PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
SCHOOL OF ENGINEERING

OUTPUT-LINEAR ENUMERATION FOR EXTENSIONS OF MSO

MARTÍN ALONSO MUÑOZ CRUCES

Thesis submitted to the Office of Graduate Studies in partial
fulfillment of the requirements for the degree of
Doctor in Engineering Sciences

Advisor:
CRISTIAN RIVEROS

Santiago de Chile, March 2025

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
SCHOOL OF ENGINEERING

OUTPUT-LINEAR ENUMERATION FOR EXTENSIONS OF MSO

MARTÍN ALONSO MUÑOZ CRUCES

Members of the Committee:
CRISTIAN RIVEROS
DOMAGOJ VRGOC
PABLO BARCELÓ
GONZALO NAVARRO
NICOLE SCHWEIKARDT
SERGIO MATURANA

Thesis submitted to the Office of Graduate Studies in partial fulfillment of the requirements for the degree of Doctor in Engineering Sciences

Santiago de Chile, March 2025

Para Lete

AGRADECIMIENTOS

Acá le dedico una página y media a las personas que me acompañaron durante estos seis años (de cuatro) mientras realizaba mi doctorado y escribía esta tesis. Y por qué no, a los que me motivaron a seguir este camino.

El agradecimiento más importante es a Cristian que me recibió cuando no sabía nada y ahora puedo decir que sé algo más que nada. Me acompañó y guió sin dejarme en ningún momento, me mostró una pincelada de temas de investigación que resultaron ser profundos e interesantes a concho y que espero me van a mantener ocupado por un buen rato. También gracias a él y a Cristian Ruz por convertirme en coach y agregarle una dimensión tan llenadora a mi doctorado.

A Marcelo por haber estado ahí cuando yo sabía aún menos que nada y por el tiempo que me ha dado entonces y ahora.

Gracias a la PUC, al IMFD, a ANID y al Fondecyt de Cristian que me han mantenido con vida, y al comité de candidatura+defensa por el apoyo.

To Antoine and Louis for welcoming me in Paris twice, letting me join on some of the most inspiring research meetings of my career, and showing me what is hopefully a preview of the next few years of my life.

A la Selección de Programación Competitiva del DCC, y a toda la comunidad actual de Programación Competitiva en Chile por la motivación y el talento, por convertirse en un grupo de alumnos de los que puedo sentir orgullo de segunda mano y en un grupo de amistades con los que puedo seguir armando proyectos y juntas. Y de acá a Bella y Sensual por no parar de ganar hasta el final y llevarme a darle la vuelta al mundo.

A la comunidad antigua de programación competitiva, a Jorge Pérez, Cristian Ruz, a Marinkovic, a Pablo, Koco y a Lete. Ahí entendí cuál es la parte que importa de la computación.

Una mención honrosa a la máquina de café del DCC y a Magnet: ambos me dieron un empujón hacia el lado de la investigación y la academia en puntos cruciales de mi carrera.

A los de la oficina de doctorado, a Alda, Ceci, Yessenia, a Antonieta y a Cristian Ruz.

A los DCCompas por el apañe, a los cabros de siempre por estar ahí siempre y al AMQ.

Gracias a mi papá, mi mamá, a la Isa y al Luciano por todo lo demás, por no fallarme nunca y por ser mi centro.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Enumeration algorithms for data extraction queries are especially relevant in modern-day systems where data reaches extremely large sizes while sometimes having little-to-no organization. This thesis proposes an enumeration framework for queries for data extraction that can be modeled by different formal languages, and in some cases, over compressed data. This is done by the development of an enumeration data structure called Enumerable Compact Set, which provides a set of basic operations over sets, each implementable in constant time, that render a concise representation of the set of results which can then be enumerated efficiently.

First, this thesis provides an algorithm for enumerating results of a query that has been modeled by a nested-word automata, over a nested document. This algorithm is built to work in a streaming setting, and requires time that is independent from the data size (constant in data-complexity) per input symbol.

Second, we detail three algorithms for enumeration for queries that are modeled by context-free grammars. We deal with the cases of unambiguous, rigid and determinable grammars, which require cubic, quadratic and linear time over the data, respectively.

Third, we show enumeration over compressed documents via an algorithm that receives a regular-language query and a document that is concisely represented as a straight-line program. The enumeration can be done after linear time preprocessing in the size of the compact representation.

This thesis also serves as a proof of concept of the Annotated Automata model. This model properly subsumes Document Spanners efficient reductions, and allows the construction of algorithms that we deem to be more intuitive and easier to implement. Needless to say, all of our results are also applicable to Document Spanners.

Keywords: Data Structures and Algorithms, Formal Languages and Automata Theory, Logic in Computer Science.

RESUMEN

Los algoritmos de enumeración para consultas de extracción de datos son especialmente relevantes en los sistemas de hoy en día, donde los datos alcanzan tamaños extremadamente grandes y, a veces, tienen poca o ninguna organización. Esta tesis propone un marco teórico de enumeración para consultas de extracción de datos que se pueden modelar mediante diferentes lenguajes formales y, en algunos casos, sobre datos comprimidos. Esto se hace mediante el desarrollo de una estructura de datos de enumeración llamada *Enumerable Compact Sets* (Conjuntos Comprimidos Enumerables en inglés), que proporciona un conjunto de operaciones básicas sobre conjuntos, cada uno implementable en tiempo constante, cuyo resultado es una representación concisa del conjunto de resultados que luego se puede enumerar de manera eficiente.

En el primer capítulo, esta tesis entrega un algoritmo para enumerar los resultados de una consulta que ha sido modelada por un autómata de palabras anidadas, sobre un documento anidado. Este algoritmo está diseñado para funcionar en un entorno de *streaming* y requiere un tiempo independiente al tamaño de los datos (constante en *data complexity*) en cada símbolo de entrada.

En el segundo capítulo, detallamos tres algoritmos de enumeración para consultas que se modelan mediante gramáticas libres de contexto. Tratamos los casos de gramáticas no-ambiguas, rígidas y deterministas, que requieren tiempo cúbico, cuadrático y lineal sobre los datos, respectivamente.

En el tercer capítulo, mostramos la enumeración sobre documentos comprimidos mediante un algoritmo que recibe una consulta en lenguaje regular y un documento que se representa concisamente como un programa *straight-line*. La enumeración se puede realizar después del preprocesamiento en tiempo lineal en el tamaño de la representación compacta.

Esta tesis también sirve como prueba de concepto del modelo de *Annotated Automata* (Autómatas anotados en inglés). Este modelo considera satisfactoriamente reducciones eficientes desde *Document Spanners* y permite la construcción de algoritmos que consideramos más intuitivos y fáciles de implementar. Vale decir que todos nuestros resultados también son aplicables a Document Spanners.

Keywords: Estructuras de datos y algoritmos, lenguajes formales y teoría de autómatas, lógica en ciencias de la computación.

1. INTRODUCTION

The term *Big Data* is believed to have been originated in the mid-1990s, back when the Internet was utilized by some tens of millions of users. Even then, as the usage of the term suggests, there was a distinct necessity of processing tremendous amounts of data efficiently. In 2023, this number grew to 5,4 billion users (ITU, 2023), and that has naturally brought a great increase in the amount of raw data that is available and required to be processed. An example of online data task that has become commonplace are recommendation algorithms. Nowadays, four out of the five most visited sites on the Internet nowadays use user-generated information to suggest content back to the users (Similarweb, 2024). To name a few numbers, Netflix reported in 2012 that their users queued 2 million movies and TV shows, and generated 4 million ratings a day (Netflix, 2012), all of which is data that feeds into their recommendation algorithms. Another task is content management and moderation. In contexts where millions of messages are being sent daily, these are considered crucial to curb undesirable data such as fake news (Fang et al., 2024) and hate speech (Fortuna, Soler Company, & Wanner, 2021). These cases show how overreaching, diverse and massive the online data processing tasks have become.

The blowup of large data processing has happened in the offline world as well. In biological research, human genomes consist of around three billion base pairs, which translate into several gigabytes of raw data that is processed when performing genome-matching or detecting genetic diseases (Boucher et al., 2021). Modern space missions, such as GAIA and LSST, are expected to produce Petabytes of data (Brahem, Zeitouni, & Yeh, 2020). To name a few other examples of offline data processing, patents are analyzed in bulk in order to predict future litigations (S. Chen & Lai, 2023); GPS records from large populations can be used in urban planning projects (Li et al., 2023); and historical crime data is massively processed to predict crime (Liang et al., 2023).

The importance and desirability of having adequate data processing solutions is unquestionable. In many cases, these algorithms need to be sharply efficient, to the point of making any solution that takes time longer than linear in the data size useless in practice.

In database systems, it is common to talk about *query evaluation* as the most elementary task which extracts relevant information from data. It can be formally defined by an instance (Q, D) in which Q is the query and D is the data. Each query Q also defines a function which maps D into the desired set of outputs $Q(D)$. As an example, consider the task of detecting if a chromosome, written as a sequence of bases, contains a certain pattern in it that may correspond to a particular disease. In this case, it would make sense to let D be the sequence of bases written as a string, and Q would be the pattern one wants to find. If the task is a yes/no question – is the pattern Q present in D? – then $Q(D)$ can evaluate to either *true* or *false*, but if one wants to also know all of the positions in the sequence that match that pattern, $Q(D)$ would be a set that contains all relevant positions, and in the case there are none, it evaluates to an empty set. With this formalization, one can talk about linear-time solutions as those algorithms that take time proportional to the size of D to compute $Q(D)$.

It is also useful to frame results around *query languages*, which define classes of queries by their syntax and semantics. For instance, one can solve the task above by using *regular expressions* as the query language (see (Pin, 2021) for a precise definition): we can make Q be the regular expression $r = \texttt{.}^{\star}p\texttt{.}^{\star}$ where p is the pattern string. In this case, we are using a usual semantics for regular expressions which is that r has to match the entire text; the sequence $\texttt{.}^{\star}$ is a wildcard that matches any string, and so r is a match if the data is equal to anything, followed by p, followed by anything. Another aspect of query languages that make them natural to work with is that each of them also comes with a certain expressive power. For example, no regular expression can define the query of checking if two strings are identical (Pin, 2021), whereas there are more powerful query languages that can. As evaluation algorithms are built for each particular query language, a weaker one can require less running time or less memory space in the worst case. It can happen that the same query becomes cheaper to compute if the user simply expresses it in a different language.

When measuring the execution time of a evaluation algorithm, the classic notion is to count the number of steps taken by the algorithm from the moment it begins, until the last output is printed. Since algorithms are built for inputs of different sizes, we typically measure this execution time as a function of the size of the input. This is a reasonable model whenever the overall output size is small, e.g., when the output is simply true or false, or a numeric value. However, for tasks in which one can expect a large amount of outputs, which occur quite naturally in many data management problems, it makes sense to treat the output printing phase separately. Naturally, the best execution time one can expect for this phase is linear in the size of the entire output set, so the overall running time that we will set as our goal is given by $|D| + |Q(D)|$. Here we are using the standard notation $|x|$ to refer to the size of an adequate encoding of x.

In database systems, there are many cases in which an evaluation problem can have a large output set. For instance, one may be interested not only in all elements in the data that satisfy a certain condition, but in tuples or subsets of them. Or perhaps the data may not fit in memory: it could be given in a streaming fashion or it could be compressed, and this implies that even an output set that has size linear in the raw data might be huge. An obvious problem here is that producing all outputs can take unreasonably long, but a more subtle one is that when one measures the running time of an algorithm, the cost of writing the outputs hides the cost of actually doing the calculations to obtain them. For this reason, a significant line of research on query evaluation has adopted the perspective of *enumeration algorithms* (Bagan, 2006a). Instead of explicitly producing all results, the task is to enumerate them, in any order and without repetition. The cost of the algorithm is then measured across two dimensions: the *preprocessing time*, which is the time needed to read the input and prepare an enumeration data structure; and the *delay*, the worst-case time that can elapse between any two solutions while enumerating using the data structure.

In the realm of enumeration algorithms, there is an even finer complexity yardstick: We say that an algorithm has *constant-delay* (Segoufin, 2013) if the delay between any two consecutive outputs is constant. One can think of this as an enumeration phase in which the outputs are produced in a fixed pace, never stopping until the last output is produced. However, this delay guarantee is not a reasonable one unless every output is expected to have constant size. Instead, in a lot of cases it makes more sense to set as a goal what we call *output-linear delay* (Florenzano, Riveros, Ugarte, Vansummeren, & Vrgoc, 2020). This is a relaxation of constant-delay which only requires the delay between two outputs to be linear in the size of the earlier one.

As an example, consider the task of receiving a list of numbers, such as $L = [5, 7, 2, 12]$, and producing a list of pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ in which $x_1, x_2, \ldots, x_n$ is the sorted list, and each y_i is the difference between x_i and x_{i+1}. For the said L, the desired set of outputs could be $\{(2, 3), (5, 2), (7, 5), (12, \infty)\}$. A constant-delay algorithm may simply sort the list, read it sequentially and produce each value along with the difference. Now, consider the task of again receiving a list of numbers, but now instead of printing the difference between two numbers, we ask for the numbers that are present in that gap. For the input L, the output set would be $\{(2, 3, 4), (5, 6), (7, 8, 9, 10, 11), (12, \infty)\}$. In this case, there is no reasonable way of solving this with a constant-delay algorithm, since the size of a single output could be arbitrarily large, even for a list of size 2. However, sorting the list, traversing it linearly and writing down the gaps as they are seen is a perfectly valid solution that satisfies output-linear delay.

One area where enumeration algorithms have been especially successful is the field of information extraction, where *document spanners* have been proposed as a suitable formalism (Fagin, Kimelfeld, Reiss, & Vansummeren, 2015). Information extraction captures the family of data management tasks where the data is a document, i.e, a string of characters, and the query is a description of the information that should be extracted from the text. A spanner formalizes this idea by defining a function that maps documents to sets of *mappings*, which themselves map variables to substrings of the document (called *spans*). The enumeration problem is then to enumerate all mappings of a spanner on an input document. The work by Florenzano et al. (Florenzano et al., 2020) showed that, if the spanner is described by a finite automaton, then the task could be solved with output-linear delay after a preprocessing that is linear in the document, and polynomial in the spanner if the automaton is deterministic; this was extended in (Amarilli, Bourhis, Mengel, & Niewerth, 2019c) to allow similar bounds for nondeterministic finite automata.

Solving enumeration tasks is a fundamental problem in database systems, and this is the main topic of this thesis: finding efficient algorithms to evaluate queries over great volumes of data with output-linear delay. In particular, we work on this problem over nested documents or query languages over documents that allow recursion.

In many practical cases, the way these large volumes of data are contained is in documents that have a pre-established recursive structure. It may be hierarchical, like XML and JSON documents, or program-like, as in some forms of compressed text (Ziv & Lempel, 1977; Storer & Szymanski, 1982; Claude & Navarro, 2011). These documents are designed to be read by computers; the way the user is expected to access the data is by writing a query using a suitable query language, and then process the query through a specialized evaluation algorithm.

Let us look at a simple XML document to show how data may be laid out in practice. The following document (separated in two columns) contains information about musical artists and albums satisfying the hierarchy `Artist->Album-> [Title, Year]`:

```
<Document>
  <Artist>
    <Name>Los Tres</Name>
    <Album>
      <Title>Los Tres</Title>
      <Year>1991</Year>
    </Album>
    <Album>
      <Title>Fome</Title>
      <Year>1997</Year>
    </Album>
  </Artist>
  <Artist>
    <Name>La Ley</Name>
    <Album>
      <Title>Invisible</Title>
      <Year>1995</Year>
    </Album>
  </Artist>
  <Artist>
    <Name>Tiro de Gracia</Name>
    <Album>
      <Title>Ser Humano!!</Title>
      <Year>1997</Year>
    </Album>
  </Artist>
</Document>
```

It can be appreciated how XML owes its classification of “semi-structured data” to the fact that the contents maintain a lot of the flexibility of unstructured data (e.g. plain-text documents), yet there is some machine-oriented structure to the data that aids its processing. In the example above, the entire document contains a list of artists, and the structure requires the info of each to be demarcated by the tags `<Document>` and `</Document>`

In this thesis, we will sometimes restrict ourselves to evaluation tasks where the data part of the query is presented with certain recursive structures—namely, nested documents, and program-like documents. For some tasks, we allow a more standard plain-text structure.

Let us now discuss the presentation aspect for the *query* part of a task. That is, the query languages that we will deal with in this thesis: these are all extensions to a logic called Monadic Second Order Logic. The logic in itself is of immense theoretical importance, but in actual data extraction tasks it is rarely used as-is. The reader might be more familiar with a query language which is equivalent in expressive power to Monadic Second Order over documents—that is, regular expressions. In database systems and data extraction tasks, regular expressions (regex for short) have long been a favoured choice for theoreticians, developers and users alike.

When a single document is huge, and queries are simple enough, it makes sense to use a language based on regular expressions, such as XPath in the case of XML. Continuing the ”albums” example above, an XPath query that produces the albums that were released in the year 1997 might look as follows:

$$Q = \texttt{//Artist/Album[Year = 1997].}$$

This would output info about albums *Fome* and *Ser Humano!!*.

In the case of compressed text, one can use a plain regular expression, and process it using some of the known algorithms that evaluate it on the compressed file itself (Lohrey, 2012). Among these solutions, one uses regular spanners (i.e. document spanners based on regular expressions) over compressed documents to enumerate with delay that is logarithmic in the size of the uncompressed document (Schmid & Schweikardt, 2021).

There are evaluation tasks in which regular expressions are not powerful enough, and the data document does not have a pre-established structure. The immediate extension to regular expressions and MSO that also allows the query itself to infer this very structure are context-free grammars. Some scenarios in which this type of queries have been famously used are code parsing and verification on nested documents—both scenarios where the data itself would ideally satisfy a desired structure, but this is not a-priori guaranteed.

Grammars by themselves do not describe how to capture substrings, so they are not immediately suited for extracting data. One model that has been proposed to bridge this gap is based on document spanners and is called Extraction Grammars (Peterfreund, 2023). These are defined by context-free grammars which allow beginning and ending position marks in its syntax. In same work, we also find a relevant enumeration result. It shows constant-delay enumeration after a preprocessing that takes quintic time on the size of the document.

Having discussed some scenarios where one would want to extract data using queries that are based on regex and their extensions, we note that each used an ad-hoc extension of the yes/no query language to capture the desired outputs. We contrast this with MSO, which has the ability to define queries that extract sets of positions in an input document in a natural fashion. This is especially useful for us, as this ability implies there is a generic adaptation from yes/no evaluation into queries with several, complex outputs, for every extension of MSO.

This idea is the core of the *Annotator* framework, which translates many rule-based binary query languages into query languages with complex outputs. It does so by continuing to use the same type of binary model that describes the query while adding a second dimension to the alphabet that describes the data—we use these as the symbols the model will treat as letters. For instance, the underlying binary model of an annotator may accept words such as $(a, x)bb(a, y)$ or $a(b, x)(b, y)a$, and if we give the word $abba$ to the annotator, it will define an output set that includes the strings $(x, 1)(y, 4)$ and $(x, 2)(y, 3)$. The first string represents the idea that if we extend $abba$ by appending an x to the first position, and a y to the fourth position, the resulting word is accepted by the model, and the second string represents the analogous idea.

We have found that some of the relevant models for document spanners that extend a binary query language into a complex one—as is the case for Regular Spanners and Extraction Grammars—can be reduced into an annotator with only a minor blowup. More importantly, we believe that this change makes the evaluation algorithms simpler, and that it is a reason behind our finding improvements in the best-known bounds for them.

The main goal of this thesis is thus to explore the power of the Annotator framework, and provide efficient enumeration algorithms for the ways it can be used to deal with known query languages. It is worth noting that while many of the query languages studied in the literature were restricted to outputs with a fixed size, annotators by default do not impose any such restriction, so all of our results are given in the best-possible delay bound that this circumstance allows; namely, output-linear delay.

