})$ as the set of all variables x that are mentioned in some transition of $\mathcal{A}$.

A *configuration* of a VA $\mathcal{A}$ 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 of $\mathcal{A}$ 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 $o_{j+1} \in \Sigma$ and $i_{j+1} = i_j$ otherwise (i.e., the automata moves one position in the document if an only if it is reading a letter) . 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: 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 $i \leq j$.[3] 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 each variable x to $[i_j, i_k\rangle \in \mathrm{span}(d)$, where $o_j = x\vdash$ and $o_k = \dashv x$ 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, but not that all variables in $\mathrm{var}(\mathcal{A})$ actually appear in the run. Valid runs that do mention all variables in $\mathrm{var}(\mathcal{A})$ 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 a *sequential* variable-set automaton (sVA) if every accepting run of $\mathcal{A}$ over d is valid, for every document d. A VA $\mathcal{A}$ is a *functional* variable-set automaton (fVA) if every accepting run over d is functional, for every d. In particular, every fVA is also sequential. Intuitively, during a run an sVA does not need to check whether variables are opened and closed correctly; the run is guaranteed to be valid whenever a final state is reached. Figure 2 illustrates this point. The VA on the left is non-sequential, as e.g., witnessed by the accepting run

$$(q,1) \xrightarrow{x\vdash} (q,1) \xrightarrow{a} (q,2) \xrightarrow{\dashv x} (q,2) \xrightarrow{x\vdash} (q,2) \xrightarrow{a} (q,3) \xrightarrow{\dashv x} (q,3)$$

on $d = aa$, where q is the VA's only state. The VA on the right is functional, and hence also sequential. It can be verified that the VA $\mathcal{A}_2$ on the right is equivalent to the VA $\mathcal{A}_1$ on the left in the sense that $[\![\mathcal{A}_1]\!]_d = [\![\mathcal{A}_2]\!]_d$, for every d.

It was shown in [17, 22] that polynomial delay enumeration (in combined complexity) is not possible for variable-set automata in general. The hardness in this case is explained by the fact that

[3]Note that this does not require that variables are well-nested, i.e., valid runs do allow variable marker sequences of the form $x\vdash \cdots y\vdash \cdots \dashv x \cdots \dashv y$.

variable-set automata cannot consider all accepting runs. However, the authors in [22] also show that for the class of sVA (and therefore also for the class of fVA), polynomial delay enumeration is possible. As such, the sequential property is important in order to obtain 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 $\alpha \in \mathcal{L}$ and that $[\![\alpha]\!]_d$ denotes the result of evaluating the spanner denoted by α on document d. Then the semantics $[\![e]\!]$ of expression e is the spanner inductively defined by

$$\begin{aligned}
[\![\pi_Y(e)]\!]_d &= \{\mu|_Y : \mu \in [\![e]\!]_d\},\\
[\![e_1 \cup e_2]\!]_d &= [\![e_1]\!]_d \cup [\![e_2]\!]_d, \text{and}\\
[\![e_1 \bowtie e_2]\!]_d &= [\![e_1]\!]_d \bowtie [\![e_2]\!]_d.
\end{aligned}$$

Here, $\mu|_Y$ is the restriction of μ to $\mathrm{dom}(\mu) \cap Y$ and $[\![e_1]\!]_d \bowtie [\![e_2]\!]_d$ is the join of two sets of mappings, as defined by (1).

It was shown by Fagin et al. [12] that VA, $\mathrm{RGX}^{\{\pi,\cup,\bowtie\}}$, and $\mathrm{VA}^{\{\pi,\cup,\bowtie\}}$ all express the same class of spanners, called *Regular Spanners*. In particular, every expression in $\mathrm{RGX}^{\{\pi,\cup,\bowtie\}}$, and $\mathrm{VA}^{\{\pi,\cup,\bowtie\}}$ is equivalent to a VA. These results were later extended to the class of sequential VA in [22], where a full overview of the relationship between different versions of VA, regex formulas, and the associated algebras is given. This will be used later in Section 5.

The evaluation problem. In this paper, we study the problem of computing $[\![\gamma]\!]_d$, 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 evaluation problem for $\mathcal{L}$ formally as follows:

Problem:	Eval$[\mathcal{L}]$
Input:	Expression $\gamma \in \mathcal{L}$ and a document d.
Output:	The set $[\![\gamma]\!]_d$.

Model of computation. As it is standard in the literature [5, 9, 25, 26], we consider algorithms on Random Access Machines (RAM) with uniform cost measure [1] and addition and substraction as basic operations. A RAM has read-only input registers (containing the input I), read-write work memory registers and write-only output registers.

In our case, the input will consist of document spanner γ and document d, whose sizes we measure as follows. We assume that each letter $\sigma \in \Sigma$ can be encoded in a single register. This implies that $\|d\| = |d|$ (recall that $|d|$ denotes the length of d). We further assume that each variable $x \in \mathcal{V}$ can be encoded in a single register. A regex formula γ is encoded on the RAM input tape as a sequence of registers, where each register encodes a single alphabet symbol or operation in γ. As such, $\|\gamma\|$ is the number of occurrences of alphabet symbols and operations in γ, as inductively defined by:

$$\|\varepsilon\| = 1 \qquad \|\sigma\| = 1 \qquad \|x\{\gamma\}\| = \|\gamma\| + \|x\| + 2$$
$$\|\gamma_1 \cdot \gamma_2\| = \|\gamma_1\| + \|\gamma_2\| + 1 \qquad \|\gamma_1 \vee \gamma_2\| = \|\gamma_1\| + \|\gamma_2\| + 1 \qquad \|\gamma^*\| = \|\gamma\| + 1$$

We are considering various forms of directed graphs throughout the paper (without labels, with node labels, with edge labels, with distinguished nodes, etc). These graphs are assumed to be encoded on the input tape in the standard way, i.e. by listing the set of nodes and the set of edges, as well as distinguished nodes (if any). A single node is assumed to be encoded in a single register (if it is unlabeled), or a single register followed by an encoding of its label. Similarly, edges are encoded by encoding the endpoints and the edge label (if any). Concretely, under this encoding for graphs, the size $\|\mathcal{A}\|$ of a VA $\mathcal{A}$ (which is a graph) is linear in the number of transitions plus the number of states plus the number of variables. Finally, we assume that algebraic expressions $e \in \mathcal{L}^{\{\pi,\cup,\bowtie\}}$ with $\mathcal{L} = \text{RGX}$ or $\mathcal{L} = \text{VA}$ are encoded by representing their parse trees: the subexpressions of e that are in $\mathcal{L}$ are encoded as defined above, while each operator in $\{\pi, \cup, \bowtie\}$ (as well as parenthesis) are encoded in a single register. As such, $\|e\| = \sum_i \|\alpha_i\| + c$ where α_i are the subexpressions of e in $\mathcal{L}$, and c is the number of occurrences of symbols in $\{\pi, \cup, \bowtie, (,)\}$ in e.

Enumeration with output-linear delay. To solve Eval$[\mathcal{L}]$ we adapt the notion of enumeration with output-linear delay, as used in [5, 9], as follows. We say that algorithm $\mathcal{E}$ is an *enumeration algorithm for* Eval$[\mathcal{L}]$ if $\mathcal{E}$ runs in two phases, for every spanner $\gamma \in \mathcal{L}$ and a document d.

- The first phase, called the *preprocessing* phase, does not produce output, but may prepare data structures for use in the next phase.
- The second phase, called the *enumeration* phase, occurs immediately after the precomputation phase. During this phase, the algorithm:
  - writes $\#y_1\#y_2\#\ldots\#y_n\#$ to the output registers where # is a distinct separator symbol, and $(y_1, \ldots, y_n)$ is an enumeration (without repetition) of the set $[\![\gamma]\!]_d$;
  - it writes the first # as soon as the enumeration phase starts,
  - it stops immediately after writing the last #.

The separation of $\mathcal{E}$'s operation into a preprocessing and enumeration phase is done to be able to make an output-sensitive analysis of $\mathcal{E}$'s complexity. Formally, we say that $\mathcal{E}$ has *preprocessing time* $f : \mathbb{N}^2 \to \mathbb{N}$ if the number of instructions that $\mathcal{E}$ executes during the preprocessing phase on input (γ, d) is at most $f(\|\gamma\|, \|d\|)$, for every $\gamma \in \mathcal{L}$ and document d.

Furthermore, we measure the delay as follows. Let $time_i(\gamma, d)$ denote the time in the enumeration phase when the algorithm writes the i-th # (if it exists) when running on input (γ, d). Define $delay_i(\gamma, d) = time_{i+1}(\gamma, d) - time_i(\gamma, d)$. Further, let $output_i(\gamma, d)$ denote the i-th element y_i that is output by $\mathcal{E}$ when running on input (γ, d), if it exists. Then $\mathcal{E}$ is said to have *output-linear delay* if there exists a constant k such that, for all $\gamma \in \mathcal{L}$ and $d \in \Sigma^*$, it holds that $delay_i(\gamma, d) \leq k \cdot \|output_i(\gamma, d)\|$ for every $1 \leq i \leq |[\![\gamma]\!]_d|$. Furthermore, if $[\![\gamma]\!]_d$ is empty, then $delay_1(\gamma, d) \leq k$ (i.e. its ends in constant time).

3 EXTENDED VARIABLE-SET AUTOMATA

In this section, we present a syntactic variant of VA that we call *extended variable-set automata*, or eVA for short. This variant is based on ideas from [23] and avoids several problems that VA have in terms of evaluation. In Section 4 we will give an enumeration algorithm for evaluating eVAs that has $O(\|\mathcal{A}\| \times \|d\|)$ preprocessing time and output-linear delay, provided that the input eVA $\mathcal{A}$ is both deterministic and sequential. Later, in Section 5, we show how this algorithm can be applied when spanners are represented by means of ordinary VA, RGX formulas, or spanner algebras, by studying the complexity of converting these spanner models into sequential and deterministic eVA.

To see why VA can be problematic to evaluate by means of enumeration algorithms, observe that VA can open and close variables in arbitrary ways, which can cause multiple runs to define the same output mapping. An example of this is given in Figure 3. On the left we have a functional VA that, on input document $d \in a^*$, admits two runs that result in the same mapping $\mu = [x \mapsto$

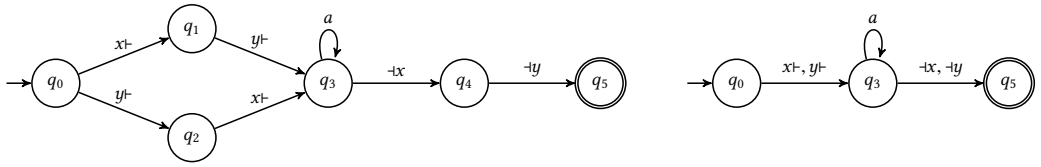

Fig. 3. A functional VA with two different runs that define the same output mapping (left) and an equivalent I/O deterministic eVA (right).

$[1, |d| + 1\rangle, y \mapsto [1, |d| + 1\rangle]$. This behavior is problematic for enumeration algorithms, as outputs must be enumerated without repetitions[4].

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 an input letter.

Formally, let $\text{Markers}_{\mathcal{V}} = \{x\vdash, \dashv x \mid x \in \mathcal{V}\}$ be the set of variable 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 (p, a, q), or *extended variable transitions* (q, S, p), where $S \subseteq \text{Markers}_{\mathcal{V}}$, $S \neq \emptyset$, and $p, q \in Q$. A run ρ over a document $d = a_1 a_2 \cdots a_n$ is a sequence of the form:

$$\rho \;=\; q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} q_1 \xrightarrow{S_2} p_1 \xrightarrow{a_2} \;\ldots\; \xrightarrow{a_n} q_n \xrightarrow{S_{n+1}} p_n \tag{3}$$

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 $S_{i+1} \neq \emptyset$, and $q_i = p_i$ otherwise. Notice that extended variable transitions and letter transitions must alternate in a run of an eVA. Furthermore, while the transition relation does not allow extended variable transitions of the form $(q, \emptyset, p)$, transitions $q_{i-1} \xrightarrow{S_i} p_i$ with $S_i = \emptyset$ are allowed in a run, but only if $q_{i-1} = p_i$. Also notice that alternation between variable transitions and letter transitions is only required in the definition of a run; no such condition is imposed on the transition function of an eVA itself.

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 $j \geq i$ with $\dashv x \in S_j$; and, conversely, for every j and every $\dashv x \in S_j$ there exists $i \leq j$ 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, $x\vdash \in S_i$ and $\dashv x \in S_j$. 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 standard VA to extended VA in the obvious way.

The following result shows that eVA have the same expressive power as standard VA. We use $\mathcal{A} \equiv \mathcal{A}'$ to denote that two automata are equivalent; that is, for every document d, we have that $[\![\mathcal{A}]\!]_d = [\![\mathcal{A}']\!]_d$.

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).*

[4]As shown in [12], this behaviour also leads to a blow-up that is factorial in the number of variables when defining the join of two VA, as all possible orders between variables need to be considered. See Section 5 for further discussion.