To this end, in this thesis we present three different formalisms for information extraction and then provide an efficient enumeration scheme for queries expressed in each of them. These are: (1.) Automata over streams of nested documents, (2.) context-free grammars which represent annotations in documents and (3.) regular automata over compressed documents.

1.1. Summary of contributions

This document is structured in three main chapters. In the following, we present a brief summary of each.

1.1.1. Nested streaming queries

Streaming query evaluation (Altınel & Franklin, 2000; Babcock, Babu, Datar, Motwani, & Widom, 2002) is the task of processing queries over data streams in one pass and with a limited amount of resources. This approach is especially useful on the web, where servers share data, and they have to extract the relevant content as they receive it. For structuring the data, the de facto structure on the web is nested documents, like XML or JSON. For querying, servers use languages designed for these purposes, like XPath, XQuery, or JSON query languages. As an illustrative example, suppose our data server (e.g., a Web API) is continuously receiving XML documents like:

```
<doc> <a> <b/> <c/> <b/> </a> <c> <b/> <b/> </c> </doc> ...
```

and for each document it has to evaluate the query $\mathcal{Q} = $ `//a/b` (i.e., to extract all b-tags that are surrounded by an a-tag). The streaming query evaluation problem consists of reading these documents and finding all b-tags without storing the entire document on memory, that is, by making one pass over the data and spending constant time per tag. In our example, we need to retrieve the 3rd and 5th tags as soon as the last tag `</doc>` is received. One could consider here that the server has to read an infinite stream and perform the query evaluation continuously, where it must enumerate partial outputs as soon as one of the XML documents ends.

Researchers have studied the streaming query evaluation problem in the past, focusing on reducing the processing time or memory usage (see, e.g. (Bar-Yossef, Fontoura, & Josifovski, 2007)). Hence, they spent less effort on understanding the enumeration time of such a problem, with respect to delay guarantees between outputs. Constant-delay enumeration is a new notion of efficiency for retrieving outputs (Durand & Grandjean, 2007; Segoufin, 2013). Given an instance of the problem, a constant-delay enumeration algorithm performs a preprocessing phase over the instance to build some indices and then continues with an enumeration phase. It retrieves each output, one by one, taking a delay that is constant between any two consecutive outcomes. These algorithms provide a strong guarantee of efficiency since a user knows that, after the preprocessing phase, she will access the output as if the algorithm had already computed it. These techniques have attracted researchers' attention, finding sophisticated solutions to several query evaluation problems (Bagan, Durand, & Grandjean, 2007; Berkholz, Gerhardt, & Schweikardt, 2020; Bagan, 2006a; Amarilli, Bourhis, Jachiet, & Mengel, 2017; Florenzano et al., 2020; Amarilli, Bourhis, Mengel, & Niewerth, 2019a).

In this chapter, we investigate the streaming query evaluation problem over nested documents by including enumeration guarantees, like constant delay. We study the evaluation of queries given by visibly pushdown annotators (VPAnn) over nested documents. These machines are an "output extension" of visibly pushdown automata, and have the same expressive power as MSO over nested documents. In particular, VPAnn can define queries like $\mathcal{Q}$ above or any fragment of query languages for XML or JSON included in MSO. Therefore, VPAnn allow considering the streaming query evaluation from a more general perspective, without getting married to a specific language (e.g., XPath).

We study the evaluation of VPAnn over a nested document in a streaming fashion. Specifically, we want to find a streaming algorithm that reads the document sequentially and spends as little time as possible per input symbol. Furthermore, whenever needed, the algorithm can enumerate all outputs with output-linear delay. The main contribution in this chapter is an algorithm with such characteristics for the class of unambiguous

VPAnn. We can extend this algorithm to all VPAnn by determinization (preserving its data complexity). Regarding memory usage, we bound the amount of memory used in terms of the nesting of the document and the output weight. We show that our algorithm is worst-case optimal in the sense that there are instances where the maximum amount of memory required by any streaming algorithm is at least one of these two measures. Finally, we present some examples that show how our result can be applied to the streaming evaluation of XML and JSON query languages. Further, we show an application of our results in the context of information extraction by document spanners (Fagin et al., 2015).

1.1.2. Annotated Grammars

A natural way to address extraction over structured data is to move from finite automata to *context-free grammars* (CFGs). Context-free grammars are a well-known formalism: they extend regular expressions and are commonly used, e.g., in programming language design. Common verification tasks on textual representations of tree documents can be expressed using CFGs, and so can parsing tasks, e.g., to extract subexpressions from source code data. However, CFGs do not describe *captures*, i.e., they do not specify how to extract the parts of interest of an input document, and thus cannot be used directly for information extraction.

This question of information extraction with grammars was studied by Peterfreund in very recent work (Peterfreund, 2023). This paper proposed a formalism of *extraction grammars*, which are CFGs extended via special terminals that describe the endpoints of spans. Further, it presents an algorithm to enumerate the mappings captured by *unambiguous* extraction grammars on an input document. However, while the algorithm achieves constant-delay, the preprocessing bound is significantly worse than in the case of regular spanners: it is quintic in the document, and exponential in the number of variables of the grammar. This complexity is also worse than CFG parsing, e.g., the standard CYK parsing algorithm runs in cubic time in the input string.

Our goal in this chapter is to study the enumeration problem for CFGs while achieving better complexities. Our algorithms ensure a constant-delay guarantee when outputs have constant size, and more generally ensure *output-linear* delay when this is not the case: the delay is linear in the size of each produced solution. Within this delay bound, the preprocessing time has lower complexity: it is at worse cubic in the input document, and improves to quadratic or even linear time for restricted classes. We achieve these results by proposing a new formalism to extend CFGs, called *annotated grammars*, on which we impose an unambiguity restriction similar to that of (Peterfreund, 2023). Let us present our specific contributions.

Our first contribution is to introduce *annotated grammars* (Section 4.1). They are a natural extension of CFGs, where terminals are optionally annotated by the information that we wish to extract. We then study the problem, given an annotated grammar $\mathcal{G}$ and document s, of enumerating all annotations of s that are derived by $\mathcal{G}$. This captures the enumeration problems for regular spanners (Florenzano, Riveros, Ugarte, Vansummeren, & Vrgoc, 2018; Amarilli, Bourhis, Mengel, & Niewerth, 2020), nested words, and even the extraction grammars of (Peterfreund, 2023) (we explain this in Section 4.5). As we explain, we aim for *output-linear delay*, which is the best possible delay in our setting where the solutions to output may have non-constant size.

Our second contribution is to study the enumeration problem for *unambiguous* annotated grammars (Section 4.2), that do not produce multiple times the same annotation of an input string. This is a natural restriction to avoid duplicate results, which is also made in (Peterfreund, 2023). For such grammars, we present an algorithm to enumerate the annotations produced by a grammar $\mathcal{G}$ on a string s with output-linear delay (independent from $\mathcal{G}$ or s), after a preprocessing time of $\mathcal{O}(|\mathcal{G}| \cdot |s|^3)$, i.e., cubic time in s, and linear time in $\mathcal{G}$. This improves over the result of (Peterfreund, 2023) whose preprocessing is quintic. Our algorithm has a modular design: it follows a standard design of a CFG parsing algorithm, but uses the abstract data structure of Chapter 3 to represent the sets of annotations and combine them with operators in a way that allows for output-linear enumeration. We further show a conditional lower bound on the best preprocessing time that can achieve output-linear delay, by reducing from the standard task of checking membership to a CFG, and using the lower bound of (Abboud, Backurs, & Williams, 2018). We show that the preprocessing time must be $\Omega(|s|^{\omega-c})$ for every $c > 0$, where ω is the Boolean matrix multiplication exponent.

Our third contribution is to improve the preprocessing time by imposing a different requirement on grammars. Thus, we introduce *rigid* annotated grammars (Section 4.3) where, for every input string, all annotations on the string are intuitively produced by parse trees that have the same shape. In contrast with general annotated grammars, we show that rigid annotated grammars can always be made unambiguous, so that our algorithm applies to them. But we also show that, under this restriction, the data complexity of our algorithm goes down from cubic to quadratic time. Further, achieving sub-quadratic preprocessing time would imply a sub-quadratic algorithm to test membership to an unambiguous CFG, which is an open problem.

Our last contribution shows how we can, in certain cases, achieve linear-time preprocessing complexity and output-linear delay (Section 4.4). This is the complexity of enumeration for regular spanners, and is by definition the best possible. We show that the same complexity can be achieved, beyond regular spanners, for a subclass of rigid grammars, intuitively defined by a determinism requirement. We define it via the formalism of *pushdown annotators* (PDAnn for short), which are the analogue of pushdown automata for CFGs, or the extraction pushdown automata of (Peterfreund, 2023). We show that PDAnn are equally expressive to annotated grammars, and that rigid CFGs correspond to a natural class of PDAnns where all runs have the same sequence of stack heights. Moreover, we show that we can enumerate with linear-time preprocessing and output-linear delay in the case of *profiled-deterministic* PDAnn, where the sequence of stack heights can be computed deterministically over the run: this generalizes regular spanners and visibly-pushdown automata.

1.1.3. Queries over compressed documents

Recently, Schmid and Schweikardt (Schmid & Schweikardt, 2021, 2022) studied the evaluation problem for regular spanners over a document compressed by a Straight-line Program (SLP). In this setting, one encodes a document through a context-free grammar that produces a single string (i.e., the document itself). This mechanism allows highly compressible documents, in some instances allowing logarithmic space compared to the uncompressed copy. The enumeration problem consists now of evaluating a regular spanner over an SLP-compressed document. In (Schmid & Schweikardt, 2021), the authors provided a logarithmic-delay (over the uncompressed document) algorithm for the problem, and in (Schmid & Schweikardt, 2022), they extended this setting to edit operations over SLP documents, maintaining the delay. In particular, these works left open whether one can solve the enumeration problem of regular spanners over SLP-compressed documents with a constant-delay guarantee.

In this chapter, we extend the understanding of the evaluation problem over SLP-compressed documents in several directions.

We study the evaluation problem of *annotated automata* (AnnA) over SLP-compressed documents. These automata are a general model for defining regular enumeration problems, which strictly generalizes the model of extended variable-set automaton used in (Schmid & Schweikardt, 2021).

We provide an output-linear delay enumeration algorithm for the problem of evaluating an unambiguous AnnA over an SLP-compressed document. In particular, this result implies a constant-delay enumeration algorithm for evaluating extended variable-set automaton, giving a positive answer to the open problem left in (Schmid & Schweikardt, 2021).

We show that this result extends to what we call a *succinctly* annotated automaton, a generalization of AnnA whose annotations are succinctly encoded by an enumeration scheme. We develop an output-linear delay enumeration algorithm for this model, showing a constant-delay algorithm for sequential (non-extended) vset automata, strictly generalizing the work in (Schmid & Schweikardt, 2021).

Finally, we show that one can maintain these algorithmic results when dealing with complex document editing as in (Schmid & Schweikardt, 2022).

The main technical result in this chapter is to show that the Enumerable Compact Sets presented in Chapter 3 can be extended to deal with shift operators (called Shift-ECS). This extension allows us to compactly represent the outputs and "shift" the results in constant time, which is to add or subtract a common value to all elements in a set. Then, by using matrices with Shift-ECS nodes, we can follow a bottom-up evaluation of the annotated automaton over the grammar (similar to (Schmid & Schweikardt, 2021)) to enumerate all outputs with output-linear delay. The combination of annotated automata and Shift-ECSs considerably simplifies the algorithm presentation, reaching a better delay bound.

This thesis is structured in the following way: In Chapter 2, we describe some terminology that will be used throughout the three main chapters. The first main chapter is Chapter 3, in which we show our results for nested streaming queries. The second is Chapter 4, in which we show our results for annotated grammars. The third is Chapter 5, in which we show our results for queries over compressed documents. Lastly, in Chapter 6, we give some closing words and list the main conclusions.

2. PRELIMINARIES

Throughout this document we will use several formalisms for processing text. In order of expressiveness, we will deal with finite automata and regular expressions, nested or visibly pushdown automata, and context-free grammars and pushdown automata. We dedicate this chapter to defining these models.

2.1. Strings and Finite State Automata

Strings and documents. Given a finite alphabet Σ, a *string* w over Σ (or just a string) is a sequence $w = a_1 a_2 \dots a_n \in \Sigma^*$. Given strings w_1 and w_2, we write $w_1 \cdot w_2$ (or just $w_1 w_2$) for the concatenation of w_1 and w_2. We denote by $|w| = n$ the length of the string $w = a_1 \dots a_n$ and by ε the string of length 0. We use Σ^* to denote the set of all strings, and Σ^+ for all strings with one or more symbols. Due to differences in the literature, in some chapters we will refer to strings as *documents* with the same indications, except we will prefer d instead of w to denote a generic document.

Regular automata. A *regular automata* is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ where Q is a finite set of *states*, Σ is an input alphabet, $I \subseteq Q$ and $F \subseteq Q$ are the initial and final set of states, respectively, and $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation.

A run ρ of $\mathcal{A}$ over a string $s = a_1 a_2 \dots a_n \in \Sigma^*$ is a sequence of the form:

$$\rho \ := \ q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \dots \xrightarrow{a_{n+1}} q_{n+1}$$

such that $q_1 \in I$ and, for each $i \in \{1, \dots, n\}$, it holds that $(q_i, a_i, q_{i+1}) \in \Delta$. We say that ρ is accepting if $q_{n+1} \in F$. We define the language of $\mathcal{A}$ as $L(\mathcal{A}) = \{s \mid \text{there is an accepting run of } \mathcal{A} \text{ over } s\}$.

We say that $\mathcal{A}$ is *deterministic* if for each pair $(p, a) \in Q \times \Sigma$ there exists exactly one $q \in Q$ such that $(p, a, q) \in \Delta$.[1] We say that $\mathcal{A}$ is *unambiguous* if for each $s \in L(\mathcal{A})$ there exists exactly one accepting run of $\mathcal{A}$ over s.

2.2. Document Spanners

Consider a document $d = a_1 \dots a_n$ over an input alphabet Σ. A *span* of d is a pair $[i, j\rangle$ with $1 \leq i \leq j \leq n + 1$. Intuitively, a span represents a substring of d by identifying the starting and ending positions. We define the substring $[i, j\rangle$ by $d[i, j\rangle = a_i \dots a_{j-1}$. We denote by $\mathsf{Spans}(d)$ the set of all possible spans of d.

Consider also a set of variables Vars such that $\Sigma \cap \mathsf{Vars} = \emptyset$. Let $X \subseteq \mathsf{Vars}$ be a finite set of variables. An (X, d)-*mapping* (or just a *mapping*) $\mu \colon X \to \mathsf{Spans}(d)$ assigns variables in X to spans of d. An (X, d)-*relation* is a finite set of (X, d)-mappings. Then a *document spanner* P (or just spanner) is a function associated with a finite set X of variables that map documents d into (X, d)-relations.

For $X \subseteq \mathsf{Vars}$, let $\mathsf{C}_X = \{\vdash^x, \dashv^x \mid x \in X\}$ be the set of captures of X where, intuitively, $\vdash^x$ denotes the opening of x, and $\dashv^x$ its closing.

In some chapters we will use ref-words as an intermediate step between the structure that defines a spanner (such as a grammar or an automaton) and the mappings themselves. Formally, a ref-word is just a string $r \in (\Sigma \cup \mathsf{C}_X)^*$. A ref-word $r = a_1 \dots a_n \in (\Sigma \cup \mathsf{C}_X)^*$ is called *valid* for X if, for every $x \in X$, there exists exactly one position i with $a_i = \vdash^x$ and exactly one position j with $a_j = \dashv^x$, such that $i < j$. In other words, a valid ref-word defines a correct match of opening and closing captures. Moreover, each $x \in X$ induces a unique factorization of r of the form $r = r_x^p \cdot \vdash^x \cdot r_x \cdot \dashv^x \cdot r_x^s$. This factorization defines an (X, d)-mapping as follows. Let $\mathsf{plain} : (\Sigma \cup \mathsf{C}_X)^* \to \Sigma^*$ be the morphism that removes the captures from ref-words, namely, $\mathsf{plain}(a) = a$ when $a \in \Sigma$ and $\mathsf{plain}(c) = \varepsilon$

[1] In some of the literature, deterministic is defined as automata for which there exist *at most one* q per (p, a). We restrict ourselves to the definition in the paragraph.

when $c \in \mathsf{C}_X$. We extend plain to operate homomorphically over strings. Furthermore, let r be a valid ref-word for X, d be a document, and assume that $\mathsf{plain}(r) = d$. Then we define the (X, d)-mapping μ^r such that $\mu^r(x) = [i, j\rangle$ iff $r = r^p_x \cdot \vdash^x \cdot r_x \cdot \dashv^x \cdot r^s_x$, $i = |\mathsf{plain}(r^p_x)| + 1$, and $j = i + |\mathsf{plain}(r_x)|$.

As an example, consider the document $d = \mathtt{hellogoodbye}$. If we let $X = \{x, y\}$, then a valid ref-word for X would be $r = \mathtt{hello} \vdash^x \vdash^y \mathtt{good} \dashv^x \mathtt{bye} \dashv^y$. Note that $\mathsf{plain}(r) = d$, and thus the (X, d)-mapping μ^r is properly defined as $\mu^r(x) = [6, 10\rangle$ and $\mu^r(y) = [6, 13\rangle$.

2.3. Enumeration Algorithms

All of the main results in this thesis are given in the framework of *enumeration algorithms*. Such algorithms consist of two *phases*. First, in the *preprocessing phase*, the algorithm receives the input $\mathcal{I} = (Q, w)$ where Q is a query and w is a string, and produces some *index* structure D. The *preprocessing time* is the worst-case running time of this preprocessing phase, measured as a function of the input $\mathcal{I}$. We say that such an algorithm has *f-preprocessing time* if there exists a constant c such that, for every input $\mathcal{I}$, the time for the preprocessing phase of $\mathcal{I}$ is bounded by $c \cdot f(|\mathcal{I}|)$.

As $\mathcal{I}$ is made of two components Q and w, we also sometimes measure the preprocessing time in terms of w only. This is referred to as the *data complexity* of the problem. On the other hand, when a problem is measured in terms of both w and Q this is referred to as its *combined complexity*.

For the purposes of this definition, we will overload Q to denote both the structure that defines the query, and the function that maps w to a desired set of outputs $Q(w)$. In the following chapters, these concepts are separated by notation, for example, if the input query is represented by a structure S, the associated function is denoted as $[\![S]\!]$. Furthermore, we assume that each element $y \in Q(w)$ is a string $y \in \Omega^*$ for some set of output symbols Ω.

Second, in the *enumeration phase*, the algorithm can use Q, w and D to produce the elements of the output set $Q(w)$ one after the other and without repetitions. During this phase the algorithm: (1) writes $\#y_1\#y_2\#\cdots\#y_m\#$ to the output registers where # is a distinct separator symbol not mentioned in any output, and $y_1, y_2, \ldots, y_m$ is an enumeration (without repetitions) of the set $Q(w)$, (2) it writes the first # as soon as the enumeration phase starts, and (3) it stops immediately after writing the last #.

For the enumeration phase, we measure the *delay between two outputs* as follows. For an input $x \in \Omega^*$, let $\#y_1\#y_2\#\cdots\#y_m\#$ be the output of the algorithm during any call to the enumeration phase. Furthermore, let $\mathsf{time}_i(x)$ be the time in the enumeration phase when the algorithm writes the i-th $\#$ when running on x for $i \leq m+1$. Define $\mathsf{delay}_i(x) = \mathsf{time}_{i+1}(x) - \mathsf{time}_i(x)$ for $i \leq m$. Then we say that the algorithm has *output-linear delay* (Florenzano et al., 2020)[2], if there exists a constant k such that for every $x \in \Omega^*$ and $i \leq m$ it holds that $\mathsf{delay}_i(x) \leq k \cdot |y_i|$. In other words, the number of instructions executed by the enumeration algorithm between the time that the i-th and the $(i+1)$-th # are written is linear on the size of y_i. Note that, in particular, an output-linear delay implies that the enumeration phase ends in constant time if there is no output for enumerating.