Proof. Let $\mathcal{A} = (Q, q_0, F, \delta)$ be a VA. First of all, we require the following notion. A *variable path* between two states p and q in $\mathcal{A}$ is a sequence $\pi : p = p_0 \xrightarrow{v_1} p_1 \xrightarrow{v_2} \ldots \xrightarrow{v_h} p_h = q$ such that $(p_i, v_{i+1}, p_{i+1}) \in \delta$ are variable transitions in $\mathcal{A}$ and $v_i \neq v_j$ for every $i \neq j$. We write Markers(π) for the set $\{v_1, \ldots, v_h\}$ of all variable markers appearing in π.

Given the VA $\mathcal{A}$, the resulting eVA $\mathcal{A}'$ should produce valid runs that alternate between letter transitions and extended variable transitions. To this end, construct the 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 Markers(π) $= S$. 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. Consider an arbitrary eVA $\mathcal{A}' = (Q', q_0', F', \delta')$. The construction of the desired equivalent VA $\mathcal{A}$ is intuitively straightforward: for every extended transition between two states in $\mathcal{A}'$, create a single variable-path between those two states in $\mathcal{A}$ that has 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, an arbitrary order $\preceq$ of variables $\mathcal{V}$ can be extended over Markers$_{\mathcal{V}}$ such that for every pair of variables $x, y \in \mathcal{V}$: $x\vdash \preceq \dashv y$, 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 follows immediately after, following the induced order.

Formally, define 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:

$$\delta_{\text{first}} = \{(p, v, q^v_{(p,S,p')}) \mid v \text{ is the } \preceq\text{-minimum element in } S\ \}$$

$$\delta_{\text{succ}} = \{(q^v_{(p,S,p')}, v', q^{v'}_{(p,S,p')}) \mid v, v' \in S \text{ and } v' \text{ is the } \preceq\text{-successor of } v \text{ in } S\}$$

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

$$\delta_{\text{one}} = \{(p, v, p') \mid (p, \{v\}, p') \in \delta'\}.$$

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. The transitions in δ_{first} define how to get to the first state of the path, using the first marker of S. The transitions in δ_{succ} define how to get to the next marker in S, and the transitions in δ_{last} define how to get back to the state p', having finished the extended transition. The transitions in δ_{one} define the case when $|S| = 1$ and no intermediate states are needed. Note that a different set of intermediate states are added for each extended transition (p, S, p'), so states do not get reused and 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.

Note that for both constructions, if the input automaton is sequential or functional, then the output automaton preserves this property. In the first construction, 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. Similar observations hold for the second construction. □

Another important notion for our output-linear delay algorithm is the notion of determinization of an extended VA. More precisely, an extended VA $\mathcal{A}$ is *I/O deterministic* (for input-output deterministic) if the transition relation δ of A is a partial function $\delta : Q \times (\Sigma \cup 2^{\text{Markers}_\mathcal{V}} \setminus \{\emptyset\}) \to Q$. If $\mathcal{A}$ is I/O deterministic, then we define $\text{Markers}_\delta(q)$ as the set $\{S \subseteq \text{Markers}_\mathcal{V} \mid (q, S) \in \text{dom}(\delta)\}$. We also define $\text{Markers}_\mathcal{A}$ as the union of all $\text{Markers}_\delta(q)$, for all $q \in \mathcal{A}$. Note that, in contrast to determinism for classical NFAs, I/O determinism as defined here does not imply 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 $\mathcal{A}$ of Figure 3 (left) into the equivalent eVA $\mathcal{A}'$ of Figure 3 (right). It is easy to see that $\mathcal{A}'$ is I/O deterministic, so all accepting runs will define a unique mapping, thus avoiding the issues that $\mathcal{A}$ has when considering the enumeration of output mappings. For the sake of presentation, in the future we will refer to I/O deterministic VA just as deterministic VA. The following result shows that all eVA can be determinized.

Proposition 3.2. *For every* eVA $\mathcal{A}$ *there exists a deterministic* eVA $\mathcal{A}'$ *such that* $\mathcal{A} \equiv \mathcal{A}'$. *Furthermore, if* $\mathcal{A}$ *is sequential (resp. functional), then* $\mathcal{A}'$ *is also sequential (resp. functional).*

Proof. This 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 $F' = \{B \in 2^Q \mid B \cap F \neq \emptyset\}$ 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 $[\![\mathcal{A}]\!]_d = [\![\mathcal{A}']\!]_d$ 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 into 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 exist if a sequence of states using the same transitions exists in the original automaton $\mathcal{A}$. Similarly, a non-valid run in $\mathcal{A}'$ can only exist if a non-valid run exists in $\mathcal{A}$. Putting all of these observations together, we conclude that if $\mathcal{A}$ is sequential (resp. functional), then all accepting runs are valid (resp. functional), and hence $\mathcal{A}'$ must also be sequential (resp. functional). □

In Section 5 we will study in detail the complexity of the translations given by Theorem 3.1 and Proposition 3.2; to present our algorithm we only require equivalence between the models.

4 EVALUATING EXTENDED VA WITH OUTPUT-LINEAR DELAY

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 output-linear delay after $O(\|\mathcal{A}\| \times \|d\|)$ preprocessing. We start with an intuitive explanation of the algorithm's underlying idea, and then develop the algorithm and analyze its complexity.

4.1 Intuition

As with the majority of bounded-delay enumeration algorithms, we build a compact representation of the output $[\![\mathcal{A}]\!]_d$ in the preprocessing phase, and use this representation to produce the output in the enumeration phase. 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 eVA $\mathcal{A}_d$ containing the same variables as $\mathcal{A}$;

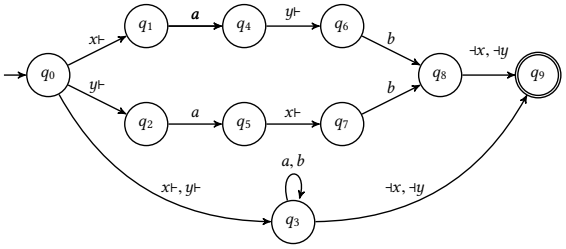

Fig. 4. An extended functional VA $\mathcal{A}$.

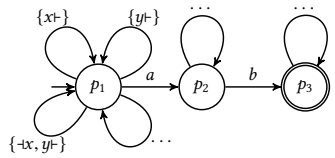

Fig. 5. The deterministic eVA $\mathcal{A}_d$ for $d = ab$.

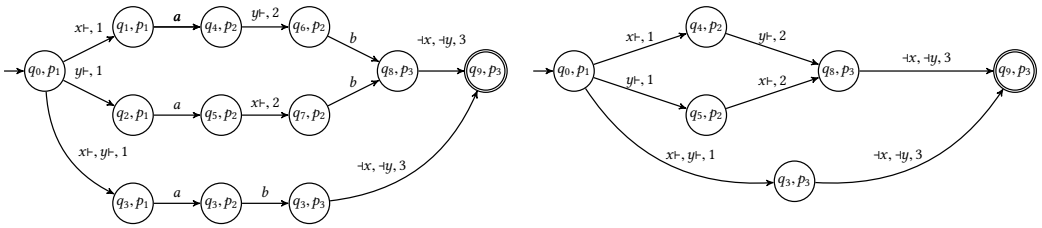

Fig. 6. The annotated product automaton (left) and its "forward"ε-closure (right).

- 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. Consider the eVA $\mathcal{A}$ from Figure 4. It is straightforward to check that this automaton is functional (hence sequential) and deterministic. To evaluate $\mathcal{A}$ over document $d = ab$ we first 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 $2^{2 \cdot |\operatorname{var}(\mathcal{A})|} - 1$ self loops, each labeled by a different non-empty subset of $\text{Markers}_{\operatorname{var}(\mathcal{A})}$. For instance, in the case of $d = ab$, the automaton $\mathcal{A}_d$ is shown in Figure 5.

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 this construction, we take care of creating only 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 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 6 (left).

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 and computing all 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 on the left of Figure 6, we would add a transition

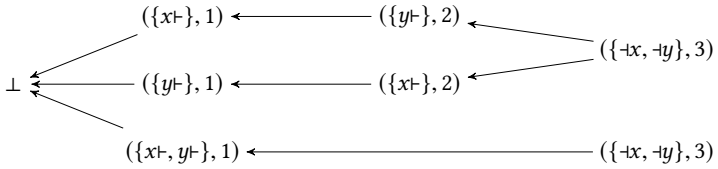

Fig. 7. The mapping DAG corresponding to the ε-closure shown in Fig. 6(right). Here the information in the brackets represents the node label.

$((q_0, p_1), (x\vdash, 1), (q_4, p_2))$, because 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 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 ε transitions from the resulting automaton. An example of this process for the automaton $\mathcal{A}$ of Figure 4 and the document $d = ab$ is given in Figure 6 (right).

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. Note that every path from the final to the initial state is of size linear in the mapping that it represents. Thus, we take linear time in producing each mapping and this time does not depend on the size of $\mathcal{A}$ or d. For example, consider in Figure 6 (right) that we start in the accepting state (q_9, p_3), move backwards to (q_3, p_3), and then further back to the initial state (q_0, p_1). From the labels along this path we can reconstruct the mapping μ with $\mu(x) = \mu(y) = [1, 3\rangle$. To enumerate the next mapping, we would backtrack to the accepting state (q_9, p_3), start again from there reaching (q_8, p_3), and continue until we reach the initial state. Note that, since $\mathcal{A}$ and $\mathcal{A}_d$ are deterministic, we will never output the same mapping twice when traversing backwards. Hence, we will successfully enumerate all the mappings with a delay that is linear in the size of the current mapping.

4.2 The mapping DAG

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 and instead *traverse* this product automaton *on the fly* by processing the input document one letter at a time, incrementally constructing the ε-closure at the same time. The traversing and construction together will have total complexity $O(\|\mathcal{A}\| \times \|d\|)$.[5]

For the description of the algorithm it will be convenient to not work with the ε-closure, which is an edge-labeled graph, but encode the same information in a node-labeled graph that we call a *mapping DAG*. To illustrate the connection between the ε-closure and the mapping DAG that we will construct, Figure 7 shows the mapping DAG corresponding to the ε-closure of Figure 6 (right). Essentially, $\perp$ in the mapping DAG plays the role of (q_0, p_0) in the forward ε-closure. Each edge in the ε-closure gets represented by a node in the mapping DAG with the same label. Moreover, for all paths $(q, p) \xrightarrow{(S,i)} (q', p') \xrightarrow{(T,j)} (q'', p'')$ in the ε-closure we will have an edge $u \leftarrow v$ in the mapping DAG, where u is labeled (S, i) and v is labeled (T, j). Note that the direction of edge in the mapping DAG is the opposite of the direction of the path in the ε-closure.

[5]Note that, in contrast, a naive implementation according to our intuitive description that explicitly constructs $\mathcal{A}_d$ has a preprocessing time $\Omega(2^{\mathrm{var}(\mathcal{A})} \times \|d\|)$ because this is the size of $\mathcal{A}_d$.

The formal definition of a mapping DAG is as follows.

Definition 4.1. A *mapping DAG* G is a tuple $G = (V, E, \bot, \lambda)$ consisting of a finite set V of nodes, a set of edges $E \subseteq V \times V$, a distinguished node $\bot \in V$ called the *sink*, and a labeling function λ that labels each node $v \in V \setminus \{\bot\}$ by a pair (S, i) where S is a non-empty set of variable-markers, and $i \in \mathbb{N}$ is a position. It is required that the graph (V, E) is acyclic, that $\bot$ has no outgoing edges in this graph, and that for every node $v \in V$ there is a directed path from v to $\bot$. If $v = \bot$ then this path consists of the single node $\bot$ itself.

Given a set of nodes $U \subseteq V$ we denote by $\text{paths}(G, U)$ the set of paths in G that start at a node in U and terminate at $\bot$. We denote with $\text{paths}(G)$ the set $\text{paths}(G, V)$, of all paths in G that end in $\bot$. Intuitively, the sequence of labels of the paths in $\text{paths}(G)$ will encode the mappings that we need to output. This notion is made formally precise as follows. For a path $\pi = v_1 \to v_2 \to \cdots \to v_k \to \bot \in \text{paths}(G)$ we define the label $\lambda(\pi)$ of π to be the sequence $\lambda(v_k), \lambda(v_{k-1}), \ldots, \lambda(v_1)$, i.e., it is the string of node labels when the path is traversed in the opposite direction (ignoring $\bot$, which does not have any label). If $k = 0$ and $\pi = \bot$ then $\lambda(\pi)$ is simply the empty sequence.

Definition 4.2. A *mapping sequence* M is a sequence $M = (S_1, i_1), \ldots, (S_m, i_m)$ of pairs (S_k, i_k) such that (1) every set S_k is non-empty; (2) the positions are strictly increasing, $i_1 < i_2 \cdots < i_m$; and (3) the variables in $S_1 \ldots S_m$ are opened and closed in a correct manner, i.e. the sets S_k are pairwise disjoint; for every k and every $x\vdash \in S_k$ there exists $l \geq k$ with $\dashv x \in S_l$; and, conversely, for every l and every $\dashv x \in S_l$ there exists $k \leq l$ with $x\vdash \in S_k$. We denote by μ^M the mapping that maps x to $[k, l\rangle$ if, and only if, $x\vdash \in S_k$, $\dashv x \in S_l$ and $k \leq l$. We say that M *represents* a mapping ν if $\nu = \mu^M$.

Note that if $m = 0$ then M represents the empty mapping. It is straightforward to see that for every mapping μ there is exactly one mapping sequence that represents it. As such, mappings and mapping sequences are in one-to-one relationship. Moreover, writing $\|M\|$ for $\sum_{j=1}^{m}(|S_{i_j}| + 1)$ (which denotes the space needed to encode M in a RAM under our assumption that variables can be encoded in a single register) we have $\|M\| = O(|\text{dom}(\mu^M)|)$, i.e., mapping sequences are of size linear in the mapping that they represent.

We can now make precise what it means for a mapping DAG to *represent* a spanner result $[\![\gamma]\!]_d$.

Definition 4.3. Let $G = (V, E, \bot, \lambda)$ be a mapping DAG and let $U \subseteq V$. We say that (G, U) *represents* a set of mappings O if for every $\mu \in O$ there exists $\pi \in \text{paths}(G, U)$ such that $\lambda(\pi)$ represents μ, and for every path $\pi \in \text{paths}(G, U)$ it is the case that $\lambda(\pi)$ is a mapping sequence which represents some $\mu \in [\![\mathcal{A}]\!]_d$.

The above definition does not exclude the possibility that a single mapping $\mu \in O$ is represented by multiple $\pi \in \text{paths}(G, U)$. To exclude this possibility, we say that (G, U) is *without duplicates* if for every two distinct paths $\pi_1, \pi_2 \in \text{paths}(G, U)$ also $\lambda(\pi_1)$ and $\lambda(\pi_2)$ are distinct.

Our algorithm for evaluating a deterministic seVA$\mathcal{A}$ on document d consists of two subroutines, called Preprocessing and Enumerate. Preprocessing represents the preprocessing phase, during which we construct a mapping DAG G and set of vertices U such that (G, U) is without duplicates and represents $[\![\mathcal{A}]\!]_d$. Enumerate then uses (G, U) to enumerate the set $\{\lambda(\pi) \mid \pi \in \text{paths}(G, U)\}$ with output-linear delay. Because (G, U) represents $[\![A]\!]_d$ and because of the one-to-one correspondence between mapping sequences and mappings, we do not make a distinction between enumerating $\{\lambda(\pi) \mid \pi \in \text{paths}(G, U)\}$ and the set of mappings $[\![\mathcal{A}]\!]_d$. We next describe both subroutines, starting with the enumeration phase.

4.3 Enumeration procedure

If (G, U) represents a set of mappings O, then enumerating O from (G, U) is conceptually simple: since $\text{paths}(G, U)$ encodes exactly the mappings in O it suffices to enumerate $\text{paths}(G, U)$ and, for

each such path π, output $\lambda(\pi)$. If (G, U) is without duplicates, then we are certain not to enumerate the same mapping twice. As we have already intuitively described in Section 4.1, enumerating paths(G, U) can be done by a depth-first traversal of G, starting from the nodes in U. In order to formally prove that this enumeration is with output-linear delay, we have to be precise about when the separator symbols # are written; after all, the delay is measured as the time elapsed between writing these symbols.

Concretely, we do depth-first traversal as follows. Assume for simplicity of exposition that U contains only a single node u. During the traversal, we maintain a path π in G. Such a path is called *complete* if the last node in π is $\perp$, it is called *partial* otherwise. Initially, π is partial and consists of a single node: u.

(1) (Start) Output #, indicating the start of the enumeration phase.
(2) (Forward) Let $\pi : u = v_0 \to v_1 \to \cdots \to v_l$ be the current partial path. Make this path complete by navigating from v_l to $\perp$. This is possible by definition of a mapping DAG.
(3) (Output) In this step, π is complete. Its label hence represent a mapping. Output $\lambda(\pi)$.
(4) (Backward) Let $\pi : u = v_0 \to \cdots \to v_k = \perp$ be the current complete path. Transform π back into a partial path by removing the suffix $v_{j+1} \to \cdots \to v_k$, where j is the largest index smaller than k such that there is an edge $v_j \to v'_{j+1}$ and the partial path $v_0 \to \cdots \to v_j \to v'_{j+1}$ has not yet been previously considered. Update π such that it becomes $\pi : v_0 \to \cdots \to v_j \to v'_{j+1}$. If no such index j exists, set π to empty.
(5) (Mark) Output #. Goto step (2) if π is non-empty, otherwise terminate.

The case where U consists of multiple nodes is an easy generalization: whenever π becomes empty, set $\pi = u'$ with u' an unprocessed node in U, and repeat. Terminate when all of U is processed. If U was empty to begin with, print # to indicate the end of enumeration immediately after step (1), and terminate.

Let us now analyze the delay of this algorithm. For the moment, assume that in step (2) the time required to complete the current partial path $u = v_0 \to \cdots \to v_l$ into a complete path $u = v_0 \to \cdots \to v_l \to v_{l+1} \to \cdots \to v_k = \perp$ is bounded by $O(k)$ (i.e. the size of the current output). Further assume that in step (4) backtracking from $u = v_0 \to \cdots \to v_l \to v_{l+1} \to \cdots \to v_k = \perp$ to $u = v_0 \to \cdots \to v_j \to v_{j+1}$ can be done in time $O(k)$. Observe that under these assumptions, we enumerate with output-linear delay. Indeed, step (1) is $O(1)$ and only executed once, before the first mapping is output. Step (2) is $O(k)$ by assumption. Observe that the size of the mapping $\lambda(\pi)$ produced in step (3) is $\Omega(k)$. Hence, step (2) is linear in the mapping sequence that is output in step (3). Furthermore, step (3) is itself clearly linear in $\|\lambda(\pi)\|$. Step (4) is $O(k)$ by assumption, hence linear in the mapping sequence $\lambda(\pi)$ that we are outputting; and finally step (5), which finishes the generation of $\lambda(\pi)$ by writing the separator symbol # is $O(1)$. Therefore, the overall delay is linear in $\|\lambda(\pi)\|$.

Implementing step (2) and step (3) such that they run in the required time $O(k)$ is straightforward when mapping DAG G uses an adjacency-list representation. Indeed, under this representation, we can retrieve, for every node v, an *iterator* `out`(G, v) to the set of all nodes $\{w \mid v \to w \text{ edge in } G\}$ in $O(1)$ time. The returned iterator is positioned before the first element and supports two $O(1)$ operations: `next()` which returns the next unvisited element in this set if it exists and at the same time advances the iterator one element; and `has_next()` which checks if there is such a next unvisited element. Steps (2) and (3) are then implemented by maintaining, in addition to $\pi : v_0 \to v_1 \to \cdots \to v_l$ a sequence of iterators $\iota : i_0, i_1, \ldots, i_l$ such that we can use iterator i_l to check if v_l has a successor v'_{l+1} that we did not process yet. Note that we use iterator i_0 to iterate over the elements of U.

Algorithm 1 Enumeration of the mappings represented by a mapping DAG.

```
1:  procedure ENUMERATE(G, U)
2:      Output #
3:      if U = ∅ then Output # and terminate
4:      Initialize iterator i_0 over U
5:      Initialize π and ι to empty sequence
6:      Push i_0 on ι
7:      while ι non-empty and
8:              ι.top().has_next() do
9:          FORWARD(G, π, ι)
10:         Output λ(π)
11:         BACKWARD(G, π, ι)
12:         Output #
13: procedure FORWARD(G, π, ι)
14:     while ι.top().has_next() do
15:         v ← ι.top().next()
16:         push v on π
17:         push out(G, v) on ι
18: procedure BACKWARD(G, π, ι)
19:     // top of π is ⊥, its iterator has no next
20:     while ι non-empty and
21:             not ι.top().has_next() do
22:         Pop ι
23:         Pop π
```

The complete pseudo-code for ENUMERATE is given in Algorithm 1. There we use π and ι as stacks equipped with the following operations. Operation `push` adds a new element at the right end of π and ι. Operation `top` retrieve the last element, and `pop` removes this last element (also returning that element). Using textbook data structures for stacks, these operations can be supported in $O(1)$ time. From the above discussion we conclude the following.

PROPOSITION 4.4. *Assume (G, U) is without duplicates. If G is given under the adjacency-list representation of graphs, then ENUMERATE (G, U) enumerates the set $\{\lambda(\pi) \mid \pi \in \text{paths}(G, U)\}$ without duplicates and with output-linear delay.*

We note that while in Line 10 of Algorithm 1 we assume that it suffices to output a mapping sequence $M = (S_1, i_1), \ldots, (S_k, i_k)$ in order to output the mapping μ^M, other representations of μ^M can be derived from M in time linear in $\|M\|$, if desired. For example, if, rather than M we wish to output the set $\{(m, i) \mid (S, i)$ a pair in $M, m \in M\}$ that contains all pairs (m, i) with m a variable marker and i the position where the marker applies, then this can clearly be computed from M in time $O(\|M\|) = O(\|\mu^M\|)$.

4.4 Preprocessing procedure

Given a deterministic seVA $\mathcal{A}$ and a document d, PREPROCESSING constructs a mapping DAG G and subset of its vertices U such that (G, U) represents $[\![\mathcal{A}]\!]_d$ without duplicates. To that end, it uses the following operations on mapping DAGs. The first method, `NewMappingGraph()`, creates a new mapping DAG containing only the sink $\perp$ and empty set of nodes V and edges E. The second method, `AddNewNode`(G, S, i), takes as it input a mapping DAG G, a non-empty set of variable markers S, and a number $i > 0$, and creates a fresh node v not belonging to the set V of nodes of G. The newly created node is then added to V. The method also extends λ by defining $\lambda(v) = (S, i)$. Finally, the method returns the node v. Note that upon executing this method on G, we will effectively extend G with a single disconnected node v. Formally, since v is disconnected, G is not a mapping DAG any longer. However, calls to `AddNewNode` will directly be followed by calls to the next operation, which connects v, and hence makes G a mapping DAG again. The final method we require is called `Connect`(G, v, A), and it takes as its input a graph G with the set of nodes $V \cup \{\perp\}$, a node $v \in V$, and a set $A \subseteq V \cup \{\perp\}$. The method adds an edge (v, v') to G, for every node v' in A.

In this subsection, we describe PREPROCESSING based on these operations. This allows us to focus on the logic behind PREPROCESSING and more easily shows it correctness. Once this is done, in

Section 4.6 we discuss how to implement these operations to ensure that PREPROCESSING runs in time $O(\|\mathcal{A}\| \times \|d\|)$.

We will need the following notation and terminology on runs. Recall that a run of eVA $\mathcal{A}$ over $d = a_1 a_2 \dots a_n$ is a sequence ρ of the form:

$$\rho\colon q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} q_1 \xrightarrow{S_2} p_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n \xrightarrow{S_{n+1}} p_n. \tag{4}$$

That is, when processing the document d, the eVA $\mathcal{A}$ first invokes a variable transition which uses the set of markers S_1, followed by a letter transition which consumes the first symbol a_1 of d. The automaton then invokes the next variable transition using the set of markers S_2, followed by a letter transition consuming a_2, etc, and ends with a variable transition. Define a *partial run* similar to a run, but ending with a letter transition. As such, a partial run ρ of $\mathcal{A}$ on $d = a_1 a_2 \dots a_n$ is a sequence of the form

$$\rho\colon q_0 \xrightarrow{S_1} p_0 \xrightarrow{a_1} q_1 \xrightarrow{S_2} p_1 \xrightarrow{a_2} \dots \xrightarrow{S_n} p_{n-1} \xrightarrow{a_n} q_n. \tag{5}$$

Note that if $n = 0$ and $d = \varepsilon$ then there is only one partial run of $\mathcal{A}$ on d, which consists only of the start state q_0. Clearly, extending a partial run by following an additional variable transition (possibly with empty set of variable-markers) yields a run and, conversely, extending a run on d by following an additional letter transition with letter a_{n+1} yields a partial run for $d' = a_1 \dots a_n a_{n+1}$. From this observation, the following lemma straightforwardly follows. Let $\mathit{runs}(\mathcal{A}, d, q)$ (resp. $\mathit{part\text{-}runs}(\mathcal{A}, d, q)$) denote the set of all runs (resp. partial runs) of $\mathcal{A}$ on d that end in state q.

LEMMA 4.5. *For every extended* eVA $\mathcal{A}$*, every document* d*, every state* p *in* $\mathcal{A}$ *and every letter* $a \in \Sigma$ *the following hold:*

$$\mathit{runs}(\mathcal{A}, d, p) = \bigcup_{(q,S,p)\in\delta} \{\rho \xrightarrow{S} p \mid \rho \in \mathit{part\text{-}runs}(\mathcal{A}, d, q)\} \cup \{\rho \xrightarrow{\emptyset} p \mid \rho \in \mathit{part\text{-}runs}(\mathcal{A}, d, p)\}$$

$$\mathit{part\text{-}runs}(\mathcal{A}, da, p) = \bigcup_{(q,a,p)\in\delta} \{\rho \xrightarrow{a} p \mid \rho \in \mathit{runs}(\mathcal{A}, d, p)\}.$$

For the purpose of representing $[\![\mathcal{A}]\!]_d$ by a mapping DAG we are of course not interested in the runs of $\mathcal{A}$ on d themselves, but in the sequences of extended variable transitions that these make. We formalize this as follows.