The *memory usage* of the algorithm is the maximum memory used across both phases, including the size of D.

2.4. Model of Computation

As it is common in the enumeration algorithms literature (Bagan, 2006a; Courcelle, 2009; Segoufin, 2013), for our algorithms we assume the computational model of *Random Access Machines* (RAM) with uniform cost measure, and addition and subtraction as basic operations (Aho, Hopcroft, & Ullman, 1974). We assume that a RAM has read-only input registers where the machine places the input, read-write work registers where it does the

[2] output-linear delay has also been called linear delay in the literature (Courcelle, 2009)

computation and write-only output registers where it gives the output (i.e., the enumeration of the results).

3. ENUMERATION FOR NESTED QUERIES

In this chapter, we present a query model named Visibly Pushdown Annotators (VPAnn), which is a computational model that extends visibly pushdown automata with outputs and has the same expressive power as Monadic Second Order over nested documents. We study the task of receiving the input data in a streaming document, and we show an algorithm which enumerates the outputs with output-linear delay after a preprocessing phase which takes data-independent time after receiving each character in the input data.

Furthermore, we show that this algorithm is worst-case optimal in terms of update-time per symbol and memory usage.

Outline of the chapter. In Section 3.1, we describe the main terminology and models for the chapter. In Section 3.2, we discuss the notion of a streaming enumeration problem. In Section 3.3, we define Visibly Pushdown Annotators (VPAnn), the logical model we will use to represent queries over nested documents and study their expressive power. In Section 3.4, we state the main result of the chapter and discuss some of its extent. In Section 3.5, we show the data structure used by the algorithm, called Enumerable Compact Set, which stores the output and handles the enumeration efficiently. Section 3.6 presents and analyzes the main algorithm. In Section 3.7, we link our results to document spanners. We close this chapter in Section 3.8 by discussing some related work.

3.1. Preliminaries

We start by stating the definitions of well-nested words and visibly pushdown automata. Further, we state some basic definitions that will be useful throughout the chapter.

Well-nested words and streams. Besides the usual definitions for strings, we will work over a *structured alphabet* $\Sigma = (\Sigma^{<}, \Sigma^{>}, \Sigma^{|})$ comprised of three disjoint sets $\Sigma^{<}$, $\Sigma^{>}$, and $\Sigma^{|}$ that contain *open*, *close*, and *neutral* symbols respectively[1]. Furthermore, we will denote symbols in $\Sigma^{<}$, $\Sigma^{>}$ or $\Sigma^{|}$ by $<a$, $a>$, and a, respectively. On the other hand, we will use s to denote any symbol in $\Sigma^{<}$, $\Sigma^{>}$, or $\Sigma^{|}$. The set of *well-nested words* over Σ, denoted as $\Sigma^{<*>}$, is defined as the smallest set satisfying the following rules: $\Sigma^{|} \cup \{\varepsilon\} \subseteq \Sigma^{<*>}$, if $w_1, w_2 \in \Sigma^{<*>}$ then $w_1 \cdot w_2 \in \Sigma^{<*>}$, and if $w \in \Sigma^{<*>}$ and $<a \in \Sigma^{<}$ and $b> \in \Sigma^{>}$ then $<a \cdot w \cdot b> \in \Sigma^{<*>}$. In addition, we will work with prefixes of well-nested words, that we call *prefix-nested words*. We denote the set of prefixes of $\Sigma^{<*>}$ as $\mathsf{prefix}(\Sigma^{<*>})$. Sometimes, we will use $w[i]$ to refer to the i-th symbol in a word w.

Visibly pushdown automata. A *visibly pushdown automaton* (Alur & Madhusudan, 2004b) (VPA) is a tuple:

$$\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, I, F)$$

where Q is a finite set of states, $\Sigma = (\Sigma^{<}, \Sigma^{>}, \Sigma^{|})$ is the structured input alphabet, Γ is the stack alphabet, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and

$$\Delta \subseteq (Q \times \Sigma^{<} \times Q \times \Gamma) \cup (Q \times \Sigma^{>} \times \Gamma \times Q) \cup (Q \times \Sigma^{|} \times Q)$$

is the transition relation. A transition $(q, <a, q', \gamma)$ is a *push-transition* where on reading $<a \in \Sigma^{<}$, the symbol γ is pushed onto the stack and the current state switches from q to q'. Conversely, $(q, a>, \gamma, q')$ is a *pop-transition* where on reading $a> \in \Sigma^{>}$ from the input and γ from the top of the stack, the current state changes from q to q', and the symbol γ is popped. Lastly, we say that (q, a, q') is a *neutral transition* if $a \in \Sigma^{|}$, where there is no stack operation.

A *stack* is a finite sequence σ over Γ where the top of the stack is the first symbol on σ. For a well-nested word $w = s_1 \cdots s_n$ in $\Sigma^{<*>}$, a *run* of $\mathcal{A}$ on w is a sequence:

$$\rho = (q_1, \sigma_1) \xrightarrow{s_1} (q_2, \sigma_2) \xrightarrow{s_2} \ldots \xrightarrow{s_n} (q_{n+1}, \sigma_{n+1}),$$

[1] In (Alur & Madhusudan, 2004b; Filiot, Raskin, Reynier, Servais, & Talbot, 2018) these sets are named *call*, *return*, and *local*, respectively.

where each $q_j \in Q$ and $\sigma_j \in \Gamma^*$ for every $j \in [1, n]$, $q_1 \in I$, $\sigma_1 = \varepsilon$, and for every $i \in [1, n]$ the following holds: (1) if $s_i \in \Sigma^{<}$, then there is $\gamma \in \Gamma$ such that $(q_i, s_i, q_{i+1}, \gamma) \in \Delta$ and $\sigma_{i+1} = \gamma\sigma_i$, (2) if $s_i \in \Sigma^{>}$, then there is $\gamma \in \Gamma$ such that $(q_i, s_i, \gamma, q_{i+1}) \in \Delta$ and $\sigma_i = \gamma\sigma_{i+1}$, and (3) if $s_i \in \Sigma^{|}$, then $(q_i, s_i, q_{i+1}) \in \Delta$ and $\sigma_{i+1} = \sigma_i$. A run ρ is *accepting* if $q_{n+1} \in F$. A well-nested word $w \in \Sigma^{<*>}$ is *accepted by* a VPA $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. The *language* $\mathcal{L}(\mathcal{A})$ is the set of well-nested words accepted by $\mathcal{A}$. Note that if ρ is a run of $\mathcal{A}$ on a well-nested word w, then $\sigma_{n+1} = \varepsilon$. A set of well-nested words $\mathcal{L} \subseteq \Sigma^{<*>}$ is called a *visibly pushdown language* if there exists a VPA $\mathcal{A}$ such that $\mathcal{L} = \mathcal{L}(\mathcal{A})$.

A VPA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, I, F)$ is said to be *deterministic* if $|I| = 1$ and δ is a function subset of the set $(Q \times \Sigma^{<} \to Q \times \Gamma) \cup (Q \times \Sigma^{>} \times \Gamma \to Q) \cup (Q \times \Sigma^{|} \to Q)$. We also say that $\mathcal{A}$ is *unambiguous* if, for every $w \in \mathcal{L}(\mathcal{A})$, there exists exactly one accepting run of $\mathcal{A}$ on w. In (Alur & Madhusudan, 2004b), it is shown that for every VPA there exists an equivalent deterministic VPA of at most exponential size.

Streams. A *stream* $\mathcal{S} = s_1 s_2 \cdots$ is an infinite sequence where $s_i \in \Sigma^{<} \cup \Sigma^{>} \cup \Sigma^{|}$. Given a stream $\mathcal{S} = s_1 s_2 \ldots$ and positions $i, j \in \mathbb{N} \setminus \{0\}$ such that $i \leq j$, the word $\mathcal{S}[i, j]$ is the sequence $s_i s_{i+1} \cdots s_j$. We also use this notation to refer to contiguous subsequences of infinite sequences that are not composed of symbols in Σ. For a stream $\mathcal{S}$, we will always assume that for each $i \in \mathbb{N} \setminus \{0\}$, the word $\mathcal{S}[1, i]$ is a prefix of some nested word (i.e., it can be completed to form a nested word). For our streaming algorithms, we also consider the method `yield`$\mathcal{S}$ which can be called to access each element of $\mathcal{S}$ sequentially.

3.2. Streaming evaluation with output-linear delay

Our first task in this chapter is to define a notion of a *streaming enumeration problem*: evaluating a query over a stream and enumerating the outputs with bounded delay. Towards this goal, we want to restrict the resources used (i.e., time and space) and impose strong guarantees on the delay. As our gold standard, we consider the notion of *output-linear delay* defined in (Florenzano et al., 2020). This notion is a refinement of the definition of constant-delay (Segoufin, 2013) or linear-delay (Courcelle, 2009) enumeration that better fits our purpose. Altogether, our plan for this section is to define a streaming enumeration problem and then provide a notion of efficiency that a solution for this problem should satisfy.

We adopt the setting of *relations* (Jerrum, Valiant, & Vazirani, 1986; Arenas, Croquevielle, Jayaram, & Riveros, 2019) to formalize a streaming enumeration problem. First, we need to define what is an enumeration problem outside the stream setting, and we will use a subtler definition than the one described in the preliminaries. Let Ω be an alphabet. An *enumeration problem* is a relation $R \subseteq (\Omega^* \times \Omega^*) \times \Omega^*$. For each pair $((q, x), y) \in R$ we view (q, x) as the input of the problem and y as a possible output for (q, x). Furthermore, we call q the *query* and x the *data*. This separation allows for a fine-grained analysis of the query complexity and data complexity of the problem. For an instance (q, x) we define the set $[\![q]\!]_R(x) = \{y \mid ((q, x), y) \in R\}$ of all outputs of evaluating q over x.

A *streaming enumeration problem* is an extension of an enumeration problem R where the input is a pair $(q, \mathcal{S})$ such that $\mathcal{S}$ is an infinite sequence of elements in Ω. We identify two ways of extending an enumeration problem R that differ in the output sets that are desired at each position in the stream:

(i) The *streaming full-enumeration problem* for R is one where the objective is to enumerate the set $[\![q]\!]_R(\mathcal{S}[1, n])$ at each position $n \geq 1$.

(ii) A *streaming Δ-enumeration problem* for R is one where the objective is to enumerate the set $[\![q]\!]^{\Delta}_R(\mathcal{S}[1, n]) = [\![q]\!]_R(\mathcal{S}[1, n]) \setminus \bigcup_{i<n} [\![q]\!]_R(\mathcal{S}[1, i])$ at each position $n \geq 1$.

These versions give us two different ways of returning the outputs. Both notions have been studied previously in the context of incremental view maintenance (Chirkova & Yang, 2012) and more recently, for dynamic query evaluation (Idris, Ugarte, & Vansummeren, 2017; Berkholz, Keppeler, & Schweikardt, 2017). For the sake of simplification, in the following, we provide all definitions for the full-enumeration scenario. Note that all definitions can be extended to Δ-enumeration by changing $[\![q]\!]_R$ to $[\![q]\!]_R^{\Delta}$.

We now turn to our efficiency notion for solving a streaming enumeration problem like the above. Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function. We say that $\mathcal{E}$ is a *streaming evaluation algorithm* for R with f*-update-time* if $\mathcal{E}$ operates in the following way: it receives a query q and reads the stream $\mathcal{S}$ by calling the $\texttt{yield}\mathcal{S}$ method sequentially. After the n-th call to $\texttt{yield}\mathcal{S}$, the algorithm processes the n-th data symbol in two phases:

- In the first phase, called the *update* phase, the algorithm updates a data structure D with the read symbol, and the time spent is bounded by $\mathcal{O}(f(|q|))$.
- The second phase, called the *enumeration* phase, occurs immediately after each update phase and outputs $[\![q]\!]_R(\mathcal{S}[1,n])$ using D. During this phase the algorithm: (1) writes $\#y_1\#y_2\#\cdots\#y_m\#$ to the output registers where # is a distinct separator symbol not contained in Ω, and $y_1, y_2, \ldots, y_m$ is an enumeration (without repetitions) of the set $[\![q]\!]_R(\mathcal{S}[1,n])$, (2) it writes the first # as soon as the enumeration phase starts, and (3) it stops immediately after writing the last #.

The purpose of separating $\mathcal{E}$'s operation into an update and enumeration phase is to make an output-sensitive analysis of $\mathcal{E}$'s complexity. Moreover, from a user perspective, this separation allows running the enumeration phase without interrupting the update phase. That is, the user could execute the enumeration phase in a separate machine, and its running time only depends on how many outputs she wants to enumerate.

We omit the time analysis of the enumeration phase and the definition of delay as it was included in Section 2.3.

As the last ingredient, we define how to measure the *memory space* of a streaming evaluation. Note that after the n-th call a streaming evaluation algorithm with f-update-time will necessarily use at most $\mathcal{O}(n \cdot f(|q|))$ space. As a refinement of this bound, we say that this algorithm *uses g-space* over a query q and stream $\mathcal{S}$ if the number of bits used by it after the n-th call is in $\mathcal{O}(g(|q|, \mathcal{S}[1,n]))$.

Given a streaming enumeration problem, we say that it can be solved with update-time f, output-linear delay, and in g-space if an algorithm such as the one described above exists. For Δ-enumeration, the notion of streaming evaluation algorithm also applies, even though it could be the case that one can find such an algorithm for full-enumeration but not for Δ-enumeration, and vice versa. To finish, we would like to remark that the enumeration problem and solutions provided here are a formal refinement of the algorithmic notions proposed in the literature of streaming evaluation (Gauwin, Niehren, & Tison, 2009b), dynamic query evaluation (Berkholz et al., 2017; Idris et al., 2017), and complex event processing (Grez, Riveros, & Ugarte, 2019; Grez & Riveros, 2020).

3.3. Visibly pushdown annotators

In this section, we present the definition of *visibly pushdown annotators* (VPAnn), which are an extension of visibly pushdown automata to produce outputs. We use VPAnn as our computational model to represent queries with output. This model is general enough to include the regular core of most query languages for nested documents, like XML or JSON, whose expressive power is included in Monadic Second Order logic (MSO) (see next section for a discussion). In the following, we present the VPAnn model and provide some examples.

A *visibly pushdown annotator* (VPAnn) is a tuple $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$ where Q, Σ, Γ, I, and F are the same as for VPA, Ω is the output alphabet of *annotations* such that $\Sigma \cap (\Sigma \times \Omega) = \emptyset$, and:

$$\begin{array}{rcl} \Delta & \subseteq & Q \times \big(\Sigma^{<} \cup (\Sigma^{<} \times \Omega)\big) \times Q \times \Gamma \;\;\cup \\ & & Q \times \big(\Sigma^{>} \cup (\Sigma^{>} \times \Omega)\big) \times \Gamma \times Q \;\;\cup \\ & & Q \times \big(\Sigma^{|} \cup (\Sigma^{|} \times \Omega)\big) \times Q \end{array}$$

is the transition relation. A symbol $s \in \Sigma^{<} \cup \Sigma^{>} \cup \Sigma^{|}$ is an *input symbol* that the machine reads and $\text{ɤ} \in \Omega$ is an *annotation symbol* that the machine produces. Intuitively, the second component of each transition in Δ decides non-deterministically whether the machine reads and annotates an input symbol (i.e., $\Sigma \times \Omega$) or just reads an input symbol without annotating it (i.e., Σ).

A run ρ of $\mathcal{T}$ over a well-nested word $w = s_1 s_2 \cdots s_n \in \Sigma^{<*>}$ is a sequence of the form:

$$\rho = (q_1, \sigma_1) \xrightarrow{b_1} (q_2, \sigma_2) \xrightarrow{b_2} \ldots \xrightarrow{b_n} (q_{n+1}, \sigma_{n+1})$$

where $q_i \in Q$, $\sigma_i \in \Gamma^*$, $q_1 \in I$, $\sigma_1 = \varepsilon$, and either $b_i = s_i$ or $b_i = (s_i, \text{ɤ})$ for some $\text{ɤ} \in \Omega$, for every $i \in [1, n]$. In addition, the following holds for every $i \in [1, n]$:

(i) if $b_i \in \Sigma^{<} \cup (\Sigma^{<} \times \Omega)$, then $(q_i, b_i, q_{i+1}, \gamma) \in \Delta$ for some $\gamma \in \Gamma$ and $\sigma_{i+1} = \gamma\sigma_i$,

(ii) if $b_i \in \Sigma^{>} \cup (\Sigma^{>} \times \Omega)$, then $(q_i, b_i, \gamma, q_{i+1}) \in \Delta$ for some $\gamma \in \Gamma$ and $\sigma_i = \gamma\sigma_{i+1}$, and

(iii) if $b_i \in \Sigma^{|} \cup (\Sigma^{|} \times \Omega)$, then $(p_i, b_i, q_{i+1}) \in \Delta$ and $\sigma_i = \sigma_{i+1}$.

We call a pair (q_i, σ_i) a *configuration* of ρ. We say that the run is *accepting* if $q_{n+1} \in F$. Regarding the output of an accepting run ρ like above, we define the output of ρ as:

$$\mathsf{out}(\rho) \ = \ \mathsf{out}(b_1, 1) \cdot \ldots \cdot \mathsf{out}(b_n, n)$$

where $\mathsf{out}((s_i, \text{ɤ}), i) = (\text{ɤ}, i)$ when $b_i = (s_i, \text{ɤ})$ and $\mathsf{out}(s_i, i) = \varepsilon$ when $b_i = s_i$. Then, given a VPAnn $\mathcal{T}$ and a $w \in \Sigma^{<*>}$, we define the set $[\![\mathcal{T}]\!](w)$ of all outputs of $\mathcal{T}$ over w as:

$$[\![\mathcal{T}]\!](w) \ = \ \{\mathsf{out}(\rho) \mid \rho \text{ is an accepting run of } \mathcal{T} \text{ over } w\}.$$

As is usual with automata, VPAnn are depicted as directed graphs. We represent an open transition $(x, {<}s, \text{ɤ}, y, \gamma)$ by an edge from state x to state y with the label ${<}s/\gamma : \text{ɤ}$, a close transition $(x, s{>}, \text{ɤ}, \gamma, y)$ with the label ${<}s, \gamma : \text{ɤ}$, and a neutral transition $(x, s, \text{ɤ}, y)$ with the label $s : \text{ɤ}$. If the transition is non-annotating (e.g., $(x, {<}s, y, \gamma)$), we omit the ': ɤ'
.

$$\begin{aligned}
&\Sigma = (\{<\}, \{>\}, \emptyset),\ \ \Omega = \{\triangleright, \triangleleft\},\ \ \Gamma = \{\mathsf{X}, \mathsf{Y}\} \\
&Q = \{\mathsf{p}, \mathsf{q}, \mathsf{r}\},\ \ I = \{\mathsf{p}\},\ \ F = \{\mathsf{r}\} \\
&\Delta = \{\ (\mathsf{p}, <, \mathsf{p}, \mathsf{X}),\ (\mathsf{p}, >, \mathsf{X}, \mathsf{p}) \\
&\quad (\mathsf{p}, (<, \triangleright), \mathsf{q}, \mathsf{Y}),\ (\mathsf{q}, <, \mathsf{q}, \mathsf{X}),\ (\mathsf{q}, >, \mathsf{X}, \mathsf{q}) \\
&\quad (\mathsf{q}, (>, \triangleleft), \mathsf{Y}, \mathsf{r}),\ (\mathsf{r}, <, \mathsf{r}, \mathsf{X}),\ (\mathsf{r}, >, \mathsf{X}, \mathsf{r})\ \}
\end{aligned}$$

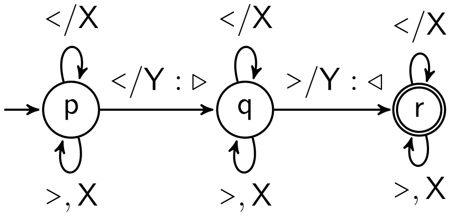


Figure 3.1. An example of a VPAnn $\mathcal{T}$ that marks all pairs of positions that correspond to matching brackets.