Definition 4.6. Let ρ be a run or partial run of $\mathcal{A}$ as shown in (4) or (5). We define $\mathit{out}(\rho)$ to be the (possibly empty) sequence obtained by concatenating all the pairs (S_j, j) with $S_j \neq \emptyset$, in an increasing order on j. We call $\mathit{out}(\rho)$ the *out*-sequence of ρ.

For instance, if $\rho = q_0 \xrightarrow{\{x\vdash\}} p_0 \xrightarrow{a_1} q_1 \xrightarrow{\emptyset} p_1 \xrightarrow{a_2} q_2 \xrightarrow{\{y\vdash\}} p_2$ then $\mathit{out}(\rho) = (\{x\vdash\}, 1), (\{y\vdash\}, 3)$. Note that, as shown by this example, $\mathit{out}(\rho)$ is not necessarily a mapping sequence, even when ρ is a run and not a partial run. In particular, variables need not be opened and closed in the correct order in $\mathit{out}(\rho)$, and some variables may be opened but not closed (or vice versa). However, if ρ is an accepting run (i.e., $p_n \in F$ with F the set of final states of $\mathcal{A}$) and $\mathcal{A}$ is sequential, then $\mathit{out}(\rho)$ is guaranteed to be a mapping sequence because every accepting run is valid in a sequential automaton, ensuring that variables are both opened and closed, and in the correct order.

Define $\mathit{out}(\mathcal{A}, d, q) = \{\mathit{out}(\rho) \mid \rho \in \mathit{runs}(\mathcal{A}, d, q)\}$ be the set of all *out* sequences of runs on d that end in state q and similarly $\mathit{part\text{-}out}(\mathcal{A}, d, q) = \{\mathit{out}(\rho) \mid \rho \in \mathit{part\text{-}runs}(\mathcal{A}, d, q)\}$. From the definition of $\mathit{out}(\rho)$ and Lemma 4.5 the following is now straightforward to obtain.

Algorithm 2 Preprocessing phase for an automaton $\mathcal{A}$ over the document $a_1 \dots a_n$

```
1:  procedure Preprocessing(A, a_1 ... a_n)
2:      G ← NewMappingGraph()
3:      for all q ∈ Q \ {q_0} do
4:          R_q^1 ← ∅
5:      R_{q_0}^1 ← {⊥}
6:      for i := 1 to n do
7:          Capturing(i)
8:          Reading(i + 1)
9:      Capturing(n + 1)
10:     return (G, ⋃_{q∈F} C_q^{n+1})

11: procedure Capturing(i)
12:     for all q ∈ Q do
13:         C_q^i ← R_q^i
14:     for all q ∈ Q with R_q^i ≠ ∅ do
15:         for all S ∈ Markers_δ(q) do
16:             p ← δ(q, S)
17:             v ← AddNewNode(G, S, i)
18:             Connect(G, v, R_q^i)
19:             C_p^i ← C_p^i ∪ {v}

20: procedure Reading(i + 1)
21:     for all q ∈ Q do
22:         R_q^{i+1} ← ∅
23:     for all q ∈ Q with C_q^i ≠ ∅ do
24:         p ← δ(q, a_i)
25:         R_p^{i+1} ← R_p^{i+1} ∪ C_q^i
```

Lemma 4.7. *For every extended* eVA $\mathcal{A}$*, every document* d*, every state* p *in* $\mathcal{A}$ *and every letter* $a \in \Sigma$ *the following hold:*

$$out(\mathcal{A}, d, p) = \bigcup_{(q,S,p)\in\delta} \{\sigma, (S, |d|+1) \mid \sigma \in part\text{-}out(\mathcal{A}, d, q)\} \cup part\text{-}out(\mathcal{A}, d, p)$$

$$part\text{-}out(\mathcal{A}, da, p) = \bigcup_{(q,a,p)\in\delta} out(\mathcal{A}, d, p).$$

With this notation and terminology, we are ready to discuss Preprocessing, which is shown in Algorithm 2. Preprocessing builds the mapping DAG G incrementally, by processing $d = a_1 \cdots a_n$ one letter at a time. During its execution, new nodes are added to G, but never deleted. Concretely, assume that $\mathcal{A} = (Q, q_0, F, \delta)$. For every i with $1 \le i \le n+1$ and every state $q \in Q$, Preprocessing computes sets of nodes R_q^i and C_q^i for Reading and Capturing, respectively. Indeed, recall that $d = a_1 a_2 \dots a_n = d(1, n+1)$. Then R_q^i will contain all nodes in G produced after reading $d(1, i)$, namely, $a_1 \dots a_{i-1}$. On the other hand, C_q^i will contain all nodes produced right after capturing a span ended at $d(1, i)$. Note that R_q^1 is used for $d(1, 1) = \varepsilon$ and C_q^1 is for adding the captures variables that start before reading the document.

More specifically, Preprocessing adds nodes to G such that (G, R_q^i) and (G, C_q^i) represent $part\text{-}out(\mathcal{A}, d(1, i), q)$ and $out(\mathcal{A}, d(1, i), q)$, respectively, in the following particular sense.

Definition 4.8. Let $G = (V, E, \perp, \lambda)$ be a mapping DAG and $U \subseteq V$ a set of its nodes. We say that (G, U) *represents* $out(\mathcal{A}, d(1, i), q)$ if $out(\mathcal{A}, d(1, i), q) = \{\lambda(\pi) \mid \pi \in \text{paths}(G, U)\}$ and similarly that (G, U) *represents* $part\text{-}out(\mathcal{A}, d(1, i), q)$ *if* $part\text{-}out(\mathcal{A}, d(1, i), q) = \{\lambda(\pi) \mid \pi \in \text{paths}(G, U)\}$.

In particular, R_q^i and C_q^i will be non-empty if, and only if, $part\text{-}out(\mathcal{A}, d(1, i), q)$ and $out(\mathcal{A}, d(1, i), q)$, respectively, are non-empty, which in turn, are non-empty if, and only if, $part\text{-}runs(\mathcal{A}, d(1, i), q)$ and $runs(\mathcal{A}, d(1, i), q)$ are non-empty, respectively. For every $1 \le i \le n+1$, the sets $\{C_q^i\}_{q\in Q}$ will be computed by a call to the procedure Capturing(i), while the sets $\{R_q^{i+1}\}_{q\in Q}$ for $1 \le i \le n$ are computed by a call to the procedure Reading$(i+1)$. Both procedures are further explained below.

At the beginning, the sets $\{R_q^1\}_{q\in Q}$ are initialized as follows: the mapping DAG G is initialized to contain only the sink $\perp$, and $R_{q_0}^1$ is initialized to contain this node and nothing else, while R_q^1 is initialized to empty, for every $q \neq q_0$ (lines 2–5 of Algorithm 2). This is consistent with the fact that there is only one partial run over $d(1,1) = \varepsilon$, namely the partial run that consists of the initial state q_0. Hence, after initialization *part-out*$(\mathcal{A}, d(1,1), q)$ is represented by (G, R_q) for every $q \in Q$.

Following this initialization, the procedures Capturing(i) and Reading$(i+1)$ alternate for every $i = 1, \ldots, n$. They operate as follows. When started, the procedure Capturing(i) simulates the extension of all partial runs on $d(1,i)$ to runs on $d(1,i)$ by traversing an extended variable transition (q, S, p) (with S possibly empty). To this end, it initializes C_q^i to contain all elements of R_q^i, for every $q \in Q$ (line 12). This simulates the case where $S = \emptyset$. It then iterates over all the sets R_q^i with $R_q^i \neq \emptyset$ (line 14) and, creates, for every extended variable transition $(q, S, p) \in \delta$ a new node v with the label (S, i), adds it to the mapping DAG, connects it to all nodes in R_q^i, and adds it to C_p^i (line 15–19). This simulates the case where $S \neq \emptyset$. The pairs (G, C_p^i) now correctly represent *out*$(\mathcal{A}, d(1,i), p)$ because of Lemma 4.7(1) and because the pairs (G, R_q^i) represent *part-out*$(\mathcal{A}, d(1,i), q)$ (by induction).

The procedure Reading$(i+1)$ simulates the extension of runs on $d(1,i)$ to partial runs on $d(1,i+1)$ by reading the letter a_i. To this end, Reading$(i+1)$ first initializes R_q^{i+1} to the empty set, for every $q \in Q$ and $1 \leq i \leq n$. It then proceeds by examining, for every q with $C_q^i \neq \emptyset$ the transition (q, a_i, p) (if it exists), and adding all elements of C_q^i to R_p^{i+1} (lines 23–25). Once this is done, the pairs (G, R_p^{i+1}) correctly represent *part-out*$(\mathcal{A}, d(1,i+1), p)$ because of the Lemma 4.7(2) and because the pairs (G, C_q^i) represent *out*$(\mathcal{A}, d(1,i), q)$ (by induction).

As a final step, Preprocessing also executes Capturing$(n+1)$ to ensure that *out*$(\mathcal{A}, d(1,n+1), q)$ is represented by (G, C_q^{n+1}), for every q.

Because $\mathcal{A}$ is sequential, every accepting run is valid. This implies that all elements of *out*$(\mathcal{A}, d, q)$ are valid mapping sequences, for every $q \in F$. In particular, $[\![\mathcal{A}]\!]_d$ is in one-to-one correspondence with the mapping sequences in $\bigcup_{q\in F}$ *out*$(\mathcal{A}, d, q)$, which is represented by $(G, \bigcup_{q\in F} C_q^{n+1})$ and returned by Preprocessing in line 10.

Next we give an example that illustrates the execution of Algorithm 2.

Example 4.9. *Consider the deterministic* seVA $\mathcal{A}$ *from Figure 4 and 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 Preprocessing *works, in Figure 8 we show the sets* R_q^i *and* C_q^i *that are computed, for* $q \in Q$ *and* $i = 1, 2, 3$*. For parsimony, we only show those sets that are non-empty. We also show the mapping DAG that is constructed in Figure 9. At the beginning, the only non-empty set is* $R_{q_0}^1 = \{\perp\}$*. When* Capturing(1) *is triggered, we create three new nodes,* v_1, v_2 *and* v_3*, each corresponding to the extended variable transitions leaving the state* q_0*, and add them to the mapping DAG. In Figure 9, the node* v_1 *corresponds to the transition* $(q_0, x\vdash, q_1)$*, the node* v_2 *to the transition* $(q_0, y\vdash, q_2)$*, and the node* v_3 *to the transition* $(q_0, \{x\vdash, y\vdash\}, q_3)$*. These nodes are also added to appropriate sets; namely,* $C_{q_1}^1 = \{v_1\}$*,* $C_{q_2}^1 = \{v_2\}$*, and* $C_{q_3}^1 = \{v_3\}$*. In* Reading(2) *we propagate the non-empty sets* C_q^1 *to the set* R_p^1*, whenever* (q, a, p) *is a letter transition of* $\mathcal{A}$*. For instance, since* $\mathcal{A}$ *can go from* q_1 *to* q_4 *while reading* $a_1 = a$*, the set* $R_{q_4}^1$ *will be equal to* $C_{q_1}^1$ *.*

Next, Capturing(2) *is executed. Here, the sets that were non-empty after* Reading(1) *will remain unchanged after* Capturing(2)*, simulating the situation where a partial run is extended to a run by taking a transition* $(q, \emptyset, q)$ *with an empty set of variable markers. Other extended variable transitions that can be triggered create new nodes and add them to the appropriate sets. For instance, the node* v_4

Stage	Non-empty sets
Initial	$R^1_{q_0} = \{ \bot \}$
Capturing(1)	$C^1_{q_0} = \{ \bot \}$
	$C^1_{q_1} = \{ v_1 \}$
	$C^1_{q_2} = \{ v_2 \}$
	$C^1_{q_3} = \{ v_3 \}$
Reading(2)	$R^2_{q_4} = C^1_{q_1} = \{ v_1 \}$
	$R^2_{q_5} = C^1_{q_2} = \{ v_2 \}$
	$R^2_{q_3} = C^1_{q_3} = \{ v_3 \}$
Capturing(2)	$C^2_{q_4} = R^2_{q_4} = \{ v_1 \}$
	$C^2_{q_5} = R^2_{q_5} = \{ v_2 \}$
	$C^2_{q_3} = R^2_{q_3} = \{ v_3 \}$
	$C^2_{q_6} = \{ v_4 \}$
	$C^2_{q_7} = \{ v_5 \}$
	$C^2_{q_9} = \{ v_6 \}$
Reading(3)	$R^3_{q_3} = C^2_{q_3} = \{ v_3 \}$
	$R^3_{q_8} = C^2_{q_6} \cup C^2_{q_7} = \{ v_4, v_5 \}$
Capturing(3)	$C^3_{q_3} = R^3_{q_3} = \{ v_3 \}$
	$C^3_{q_8} = R^3_{q_8} = \{ v_4, v_5 \}$
	$C^3_{q_9} = \{ v_7, v_8 \}$

Fig. 8. The non-empty sets R^i_q and S^i_q after executing each stage of Preprocessing on the automaton and document of Example.

is created because $(q_4, y\vdash, q_6)$ is an extended variable transition, and $R^1_{q_4}$ was non empty. Similarly, v_6 is created because $R^1_{q_3} \neq \emptyset$, and $(q_3, \{\dashv x, \dashv y\}, q_9)$ is an extended variable transition of $\mathcal{A}$.

Reading(3) again "propagates" the sets C^2_q according to what $\mathcal{A}$ does when reading $a_2 = b$. The set $C^2_{q_3}$ is simply copied into $R^3_{q_3}$ (simulating a self loop). A more interesting situation occurs when the transitions (q_6, b, q_8) and (q_7, b, q_8) are processed. Since they both reach q_8, we first copy the set $C^2_{q_6}$ into (the initially empty set) $R^3_{q_8}$, and then to keep track that one can also get to q_8 from q_7, we also add to $R^3_{q_8}$ all the elements of the set $C^2_{q_7}$. Since these are the only two ways that $\mathcal{A}$ can move while reading b, the other sets R^3_q remain empty.

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 create the new nodes v_7 and v_8 to be added to $C^3_{q_9}$. Note that $C^3_{q_3}$ and $C^3_{q_8}$ are also non-empty at this stage.

From our previous description of Algorithm 2 and Example 4.9, we hence conclude the following.

Proposition 4.10. *Given a* seVA $\mathcal{A}$ *and document d, Preprocessing returns a pair (G, U) consisting of a mapping DAG G and subset of its vertices U such that (G, U) represents $[\![\mathcal{A}]\!]_d$.*

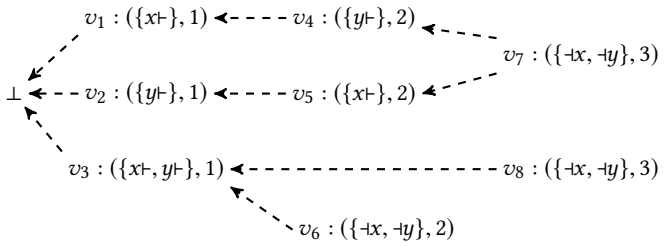

Fig. 9. The mapping DAG created by Preprocessing in Example 4.9 to record the output mappings.

4.5 Correctness of the preprocessing procedure

While the above proposition establishes that the result of Preprocessing$(\mathcal{A}, d)$ represents $[\![\mathcal{A}]\!]_d$, we also require that this result is without duplicates in order to prove the correctness of the preprocessing procedure. We dedicate this subsection to show that this is actually the case, provided that $\mathcal{A}$ is deterministic. We start by proving the following auxiliary lemma.

Lemma 4.11. *Let $\mathcal{A}$ be a deterministic* seVA *and $d = a_1 \cdots a_n$ be a document. While running the procedure Preprocessing$(\mathcal{A}, d)$ of Algorithm 2, for every $i \in \{1, \ldots, n+1\}$ and every two distinct states q_1 and q_2 of $\mathcal{A}$, it is the case that $R^i_{q_1} \cap R^i_{q_2} = \emptyset$ and $C^i_{q_1} \cap C^i_{q_2} = \emptyset$.*

Proof. We proceed by induction over i. For $i = 1$ we have $R^i_q = \emptyset$ for every $q \neq q_0$, and therefore $R^i_{q_1} \cap R^i_{q_2} = \emptyset$. The sets $C^i_{q_1}$ and $C^i_{q_2}$ are initialized as $R^i_{q_1}$ and $R^i_{q_2}$, respectively (Line 13), and therefore, they are initially disjoint. Afterwards, they can only be modified by adding fresh elements (Line 19), so they will always be disjoint.

For the inductive step, assume that the lemma holds for i. We show that it also holds for $i+1$. We first prove that $R^{i+1}_{q_1} \cap R^{i+1}_{q_2} = \emptyset$. Initially, both $R^{i+1}_{q_1}$ and $R^{i+1}_{q_2}$ are initialized as the empty set by Reading$(i+1)$, and thus they are disjoint. Afterwards, Reading$(i+1)$ adds to $R^{i+1}_{q_1}$ (resp. $R^{i+1}_{q_2}$) all the elements of all sets C^i_q such that $(q, a_i, q_1) \in \delta$ (resp. $(q, a_i, q_2) \in \delta$). Since $\mathcal{A}$ is deterministic and q_1, q_2 are distinct, there is no state q such that both (q, a_i, q_1) and (q, a_i, q_2). Therefore, there is no state q such that C^i_q is included in both $R^{i+1}_{q_1}$ and $R^{i+1}_{q_2}$. Then, since the C^i_q are disjoint from $C^i_{q'}$ for all $q' \neq q$, this implies that $R^i_{q_1} \cap R^i_{q_2} = \emptyset$. Using the fact $R^{i+1}_{q_1} \cap R^{i+1}_{q_2} = \emptyset$ we can now show that also $C^{i+1}_{q_1}$ and $C^{i+1}_{q_2}$ are disjoint by repeating the same reasoning as in the base case. □

It is important to note that we say that (G, U) is "without duplicates" when the enumeration procedure starting from U enumerates all outputs without duplicates. Actually, this property does not forbid the case that the enumeration of (G, U_1) and (G, U_2) produce duplicate outputs for two different sets U_1 and U_2. This motivates the following definition: we call two pairs (G, U_1) and (G, U_2) non-overlapping if for every $\pi_1 \in \text{paths}(G, U_1)$ and $\pi_2 \in \text{paths}(G, U_2)$ it holds that $\lambda(\pi_1) \neq \lambda(\pi_2)$.

The following lemma shows that after each reading and capturing call the sets R^i_q and C^i_q do not contain duplicates, respectively.

Lemma 4.12. *Let $\mathcal{A}$ be a deterministic* seVA *and $d = a_1 \ldots a_n$ be a document. While running Preprocessing$(\mathcal{A}, d)$ the following hold, for every $i \in \{1, \ldots, n+1\}$ and every state q:*

(R) (G, R^i_q) is without duplicates and (G, R^i_q) and $(G, R^i_{q'})$ are non-overlapping for every $q' \neq q$.

(C) (G, C^i_q) is without duplicates and (G, C^i_q) and $(G, C^i_{q'})$ are non-overlapping for every $q' \neq q$.

Proof. The proof is by induction on i, proving both statements (R) and (C) simultaneously for all q. For $i = 1$ we have $R^1_{q_0} = \{\perp\}$ and $R^1_p = \emptyset$, for $p \neq q_0$. Statement (R) then trivially follow. To

show statement (C), we use the fact that (R) already holds. Actually, the reasoning here is exactly the same as the one used in the inductive step below.

For the induction step assume that statements (R) and (C) hold for i; we also show that they hold for $i+1$. We first show that (R) holds and, specifically, that (G, R_q^{i+1}) is without duplicates. To that end, first observe that $R_q^{i+1} = \bigcup_{(p,a_i,q)\in\delta} C_p^i$, for every state q. Then (G, R_q^{i+1}) is without duplicates, for every state q, because all the (G, C_p^i) are pairwise non-overlapping and without duplicates by the induction hypothesis.

To show the second statement of (R), let $q' \neq q$ and consider two paths $\pi \in \text{paths}(G, R_q^{i+1})$ and $\pi' \in \text{paths}(G, R_{q'}^{i+1})$, respectively. Because $R_q^{i+1} = \bigcup_{(p,a_i,q)\in\delta} C_p^i$ and $R_{q'}^{i+1} = \bigcup_{(p',a_i,q')\in\delta} C_{p'}^i$ there exist some states p and p' such that also $\pi \in \text{paths}(G, C_p^i)$ and $\pi' \in \text{paths}(G, C_{p'}^i)$. We consider two cases. If $p \neq p'$ then $\lambda(\pi) \neq \lambda(\pi')$ since (G, C_p^i) and $(G, C_{p'}^i)$ are non-overlapping by the induction hypothesis. If $p = p'$ then we first observe that, since R_q^{i+1} is disjoint with $R_{q'}^{i+1}$ by Lemma 4.11, the first node of π must be distinct from the first node of π'. Hence π and π' are distinct elements of $\text{paths}(G, C_p^i) = \text{paths}(G, C_{p'}^i)$. It follows that $\lambda(\pi) \neq \lambda(\pi')$ since (G, C_p^i) is without duplicates by the induction hypothesis.

Let us next show that statement (C) holds. For this, we use the fact that we have already shown (R). For the sake of simplification, we show the two statements inside C simultaneously (i.e. the arguments are very similar). Specifically, fix states q and q' (not necessarily different) and consider two distinct paths $\pi \in \text{paths}(G, C_q^{i+1})$ and $\pi' \in \text{paths}(G, C_{q'}^{i+1})$. Let v be the first node of π and v' the first node of π'. We make the following case analysis (highlighting the distinction of the argument when $q = q'$ or $q \neq q'$):

- if $v \in R_q^{i+1}$ and $v' \in R_{q'}^{i+1}$, then $\lambda(\pi_1) \neq \lambda(\pi_2)$ by (R) because (G, R_q^{i+1}) is without duplicates when $q = q'$ or (G, R_q^{i+1}) and $(G, R_{q'}^{i+1})$ do not overlap when $q \neq q'$.
- if $v \in R_q^{i+1}$ but $v' \notin R_{q'}^{i+1}$, then since $v' \in C_{q'}^{i+1}$ and since $C_{q'}^{i+1}$ is computed by Capturing$(i+1)$, it follows that v' was created by Capturing$(i+1)$ in line 17. In particular, v' is labeled by $(S, i+1)$ for some non-empty set S. It is straightforward to observe, by induction on i that all nodes in R_q^{i+1} either do not have a label (in case of $\perp$) or have a label of the form (S', j) with $j < i+1$. Therefore, the labels of v and v' are distinct, and as such $\lambda(\pi_1) \neq \lambda(\pi_2)$, as desired. Note that the previous argument holds regardless whether $q = q'$ or $q \neq q'$.
- The case where $v \notin R_q^{i+1}$ but $v' \in R_{q'}^{i+1}$ is similar.
- $v \notin R_q^{i+1}$ and $v' \notin R_{q'}^{i+1}$. Then, since $v \in C_q^{i+1}$ and $v' \in C_{q'}^{i+1}$, v and v' are both new nodes created by Capturing$(i+1)$ in line 17. In particular, there exist states p and p' and transitions $(p, S, q) \in \delta$ and $(p', S', q') \in \delta$ such that v has label $(S, i+1)$, v' has label $(S', i+1)$. Clearly, if $S \neq S'$ then we are done. Otherwise, let τ and τ' be the paths obtained by removing v and v' from π and π', respectively. In particular, because of the way that Capturing$(i+1)$ connects newly created nodes to the already existing nodes in lines 16–19, we know that $\tau \in \text{paths}(G, R_p^{i+1})$ and $\tau' \in \text{paths}(G, R_{p'}^{i+1})$. We consider two further cases.
  - if $p \neq p'$, then $\lambda(\tau) \neq \lambda(\tau')$ since (G, R_p^{i+1}) and $(G, R_{p'}^{i+1})$ do not overlap by (R). As such, also $\lambda(\pi) \neq \lambda(\pi')$.
  - if $p = p'$, then $S = S'$ and we hence have $(p, S, q) \in \delta$ and $(p, S, q') \in \delta$ which implies that $q = q'$ given that $\mathcal{A}$ is deterministic. Then both τ and τ' belong to $\text{paths}(G, R_p^{i+1}) = \text{paths}(G, R_{p'}^{i+1})$. Since (G, R_p^{i+1}) is without duplicates, we know that $\lambda(\tau) \neq \lambda(\tau')$ and hence also $\lambda(\pi) \neq \lambda(\pi')$.

Given that in all cases we conclude that $\lambda(\pi) \neq \lambda(\pi')$, the statement (C) holds as well. □

From the previous two lemmas, Lemma 4.11 and Lemma 4.12, we conclude the correctness of the Preprocessing algorithm.

Corollary 4.13. *The pair* (G, U) *returned by* Preprocessing$(\mathcal{A}, d)$ *is without duplicates, for every document* d *and every deterministic* seVA $\mathcal{A}$.

4.6 Implementation and complexity

From Proposition 4.10 and Corollary 4.13 we know that Preprocessing, shown in Algorithm 2, computes a pair (G, U) consisting of a mapping DAG G and subset U of its vertices such that (G, U) represents $[\![\mathcal{A}]\!]_d$ without duplicates. From Section 4.4 and Proposition 4.4 we know that we can hence enumerate $[\![\mathcal{A}]\!]_d$ with output-linear delay by running Enumerate as shown in Algorithm 1. To obtain Theorem 1.1 it hence suffices to show that Preprocessing can be implemented to run in the desired complexity of $O(\|\mathcal{A}\| \times \|d\|)$. In this section, we therefore present an implementation of Preprocessing that uses concrete data structures to represent sets and graphs, and achieves the desired time complexity. Concretely, we prove that, with these data structures, a single call to the procedures Capturing(i) and Reading(i) of Algorithm 2 takes time linear in the size of the automaton, for every $1 \leq i \leq n + 1$.