Example 3.1. *Consider the VPAnn $\mathcal{T}$ depicted in Figure 3.1. On the left side is its formal definition and on the right is its graphical representation.*

As an important note regarding the notation, this VPAnn in particular uses names $\mathsf{p}, \mathsf{q}, \mathsf{r}$ *(sans serif) for its states. These are not to be confused with the symbols p, q, r that we use elsewhere for referring to a generic state in an arbitrary VPAnn.*

The VPAnn $\mathcal{T}$ in Figure 3.1 receives a well-nested word over the input alphabet $\Sigma = (\{<\}, \{>\}, \emptyset)$ and marks all pairs of positions that correspond to matching brackets. For the sake of illustration, below we show the three accepting runs of $\mathcal{T}$ over the word $w = \, <<><>>$.

$$\begin{aligned}
&\mathsf{p}, \varepsilon \xrightarrow{(<,\triangleright)} \mathsf{q}, \mathsf{Y} \xrightarrow{<} \mathsf{q}, \mathsf{XY} \xrightarrow{>} \mathsf{q}, \mathsf{Y} \xrightarrow{<} \mathsf{q}, \mathsf{XY} \xrightarrow{>} \mathsf{q}, \mathsf{Y} \xrightarrow{(>,\triangleleft)} \mathsf{r}, \varepsilon \\
&\mathsf{p}, \varepsilon \xrightarrow{<} \mathsf{p}, \mathsf{X} \xrightarrow{(<,\triangleright)} \mathsf{q}, \mathsf{YX} \xrightarrow{(>,\triangleleft)} \mathsf{r}, \mathsf{X} \xrightarrow{<} \mathsf{r}, \mathsf{XX} \xrightarrow{>} \mathsf{r}, \mathsf{X} \xrightarrow{>} \mathsf{r}, \varepsilon \\
&\mathsf{p}, \varepsilon \xrightarrow{<} \mathsf{p}, \mathsf{X} \xrightarrow{<} \mathsf{p}, \mathsf{XX} \xrightarrow{>} \mathsf{p}, \mathsf{X} \xrightarrow{(<,\triangleright)} \mathsf{q}, \mathsf{YX} \xrightarrow{(>,\triangleleft)} \mathsf{r}, \mathsf{X} \xrightarrow{>} \mathsf{r}, \varepsilon
\end{aligned}$$

Then the reader can check that the output set of running $\mathcal{T}$ over w is:

$$[\![\mathcal{T}]\!](w) = \big\{ (\triangleright, 1)(\triangleleft, 6),\ (\triangleright, 2)(\triangleleft, 3),\ (\triangleright, 4)(\triangleleft, 5) \big\}.$$

Strictly speaking, our definition of VPAnn is richer than the one studied in (Filiot et al., 2018). In our definition of VPAnn each output element is a tuple (ϑ, i) where ϑ is the symbol and i is the output position, where for a standard VPT (Filiot et al., 2018) an output element is just the symbol ϑ. The extension presented here is indeed important for practical applications like in document spanners (Florenzano et al., 2020; Amarilli et al.,

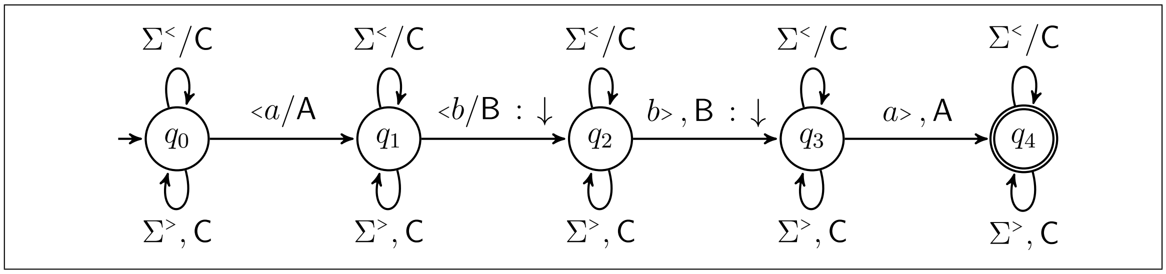


Figure 3.2. A VPAnn implementing the CPath query $\mathcal{Q}$ = `//a/b`. Its input alphabet consists of the sets $\Sigma^{<}$ and $\Sigma^{>}$ of open and closed tags, respectively, $\{\mathsf{A},\mathsf{B},\mathsf{C}\}$ is the stack alphabet, and $\{\downarrow\}$ is the output alphabet.

2019a) or in XML query evaluation (Bar-Yossef, Fontoura, & Josifovski, 2005; Shalem & Bar-Yossef, 2008), as we show next.

Expressiveness of visibly pushdown annotators. How useful are VPAnn as a computational model for representing queries over nested words? To motivate this question, recall the XPath query $\mathcal{Q}$ = `//a/b` presented in the introduction, whose outputs are all pairs of nested tags `<b>` and `</b>` inside some nested tags `<a>` and `</a>`. In Figure 3.2 we show a VPAnn equivalent to $\mathcal{Q}$. The VPAnn processes an XML document by moving non-deterministically from q_0 to q_1 when it reads an open tag `<a>` and then to q_2 when it reads an open tag `<b>` and marks it with $\downarrow$. Then it moves to q_3 when it reads the matching close tag `</b>` marking it with $\downarrow$, and it moves to an accepting state when it reads the corresponding close tag `</a>`. The reader can easily check that this VPAnn marks all pairs of b-tags satisfying $\mathcal{Q}$.

As in the previous example, one could ask whether VPAnn can encode every XPath query or any other query language over nested documents (e.g., JSON). To answer this question, we study the expressiveness of VPAnn by comparing it to MSO over nested words. We show that VPAnn is equally expressive to MSO over nested words with open MSO variables. Given that fragments of query languages over nested documents (e.g., navigational XPath (ten Cate & Marx, 2007), JSON Navigational Logic (Bourhis, Reutter, & Vrgoc, 2020)) are included in MSO logics, this result shows that VPAnn is a useful computational model to express query evaluation problems over nested documents. We continue this section by defining MSO over nested words in order to state and prove the main result of the chapter.

In (Alur & Madhusudan, 2004b), it was shown that VPA describe the same class of queries as MSO over nested words, called $\text{MSO}_{\mathsf{match}}$. Formally, fix a structured alphabet Σ and let $w \in \Sigma^{<*>}$ be a word of length n. We encode w as a *logic structure*:

$$\big(A,\ \leq,\ \{P_a\}_{a\in\Sigma},\ \mathsf{match}\big)$$

where $A = [1, n]$ is the domain, $\leq$ is the total order over $[1, n]$, each P_a is a unary predicate encoding the appearance of letter $a \in \Sigma$ such that $P_a = \{i \mid w[i] = a\}$, and match is a binary relation over $[1, n]$ that corresponds to the matching relation of open and close symbols. Formally, for every $i, j \in A$, $\mathsf{match}(i, j)$ is true if, and only if, $w[i]$ is an open symbol and $w[j]$ is its matching close symbol. Then, *an $\text{MSO}_{\mathsf{match}}$ formula φ over Σ* is given by the following syntax:

$$\varphi \ := \ P_a(x) \ \mid \ x \in X \ \mid \ x \leq y \ \mid \ \mathsf{match}(x, y) \ \mid \ \neg\varphi \ \mid \ \varphi \vee \varphi \ \mid \ \exists x.\varphi \ \mid \ \exists X.\varphi$$

where $a \in \Sigma$, x and y are first-order variables and X is a monadic second order (MSO) variable. We will use $\bar{x}$ as a shorthand for a list of first-order variables $x_1, \ldots, x_\ell$ and $\bar{X}$ as a shorthand for a list of MSO variables $X_1, \ldots, X_m$. Then, we can write $\varphi(\bar{x}, \bar{X})$ to denote an $\text{MSO}_{\mathsf{match}}$ formula φ where $\bar{x}$ and $\bar{X}$ are the free variables of φ. By some abuse of notation, we will also use $\bar{x}$ and $\bar{X}$ as sets and write $\bar{x} \cup \bar{X}$ to denote the union of $\bar{x}$ and $\bar{X}$.

An *assignment* σ for w is a function $\sigma \colon \bar{x} \cup \bar{X} \to 2^{[1,n]}$ such that $|\sigma(x)| = 1$ for every $x \in \bar{x}$. Here, we treat first-order variables as a special case of MSO variables. As usual, we denote by $\mathsf{dom}(\sigma) = \bar{x} \cup \bar{X}$ the domain of the function σ. Then we write $(w, \sigma) \models \varphi(\bar{x}, \bar{X})$ when σ is an assignment over w, $\mathsf{dom}(\sigma) = \bar{x} \cup \bar{X}$, and w satisfies $\varphi(\bar{x}, \bar{X})$ when each variable in $\bar{x} \cup \bar{X}$ is instantiated by σ (see (Libkin, 2004) for the formal semantics of MSO). Given a formula $\varphi(\bar{x}, \bar{X})$, we define:

$$[\![\varphi]\!](w) \;=\; \{\sigma \mid (w, \sigma) \models \varphi(\bar{x}, \bar{X})\}.$$

For the sake of simplification, from now on we will only use $\bar{X}$ to denote the free variables of $\varphi(\bar{X})$ and use $X \in \bar{X}$ for a first-order or monadic second-order variable.

Given an assignment σ over w, we can represent σ as a sequence of annotations and positions as follows. First, define the *support* of σ, denoted by $\mathsf{supp}(\sigma)$, as the set of positions mentioned in σ; formally, $\mathsf{supp}(\sigma) = \{i \mid \exists X \in \mathsf{dom}(\sigma) \text{ such that } i \in \sigma(X)\}$. Next, assume that $\mathsf{supp}(\sigma) = \{i_1, \ldots, i_m\}$ such that $i_j < i_{j+1}$ for every $j < m$. Then, we define the *sequence encoding* of σ as:

$$\mathsf{enc}(\sigma) \;=\; (\bar{X}_1, i_1)\,(\bar{X}_2, i_2)\,\ldots\,(\bar{X}_m, i_m)$$

such that $\bar{X}_j = \{X \in \mathsf{dom}(\sigma) \mid i_j \in \sigma(X)\}$ for every $j \leq m$. In other words, we represent σ as an increasing sequence, where each position is labeled with the variables of σ where it belongs.

We can now precisely state the equivalence between VPAnn and queries defined by MSO over nested words. Fix a structured alphabet Σ and a set of MSO variables $\bar{X}$. We say that a VPAnn $\mathcal{T}$ with output alphabet $2^{\bar{X}}$ *is equivalent to* an MSO$_{\mathsf{match}}$ formula $\varphi(\bar{X})$, denoted by $\mathcal{T} \equiv \varphi$, if, and only if, $[\![\mathcal{T}]\!](w) = \{\mathsf{enc}(\sigma) \mid \sigma \in [\![\varphi]\!](w)\}$ for every $w \in \Sigma^{<*>}$. In other words, the outputs of $\mathcal{T}$ are equivalent to the satisfying assignments of φ encoded as sequences.

PROPOSITION 3.1. *For any MSO$_{\mathsf{match}}$ formula $\varphi(\bar{X})$ there exists a VPAnn $\mathcal{T}$ with output alphabet $2^{\bar{X}}$ such that $\mathcal{T} \equiv \varphi$, and conversely.*

PROOF. The following proof is largely based on the proof of Theorem 4 in (Alur & Madhusudan, 2004b). We start by showing how to convert a MSO$_{\mathsf{match}}$ formula $\varphi(\bar{X})$ into an equivalent VPAnn $\mathcal{T}$. For this, we can follow the exact same argument as the *if* direction of the proof in (Alur & Madhusudan, 2004b) and assume we can obtain a VPA

$\mathcal{A}_\varphi$ over the input alphabet $\Sigma^{\bar{X}} = \Sigma \times 2^{\bar{X}}$ whose language is the set of words which encode a valuation σ of $\bar{X}$ along with a word w for which $(w, \sigma) \models \varphi(\bar{X})$. We define a straightforward transformation from $\mathcal{A}_\varphi$ to $\mathcal{T}$ as follows. Let t be a transition in $\mathcal{A}_\varphi$ and let (a, V) be its input symbol with $V \in 2^{\bar{X}}$. If $V \neq \emptyset$ the corresponding transition t' in $\mathcal{T}$ is kept the same, except it has an input symbol a, and an output symbol V, and if $V = \emptyset$, then t' is obtained by simply replacing (a, V) by a. One can easily check that $\mathcal{T} \equiv \varphi$, proving the first direction.

To prove the other direction, we can convert $\mathcal{T}$ into a VPA $\mathcal{A}_\mathcal{T}$ with input alphabet $\Sigma^{\bar{X}}$ in the opposite way and use the result of (Alur & Madhusudan, 2004b) itself to obtain a MSO$_{\textsf{match}}$ formula with no free variables φ' over the same input alphabet. We replace any instance of $P_{(a,V)}(x)$ in φ by the expression $P_a(x) \wedge \bigwedge_{X \in V} x \in X \wedge \bigwedge_{X \in \bar{X} \setminus V} x \notin X$ to obtain a formula $\varphi(\bar{X})$ over Σ which proves the statement. □

We conclude that VPAnn has the same expressive power as MSO over nested words, which implies that VPAnn can represent the navigational fragments of languages like XPath and JSON Navigational Logic subsumed by MSO. In the next section, we complement this result by stating our main algorithmic results regarding the streaming evaluation of VPAnn. Before that, we introduce a class of visibly pushdown annotators that will be crucial for our algorithmic results.

Deterministic and unambiguous visibly pushdown annotators. We say that a VPAnn $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$ is *input/output deterministic* (I/O-deterministic for short) if $|I| = 1$ and Δ is a partial function of the form:

$$\begin{aligned} \Delta \;\subseteq\; & Q \times \big(\Sigma^{<} \cup (\Sigma^{<} \times \Omega)\big) \to Q \times \Gamma \;\;\cup \\ & Q \times \big(\Sigma^{>} \cup (\Sigma^{>} \times \Omega)\big) \times \Gamma \to Q \;\;\cup \\ & Q \times \big(\Sigma^{|} \cup (\Sigma^{|} \times \Omega)\big) \to Q \end{aligned}$$

Also, we say that $\mathcal{T}$ is *input/output unambiguous* (I/O-unambiguous for short) if for every $w \in \Sigma^{<*>}$ and every $\mu \in [\![\mathcal{T}]\!](w)$ there is exactly one accepting run ρ of $\mathcal{T}$ over w such that $\mu = \mathsf{out}(\rho)$.

Notice that an I/O-deterministic VPAnn is also I/O-unambiguous. Intuitively, a VPAnn is I/O-deterministic (I/O-unambiguous) if, given the input word and a sequence of annotations, the automaton behaves in a deterministic (unambiguous, resp.) manner. The definition is in line with the notion of I/O-deterministic variable automata of (Florenzano et al., 2020) and I/O-unambiguous is a generalization of this idea that is enough for the purpose of our enumeration algorithm.

In the next result, we show that for every VPAnn $\mathcal{T}$ there exists an equivalent I/O-deterministic VPAnn and, therefore, an equivalent I/O-unambiguous VPAnn.

PROPOSITION 3.2. *For every VPAnn $\mathcal{T}$ there exists an I/O-deterministic VPAnn $\mathcal{T}'$ of size $\mathcal{O}(2^{|Q|^2|\Gamma|})$ such that $[\![\mathcal{T}]\!](w) = [\![\mathcal{T}']\!](w)$ for every $w \in \Sigma^{<*>}$.*

PROOF. Let $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$. For the sake of presentation, assume that Δ contains only transitions with an output symbol, the proof can be extended straightforwardly to include transitions with no output symbol. We will construct an I/O-deterministic VPAnn $\mathcal{T}' = (Q', \Sigma, \Gamma', \Omega, \delta^{\text{det}}, S_I, F')$ as follows. Let $Q' = 2^{Q\times Q}$ and $\Gamma' = 2^{Q\times\Gamma\times Q}$. Let $S_I = \{(q,q) \mid q \in I\}$ and let $F' = \{S \mid (p,q) \in S$ for some $p \in I$ and $q \in F\}$. Let δ be defined as follows:

- For $<a \in \Sigma^{<}$ and $o \in \Omega$, $\delta(S, <a, o) = (S', T)$, where:

$T = \{(p,\gamma,q) \mid (p,p') \in S$ and $(p', <a, o, \gamma, q) \in \Delta$ for some $q \in Q\}$,

$S' = \{(q,q) \mid (p,p') \in S$ and $(p', <a, o, \gamma, q) \in \Delta$ for some $p, p' \in Q$ and $\gamma \in \Gamma\}$

- For $a> \in \Sigma^{>}$ and $o \in \Omega$, $\delta(S, a>, o, T) = S'$ where, if $T \subseteq Q \times \Gamma \times Q$, then:

$$S' = \{(p, q) \mid (p, \gamma, p') \in T \text{ and } (p', q') \in S \text{ and } (q', a>, o, \gamma, q) \in \Delta$$
$$\text{for some } p', q' \in Q, \gamma \in \Gamma\},$$

- For $a \in \Sigma^{|}$ and $o \in \Omega$, $\delta(S, a, o) = S'$ where:

$$S' = \{(q, q'') \mid (q, q') \in S \text{ and } (q', a, o, q'') \in \Delta \text{ for some } q' \in Q\}.$$

One can immediately check that this automaton is I/O-deterministic since the transition relation is given as a partial function.

We will prove that $\mathcal{T}$ and $\mathcal{T}'$ are equivalent by induction on well-nested words. To aid our proof, we will introduce a couple of ideas. First, we extend the definition of a run to include sequences that start on an arbitary configuration. Also, given a run

$$\rho = (q_1, \sigma_1) \xrightarrow{b_1} (q_2, \sigma_2) \xrightarrow{b_2} \cdots \xrightarrow{b_n} (q_{n+1}, \sigma_{n+1}),$$

and a span $[i, j\rangle$, define a subrun of ρ as the subsequence

$$\rho[i, j\rangle = (q_i, \sigma_i) \xrightarrow{b_i} (q_{i+1}, \sigma_{i+1}) \xrightarrow{b_{i+1}} \cdots \xrightarrow{b_{j-1}} (q_j, \sigma_j).$$

In this proof, we only consider subruns such that $w[i, j\rangle = s_i s_{i+1} \cdots s_{j-1}$ is a well-nested word. A second definition we will use is that of a VPAnn with arbitrary initial states. Formally, let $q \in Q$. We define $\mathcal{T}_q$ as the VPAnn that simulates $\mathcal{T}$ by starting on the configuration (q, ε). Note that for a run $\rho = (q_1, \sigma_1) \xrightarrow{b_1} \cdots \xrightarrow{b_n} (q_{n+1}, \sigma_{n+1})$ of $\mathcal{T}$ over $w = s_1 \cdots s_n$ and a well-nested span $[i, j\rangle$, the subrun $\rho[i, j\rangle$ is one of the runs of $\mathcal{T}_q$ over $w[i, j\rangle$ modulo σ_i, which is present in all of the stacks in ρ as a common suffix.

We shall prove first that $[\![\mathcal{T}]\!](w) \subseteq [\![\mathcal{T}']\!](w)$ for every well-nested word w. This is done with the help of the following result.