From the description of Algorithm 2 we can see that Capturing and Reading iterate over the states and transitions of the automaton and, for each of them, perform variable assignments, graph operations and set operations. We first explain how the required set operations can be performed in constant time, and then explain how this can be used for implementing graph operations in constant time. The set operations consist of initializing an empty set (Line 22 of Algorithm 2), copying a set (Line 13) and taking the union of two sets (lines 19 and 25). These operations cannot be carried out in constant time in general, but we will show that some invariants of our algorithm allow for a particular representation of sets that indeed allow more efficient manipulation.

Concretely, we represent a set as a pair of two pointers that point to the first and last elements of a reverse linked list. This means that a set $S = \{v_1, \ldots, v_n\}$ is represented by a pair of pointers (a, b), where a points to v_1, b points to v_n, and for each $i \in \{2, \ldots, n\}$ there is a pointer from v_i to v_{i-1} denoted $v_i.prev$. An empty set is represented by two null pointers, and is denoted in Algorithm 3 by ϵ. Note that this representation by means of pairs of pointers allows us to iterate over a set in reverse order with constant delay by simply starting at the end of the set and following the reverse linked list. This is, of course, asuming that there are no repeated elements in the linked list, which is an additional requirement of our set representation. It is easy to see that our representation of sets allows for the desired set operations in constant time:

(1) Set union: Consider two sets $S_1 = \{v_1, \ldots, v_n\}$ and $S_2 = \{u_1, \ldots, u_m\}$, represented by (a_1, b_1) and (a_2, b_2), respectively. First we define $u_1.prev = v_n$, and then represent $S_1 \cup S_2$ by a new pair (a, b), where $a = a_1$ and $b = b_2$ (i.e., a points to v_1 and b points to u_m).
(2) Set copy: A set $S = \{v_1, \ldots, v_n\}$ represented by (a, b) is simply copied by defining a new pair of pointers (a', b'), where $a' = a$ and $b' = b$.

However, performing the operations defined above might generate inconsistencies. In particular, if we take the union between two different sets that are not disjoint, the representation will contain repeated elements (and the resulting linked list could contain loops). Moreover, when assigning $u_1.prev = v_n$ we could be overwriting the previous value of $u_1.prev$, corrupting some previously defined set. Thanks to Lemma 4.11 and Lemma 4.12 the following two invariants are always satisfied:

- We always take the union of disjoint sets, and
- Whenever we take the union of two sets, at least one of them has a first element without a defined *prev* pointer.

Note that Lemma 4.11 and Lemma 4.12 imply that in Algorithm 2 we are always taking the union of disjoint sets. Moreover, since every set C_q^i is only used at most in a single set union, if we use the representation of sets described above we can always modify the *prev* pointer of its first element. This implies the following corollary.

Corollary 4.14. *By encoding every set as a pair of pointers to the first and last element of a reversed linked list, which contains the elements of the set without repetitions, Algorithm 2 can be implemented in such a way that every set operation takes constant time.*

Taking this into account, we can now also implement graph operations used in Algorithm 2 in constant time. More precisely, we need to show that node creation (line 18 in Algorithm 2) and adding new edges (line 18) can be carried out in constant time. For this, we will represent the mapping DAG as a set of nodes, where each node has a label (in form of a set of variable markers and a position where they are invoked), and a set of neighbours of this node. This is analogous to the adjacency list representation of a graph, so the edges are encoded in the neighbour set of each node. With this representation creating a node is trivially done in constant time, as it only includes assigning a label to a node, and adding it to the set of nodes of our graph. On the other hand, taking into consideration that the newly created node is always connected to some set R_q^i (line 18), and the fact that we can represent the set R_q^i with a pair of pointers, the neighbour set of this node can be represented by copying the pointers to the beginning and the end of the list representing R_q^i, and is thus carried out in constant time.

With these observations it is now easy to obtain the desired complexity: as discussed before, for every symbol of the input document, Algorithm 2 executes the procedures Capturing and Reading. The procedure Capturing first copies a set for each state (Line 13), and then performs at most four operations (lines 16 to 19) for each transition of $\mathcal{A}$. The procedure Reading first initializes one empty set per each state (Line 22) and then performs at most two operations per state (lines 24 to 25). By the previous discussion and Corollary 4.14, all of these per-state operations can be performed in constant time, giving us the expected complexity. Finally, Preprocessing takes the union of the sets C_q^{n+1}, for every $q \in F$ with F the set of final states (line 10). This takes time $O(|F|)$ by the same reasoning. Note that the return statement itself just involves copying the pointer to G and this union, which is also constant time. We hence obtain:

Theorem 4.15. *Let $\mathcal{A}$ be a sequential deterministic* eVA *and $d = a_1 \cdots a_n$ be a document. The procedure Preprocessing of Algorithm 2 can be implemented in time $O(\|\mathcal{A}\| \times \|d\|)$.*

This theorem plus the fact that from the graph constructed by the algorithm allows for output-linear delay enumeration already give us the desired result. However, we aim at further closing the gap between theory and practice, and therefore work further in optimizing our algorithm in practice while keeping the implementation simple. Therefore, we present another version of the algorithm that uses the previously mentioned representation of sets and graphs. Moreover, to optimize space we avoid defining a list for each set R_q^i and C_q^i. Instead, we reuse the variables by keeping only two lists per state. This more low-level version of the algorithm is presented in Algorithm 3. Here, instead of sets we use reversed linked lists and, as explained before, we use two methods to represent set union and set copy by means of the methods `extend` and `copyByReference`, respectively, as defined in points (1) and (2). Creating a node and defining its set of neighbours is implemented using the method `Node`, which takes the label (S, i) assigned to the node, and the list representing the set of its neighbours given by the two pointers to its beginning and end node, and copies these two pointers to represent the neighbours of the node. In Figure 10 we illustrate the data structure created by Algorithm 3, which can be compared to the DAG presented in Figure 9.

Algorithm 3 Evaluate $\mathcal{A}$ over the document $a_1 \dots a_n$

```
1:  procedure Evaluate(A, a_1 ... a_n)
2:      for all q ∈ Q \ {q_0} do
3:          list_q ← ϵ
4:      list_{q_0} ← [⊥]
5:      for i := 1 to n do
6:          Capturing(i)
7:          Reading(i + 1)
8:      Capturing(n + 1)
9:      resultList ← ϵ
10:     for all q ∈ F with list_q ≠ ϵ do
11:         resultList.extend(list_q)
12:     return (G, resultList)

13: procedure Capturing(i)
14:     for all q ∈ Q do
15:         list_q^old ← list_q.copyByReference
16:     for all q ∈ Q with list_q^old ≠ ϵ do
17:         for all S ∈ Markers_δ(q) do
18:             node ← Node((S, i), list_q^old)
19:             p ← δ(q, S)
20:             list_p.extend(node)

21: procedure Reading(i + 1)
22:     for all q ∈ Q do
23:         list_q^old ← list_q
24:         list_q ← ϵ
25:     for all q ∈ Q with list_q^old ≠ ϵ do
26:         p ← δ(q, a_i)
27:         list_p.extend(list_q^old)
```

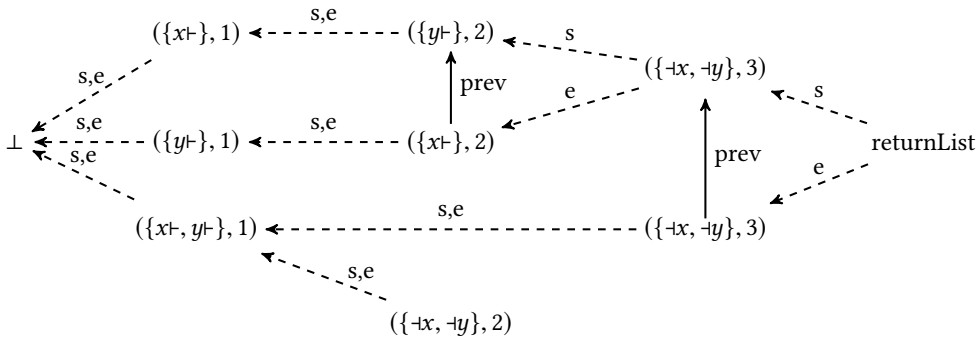

Fig. 10. DAG created by Algorithm 3 to record the output mappings, and the content of *returnList*. Dashed edges represent the start/end pointer for a list (denoted by s and e, respectively).

5 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 output-linear delay enumeration after $O(\|\mathcal{A}\| \times \|d\|)$ preprocessing. 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 output-linear delay algorithm for evaluating regular spanners.

Because it is well-known that RGX formulas can be translated into VA in linear time [12], 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 [15]. 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 5.1. *For any* VA $\mathcal{A}$ *there is a deterministic* seVA $\mathcal{A}'$ *with at most* $2^n 3^\ell$ *states and* $2^n 6^\ell + 2^n 3^\ell |\Sigma|$ *transitions such that* $\mathcal{A}' \equiv \mathcal{A}$.

Proof. Given an arbitrary VA $\mathcal{A} = (Q, q_0, F, \delta)$ 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 $2^n 3^\ell$ states, $2^n 6^\ell + 2^n 3^\ell |\Sigma|$ transitions and ℓ variables. Let us first describe the set Q' of states of $\mathcal{A}'$. Intuitively, every state will correspond to a pair $(\{q_1, \ldots, q_k\}, S)$, where $q_1, \ldots, q_k \in Q$ are the states reached by $\mathcal{A}$ after reading exactly all (and only) the variable markers in the set of variable markers S. Since there are n states, the first component (the set of reached states) can be chosen from 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 $\sum_{i=1}^{\ell} \binom{\ell}{i} 2^i$, where i represents the number of opened variables, $\binom{\ell}{i}$ the different ways of choosing those i variables, and 2^i possible ways of closing (or leaving open) those i variables. From this we obtain:

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

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

Let us now define the set of transitions δ'. Given a symbol $c \in \Sigma$, 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. As defined in the proof of Theorem 3.1, a variable path in $\mathcal{A}$ is a sequence of transitions $\pi : p = p_0 \xrightarrow{v_1} p_1 \xrightarrow{v_2} \ldots \xrightarrow{v_h} p_h = q$ such that $(p_i, v_{i+1}, p_{i+1}) \in \delta$ are variable transitions and $v_i \neq v_j$ for every $i \neq j$. If $S = \text{Markers}(\pi) = \{v_1, \ldots, v_h\}$ we say that π is an S-path from p to q. Then, for every $P \subset Q$ and every pair (S, S') of variable markers that are valid (i.e. all the closed variables is S and S' were also opened), and 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, $\delta'((\{q_0, \ldots, q_k\}, S), S')$ 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 number of variable transitions, for each $i \in \{0, \ldots, \ell\}$ consider a state (P, S) where S uses exactly i variables. The number such states is $2^n \binom{\ell}{i} 2^i$, since any subset of Q can be in place of P, and a set of markers S using precisely i variables is specified by which i variables we choose (the $\binom{\ell}{i}$ factor), and whether each one of them is only opened, or both opened and closed. The number of transitions leaving such a state (P, S) is defined by a set of variable markers S' telling us what to do with the remaining

$\ell - i$ variables (if anything), as well as whether we will close some of the (at most) i variables in S that remained opened. This means that the number of different sets S' of markers which use precisely $0 \leq j \leq \ell$ is bounded by $\binom{\ell-i}{j}2^j 2^i$, since we have to choose which j of the remaining $\ell - i$ variables we use, whether they are only opened, or both opened and closed (the 2^j factor), and whether we choose to close any of the (at most) i variables opened in S. Notice that the 2^i factor can be replaced with $2^{\ell-j}$, given that $i + j \leq \ell$, thus making the upper bound on the number of sets S' which use precisely j new variables $\binom{\ell-i}{j}2^j 2^{\ell-j}$. With this in hand, we can bound the number of variable transitions in $\mathcal{A}'$ as follows:

$$\sum_{i=0}^{\ell}\left[2^n\binom{\ell}{i}2^i\sum_{j=0}^{\ell-i}\binom{\ell-i}{j}2^j 2^{\ell-j}\right] \;=\; 2^n\sum_{i=0}^{\ell}\binom{\ell}{i}2^i 4^{\ell-i} \;=\; 2^n(2+4)^\ell \;=\; 2^n 6^\ell.$$

This gives us the number of variable transitions in $\mathcal{A}'$. To this number, we must add the number of symbol transitions, which is at most one transition per state per symbol, i.e. $2^n 3^\ell |\Sigma|$. Then, the total number of transitions is $2^n 6^\ell + 2^n 3^\ell |\Sigma|$. Finally, we define the set F' of final states as those states (P, S) in which $P \cap F \neq \emptyset$ and all variables opened in S are also closed.