Claim 3.1. *For a well-nested word w, output μ, states $p, q \subseteq Q$, and a set S that contains (p, q), if there is a run of $\mathcal{T}_q$ over w with output μ such that its last state is q', the (only) run of $\mathcal{T}'_S$ over w with output μ ends in a state S' which contains (p, q').*

Proof. We will prove the claim by induction on w. If $w = \varepsilon$, the proof is trivial since $q = q'$. If $w = a \in \Sigma^{|}$ the proof follows straightforwardly from the construction of δ.

If $w, v \in \Sigma^{<*>}$, let $p, q \in Q$, let S be a set that contains (p, q), and let ρ be a run of $\mathcal{T}_q$ over wv with output $\mu \cdot \kappa$, which ends in a state q', for some μ and κ. Our goal is to prove that the run ρ' of $\mathcal{T}'_S$ over $w \cdot v$ with output $\mu \cdot \kappa$ as output ends in a state that contains (p, q'). Let $n = |w|$, $m = |v|$, and let q^w be the last state of the subrun $\rho[1, n+1\rangle$. Consider as well $\rho[n+1, n+m+1\rangle$, which is a run of $\mathcal{T}_{q^w}$ over v with output κ that ends in q'. From the hypothesis two conditions follow: (1) In the run of $\mathcal{T}'_S$ over w with output μ, the last state S' contains (p, q^w), and (2) in the run of $\mathcal{T}'_{S'}$ over v that has κ as output the last state contains (p, q'). It can be seen that ρ' is the concatenation of these two runs, so this proves the claim.

If $w \in \Sigma^{<*>}$, $<a \in \Sigma^{<}$, $b> \in \Sigma^{>}$, let $p, q \in Q$, S be a set that contains (p, q), and let ρ be a run of $\mathcal{T}_q$ over $<awb>$ with output $(\eth_1, 1)\mu(\eth_2, n+2)$ for some $\mu \in \Omega^*$, $\eth_1, \eth_2 \in \Omega$, where $n = |w|$. Let $q, q_2, \ldots, q_{n+2}, q_{n+3}$ be the states of ρ in order. Our goal is to prove that the run ρ' of $\mathcal{T}'_S$ over $<awb>$ with output $(\eth_1, 1)\mu(\eth_2, n+2)$ ends in a state that contains (p, q_{n+3}). Let (q_2, γ) be the second configuration of ρ. This implies that $(q, <a, \eth_1, q_2, \gamma) \in \Delta$ and $(q_{n+2}, b>, \eth_2, \gamma, q_{n+3}) \in \Delta$. Let S' and T be such that $\delta(S, <a, \eth_1) = (S', T)$. Therefore, $(q_2, q_2) \in S'$ and $(p, \gamma, q_2) \in T$. Consider the subrun $\rho[2, n+2\rangle$, which can be found as a run of $\mathcal{T}_{q_2}$ over w with output μ, modulo the stack suffix γ, and note that it ends in q_{n+2}. Since $(q_2, q_2) \in S'$, from the hypothesis it follows that the run of $\mathcal{T}'_{S'}$ over w with output μ ends in a state S'' that contains (q_2, q_{n+2}). This run starts on the configuration (S', ε) and ends in (S'', ε), so a run on the same automaton that starts on (S', T) and reads the same symbols will end in (S'', T), which is the case for the subrun

$\rho'[2, n+2\rangle$. Therefore, the construction of δ implies that (p, q_{n+3}) is contained in the last state of ρ', which proves the claim. □

Now let w be a well-nested word and let $\mu \in [\![\mathcal{T}]\!](w)$. Let ρ be some accepting run of $\mathcal{T}$ over w with output μ, and let $p^* \in I$ be its first state and $q^* \in F$ its ending state. Clearly, ρ is also a run of $\mathcal{T}_p$. Now, let us use the claim over the set $S = S_I$ and states $p = q = p^*$, using the fact that $(p, p) \in S_I$. We obtain that the run of $\mathcal{T}'_{S_I}$ over w with output μ ends in a state S' which contains (p, q) and so, is it accepting. Since $\mathcal{T}'_{S_I} = \mathcal{T}'$, this proves that $[\![\mathcal{T}]\!](w) \subseteq [\![\mathcal{T}']\!](w)$.

To prove that $[\![\mathcal{T}']\!](w) \subseteq [\![\mathcal{T}]\!](w)$ we use a similar result:

CLAIM 3.2. *For a well-nested word w, output μ, states $q, p, q' \subseteq Q$, and a set S that contains (p, q), if the run of $\mathcal{T}'_S$ over w with output μ ends on a state S' that contains (p, q'), then there is a run of $\mathcal{T}_q$ over w with output μ such that its last state is q'.*

PROOF. We will prove the claim by induction on w. If $w = \varepsilon$, the proof is trivial since $q = q'$. If $w = a \in \Sigma^{|}$ the proof follows straightforwardly from the construction of δ.

If $w, v \in \Sigma^{<*>}$, let $p, q, q' \in Q$, let S be a set that contains (p, q), and let ρ be the run of $\mathcal{T}'_S$ over $w \cdot v$ with output $\mu \cdot \kappa$, for some μ and κ, which ends in a state S' that contains (p, q'). Our goal is to prove that there is a run ρ' of $\mathcal{T}_q$ over $w \cdot v$ with output $\mu \cdot \kappa$ such that its last state is q'. Let $n = |w|$, $m = |v|$, and let S^w be the last state of the subrun $\rho[1, n+1\rangle$. Consider as well $\rho[n+1, n+m+1\rangle$, which is also a run of $\mathcal{T}'_{S^w}$ over v with output κ that ends in S'. From the construction of δ, it is clear that if a non-empty state S' follows from S in a run of $\mathcal{T}'$, then S is not empty. Let $(p, q^w) \in S^w$. From the hypothesis two conditions follow: (1) there is a run ρ_1 of $\mathcal{T}_q$ over w with output μ such that its last state is q^w (2) there is a run ρ_2 of $\mathcal{T}_{q^w}$ over v with output κ such that its last state is q'. We then construct ρ' by concatenating ρ_1 and ρ_2 which ends in q', and this proves the claim.

If $w \in \Sigma^{<*>}$, $<a \in \Sigma^{<}$, $b> \in \Sigma^{>}$, let $p, q, q' \in Q$, S be a set that contains (p, q), and let ρ be the run of $\mathcal{T}'_S$ over $<awb>$ with output $(\eth_1, 1)\mu(\eth_2, n+2)$ for some μ and $\eth_1, \eth_2 \in \Omega$, where $n = |w|$. Let $S, S_2, \ldots, S_{n+2}, S_{n+3}$ be the states of ρ in order, and suppose there is a pair $(p, q') \in S_{n+3}$. Our goal is to prove that there is a run ρ' of $\mathcal{T}_q$ over $\text{<}awb\text{>}$ with output $(\vartheta_1, 1)\mu(\vartheta_2, n+2)$ that ends in q'. Let (S_2, T) be the second configuration of ρ. From the construction of δ, there exist $q_2, q_{n+2} \in Q$ and $\gamma \in \Gamma$ such that $(q_{n+2}, b\text{>}, \vartheta_2, \gamma, q_{n+3}) \in \Delta$, $(p, \gamma, q_2) \in T$ and $(q_2, q_{n+2}) \in S_{n+2}$. Since w is well-nested, this T could only have been pushed after a step in the run with label $(\text{<}a, \vartheta_1)$, which implies that $(q, \text{<}a, \vartheta_1, q_2, \gamma) \in \Delta$. This, in turn, means that $(q_2, q_2) \in S_2$. Let us consider the subrun $\rho[2, n+2\rangle$, which is also a run of $\mathcal{T}'_{S_2}$ over w with output μ that ends in S_{n+2} modulo the common stack suffix T. We now have that $(q_2, q_2) \in S_2$ and $(q_2, q_{n+2}) \in S_{n+2}$, and so, from the hypothesis it follows that there is a run ρ'' of $\mathcal{T}_{q_2}$ over w with output μ and ending state q_{n+2}. In a similar fashion as in the previous claim, we modify the run slightly to obtain one that starts and ends on the stack γ. This new run can be easily extended with the transitions $(q, \text{<}a, \vartheta_1, q_2, \gamma), (q_{n+2}, b\text{>}, \vartheta_2, \gamma, q_{n+3}) \in \Delta$, and as a result, we obtain a run ρ' of $\mathcal{T}_q$ that fulfils the conditions of the claim. □

Now, let w be a well nested word and let $\mu \in \llbracket \mathcal{T}' \rrbracket(w)$. Let ρ be a run of $\mathcal{T}'$ over w with output μ that ends in accepting state F, and let $p^* \in I$ and $q^* \in F$ be such that $(p^*, p^*) \in I$ and $(p^*, q^*) \in F$. Note that $\mathcal{T}' = \mathcal{T}'_{S_I}$, and by using the previous claim with $p = q = p^*$ and $q' = q^*$, we obtain that there is a run of $\mathcal{T}_{p^*}$ over w with output μ that ends in state in q^*. Clearly, this is also a run of $\mathcal{T}$, so we obtain that $\mu \in \llbracket \mathcal{T} \rrbracket(w)$. This proves that $\llbracket \mathcal{T}' \rrbracket(w) \subseteq \llbracket \mathcal{T} \rrbracket(w)$.

We conclude that $\llbracket \mathcal{T} \rrbracket(w) = \llbracket \mathcal{T}' \rrbracket(w)$ for every well-nested word w. □

3.4. Results and discussion

In this chapter, we are interested in the following streaming enumeration problem for a class $\mathcal{C}$ of VPAnn (e.g. I/O-deterministic VPAnn).

Problem:	EnumVPAnn[$\mathcal{C}$]
Input:	A VPAnn $\mathcal{T} \in \mathcal{C}$ and $w \in \Sigma^{<*>}$
Output:	Enumerate $[\![\mathcal{T}]\!](w)$

The main result of the chapter is that for the class of I/O-unambiguous VPAnn, the streaming full-enumeration version of this problem can be solved efficiently.

Theorem 3.1. *The streaming full-enumeration problem of* EnumVPAnn *for I/O-unambiguous VPAnn can be solved with update-time* $\mathcal{O}(|Q|^2|\Delta|)$ *and output-linear delay. For the class of all VPAnn, it can be solved with update-time* $\mathcal{O}(2^{|Q|^2|\Delta|})$ *and output-linear delay.*

The general case is basically a consequence of Proposition 3.2 and the enumeration algorithm for I/O-unambiguous VPAnn. For both cases, if the VPAnn is fixed (i.e., in data complexity), then the update-time of the streaming algorithm is constant. In Sections 3.5 and 3.6, we present this algorithm. For the rest of this section, we discuss some further details of this result.

Δ-enumeration. The natural question is how we move from full-enumeration to Δ-enumeration. In fact, we can have Δ-enumeration with a slight loss of efficiency by solving the full-enumeration problem. Specifically, we can show that for any I/O-unambiguous VPAnn $\mathcal{T}$ there is an I/O-unambiguous VPAnn $\mathcal{T}'$ of linear size with respect to $|\mathcal{T}|$ that only outputs new results at each position. Then combining this construction with the algorithm of Theorem 3.1, we derive the following algorithm for Δ-enumeration of VPAnn.

Theorem 3.2. *The streaming Δ-enumeration problem of* EnumVPAnn *for I/O-unambiguous VPAnn can be solved with update-time* $\mathcal{O}(|Q|^2|\Delta|)$ *and output-linear delay. For the general class of VPAnn, it can be solved with update-time* $\mathcal{O}(2^{|Q|^2|\Delta|})$ *and output-linear delay.*

Proof. The proof of the theorem is a consequence of Theorem 3.1 and the following lemma.

Lemma 3.1. *For every I/O-unambiguous VPAnn $\mathcal{T}$ there exists an I/O-unambiguous VPAnn $\mathcal{T}'$ such that $\llbracket\mathcal{T}'\rrbracket(w) = \llbracket\mathcal{T}\rrbracket(w) \setminus \bigcup_{i<|w|}\llbracket\mathcal{T}\rrbracket(w[1, i])$ for every $w \in \Sigma^{<*>}$. Furthermore, the size of $\mathcal{T}'$ is linear in the size of $\mathcal{T}$.*

Let $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$ be an I/O-unambiguous VPAnn. We construct a VPAnn $\mathcal{T}' = (Q', \Sigma, \Gamma, \Omega, \Delta', I, F')$ such that $Q' = Q \times \{1, 2\}$, $I' = I \times \{1\}$, $F' = F \times \{1\}$ and Δ' is as follows:

$$\begin{aligned}
\Delta' = \{&((p, 1), {<}a, o, (q, 1), \gamma), ((p, 2), {<}a, o, (q, 1), \gamma) \mid (p, {<}a, o, q, \gamma) \in \Delta\} \cup \\
&\{((p, 1), {<}a, (q, 1), \gamma), ((p, 2), {<}a, (q, 2), \gamma) \mid (p, {<}a, q, \gamma) \in \Delta \text{ where } p \notin F\} \cup \\
&\{((p, 1), {<}a, (q, 2), \gamma), ((p, 2), {<}a, (q, 2), \gamma) \mid (p, {<}a, q, \gamma) \in \Delta \text{ where } p \in F\} \cup \\
&\{((p, 1), a{>}, o, \gamma, (q, 1)), ((p, 2), a{>}, o, \gamma, (q, 1)) \mid (p, a{>}, o, \gamma, q) \in \Delta\} \cup \\
&\{((p, 1), a{>}, \gamma, (q, 1)), ((p, 2), a{>}, \gamma, (q, 2)) \mid (p, a{>}, \gamma, q) \in \Delta \text{ where } p \notin F\} \cup \\
&\{((p, 1), a{>}, \gamma, (q, 2)), ((p, 2), a{>}, \gamma, (q, 2)) \mid (p, a{>}, \gamma, q) \in \Delta \text{ where } p \in F\} \cup \\
&\{((p, 1), a, o, (q, 1)), ((p, 2), a, o, (q, 1)) \mid (p, a, o, q) \in \Delta\} \cup \\
&\{((p, 1), a, (q, 1)), ((p, 2), a, (q, 2)) \mid (p, {<}a, q) \in \Delta \text{ where } p \notin F\} \cup \\
&\{((p, 1), a, (q, 2)), ((p, 2), a, (q, 2)) \mid (p, {<}a, q) \in \Delta \text{ where } p \in F\}
\end{aligned}$$

The idea behind this construction is to separate the VPAnn in two halves. Each run starts in the first half (marked 1) and once it reaches a final state, it changes into the second half (marked 2). The run then stays on the second half until it sees an output symbol that extends the current output, upon which it returns to the first half. It is straightforward to see that $\llbracket\mathcal{T}'\rrbracket(w) = \llbracket\mathcal{T}\rrbracket(w) \setminus \bigcup_{i<|w|}\llbracket\mathcal{T}\rrbracket(w[1, i\rangle)$.

To show that $\mathcal{T}'$ is I/O-unambiguous, consider a $w \in \Sigma^{<*>}$. Let $\mu \in \llbracket\mathcal{T}'\rrbracket(w)$ and consider two accepting runs ρ_1 and ρ_2 such that $\mathsf{out}(\rho_1) = \mathsf{out}(\rho_2) = \mu$. Let us build a run ρ of $\mathcal{T}$ over w by replacing each state (q, k) in ρ_1 by q. Note that starting from ρ_2 renders the same run because they have the same output and $\mathcal{T}$ is I/O-unambiguous. This implies that $\rho_1 = ((q_1, \ell_1), \sigma_1) \xrightarrow{b_1} \cdots \xrightarrow{b_n} ((q_{n+1}, \ell_{n+1}), \sigma_{n+1})$ and $\rho_2 = ((q_1, k_1), \sigma_1) \xrightarrow{b_1} \cdots \xrightarrow{b_n} ((q_{n+1}, k_{n+1}), \sigma_{n+1})$ for some q_i, σ_i and b_i. Note that $\ell_1 = k_1 = 1$, and suppose there is some i for which $\ell_i = k_i$ and $\ell_{i+1} \neq k_{i+1}$. This is immediately false from the construction above since from any transition of Δ that is from state p, only two transitions are included in Δ': one from $(p, 1)$ and one from $(p, 2)$. This implies that $\rho_1 = \rho_2$ so $\mathcal{T}'$ is I/O-unambiguous. □

We could have considered a more general definition of VPAnn to produce outputs for prefix-nested words. This would be desirable for having some sort of *earliest query answering* (Gauwin et al., 2009b) which is important in practical scenarios. We remark that the algorithm of Theorem 3.1 can be extended for this case at the cost of making the presentation more complicated. For the sake of presentation, we give a brief idea of how to handle this extension after the main part of Section 3.6.

Space lower bounds of evaluating a VPAnn. This last subsection deals with the space used by the streaming evaluation algorithm of Theorem 3.1. Indeed, this algorithm could use linear space in the worst case. In the following, we explore some lower bounds in the space needed by any algorithm and show that this bound is tight for a certain type of VPAnn.

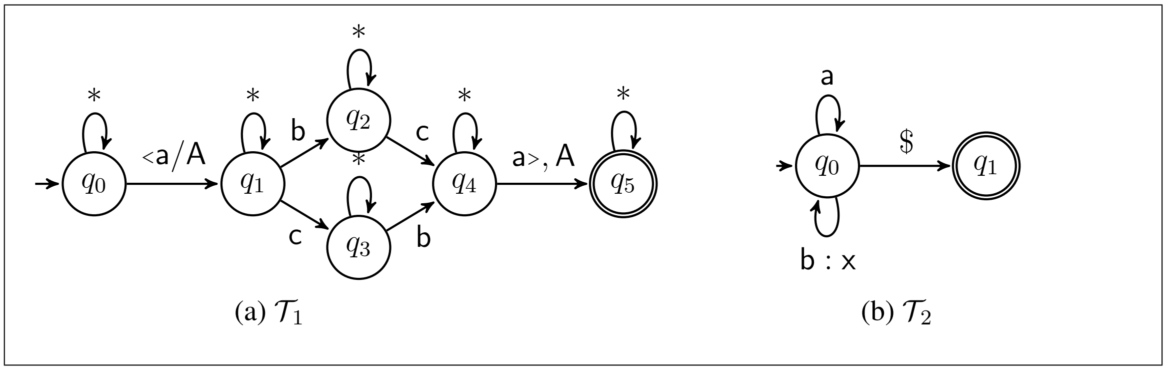
To study the minimum number of bits needed to solve ENUMVPANN we need to introduce some definitions. Fix a VPAnn $\mathcal{T}$ and $w \in \mathsf{prefix}(\Sigma^{<*>})$. Let $\mathsf{outputweight}(\mathcal{T}, w)$ be the number of positions less than $|w|$ that appear in some output of $[\![\mathcal{T}]\!](w \cdot w')$ for some $w \cdot w' \in \Sigma^{<*>}$. Furthermore, for a well-nested word u let $\mathsf{depth}(u)$ be the maximum number of nesting pairs inside u. Formally, $\mathsf{depth}(u) = 0$ for $u \in \Sigma^{|*}$, $\mathsf{depth}(u_1 \cdot u_2) = \max\{\mathsf{depth}(u_1), \mathsf{depth}(u_2)\}$, and $\mathsf{depth}(<a \cdot u \cdot b>) = \mathsf{depth}(u) + 1$. For $w \in \mathsf{prefix}(\Sigma^{<*>})$, we define $\mathsf{depth}(w) = \min\{\mathsf{depth}(w') \mid w' \in \Sigma^{<*>}$ and w is a prefix of $w'\}$. Below, we provide some worst-case space lower bounds for ENUMVPANN that are dependent on $\mathsf{outputweight}(\mathcal{T}, w)$ and $\mathsf{depth}(w)$.

PROPOSITION 3.3. *(a) There exists a VPAnn $\mathcal{T}_1$ such that every streaming evaluation algorithm for* ENUMVPANN *with input $\mathcal{T}_1$ and $\mathcal{S}$ requires $\Omega(\mathsf{depth}(\mathcal{S}[1,n]))$ bits of space. (b) There exists a VPAnn $\mathcal{T}_2$ such that every streaming evaluation algorithm for* ENUMVPANN *with input $\mathcal{T}_2$ and $\mathcal{S}$ requires $\Omega(\mathsf{outputweight}(\mathcal{T}_2, \mathcal{S}[1,n]))$ bits of space.*

PROOF. (a) The existence and lower bound for $\mathcal{T}_1$ is a corollary of Theorem 4.5 in (Bar-Yossef et al., 2007). The proof of this result implies that for the XPath query $Q = //\mathsf{a}[\mathsf{b} \texttt{ and } \mathsf{c}]$, any streaming algorithm that verifies if an XML document matches Q (the problem BOOLEVAL$_Q$) and any integer $r \geq 1$, there exists a document of depth at most $r + C$, where C is a constant value, on which the algorithm requires $\Omega(r)$ bits of space.

The VPAnn $\mathcal{T}_1$ is shown in Figure 3.3a. It can simulate the query Q for a direct mapping ν of the documents that are constructed in (Bar-Yossef et al., 2007), where $\nu(\langle \mathsf{a} \rangle) = \texttt{<a}$, $\nu(\langle / \mathsf{a} \rangle) = \texttt{a>}$, $\nu(\langle \mathsf{b}/ \rangle) = \mathsf{b}$, and $\nu(\langle \mathsf{c}/ \rangle) = \mathsf{c}$. Note that for any well-nested document w the set $[\![\mathcal{T}]\!](w)$ is either empty or $\{\varepsilon\}$. Now suppose there is a streaming evaluation algorithm $\mathcal{E}$ that solves ENUMVPANN. We can solve BOOLEVAL$_Q$ by receiving an XML stream $\mathcal{S}$, and running $\mathcal{E}$ with input $\mathcal{T}$ while applying the mapping ν to each character. Let w be the resulting string. At the end of the stream, we enumerate the set $[\![\mathcal{T}]\!](w)$ and it will enumerate the output ε iff $\mathcal{S}$ matches Q. We conclude that $\mathcal{E}$ runs in $\Omega(r)$ space.

(b) The existence and lower bound for $\mathcal{T}_2$ uses the main ideas of the proof of Theorem 1 in (Bar-Yossef et al., 2005). Here, the authors describe a set-computing communication complexity problem. In the problem $\mathcal{P}$, Alice and Bob compute a two-argument function $p(\cdot,\cdot)$, defined as follows. Alice's input is a subset $A \subseteq \{1,\ldots,k\}$, Bob's input is a bit $\beta \in \{0,1\}$, and $p(A,\beta)$ is defined to be A, if $\beta = 1$, and $\emptyset$ otherwise. Proposition 1 in (Bar-Yossef et al., 2005) proves that the one-way communication complexity of $\mathcal{P}$ is at least k.



(a) $\mathcal{T}_1$

(b) $\mathcal{T}_2$

Figure 3.3. VPAnn used in the proof of Proposition 3.3. On $\mathcal{T}_1$, a loop over a node p labeled by $*$ represents the four transitions $(p, \texttt{<a}, p, \mathsf{X})$, $(p, \texttt{a>}, \mathsf{X}, p)$, (p, b, p) and (p, c, p).

Let $\mathcal{T}_2 = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$ be a VPAnn such that $\Sigma^{|} = \{\mathsf{a}, \mathsf{b}, \$\}$, $\Omega = \{\mathsf{x}\}$, and Q, Δ, I, F be as presented in Figure 3.3b. Let $w \in (\Sigma^{|})^*$ be a word such that $i_1 < i_2 < \ldots < i_k \leq |w|$ are all positions of w where $w[i_\ell] = b$ for every $\ell \leq k$. Then one can check that $\mathcal{T}_2$ defines the following function:

$$[\![\mathcal{T}_2]\!](w) = \begin{cases} \{(\mathsf{x}, i_1) \ldots (\mathsf{x}, i_k)\} & \text{if } w \text{ ends in } \$ \\ \emptyset & \text{otherwise.} \end{cases}$$

Consider an arbitrary algorithm $\mathcal{E}$ that solves ENUMVPANN with input $\mathcal{T}_2$. We will now present a reduction that creates a protocol for $\mathcal{P}$ which makes use of the algorithm $\mathcal{E}$. Here, Alice receives the set A and generates a word w of size k such that $w[i] = \mathsf{b}$ if $i \in A$ and $w[i] = \mathsf{a}$ otherwise. Alice then executes $\mathcal{E}$ on input $\mathcal{T}_2$ and w as the first k characters of a stream. She sends the state of the algorithm to Bob, who receives the bit β, and does the following: If $\beta = 1$ he continues running $\mathcal{E}$ as if the last character of the input was \$. If $\beta = 0$, he stops executing $\mathcal{E}$ immediately. In either case, the output given by $\mathcal{E}$ contains all the information necessary to compute the set $p(A, \beta)$, so the reduction is correct. This proves that $\mathcal{E}$ requires at least k bits for an input of size less than k, and so $\mathcal{E}$ for any $n \geq 1$, requires at least n bits of space in a worst-case stream $\mathcal{S}$, which is in $\Omega(\mathsf{outputweight}(\mathcal{T}_2, S[1, n]))$. $\square$

In (Bar-Yossef et al., 2005, 2007), the authors provide lower bounds on the amount of space needed for evaluating XPath in terms of the nesting and the concurrency (see (Bar-Yossef et al., 2005) for a definition). One can show that the output weight of $\mathcal{T}$ and w is always above the concurrency of $\mathcal{T}$ and w. Despite this, one can check that both notions coincide for the space lower bound given in Proposition 3.3.

The previous results show that, in the worst case, any streaming evaluation algorithm for VPAnn will require space of at least the depth of the document or the output weight. To show that Theorem 3.1 is optimal in the worst-case, we need to consider a further assumption of our VPAnn. We say that a VPAnn $\mathcal{T}$ is *trimmed* (Caralp, Reynier, & Talbot, 2015) if for every $w \in \mathsf{prefix}(\Sigma^{<*>})$ and every (partial) run ρ of $\mathcal{T}$ over w, there exists w' and an accepting run ρ' of $\mathcal{T}$ over $w \cdot w'$ such that ρ is a prefix of ρ'. This notion is the analog of trimmed non-deterministic automata. Similarly to Proposition 3.2, one can show that for every VPAnn $\mathcal{T}$ there exists a trimmed I/O-deterministic VPAnn $\mathcal{T}'$ equivalent to $\mathcal{T}$ (i.e., by extending the construction in (Caralp et al., 2015) to VPAnn). The next result shows that, if the input to ENUMVPANN is a trimmed I/O-unambiguous VPAnn, then the memory footprint is at most the maximum between the depth and output weight of the input.

PROPOSITION 3.4. *The streaming enumeration problem of* ENUMVPANN *for the class of trimmed I/O-unambiguous VPAnn can be solved with update-time* $\mathcal{O}(|Q|^2|\Delta|)$*, output-linear delay, and* $\mathcal{O}(\max\{\mathsf{depth}(\mathcal{S}[1,n]), \mathsf{outputweight}(\mathcal{T}, \mathcal{S}[1,n])\} \times |Q|^2|\Delta|)$ *space for every stream* $\mathcal{S}$.

PROOF. Before reading this proof, it is important to note that the argument assumes the understanding of Algorithm 2. Furthermore, this proof uses the notation introduce in Section 3.6.

First of all, note that the time bounds are implied by Theorem 3.1, so we will restrict to prove the space bounds. Algorithm 2 has an update phase and an enumeration phase, and the enumeration phase only processes the data structure that was built on the update phase, using at most linear extra space, as is explained in Section 3.5. As such, we will prove that Algorithm 2 on input $(\mathcal{T}, w)$ uses $\mathcal{O}((\mathsf{depth}(w) + \mathsf{outputweight}(\mathcal{T}, w)) \times |Q|^2|\Delta|)$ space at every point in its execution, which implies the statement of the proposition, where $w = \mathcal{S}[1, n]$ for some stream $\mathcal{S}$ and n.

As it was explained in Section 3.6, Algorithm 2 uses a hash table S, and a stack T that stores hash tables. The size of the stack at each point is bounded $\mathsf{depth}(w)$, and the size of each hash table is bounded by $|Q|^2|\Gamma|$, so the size of S and T combined is in $\mathcal{O}(\mathsf{depth}(w)|Q|^2|\Delta|)$. The rest of the space used is related to the ε-ECS $\mathcal{D}$, which we will now bound by $\mathcal{O}(\mathsf{outputweight}(\mathcal{T}, w[1, k])|Q|^2|\Delta|)$ at each step k.

For every step k of the algorithm, consider an ε-ECS $\mathcal{D}_k^{\mathsf{trim}}$ which is composed solely of the nodes that are reachable from of the ones stored in S^k, or the ones stored in some hash table in T^k. A simple induction argument on k shows that the rest of the nodes in $\mathcal{D}$ can be discarded with no effect over the correctness of the algorithm, so they are not considered in the memory used by it. Therefore, proving that at each step $|\mathcal{D}_k^{\mathsf{trim}}| \in \mathcal{O}(\mathsf{outputweight}(\mathcal{T}, w[1, k])|Q|^2|\Delta|)$ is enough to complete the proof.

Let $\mathcal{I}$ be the set of positions less than k that appear in some output of $[\![\mathcal{T}]\!](w[1, k] \cdot w')$ for some $w \cdot w' \in \mathsf{prefix}(\Sigma^{<*>})$. We now refer to Lemma 3.2 since it implies that for each node v stored in S^k or the topmost hash table in T^k, each sequence in $\text{Ł}_{\mathcal{D}}(v)$ corresponds to at least one valid run of $\mathcal{T}$ over $w[1, k]$, and since $\mathcal{T}$ is trimmed, each one of these runs is part of an accepting run of $\mathcal{T}$ over $w[1, k] \cdot w'$, for some word w'. Therefore, each of the positions that appear in some of these sets is in $\mathcal{I}$. Furthermore, we can use this lemma to characterize the positions in the rest of the hash tables in T^k, since appending any close symbol $a{>}$ to $w[1, k]$ will make the algorithm pop an element from T, which will make the next hash table the topmost. This argument can be extended to any of the hash tables in T^k, so in all, Lemma 3.2 implies that all of the positions that appear in some non-empty leaf in $\mathcal{D}_k^{\mathsf{trim}}$ are in $\mathcal{I}$. Theorem 3.1 implies that the set of these positions corresponds exactly to $\mathcal{I}$, since if there was any position in $\mathcal{I}$ missing from the leaves in $\mathcal{D}$, the algorithm would not be correct.

Lastly, we will show that $|\mathcal{D}_k^{\text{trim}}| \leq |\mathcal{I}| \times |Q|^2|\Delta| \times d$, where d is a constant. Towards this goal, we will bound the number of ε-leaves, non-empty leaves, and product nodes by $\mathcal{O}(|\mathcal{I}| \times |Q|^2|\Delta|)$ independently. Union nodes can be bounded by counting the other types of nodes: The only cases where a union node is created are (1) in line 38, only after a product node had been created, (2) during the creation of a product node (as described in Theorem 3.4), (3) in line 50, but only whenever one of the previous lines had created either a product node or a non-empty leaf node, and (4) in line 16, which only happens once at the end of the update phase, and iterates by nodes in S. Thus, the number of union nodes created at this for loop at most $|\mathcal{D}_k^{\text{trim}}|$. The number of ε-nodes is at most one, owing to Theorem 3.4, since its proof shows that at the end of step k, each of the nodes in $\mathcal{D}_k^{\text{trim}}$ is ε-safe. The number of non-empty leaves can be straightforwardly shown to be $\mathcal{O}(|\mathcal{I}| \times |Q|^2|\Delta|)$ since each of these leaves was introduced in some step in $\mathcal{I}$, and in each one of these steps, the number of operations that the algorithm does is in $\mathcal{O}(|Q|^2|\Delta|)$.

To show a bound over the number of product nodes, consider a slight modification of Algorithm 2: product nodes that are created in line 48 are labeled with the step k in which the algorithm is at the moment. Now, for a set of nodes A let $\mathcal{D}_A^{\text{trim}}$ be the ε-ECS that is obtained by removing all of the nodes that are not reachable from some node in A from $\mathcal{D}$. Let $\mathcal{I}_A$ be the set of positions that appear in some non-empty leaf node in $\mathcal{D}_A^{\text{trim}}$, and let $\mathcal{P}_A$ be the set of step labels that appear in some product node in $\mathcal{D}_A^{\text{trim}}$ excluding the steps in $\mathcal{I}_A$. Also, let V_k be the the set of nodes in $\mathcal{D}_k^{\text{trim}}$. We will show by induction on k that $|\mathcal{P}_A| \leq |\mathcal{I}_A| - 1$ for any $A \subseteq V_k$ which contains at least one node that is not an ε-node. Consider any set $A \subseteq V_k$. The first observation we make here is that we can partition the nodes in A to a collection $\{A_H\}$ of sets of nodes depending on the hash table H they are reachable from, given that they are in $\mathcal{D}_k^{\text{trim}}$. Let $\mathcal{Q}_A = \mathcal{P}_A \cup \mathcal{I}_A$. From Lemma 3.2 we get that for two different sets A_{H_1} and A_{H_2} in the collection, the sets $\mathcal{Q}_{H_1}$ and $\mathcal{Q}_{H_2}$ are disjoint. Therefore, in step k, if the algorithm enters CLOSESTEP, we only need to focus on the set A_S, and if the algorithm enters OPENSTEP on the set and A_{T^k} (note that in this case, S^k is composed only of ε-nodes). The rest of the hash tables were reachable on a previous step, so the inequality can be reached by adding up the inequalities that held in those steps. First, note that if none of the product nodes in A were created in step k, then we can consider the set B of nodes reachable from A that were created in a previous step and notice that $\mathcal{P}_A = \mathcal{P}_B$ and $\mathcal{I}_B \subseteq \mathcal{I}_A$, so the statement follows since $B \subseteq V_{k-1}$. Also, note that if the algorithm in step k enters OPENSTEP, all of the product nodes created in this step are directly connected to a non-ε leaf created in this same step, so the statement also follows. From this point on, we can assume that the algorithm enters CLOSESTEP on step k, and all of the nodes in A are reachable from some node in S^k, and there is at least one product node in A that was created in step k. Let P be the set of product nodes in A that were created on step k. Consider the span $\mathsf{currlevel}(k) = [j, k\rangle$. The prod operation in line 48 either creates a new product node, or makes v reference a node that already existed in S^{k-1} or the topmost table in T^j. Furthermore, if a product node is created in line 48, then Theorem 3.4 tells us that it must be connected to a node in S^{k-1} that is not an ε-node, and to a node in the topmost table in T^j that is also not an ε-node. Consider now the set of nodes B that is made up of (1) nodes in A that are reachable from S^{k-1} and (2) nodes in S^{k-1} that are connected to a product node in P. Consider also the set of nodes C that is made up of (1) nodes in A that are reachable from the topmost table in T^j, and nodes in the topmost table in T^j that are connected to a node in P. Note that both sets B and C contain a non-ε node, and are composed of nodes created in a previous step, so assume that $|\mathcal{P}_B| \leq |\mathcal{I}_B| - 1$ and that $|\mathcal{P}_C| \leq |\mathcal{I}_C| - 1$. It can be seen that every node in $\mathcal{D}_A^{\mathsf{trim}}$ is either in B, C, or was created on step k, so we get that $\mathcal{P}_A = \mathcal{P}_B \cup \mathcal{P}_C \cup \{k\}$ and $\mathcal{I}_A \supseteq \mathcal{I}_B \cup \mathcal{I}_C$. From Lemma 3.2 we get that $\mathcal{Q}_B$ and $\mathcal{Q}_C$ are disjoint, and putting these facts to together gives us that $|\mathcal{P}_A| = |\mathcal{P}_B| + |\mathcal{P}_C| + 1 \leq |\mathcal{I}_B| + |\mathcal{I}_C| - 1 \leq |\mathcal{I}_A| - 1$.

Having proven this statement, we can deduce that the number of product nodes in $\mathcal{D}_k^{\mathsf{trim}}$ is in $\mathcal{O}(|\mathcal{I}| \times |Q|^2|\Delta|)$ since the number of steps where they are created is bounded by $|\mathcal{I}|$. Therefore, $|\mathcal{D}_k^{\mathsf{trim}}| \leq |\mathcal{I}| \times |Q|^2|\Delta| \times d$, for some constant d. This concludes the proof.

□ □

Unfortunately, the algorithm provided in Theorem 3.1 is not *instance optimal*, in the sense of using the lowest number of bits needed for each specific VPAnn. Specifically, there exist VPAnn for which only logarithmic space in $\mathsf{outputweight}(\mathcal{T}, w)$ is enough for any stream $\mathcal{S}$. For example, let $\propto$ be any output symbol and consider a VPAnn $\mathcal{T}$ for which the output set is $[\![\mathcal{T}]\!](w) = \{(\propto, i) \mid 1 \leq i \leq |w|\}$ if the last symbol in w is $\$$ and the empty set otherwise. Clearly, the output weight of any w with respect to $\mathcal{T}$ is linear in $|w|$. However, one could design a streaming evaluation algorithm that has only a counter that stores the length of the input so far and produces the correct output set after reading the last symbol in w. The enumeration phase can easily be done with output-linear delay (i.e., by counting from 1 to $|w|$). Furthermore, note that an instance optimal algorithm for the streaming enumeration problem of VPAnn will imply a solution to the *weak evaluation problem*, stated by Segoufin and Vianu (Segoufin & Vianu, 2002), which is an open problem in the area (see (Barloy, Murlak, & Paperman, 2021) for some recent results). We leave the study of *instance optimal* streaming evaluation algorithms for future work.

3.5. Enumerable compact sets: a data structure for output-linear delay

This section presents a data structure, called Enumerable Compact Set with Shifts (Shift-ECS), which is the cornerstone of our enumeration algorithm for VPAnn. This data structure is strongly inspired by the work in (Amarilli et al., 2017, 2019a). Indeed, Shift-ECS can be considered a refinement of the d-DNNF circuits used in (Amarilli et al., 2017) or of the set circuits used in (Amarilli et al., 2019a). Several papers (Olteanu & Závodný, 2015; Amarilli et al., 2017; Amarilli, Bourhis, Mengel, & Niewerth, 2019b; Torunczyk, 2020) have considered circuits-like structures for encoding outputs and enumerate them with constant delay. The novelty of ECS is twofold. First, we use ECS for solving a streaming evaluation problem. Although people have studied streaming query evaluation with enumeration before (Idris et al., 2017; Berkholz et al., 2017), this is the first work that uses a circuit-like data structure in an online setting. Second and more important, there is a difference in performance if we compare ECS to the previous approaches. In offline evaluation, constant-delay algorithms usually create an initial circuit from the input, making several passes over the structure, building indices, and then running the enumeration process. Given time restrictions for the online evaluation, we cannot create a circuit and do this linear-time preprocessing before enumerating. On the contrary, we must extend the circuit-like data structure for each data item in constant time and then be ready to start the enumeration. This requirement justifies the need for a new data structure for representing and enumerating outputs. Therefore, ECS differs from previous proposals because each operation must take constant time, and we can run the enumeration process with output-linear delay, at any time and without any further preprocessing. In the following, we present Shift-ECS step-by-step to use them later in the next section.