It is trivial to see that $\mathcal{A}'$ is sequential. As 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:

$$f(i) = \begin{cases} k & \text{if } o_i \text{ is a variable marker; } i \leq k; o_j \text{ is a variable marker for every } i \leq j \leq k \\ & \text{and either } k = m \text{ or } o_{k+1} \text{ is not a variable marker} \\ (o_i, \emptyset) & \text{if } o_i \in \Sigma \text{ and } o_{i+1} \in \Sigma \\ o_i & \text{otherwise.} \end{cases}$$

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 $f(i) = (o_i, \emptyset)$, update ρ' to $\rho' \xrightarrow{o_i} \delta'(P', S') \xrightarrow{\emptyset} \delta'((P', S'), \emptyset)$, 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 $S \neq \emptyset$). The $\emptyset$ and character transitions immediately yield valid transitions for ρ'. The fact that ρ' is valid 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 accepting 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 $(S' \setminus S)$-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. □

Therefore, evaluating an arbitrary VA with output-linear delay can be done with preprocessing that is exponential 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, one could argue that the exponential blow-up depending on the number of variable comes from the extended restriction. It is then natural to ask whether there exists a subclass of VA where the blow-up in the size of the automata can be avoided when a VA is translated to its extended version (not necessarily deterministic).

Two subclasses of VA that are known to have good algorithmic properties [17, 22] are sequential VA and functional VA. Next, we will consider if the cost of translation to extended VA is smaller in these cases. In the more general case of sequential VA we can actually show that a blow-up in the size of the automaton 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 11. 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 variable transitions between the initial and the final state, one for each possible set of variables. Formally, we have the following proposition.

Proposition 5.2. *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.*

Proof. For every ℓ consider the VA $\mathcal{A}$ with $3\ell + 2$ states, $4\ell + 1$ transitions and 2ℓ variables $(x_1, \ldots, x_\ell, y_1, \ldots, y_\ell)$ depicted in Figure 11. $\mathcal{A}$ only produces valid runs for the document $d = a$ and the resulting mapping is always valid but never total. The reason is that the automaton properly opens and closes variables, but at each intermediate state the run has the option to choose opening and closing either x_i or y_i, for every $1 \leq i \leq \ell$, 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 12. 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 these transitions has a corresponding and different ϵ mapping, the one where the contained variables are defined. Therefore, it has 2^ℓ transitions, as well as any other equivalent eVA. □

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 an equivalent deterministic eVA. To show this, we first determine the cost of transforming a functional VA into an extended VA which is not necessarily deterministic.

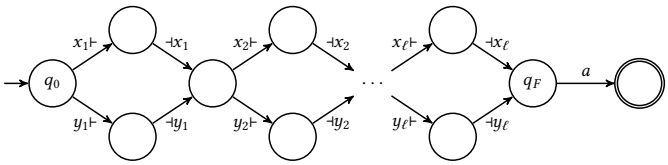

Fig. 11. A sequential VA with 2ℓ variables such that every equivalent eVA has $O(2^\ell)$ transitions.

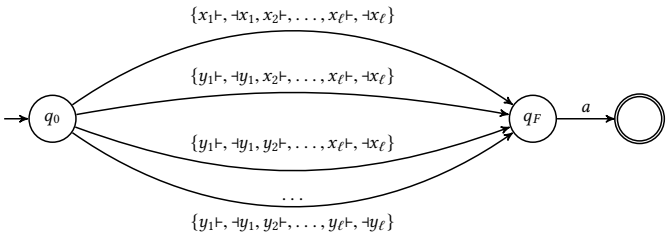

Fig. 12. The smallest eVA $\mathcal{A}'$ equivalent to $\mathcal{A}$ with 2^ℓ transitions.

Proposition 5.3. *For any functional* VA $\mathcal{A}$ *there exists an equivalent functional* eVA $\mathcal{A}'$ *with at most* n *states and* $(m + n^2)$ *transitions.*

For converting $\mathcal{A}$ into a functional eVA $\mathcal{A}_{\text{ext}}$ we require the following lemma[6] (the notion of variable-path is formally defined in the proof of Theorem 3.1).

Lemma 5.4. *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. Suppose, for the purpose of contradiction, that there are two states q and q' in $\mathcal{A}$ such that there 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. □

Proof of Proposition 5.3. The proof of Theorem 3.1 shows how to construct, given a functional VA $\mathcal{A}$, an equivalent functional eVA $\mathcal{A}_{\text{ext}}$. We next show that if $\mathcal{A}$ has n states and m transitions, then by this construction, $\mathcal{A}_{\text{ext}}$ 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: because in the construction of $\mathcal{A}_{\text{ext}}$ (cf. the proof of Theorem 3.1) we add an extended variable transition S between states p and q only if there is a variable path π between p and q in $\mathcal{A}$ such that $S = Markers(\pi)$ we obtain by Lemma 5.4 that in $\mathcal{A}_{\text{ext}}$ we can have at most one extended variable transition per pair of states (p, q). Therefore, in addition to the m transitions in $\mathcal{A}$, at most n^2 extended variable

[6] A similar lemma appears in [17].

transitions can be added to $\mathcal{A}_{\text{ext}}$. We conclude that $\mathcal{A}_{\text{ext}}$ has at most $m + n^2$ transitions and that at most n^2 of these transitions can be extended variable transitions. □

From this, we can determine the size of an extended VA that is equivalent to a functional VA.

Corollary 5.5. *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.*

Proof. From Proposition 5.3, we know that from a functiona VA $\mathcal{A}$ we can construct an equivalent functional and extended VA $\mathcal{A}_{\text{ext}}$ that has n states and at most $m + n^2$ transitions, where n^2 of these can be extended variable transitions. As shown in Proposition 3.2, deterministic seVA $\mathcal{A}'$ can be constructed from $\mathcal{A}_{\text{ext}}$ such that $\mathcal{A}_{\text{ext}} \equiv \mathcal{A}'$, where, $\mathcal{A}'$ has at most 2^n states. Now, note that since $\mathcal{A}'$ is deterministic, the number of transitions per state in $\mathcal{A}'$ can be at most the number of symbols in Σ plus the number of distinct variable sets S with (q, S, q') an extended variable transition in $\mathcal{A}_{\text{ext}}$. Since the latter is at most n^2, the number of transitions for $\mathcal{A}'$ is at most $2^n(n^2 + |\Sigma|)$. □

Because of Proposition 5.3 and Corollary 5.5, and the fact that functional VA are probably the class of VA most studied in the literature [12, 15, 17], we will work exclusively with functional VA for the remainder of this section.

Now we proceed to study how to apply the algebraic operators to evaluate regular spanners. In [12], 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 [12]). On the other hand, as shown by a subset of the author's in their previous work [24], and independently in [17], this blow-up can be avoided when working with functional VA. In the next proposition, we generalize this result to extended VA.

Proposition 5.6. *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* $\mathcal{A}_{\bowtie}$ *has* $n_1 \cdot n_2$ *states and at most* $m_1 \cdot m_2$ *transitions,*
- $\mathcal{A}_{\cup} \equiv \mathcal{A}_3 \cup \mathcal{A}_4$*, and* $\mathcal{A}_{\cup}$ *has* $(n_3 + n_4 + 1)$ *states and at most* $2 \cdot (m_3 + m_4)$ *transitions,*
- $\mathcal{A}_{\pi} \equiv \pi_Y(\mathcal{A}_1)$*, and* $\mathcal{A}_{\pi}$ *has* n_1 *states and at most* m_1^2 *transitions,*

where n_i *and* m_i *are the number of states and transitions of* $\mathcal{A}_i$*, respectively.*

Proof. Let $\mathcal{A}_1 = (Q_1, q_0^1, F_1, \delta_1)$, $\mathcal{A}_2 = (Q_2, q_0^2, F_2, \delta_2)$ and $Y \subset \mathcal{V}$. First, we construct the join of 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: 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 $n_1 \cdot n_2$ states and at most $m_1 \cdot m_2$ transitions.

Second, we tackle the projection of automata. To prove this, 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 $U = \text{Markers}_{\mathcal{V}} \setminus \text{Markers}_Y$ be markers for unprojected variables, then $\mathcal{A}_\pi = (Q_1, q_0^1, F_1, \delta')$ where $(q, a, p) \in \delta'$ whenever $(q, a, p) \in \delta_1$ for every $a \in \Sigma$, $(q, S \setminus U, p) \in \delta'$ whenever $(q, S, p) \in \delta_1$ and $S \setminus U \neq \emptyset$, and $(q, \epsilon, p) \in \delta'$ whenever $(q, S, p) \in \delta_1$ and $S \setminus U = \emptyset$.

The equivalence between $\mathcal{A}_1$ and $\mathcal{A}_\pi$ is straightforward. For every $\mu \in [\![\mathcal{A}_1]\!]_d$, there exists an accepting and valid run ρ in $\mathcal{A}_1$ 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 $[\![\pi_Y(\mathcal{A}_1)]\!]_d \subseteq [\![\mathcal{A}_\pi]\!]_d$. The other direction, $[\![\mathcal{A}_\pi]\!]_d \subseteq [\![\pi_Y(\mathcal{A}_1)]\!]_d$, 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.

It is important to note that, as for classical NFAs, from $\mathcal{A}_\pi$ an equivalent ϵ-transition free eVA $\mathcal{A}_\pi^{\epsilon\text{-free}}$ can be constructed using ϵ-closure over states. The ϵ-closure can produce a quadratic blow-up in the number of transitions. Given that $\mathcal{A}_\pi$ is of size linear in $\mathcal{A}_1$, then after removing the ϵ-transitions $\mathcal{A}_\pi^{\epsilon\text{-free}}$ will have n_1 states and at most m_1^2 transitions.

Finally, we construct the union of automaton. This construction is the standard disjoint union of automaton, with ϵ-transitions to each corresponding initial state. Let $\mathcal{A}_3 = (Q_3, q_0^3, F_3, \delta_3)$ and $\mathcal{A}_4 = (Q_4, q_0^4, F_4, \delta_4)$ be two feVA such that $\text{var}(\mathcal{A}_3) = \text{var}(\mathcal{A}_4)$ and $Q_3 \cap Q_4 = \emptyset$. Then, $\mathcal{A}_\cup = (Q_3 \cup Q_4, q_0, F_3 \cup F_4, \delta_3 \cup \delta_4 \cup \{(q_0, \epsilon, q_0^3), (q_0, \epsilon, q_0^4)\})$ where q_0 is a fresh new state. This simply adds a new

initial state connected with ϵ-transitions to the initial states of $\mathcal{A}_3$ and $\mathcal{A}_4$, respectively. Therefore every run in $\mathcal{A}_\cup$ must produce a run from $\mathcal{A}_3 \cup \mathcal{A}_4$ and vice versa. An equivalent ϵ-transition free automaton $\mathcal{A}_\cup^{\epsilon\text{-free}}$ can be constructed as in the projection case. Contrary to the projection, removing the transitions (q_0, ϵ, q_0^3) and (q_0, ϵ, q_0^4) from $\mathcal{A}_\cup$ adds at most $(m_3 + m_4)$ transitions to $\mathcal{A}_\cup^{\epsilon\text{-free}}$. Thus $\mathcal{A}_\cup^{\epsilon\text{-free}}$ will have $(n_3 + n_4 + 1)$ states and at most $2 \cdot (m_3 + m_4)$ transitions. □

It is important to mention that Proposition 5.6 differs from the corresponding proposition in the conference version of this article [14]. There, we incorrectly stated that $\mathcal{A}_\pi$ is of size linear in $|\mathcal{A}_1|$. In Proposition 5.6, in contrast, we prove that the size of $\mathcal{A}_\pi$ is in fact in $O(|\mathcal{A}_1|^2)$ and we do not know whether one can always construct an equivalent eVA of size linear in $|\mathcal{A}_1|$.

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 4 to enumerate $[\![\gamma]\!]_d$ with output-linear delay, for an arbitrary document d. More precisely, we have the following.

Proposition 5.7. *Let γ be a regular spanner in* $\mathrm{VA}^{\{\pi,\cup,\bowtie\}}$ *using k algebraic operations, and at most $k+1$ functional* VAs *as input, each of them with at most n states. Then there exists an equivalent deterministic* seVA $\mathcal{A}_\gamma$ *with at most $2^{n^{k+1}}$ states, and at most $2^{n^{k+1}} \cdot (n^{2(k+1)} + |\Sigma|)$ transitions.*

Proof. By Proposition 5.6, we know that we can construct the join between two automata with r and s states such that the resulting automaton will have at most $r \times s$ states. By the same proposition, automata for projections and unions can be obtained whose number of states are linear in the number of states of the input automata. Therefore, if we apply the transformations of Proposition 5.6 in a bottom-up fashion to γ, it is trivial to prove by induction that the final resulting automaton will be functional and will have at most n^{k+1} states. By Corollary 5.5, we can subsequently determinize this automaton, resulting in an equivalent deterministic seVA with $2^{n^{k+1}}$ states and $2^{n^{k+1}}(n^{2(k+1)} + |\Sigma|)$ transitions, concluding the proof. □

In this case the 2^n factor from Corollary 5.5 turns to $2^{n^{k+1}}$, 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 $2^{n^{k+1}}$ 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 $\mathrm{VA}^{\{\cup,\bowtie\}}$, we can obtain the following.

Proposition 5.8. *Let γ be a regular spanner in* $\mathrm{VA}^{\{\cup,\bowtie\}}$ *using k algebraic operations, and at most $k+1$ functional* VAs *as input, each of them with at most n states. Then, there exists an equivalent deterministic* seVA $\mathcal{A}_\gamma$ *with at most $2^{n\cdot(k+1)}$ states and at most $2^{n\cdot(k+1)} \cdot (n^{2(k+1)} + |\Sigma|)$ transitions.*

Proof. 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 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 $O(2^{(k+1)n})$ where $k+1$ is the number of functional eVAs in the expression.

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be the determinization of two functional eVA with n states. By Corollary 5.5 each automaton will have at most 2^n states and at most $2^n(n^2 + |\Sigma|)$ transitions. Indeed, from the proof of Corollary 5.5 one can notice that at most $2^n n^2$ transitions are variable transitions and at most $2^n|\Sigma|$ are letter transitions. Consider now the construction of $\mathcal{A}_\bowtie \equiv \mathcal{A}_1 \bowtie \mathcal{A}_2$ from Proposition 5.6. One can easily check that the construction preserves determinism, namely, $\mathcal{A}_\bowtie$ is deterministic, and the number of states is at most 2^{2n}. Furthermore, the number of variable transitions $\mathcal{A}_\bowtie$ is at

most $2^{2n}n^{2+2}$ and the number of letter transitions is at most $2^{2n}|\Sigma|$ (given that the automata are deterministic, the number of letters per state does not increase).

Unfortunately, the linear construction of the union of two functional eVA does not preserve the deterministic property of the input automaton. Instead, we can define the union of two deterministic functional eVA by taking the product and accept if either of the two automata accept. Formally, 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 deterministic functional eVA such $Q_1 \cap Q_2 = \emptyset$. Without loss of generality, we can assume that $\mathcal{A}_1$ and $\mathcal{A}_2$ contain sink states s_1 and s_2, respectively. Then, from s_1 and s_2 one cannot reach final states of $\mathcal{A}_1$ and $\mathcal{A}_2$. Define now $\mathcal{A}_\cup = (Q, (q_0^1, q_0^2), F, \delta)$ such that $Q = Q_1 \times Q_2$, $F = F_1 \times Q_2 \cup Q_1 \times F_2$, and δ satisfies that:

- $\big((p_1, p_2), o, (q_1, q_2)\big) \in \delta$ whenever $(p_1, o, q_1) \in \delta_1$, and $(p_2, o, q_2) \in \delta_2$,
- $\big((p_1, p_2), o, (q_1, s_2)\big) \in \delta$ 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
- $\big((p_1, p_2), o, (s_1, q_2)\big) \in \delta$ whenever $(p_2, o, q_2) \in \delta_2$, and $(p_1, o, q_1) \notin \delta_1$ for every $q_1 \in Q_1$.

It is straightforward to show $\mathcal{A}_\cup$ is functional, deterministic, and $[\![\mathcal{A}_\cup]\!]_d = [\![\mathcal{A}_1]\!]_d \cup [\![\mathcal{A}_2]\!]_d$. Regarding the size of $[\![\mathcal{A}_\cup]\!]_d$, if each automaton has 2^n states (i.e. after the determinization procedure), then the union automaton will have at most 2^{2n} states. Furthermore, if each automaton has at most $2^n n^2$ variable transitions and at most $2^n|\Sigma|$ letter transitions, then $\mathcal{A}_\cup$ will have at most $2^{2n}n^{2+2}$ variable transitions and at most $2^{2n} \cdot |\Sigma|$ letter transitions (i.e. the number of letter transitions per state does not increase given that $\mathcal{A}_\cup$ is deterministic).

Finally, it is easy to show by induction that the determinization, join and union of $k+1$ functional eVA will have at most $2^{n(k+1)}$ states and at most $2^{n(k+1)}(n^{2(k+1)} + |\Sigma|)$ 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 Preprocessing on the fly, thus rarely needing to materialize the entire deterministic seVA.

Given the previous results, we can convert an arbitrary regular spanner into a deterministic seVA and then applying Preprocessing followed by Enumerate to get an algorithm with output-linear delay enumeration after linear time preprocessing (in the size of the newly constructed automaton). Table 1 summarizes the total cost of the precomputation phase when using this approach.

6 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 [25] and can give some evidence on the limitations of finding bounded 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 enumeration algorithms with output-linear delay can be extended to count the number of outputs efficiently [25]. We show that this is the case for our algorithm over deterministic seVA.

Theorem 6.1. *Given a deterministic sequential extended* VA $\mathcal{A}$ *and a document* d, $|[\![\mathcal{A}]\!]_d|$ *can be computed in time* $O(\|\mathcal{A}\| \times \|d\|)$.

Algorithm 4 Count the number of mappings in $[\![\mathcal{A}]\!]_d$ over the document $d = a_1 \dots a_n$

```
1:  function Count(A, a_1 ... a_n)
2:      for all q ∈ Q \ {q_0} do
3:          N[q] ← 0
4:      N[q_0] ← 1
5:      for i := 1 to n do
6:          Capturing(i)
7:          Reading(i + 1)
8:      Capturing(n + 1)
9:      return ∑_{q∈F} N[q]
10: procedure Capturing(i)
11:     N' ← N
12:     for all q ∈ Q with N'[q] > 0 do
13:         for all S ∈ Markers_δ(q) do
14:             p ← δ(q, S)
15:             N[p] ← N[p] + N'[q]
16: procedure Reading(i + 1)
17:     N' ← N
18:     N ← 0
19:     for all q ∈ Q with N'[q] > 0 do
20:         p ← δ(q, a_i)
21:         N[p] ← N[p] + N'[q]
```

Proof. The Count function in Algorithm 4 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 Preprocessing as shown in Algorithm 3 in Section 4. 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 accepting run encodes a mapping) and deterministic (i.e. each run or partial run yields a distinct *out*-sequence), 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.

As we said, Algorithm 4 is very similar to Preprocessing. At the beginning (i.e. lines 2-4), the array N is initialized 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($i + 1$) 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($i + 1$) 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 $a_1 \dots a_{i-1}$ can be extended to reach p after reading $a_1 \dots a_i$. After reading the whole document and alternating between Capturing(i) and Reading($i + 1$), 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 4 follows by a straightforward induction over i. Indeed, the inductive hypothesis states that after executing Capturing(i) and Reading($i + 1$), $N[q]$ has the number of elements of $runs(\mathcal{A}, d(1, i), q)$. Then, by following the same arguments as for the correctness of Preprocessing, one can show that $N[q]$ also store the number elements of $runs(\mathcal{A}, d(1, i + 1), q)$ after executing Capturing($i + 1$) and Reading($i + 2$). □

Therefore, Count[$\mathcal{L}_1$], where $\mathcal{L}_1$ is the class of deterministic seVA, can be computed in polynomial time in combined complexity.

Unfortunately, the efficient algorithm of Theorem 6.1 cannot be extended 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 [29]. 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-hard 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 6.2. Count[fVA] *is SpanL-complete.*

Proof. 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.

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 4. 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 counting the number of words of length n that $\mathcal{B}$ accepts, is equivalent to counting the number of mappings that $\mathcal{A}_{\mathcal{B},n}$ generates 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:

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

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. □

From Proposition 5.3, we know that every functional VA can be converted in polynomial time into a functional extended VA. Therefore, Theorem 6.2 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 class $\mathcal{L}$ studied in this paper that is different from $\mathcal{L}_1$, the class of deterministic seVA.

Discussion: In Section 5 we have shown that enumerating the answers to a functional VA with output-linear delay can be done after a pre-computation phase that takes 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 bounded delay algorithms with efficient pre-computation phases usually imply the existence of efficient counting algorithms [25], in [14] we conjectured that it may be impossible to find an *efficient* algorithm that has pre-computation time better than $O(2^{\|\mathcal{A}\|} \times \|d\|)$, that is obtained by determinizing an fVA and running the algorithm from Section 4. However, the notion of efficiency that we used was that of data-constant delay, which was not appropriate since we were relying on the conjecture that output-linear delay algorithms with efficient precomputation phase imply efficient counting algorithms.

Interestingly, Amarilli et al. realized this difference and refuted our conjecture in [3], showing an enumeration algorithm with preprocessing time $O(\|\mathcal{A}\|^{\omega+1} \times \|d\|)$ and delay $O(\|\mathcal{A}\|^4)$ between outputs, where $2 \leq \omega \leq 3$ and ω is an exponent for Boolean matrix multiplication (see [3] for a more precise bound). Naturally, the results of [3] do not contradict Theorem 6.2 given that from the enumeration algorithm in [3] one cannot derive a polynomial time algorithm for Count[fVA]. More importantly, we now understand that the conjecture should have stated the impossibility of an algorithm that has better pre-computation than $O(2^{\|\mathcal{A}\|} \times \|d\|)$ and output-linear delay (i.e. in combined complexity). We leave this stronger conjecture as an open problem.

7 CONCLUSIONS

We believe that the algorithm described in Section 4 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 asymptotic complexity is very efficient for the class of deterministic sequential extended VA. Third, we have shown the cost of executing our algorithm on arbitrary regular spanners, by first converting regular spanners into deterministic sequential extended VA. The resulting bounds, although not ideal, are reasonable for a wide range of spanners usually encountered in practice. Finally, we have shown that better pre-computation 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 4 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 5. As far as theoretical aspects of this work are concerned, we are also interested in establishing hard lower bounds for output-linear delay algorithms that do not relay on conjectured claims.

Acknowledgements. Florenzano, Riveros and Vrgoč were partially supported by Millennium Institute for Foundational Research on Data. Vrgoč was also supported by the FONDECYT project nr. 11160383, and Riveros by the FONDECYT project nr. 11150653. Ugarte acknowledges support from Innoviris, the Brussels Institute for Research and Innovation (project SPICES).

REFERENCES

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

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

[3] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2019. Constant-Delay Enumeration for Nondeterministic Document Spanners. In *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal.* 22:1–22:19.

[4] Marcelo Arenas, Francisco Maturana, Cristian Riveros, and Domagoj Vrgoč. 2016. A Framework for Annotating CSV-like Data. *Proc. VLDB Endow.* 9, 11 (July 2016), 876–887. https://doi.org/10.14778/2983200.2983204

[5] Guillaume Bagan. 2006. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, Vol. 4207. Springer, 167–181.

[6] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Proc. of CSL.* 208–222.

[7] Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, and Shivakumar Vaithyanathan. 2010. 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.* 128–137.

[8] Laura Chiticariu, Yunyao Li, and Frederick R. Reiss. 2013. 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.* 827–832.

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

[10] Russ Cox. 2007. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...). *http://swtch.com/~rsc/regexp/regexp1.html* (2007).

[11] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2014. Cleaning inconsistencies in information extraction via prioritized repairs. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014.* 164–175.

[12] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document Spanners: A Formal Approach to Information Extraction. *Journal of the ACM* 62, 2 (2015).

[13] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2016. Declarative Cleaning of Inconsistencies in Information Extraction. *ACM Trans. Database Syst.* 41, 1 (2016), 6:1–6:44. https://doi.org/10.1145/2877202

[14] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. 2018. Constant Delay Algorithms for Regular Document Spanners. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018.* 165–177.

[15] Dominik D. Freydenberger. 2017. A Logic for Document Spanners. In *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy.* 13:1–13:18.

[16] Dominik D Freydenberger and Mario Holldack. 2016. Document Spanners: From Expressive Power to Decision Problems. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 48. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[17] Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. 2018. Joining Extractions of Regular Expressions. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018.* ACM, 137–149.

[18] R. Hoffmann. 2012. *Interactive Learning of Relation Extractors with Weak Supervision.* Ph.D. Dissertation. University of Washington.

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

[20] Benny Kimelfeld. 2014. 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.* 156–163.

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

[22] Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. 2018. Document Spanners for Extracting Incomplete Information: Expressiveness and Complexity. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018.* ACM, 125–136.

[23] Andrea Morciano. 2017. Engineering a Runtime System for AQL. *Master Thesis, Université Libre de Bruxelles and Politecnico di Milano* (2017).

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

[25] Luc Segoufin. 2013. Enumerating with constant delay the answers to a query. In *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013*. 10–20.
[26] Luc Segoufin. 2014. 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*. 13–27.
[27] Luc Segoufin. 2015. Constant Delay Enumeration for Conjunctive Queries. *SIGMOD Record* 44, 1 (2015), 10–17.
[28] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. 2007. Declarative Information Extraction Using Datalog with Embedded Extraction Predicates. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. 1033–1044.
[29] Leslie G Valiant. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput.* 8, 3 (1979), 410–421.
[30] Stijn Vansummeren. 2006. Type inference for unique pattern matching. *ACM Trans. Program. Lang. Syst.* 28, 3 (2006), 389–428.