Let Ω be a (possibly infinite) alphabet. We define an *Enumerable Compact Set with Shifts* (Shift-ECS) as a tuple:

$$\mathcal{D} = (\Omega, V, \ell, r, \lambda)$$

such that V is a finite set of nodes, $\ell\colon V \to V$ and $r\colon V \to V$ are the *left* and *right* partial functions, and $\lambda\colon V \to \Omega \cup \{\cup, \odot\}$ is a labeling function. For every node $v \in V$, both $\ell(v)$ and $r(v)$ are defined if $\lambda(v) \in \{\cup, \odot\}$, and neither is defined otherwise. Further, we assume that the directed graph $(V, \{(v, \ell(v)), (v, r(v)) \mid v \in V\})$ is acyclic. Note how this implies that nodes labeled by Ω are bottom nodes in the DAG and nodes labelled by $\cup$ or $\odot$ are inner nodes. We will use the terms *inner* and *bottom* to refer to such nodes, respectively. Furthermore, we say that v is a *product node* if $\lambda(v) = \odot$, and a *union node* if $\lambda(v) = \cup$. We define the size of $\mathcal{D}$ as $|\mathcal{D}| = |V|$. For each node v in $\mathcal{D}$, we associate a set of words $[\![\mathcal{D}]\!](v)$ recursively as follows: (1) $[\![\mathcal{D}]\!](v) = \{a\}$ whenever $\lambda(v) = a \in \Omega$, (2) $[\![\mathcal{D}]\!](v) = [\![\mathcal{D}]\!](\ell(v)) \cup [\![\mathcal{D}]\!](r(v))$ whenever $\lambda(v) = \cup$, and (3) $[\![\mathcal{D}]\!](v) = [\![\mathcal{D}]\!](\ell(v)) \cdot [\![\mathcal{D}]\!](r(v))$ whenever $\lambda(v) = \odot$, where $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$.

Since the size of the set $[\![\mathcal{D}]\!](v)$ can be exponential with respect to $|\mathcal{D}|$, we say that $\mathcal{D}$ is a *compact* representation of $[\![\mathcal{D}]\!](v)$ for any $v \in V$. Although $[\![\mathcal{D}]\!](v)$ is very large, the goal is to enumerate all of its elements efficiently. Specifically, we consider the following problem:

Problem: Enum-ECS
Input: A Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ and $v \in V$.
Output: Enumerate the set $\llbracket \mathcal{D} \rrbracket(v)$ without repetitions.

Additionally, we want to solve Enum-ECS with output-linear delay. A reasonable strategy to enumerate $\llbracket \mathcal{D} \rrbracket(v)$ is to do a sequence of traversals on the structure, each of which builds a different output from the set. However, to be able to do this without repetitions and output-linear delay, we need to guarantee two conditions: first that one can obtain every output from $\mathcal{D}$ in only one way and, second, that union nodes are *close* to a bottom node or a product node, in the sense that we can always reach one of them in a bounded number of steps. To ensure that these conditions hold, we impose two restrictions on an ECS:

(i) We say that $\mathcal{D}$ is *duplicate-free* if $\mathcal{D}$ satisfies the following two properties: (1) for every union node v it holds that $\llbracket \mathcal{D} \rrbracket(\ell(v))$ and $\llbracket \mathcal{D} \rrbracket(r(v))$ are disjoint, and (2) for every product node v and for every $w \in \llbracket \mathcal{D} \rrbracket(v)$, there exists a unique way to decompose $w = w_1 \cdot w_2$ such that $w_1 \in \llbracket \mathcal{D} \rrbracket(\ell(v))$ and $w_2 \in \llbracket \mathcal{D} \rrbracket(r(v))$.

(ii) We define the notion of *k-bounded* Shift-ECS as follows. Given a Shift-ECS $\mathcal{D}$, define the (left) output-depth of a node $v \in V$, denoted by $\mathsf{odepth}_{\mathcal{D}}(v)$, recursively as follows: $\mathsf{odepth}_{\mathcal{D}}(v) = 0$ whenever $\lambda(v) \in \Omega$ or $\lambda(v) = \odot$, and $\mathsf{odepth}_{\mathcal{D}}(v) = \mathsf{odepth}_{\mathcal{D}}(\ell(v)) + 1$ whenever $\lambda(v) = \cup$. Then, for $k \in \mathbb{N}$ we say that $\mathcal{D}$ is k-bounded if $\mathsf{odepth}_{\mathcal{D}}(v) \leq k$ for all $v \in V$.

Given the definition of output depth, we say that v is an output node of $\mathcal{D}$ if v is a bottom node or a product node. Note that if $\mathcal{D}$ only has output nodes, then it is 0-bounded, and one can easily check that $\llbracket \mathcal{D} \rrbracket(v)$ can be enumerated with output-linear delay. Indeed, for a fixed k the same happens with every duplicate-free and k-bounded Shift-ECS.

PROPOSITION 3.5. *Fix $k \in \mathbb{N}$. Let $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ be a duplicate-free and k-bounded Shift-ECS. Then one can enumerate the set $[\![\mathcal{D}]\!](v)$ with output-linear delay for any $v \in V$.*

PROOF. Let $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ be a duplicate-free and k-bounded Shift-ECS. The algorithm that we present is a depth-first traversal of the DAG, done in a recursive fashion to ensure that after retrieving some output w, the next one w' can be printed in $O(k \cdot (|w| + |w'|))$ time. We will show this easier bound first, and then show how this implies $O(k' \cdot |w'|)$ delay by the end of the proof. The entire procedure is detailed in Algorithm 1.

To simplify the presentation of the algorithm, we use the interface of an *iterator* that, given a node v, it contains all information and methods to enumerate the outputs $[\![\mathcal{D}]\!](v)$. Specifically, an iterator τ must implement the following three methods:

$$\text{CREATE}(v) \rightarrow \tau \qquad \tau\,.\text{NEXT} \rightarrow b \qquad \tau\,.\text{PRINT} \rightarrow \emptyset$$

where v is a node, b is either **true** or **false**, and $\emptyset$ means "no output". The first method, CREATE, receives a node v and creates an iterator τ of the type of v. We will implement three types of iterators, one for each node type: bottom, product, and union nodes. The second method, τ .NEXT, moves the iterator to the next output, returning **true** if, and only if, there is an output to print. Then the last method, τ .PRINT, write the current output pointed by τ to the output registers. We assume that, after creating an iterator τ, one must first call τ .NEXT to move to the first output before printing. Furthermore, if τ .NEXT outputs **false**, then the behavior of τ .PRINT is undefined. Note that one can call τ .PRINT several times, without calling τ .NEXT, and the iterator will write the same output each time in the output registers.

Assume we can implement the iterator interface for each node type. Then the procedure ENUMERATE(v) in Algorithm 1 (lines 61-64) shows how to enumerate the set $[\![\mathcal{D}]\!](v)$

Algorithm 1 Enumeration over a node u from some ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$.

```
▷ Bottom node iterator $\tau_\Omega$
procedure CREATE($v$)    ▷ Assume $\lambda(v) \in \Omega$
    $u \leftarrow v$
    $\mathrm{hasnext} \leftarrow \mathbf{true}$

procedure NEXT
    if $\mathrm{hasnext} = \mathbf{true}$ then
        $\mathrm{hasnext} \leftarrow \mathbf{false}$
        return $\mathbf{true}$
    return false

procedure PRINT
    $\mathbf{print}(\lambda(u))$

▷ Product node iterator $\tau_\odot$
procedure CREATE($v$)    ▷ Assume $\lambda(v) = \odot$
    $u \leftarrow v$
    $\tau_\ell \leftarrow$ CREATE$(\ell(u))$
    $\tau_\ell$.NEXT
    $\tau_r \leftarrow$ CREATE$(r(u))$

procedure NEXT
    if $\tau_r$.NEXT $=$ false then
        if $\tau_\ell$.NEXT $=$ false then
            return false
        $\tau_r \leftarrow$ CREATE$(r(u))$
        $\tau_r$.NEXT
    return $\mathbf{true}$

procedure PRINT
    $\tau_\ell$.PRINT
    $\tau_r$.PRINT

▷ Union node iterator $\tau_\cup$
procedure CREATE($v$)    ▷ Assume $\lambda(v) = \cup$
    $\mathrm{St} \leftarrow \mathsf{push}(\mathrm{St}, v)$
    $\mathrm{St} \leftarrow$ TRAVERSE$(\mathrm{St})$
    $\tau \leftarrow$ CREATE$(\mathsf{top}(\mathrm{St}))$

procedure NEXT
    if $\tau$ .NEXT $=$ false then
        $\mathrm{St} \leftarrow \mathsf{pop}(\mathrm{St})$
        if $\mathsf{length}(\mathrm{St}) = 0$ then
            return false
        else if $\lambda(\mathsf{top}(\mathrm{St})) = \cup$ then
            $\mathrm{St} \leftarrow$ TRAVERSE$(\mathrm{St})$
        $\tau \leftarrow$ CREATE$(\mathsf{top}(\mathrm{St}))$
        $\tau$ .NEXT
    return $\mathbf{true}$

procedure PRINT
    $\tau$ .PRINT

procedure TRAVERSE($\mathrm{St}$)
    while $\lambda(\mathsf{top}(\mathrm{St})) = \cup$ do
        $u \leftarrow \mathsf{top}(\mathrm{St})$
        $\mathrm{St} \leftarrow \mathsf{pop}(\mathrm{St})$
        $\mathrm{St} \leftarrow \mathsf{push}(\mathrm{St}, r(u))$
        $\mathrm{St} \leftarrow \mathsf{push}(\mathrm{St}, \ell(u))$
    return $\mathrm{St}$

procedure ENUMERATE($v$)
    $\tau \leftarrow$ CREATE$(v)$
    while $\tau$ .NEXT $= \mathbf{true}$ do
        $\tau$ .PRINT
```

by using an iterator τ for v. In the following, we show how to implement the iterator interface for each node type and how the size of the following output bounds the delay between two outcomes.

We start by presenting the iterator τ_Ω for a bottom node v (lines 1-13), called a *bottom node iterator*. We assume that each τ_Ω has internally two values u and $\mathrm{hasnext}$, where u is a reference to v and $\mathrm{hasnext}$ is a boolean variable. The purpose of a bottom node iterator is only to print $\lambda(u)$. For this goal, when we create τ_Ω, we initialize u equal to v and $\mathrm{hasnext} = \mathbf{true}$ (lines 3-4). Then, when we call τ_Ω.NEXT for the first time, we swap $\mathrm{hasnext}$ from **true** to **false** and output **true** (i.e., there is one output ready to be returned). Then any following call to τ_Ω.NEXT will be false (lines 6-10). Finally, the τ_Ω.PRINT writes the value $\lambda(u)$ to the output registers (lines 12-13). Here, we assume the existence of a method **print** on the RAM model for writing to the output registers.

For a product node, we present a *product node iterator* $\tau_\odot$ in Algorithm 1 (lines 15-32). This iterator receives a product node v with $\lambda(v) = \odot$ and stores a reference of v, called u, and two iterators τ_ℓ and τ_r, for iterating through the left and right nodes $\ell(u)$ and $r(u)$, respectively. The CREATE method initializes u with v, creates the iterators τ_ℓ and τ_r, and calls τ_ℓ.NEXT to be prepared for the first call of $\tau_\odot$.NEXT (lines 16-20). The idea of $\tau_\odot$.NEXT is to fix one output for the left node $\ell(u)$ and iterate over all outputs of $r(u)$ (lines 22-28). When we stop enumerating all outputs of $[\![\mathcal{D}]\!](r(u))$, we move to the next output of τ_ℓ, and iterate again over all $[\![\mathcal{D}]\!](r(u))$ (lines 24-27). For printing, we recursively call first the printing method of τ_ℓ, and then the one of τ_r (lines 30-32).

The most involved case is the *union node iterator* $\tau_\cup$ (lines 33-59). Similarly to a product node iterator, it receives a union node v with $\lambda(v) = \cup$ and keeps a *stack* of nodes St and an iterator τ. We assume the standard implementation of a stack with the native methods push, pop, top, and length over stacks – the first three define the standard operations over stacks, and length counts the elements in it. The purpose of the stack is to perform a *depth-first-search* traversal of all union nodes below v, reaching all possible output nodes u such that there is a path of union nodes between v and u. If the top node of St is an output node, then τ is an iterator for that node, which enumerates all their outputs.

For performing the *depth-first-search* traversal of union nodes, we use the auxiliary method TRAVERSE(St) (lines 53-59). While the top node u in St is a union node, it pops

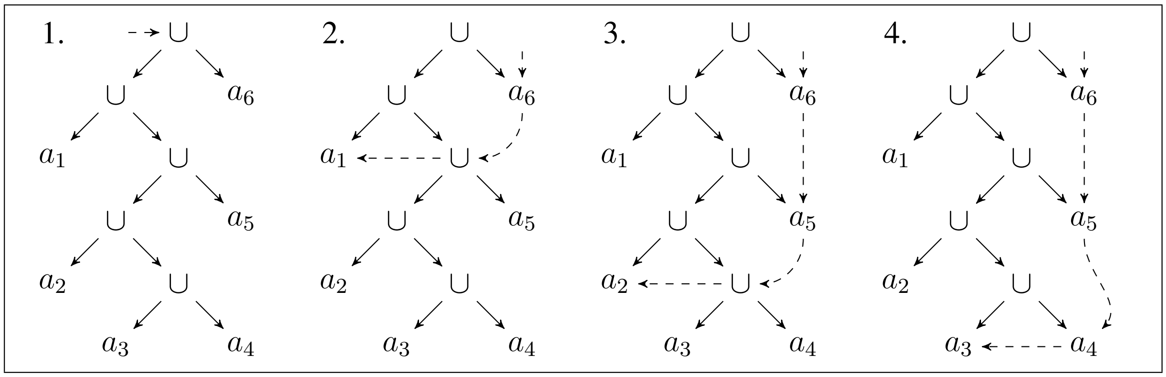


Figure 3.4. Evolution of the stack St (represented by dashed arrows) for an iterator over the topmost union node in the figure. The underlying ECS contains only union nodes and six bottom nodes. The first figure is St after calling $\mathrm{St} \leftarrow \mathsf{push}(\mathrm{St}, v)$, the second is after calling $\mathrm{St} \leftarrow \textsc{Traverse}(\mathrm{St})$. The last two figures represent successive calls to $\mathsf{pop}(\mathrm{St}), \mathrm{St} \leftarrow \textsc{Traverse}(\mathrm{St})$.

u and pushes its right and left nodes in that order. This method will eventually reach an output node (i.e., a non-union node) at the top of the stack and end. It is important to note that $\textsc{traverse}(\mathrm{St})$ takes $\mathcal{O}(k)$-steps, given that the ECS is k-bounded. Then if k is fixed, the TRAVERSE procedure takes constant time. In Figure 3.4, we illustrate the evolution of a stack St inside a union node iterator when we call $\textsc{traverse}(\mathrm{St})$ several times.

The methods of a union node iterator $\tau_\cup$ are then straightforward. For CREATE (lines 34-37), we push v into St (line 35) and traverse St, finding the first rightmost output node from v (line 36). Then we build the iterator τ of this output node for being ready to start enumerating their outputs (line 37). For NEXT, we consume all outputs by calling τ .NEXT (line 40). When there are no more outputs (lines 41-47), we pop the top node from St and check if the stack is empty or not (lines 41-42). If this is the case, there are no more outputs and we output **false**. Instead, if St is non-empty but the top node $\mathsf{top}(\mathrm{St})$ is a union node, we apply the TRAVERSE method for finding the rightmost output node from $\mathsf{top}(\mathrm{St})$ (lines 44-45). When the procedure is done, we know that the top node is an output node, and then we create an iterator and move to its first output (lines 46-47). For PRINT, we call the print method of τ which is ready to write the current output (lines 50-51).

For proving the correctness of the enumeration procedure, since $\mathcal{D}$ is duplicate-free, one can verify that ENUMERATE(v) in Algorithm 1 enumerates all the set $[\![\mathcal{D}]\!](v)$, one by one, and without repetitions. For the delay between outputs, since $\mathcal{D}$ is k-bounded, it is straightforward to prove by induction that, if w_0 is the first output, then:

- CREATE(v) takes time $\mathcal{O}(k \cdot |w_0|)$,
- NEXT takes time $\mathcal{O}(k \cdot |w_0|)$ for the first call, and $\mathcal{O}(k \cdot |w| + |w'|)$ for the next call where w and w' are the previous and next outputs, respectively, and
- PRINT takes time $\mathcal{O}(k \cdot |w|)$ where w is the current output to be printed.

Overall, ENUMERATE(v) in Algorithm 1 have delay $O(k \cdot (|w| + |w'|))$ to write the next output w' in the output register, after printing the previous output w.

We end by pointing out that the existence of an enumeration algorithm $\mathcal{E}$ with delay $O(k \cdot (|w| + |w'|))$ between any consecutive outputs w and w', implies the existence of an enumeration algorithm $\mathcal{E}'$ with output-linear delay as defined in Section 3.2. We start noting that k is a fixed value and then the delay of $\mathcal{E}$ only depends on $|w| + |w'|$. For depending only on the next output w', one can perform the following strategy for $\mathcal{E}'$: start by running $\mathcal{E}$, enumerate the first output w_0, proceed $k \cdot |w_0|$ more steps of $\mathcal{E}$, stop, and print the separator symbol $\#$. Then continue running $\mathcal{E}$ to prepare the next output w_1, proceed $k|w_1|$ more steps, stop, and print the separator symbol $\#$. By repeating this enumeration process, one can verify that the delay between the i-th output w_i and the $(i+1)$-th output w_{i+1} is $O(|w_{i+1}|)$. Therefore, $\mathcal{E}'$ has output-linear delay. □

The enumeration algorithm above does not require any preprocessing over $\mathcal{D}$. By this proposition, from now we assume that all Shift-ECS are duplicate-free and k-bounded for some fixed k.

Operations allowed by an ECS. The next step is to provide a set of operations that allow extending a Shift-ECS $\mathcal{D}$ while maintaining k-boundedness. Furthermore, we require these operations to be *fully-persistent*: a data structure is called fully-persistent if every version can be both accessed and modified (Driscoll, Sarnak, Sleator, & Tarjan, 1986a). In other words, the previous version of the data structure is always available after each operation. To satisfy the last requirement, the strategy will consist in extending $\mathcal{D}$ to $\mathcal{D}'$ for each operation, by adding new nodes and maintaining the previous nodes unmodified. Then $\mathcal{L}_{\mathcal{D}'}(v) = \mathcal{L}_{\mathcal{D}}(v)$ for each node $v \in V$, and so, the data structure will be fully-persistent.

Fix a Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$. In the following, we say that $\mathcal{D}' = (\Omega, V', \ell', r', \lambda')$ is an *extension* of $\mathcal{D}$ if, and only if, $\mathsf{X} \subseteq \mathsf{X}'$ for every $\mathsf{X} \in \{V, \ell, r, \lambda\}$. Further, we write $\mathsf{op}(I) \to O$ to define the signature of an operation op where I is the input and O is the output. Then for any $a \in \Omega$ and $v_1, \ldots, v_4 \in V$, we define the operations:

$$\mathsf{add}(\mathcal{D}, a) \to (\mathcal{D}', v') \qquad \mathsf{prod}(\mathcal{D}, v_1, v_2) \to (\mathcal{D}', v') \qquad \mathsf{union}(\mathcal{D}, v_3, v_4) \to (\mathcal{D}', v')$$

such that $\mathcal{D}'$ is an extension of $\mathcal{D}$ and $v' \in V' \setminus V$ is a fresh node such that $\mathcal{L}_{\mathcal{D}'}(v') = \{a\}$, $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_{\mathcal{D}}(v_1) \cdot \mathcal{L}_{\mathcal{D}}(v_2)$, and $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_{\mathcal{D}}(v_3) \cup \mathcal{L}_{\mathcal{D}}(v_4)$, respectively. We assume that the union and prod satisfy properties (1) and (2) of a duplicate-free Shift-ECS, namely, $\mathcal{L}_{\mathcal{D}}(v_1)$ and $\mathcal{L}_{\mathcal{D}}(v_2)$ are disjoint and, for every $w \in [\![\mathcal{D}]\!](v_3) \cdot [\![\mathcal{D}]\!](v_4)$, there exists a unique way to decompose $w = w_1 \cdot w_2$ such that $w_1 \in [\![\mathcal{D}]\!](v_3)$ and $w_2 \in [\![\mathcal{D}]\!](v_4)$.

Next, we show how to implement these operations, where the case of add and prod are straightforward. For $\mathsf{add}(\mathcal{D}, a) \to (\mathcal{D}', v')$ define $V' := V \cup \{v'\}$ and $\lambda'(v') = a$. One can easily check that $\text{Ł}_{\mathcal{D}'}(v') = \{a\}$ as expected. For $\mathsf{prod}(\mathcal{D}, v_1, v_2) \to (\mathcal{D}', v')$ we proceed in a similar way: define $V' := V \cup \{v'\}$, $\ell'(v') := v_1$, $r'(v') = v_2$, and $\lambda'(v') = \odot$. Then $\text{Ł}_{\mathcal{D}'}(v') = [\![\mathcal{D}]\!](v_1) \cdot [\![\mathcal{D}]\!](v_2)$. Furthermore, one can check that each operation takes constant time, $\mathcal{D}'$ is a valid Shift-ECS (i.e. duplicate-free and k-bounded), and the operations are fully-persistent (i.e. the previous version $\mathcal{D}$ is available).

To define the union, we need to be a bit more careful to guarantee output-linear delay, specifically, the k-bounded property. For $v \in V$, we say that v is *safe* if (1) $\mathsf{odepth}_{\mathcal{D}}(v) \leq 1$, and (2) if $\mathsf{odepth}_{\mathcal{D}}(v) = 1$, then $\mathsf{odepth}_{\mathcal{D}}(r(v)) \leq 1$. In other words, v is safe if either v is an output node, or its left child is an output node and its right child is either an output node or has output depth 1. Note that a leaf or a product node are safe nodes by definition and, thus, the add and prod operations always produce safe nodes.

The strategy then is to show that, if v_3 and v_4 are safe nodes, then we can implement the operation $\mathsf{union}(\mathcal{D}, v_3, v_4) \to (\mathcal{D}', v')$ and produce a safe node v'. To reach this goal, define $(\mathcal{D}', v')$ as follows:

- If v_3 or v_4 are output nodes then $V' := V \cup \{v'\}$ and $\lambda(v') := \cup$. Moreover, if v_3 is the output node, then $\ell'(v') := v_3$ and $r'(v') := v_4$. Otherwise, we connect $\ell'(v') := v_4$ and $r'(v') := v_3$.
- If v_3 and v_4 are not output nodes (i.e. both are union nodes), then $V' := V \cup \{v', u_1, u_2\}$, $\ell'(v') := \ell(v_3)$, $r'(v') := u_1$, and $\lambda'(v') := \cup$; $\ell'(u_1) := \ell(v_4)$, $r'(u_1) := u_2$, and $\lambda'(u_1) := \cup$; $\ell'(u_2) := r(v_3)$, $r'(u_2) := r(v_4)$, and $\lambda'(u_2) := \cup$.

This gadget[2] is depicted in Figure 3.5. This construction has several properties. First, one can easily check that $Ł_{\mathcal{D}'}(v') = [\![\mathcal{D}]\!](v_1) \cup [\![\mathcal{D}]\!](v_2)$ and so its semantics is well-defined. Second, union can be computed in constant time in $|\mathcal{D}|$ given that we only need to add three fresh nodes, and the operation is fully-persistent given that we connect them without modifying the nodes of $\mathcal{D}$. Third, the produced node v' is safe in $\mathcal{D}'$, although nodes u_1 and u_2 are not necessarily safe. Finally, $\mathcal{D}'$ is 2-bounded whenever $\mathcal{D}$ is 2-bounded. This is straightforward to verify for the first case when v_3 or v_4 are output nodes. For the second case (i.e., Figure 3.5), recall that v_3 and v_4 are safe nodes, which means that $\ell(v_3)$ and $\ell(v_4)$ are output nodes, and then $\mathsf{odepth}_{\mathcal{D}'}(v') = \mathsf{odepth}_{\mathcal{D}'}(u_1) = 1$. Further, given that v_3 is safe, we know that $\mathsf{odepth}_{\mathcal{D}}(r(v_3)) \leq 1$, so $\mathsf{odepth}_{\mathcal{D}'}(u_2) \leq 2$. Given that the output depths of all fresh nodes in $\mathcal{D}'$ are bounded by 2 and $\mathcal{D}$ is 2-bounded, then we conclude that $\mathcal{D}'$ is 2-bounded as well.

By the previous discussion, if we start with a Shift-ECS $\mathcal{D}$ which is 2-bounded (or empty) and we apply the add, prod, or union operators between safe nodes (which also

[2] Note that a similar trick was used in (Amarilli et al., 2017) for computing an index over a circuit.

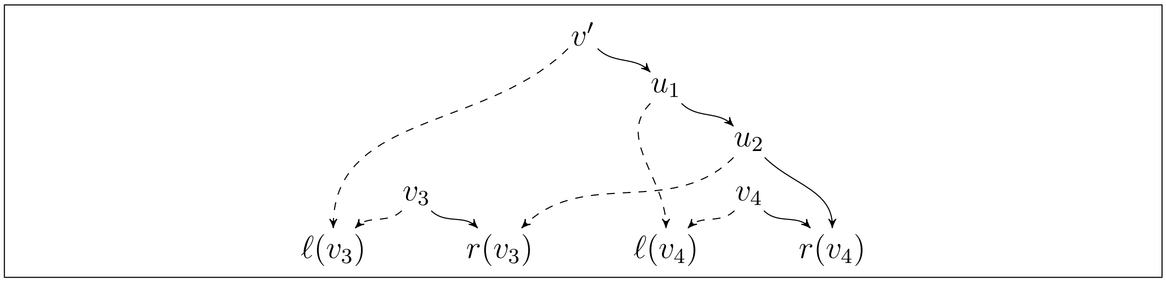


Figure 3.5. Gadget for $\mathsf{union}(\mathcal{D}, v_3, v_4)$. Nodes v', u_1, u_2, v_3 and v_4 are labeled as $\cup$. Dashed and solid lines denote the mappings in ℓ' and r' respectively.

produce safe nodes), then the result is 2-bounded as well. Finally, by Proposition 3.5, the result can be enumerated with output-linear delay.

Theorem 3.3. *The operations* add, prod, *and* union *require constant time and are fully-persistent. Furthermore, if we start from an empty Shift-ECS $\mathcal{D}$ and apply* add, prod, *or* union *operations over safe nodes, the partial results $(\mathcal{D}', v')$ satisfy that v' is always a safe node and the set $Ł_{\mathcal{D}'}(v)$ can be enumerated with output-linear delay for every node v.*

We want to remark that restricting the operations over safe nodes does not limit the final user of the data structure. Indeed, since we will usually start from an empty Shift-ECS and apply these operations over previously returned nodes, the whole algorithm will always use safe nodes during its computation, satisfying the conditions of Theorem 3.3.

Extending ECS with ε-nodes. For technical reasons, our algorithm of the next section needs a slight extension of Shift-ECS by allowing leaves that produce the empty string ε. Let $\varepsilon \notin \Omega$ be a symbol representing the empty string (i.e. $w \cdot \varepsilon = \varepsilon \cdot w = w$). We define an enumerable compact set with ε (called ε-ECS) as a tuple $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ defined identically to a Shift-ECS except that:

$$\lambda \colon V \to \Omega \cup \{\cup, \odot, \varepsilon\}$$

and $\lambda(v) \in \{\cup, \odot\}$ if, and only if, both $\ell(v)$ and $r(v)$ are defined. Further, $[\![\mathcal{D}]\!](v) = \{\varepsilon\}$ whenever $\lambda(v) = \varepsilon$. The duplicate-free restriction is the same for ε-ECS and one has to slightly extend k-boundedness to consider ε-nodes. However, to support the prod and union operations in constant time and to maintain the k-boundedness invariant, we need to extend the notion of safe nodes, called *ε-safe*, and the gadgets for prod and union. We summarize all these details in the proof of the following theorem which is a straightforward extension of Theorem 3.3.

Theorem 3.4. *The operations* add*,* prod*, and* union *over ε-ECS take constant time and are fully-persistent. Furthermore, if we start from an empty ε-ECS $\mathcal{D}$ and apply* add*,* prod*, and* union *over ε-safe nodes, the partial results $(\mathcal{D}', v')$ satisfy that v' is always an ε-safe node and the set $Ł_{\mathcal{D}'}(v)$ can be enumerated with output-linear delay for every node v.*

PROOF. For dealing with ε-nodes, we need to revisit the notions of output depth, k-bounded, and safeness. For the sake of presentation, it is simpler first to introduce the notion of ε-safe nodes to then revisit all previous definitions. Specifically, for a given ε-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ we say that $v \in V$ is *ε-safe* if it is exactly one of the following situations:

(i) $\lambda(v) = \varepsilon$, or

(ii) $\lambda(v) \neq \varepsilon$, v is a safe node, and $\lambda(u) \neq \varepsilon$ for any node u which is reachable from v, or

(iii) $\lambda(v) = \cup$, $\lambda(\ell(v)) = \varepsilon$, and $r(v)$ satisfies (ii).

In other words, ε can only occur as the left child of an ε-safe node or being the node itself.

From now on, we can assume that we will only work with ε-safe nodes as the input for enumerating or for operating them with other (ε-safe) nodes. If this is the case, then the enumeration with output-linear delay and the notions of output depth and k-boundedness are similar to before. Therefore, we dedicate the rest of the proof to revisit the operations prod and union when the inputs are ε-safe nodes (the add operation is straightforward similar to the case without ε).

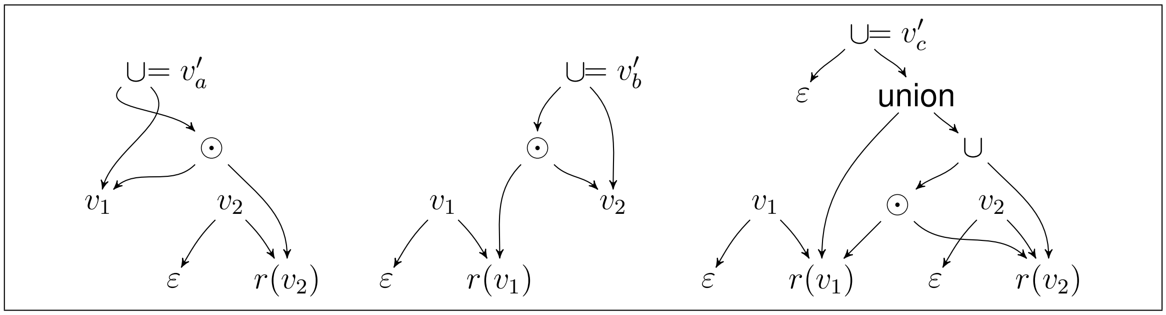


Figure 3.6. Gadgets for prod as defined for an ε-ECS. Nodes v'_a, v'_b, and v'_c correspond to v' as defined for cases (a), (b), and (c), respectively.

Assume v_1 and v_2 are ε-safe. Since both v_1 and v_2 may fall in one of three cases above, we define $\mathsf{prod}(\mathcal{D}, v_1, v_2) \to (\mathcal{D}', v')$ by separating into nine cases, of which the first six are straightforward:

- If $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_1)$ and $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_2)$, we use the construction given for a regular Shift-ECS.
- If $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_1)$ and $\lambda(v_2) = \varepsilon$, we define $v' = v_1$, and $\mathcal{D}' = \mathcal{D}$.
- If $\lambda(v_1) = \varepsilon$ and $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_2)$, we define $v' = v_2$, and $\mathcal{D}' = \mathcal{D}$.
- If $\lambda(v_1) = \varepsilon$ and $\lambda(v_2) = \varepsilon$, we define $v' = v_1$, and $\mathcal{D}' = \mathcal{D}$.
- If $\lambda(v_1) = \varepsilon$ and v_2 is in case (iii), we define $v' = v_2$, and $\mathcal{D}' = \mathcal{D}$.
- If v_1 is in case (iii) and $\lambda(v_2) = \varepsilon$, we define $v' = v_1$, and $\mathcal{D}' = \mathcal{D}$.

The other three cases are more involved and they are presented graphically in Figure 3.6. Formally, they are defined as follows:

(a) If $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_1)$ and v_2 is in case (iii), then $V' = V \cup \{v', v''\}$, $\ell'(v') = v''$, $r'(v') = v_1$, $\ell(v'') = v_1$, $r(v'') = r(v_2)$, $\lambda'(v') = \cup$ and $\lambda'(v'') = \odot$.

(b) If v_1 is in case (iii) and $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_1)$, then $V' = V \cup \{v', v''\}$, $\ell'(v') = v''$, $r'(v') = v_2$, $\ell(v'') = r(v_1)$, $r(v'') = v_2$, $\lambda'(v') = \cup$ and $\lambda'(v'') = \odot$.

(c) If both v_1 and v_2 are in case (iii), we do a slightly more delicate construction. First, we define a $\mathcal{D}''$ with $V'' = V \cup \{v^3, v^4\}$, $\ell''(v^3) = v^4$, $r''(v^3) = r(v_2)$, $\ell''(v^4) = r(v_1)$, $r''(v^4) = r(v_2)$, $\lambda''(v^3) = \cup$, $\lambda''(v^4) = \odot$. Now, let $(\mathcal{D}^3, v^2) \leftarrow$

$\mathsf{union}(\mathcal{D}'', r(v_1), v_3)$. Lastly, let $V' = V^3 \cup \{v^*, v'\}$, $\ell'(v') = v^*$, $r(v') = v_2$, $\lambda(v') = \cup$ and $v^* = \varepsilon$.

Note that the union operation in case (c) does not recurse since $r(v_1)$ is safe. In particular, it does not reach any ε-leaf.

For the union-operation, assume v_1 and v_2 are ε-safe and define $\mathsf{union}(\mathcal{D}, v_1, v_2) \to (\mathcal{D}', v')$ as:

- If $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_1)$ and $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_2)$, we use the construction given for a regular Shift-ECS.
- If $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_1)$ and $\lambda(v_2) = \varepsilon$, we define $V' = V \cup \{v'\}$ and $\lambda'(v') = \cup$. We connect $\ell'(v') = v_2$ and $r'(v') = v_1$.
- If $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_1)$ and v_2 is in case (iii), let $(\mathcal{D}'', v'') = \mathsf{union}(\mathcal{D}, v_1, r(v_2))$ as defined for a regular Shift-ECS. We define $V' = V'' \cup \{v'\}$, and $\lambda'(v') = \cup$ where λ' is an extension of λ''. We connect $\ell'(v') = \ell(v_2)$ and $r'(v') = v''$.
- If $\lambda(v_1) = \varepsilon$ and $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_2)$, we define $V' = V \cup \{v'\}$ and $\lambda'(v') = \cup$. We connect $\ell'(v') = v_1$ and $r'(v') = v_2$.
- If $\lambda(v_1) = \varepsilon$ and $\lambda(v_2) = \varepsilon$, we define $\mathcal{D}' = \mathcal{D}$ and $v' = v_1$.
- If $\lambda(v_1) = \varepsilon$ and v_2 is in case (iii), we define $\mathcal{D}' = \mathcal{D}$ and $v' = v_2$.
- (*) If v_1 is in case (iii) and $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v_2)$, let $(\mathcal{D}'', v'') = \mathsf{union}(\mathcal{D}, r(v_1), v_2)$ as defined for a regular Shift-ECS. We define $V' = V'' \cup \{v'\}$ and $\lambda'(v') = \cup$ where λ' is an extension of λ''. We connect $\ell'(v') = \ell(v_2)$ and $r'(v') = v''$.
- If v_1 is in case (iii) and $\lambda(v_2) = \varepsilon$, we define $\mathcal{D}' = \mathcal{D}$ and $v' = v_1$.
- If both v_1 and v_2 are in case (iii), let $(\mathcal{D}', v') = \mathsf{union}(\mathcal{D}, v_1, r(v_2))$ by using the construction of the case marked with (*).

It is straightforward to check that each operation behaves as expected. Moreover, if both v_1 and v_2 are ε-safe, then the resulting node v' is ε-safe as well for each operation.

We finish this proof by noticing that each operation falls into a fixed number of cases that can be checked exhaustively, and each construction has a fixed size, so they take constant time. Furthermore, each operation is fully persistent as expected.

□

3.6. Evaluating visibly pushdown annotators with output-linear delay

The goal of this section is to describe an algorithm that takes an I/O-unambiguous VPAnn $\mathcal{T}$ plus a stream $\mathcal{S}$, and enumerates the set $[\![\mathcal{T}]\!](\mathcal{S}[1,n])$ for an arbitrary $n \geq 0$ with $\mathcal{O}(|Q|^2|\Delta|)$-update-time and output-linear delay. We divide the presentation of the algorithm into two parts. The first part explains the determinization of a VPA, which is instrumental in understanding the update phase. The second part gives the algorithm and proves its correctness. Given that a neutral symbol a can be represented as a pair $<\!a \cdot a\!>$, in this section we present the algorithm and definitions without neutral letters, that is, the structured alphabet is $\Sigma = (\Sigma^{<}, \Sigma^{>})$.

Determinization of visibly pushdown automata. An important result in Alur and Madhusudan's paper (Alur & Madhusudan, 2004b) that introduces VPA was that one can always determinize them. We provide here an alternative proof for this result that requires a somewhat more direct construction. This determinization process is behind our update algorithm and serves to give some crucial notions of how it works. We start by providing the determinization construction, introducing some useful notation, and then giving some intuition.

Given a VPA $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, I, F)$, we define the following deterministic VPA:

$$\mathcal{A}^{\text{det}} = (Q^{\text{det}}, q_0^{\text{det}}, \Gamma^{\text{det}}, \delta^{\text{det}}, F^{\text{det}})$$

with state set $Q^{\text{det}} = 2^{Q \times Q}$ and stack symbol set $\Gamma^{\text{det}} = 2^{Q \times \Gamma \times Q}$. The initial state is $q_0^{\text{det}} = \{(q, q) \mid q \in I\}$ and the set of final states is $F^{\text{det}} = \{S \in Q^{\text{det}} \mid S \cap (I \times F) \neq \emptyset\}$. Finally, we define the transition function δ^{det} such that if $<\!a \in \Sigma^{<}$, then $\delta^{\text{det}}(S, <\!a) =$

(S', T') where $S' = \{(q,q) \mid \exists p, p', \gamma.\ (p,p') \in S \wedge (p', {<}a, q, \gamma) \in \Delta\}$ and $T' = \{(p,\gamma,q) \mid \exists p'.\ (p,p') \in S \wedge (p', {<}a, q, \gamma) \in \Delta\}$; if $a{>} \in \Sigma^{>}$, then $\delta^{\det}(S, T, a{>}) = S'$ where $S' = \{(p,q) \mid \exists p', q', \gamma.\ (p,\gamma,p') \in T \wedge (p',q') \in S \wedge (q', a{>}, \gamma, q) \in \Delta\}$.

To explain the purpose of this construction, first we need to introduce some notation. Fix a well-nested word $w = a_1 a_2 \cdots a_n$. A span s of w is a pair $[i, j\rangle$ of natural numbers i and j with $1 \leq i \leq j \leq n+1$. We denote by $w[i,j\rangle$ the subword $a_i \cdots a_{j-1}$ of w and, when $i = j$, we assume that $w[i,j\rangle = \varepsilon$. Intuitively, spans are indexing w with intermediate positions like:

$${}_{\mathtt{1}}a_1\,{}_{\mathtt{2}}a_2\,{}_{\mathtt{3}}\ \ldots\ {}_{\mathtt{n}}a_n\,{}_{\mathtt{n+1}}$$

where i is between symbols a_{i-1} and a_i. Then $[i,j\rangle$ represents an interval $\{i, \ldots, j\}$ that captures the subword $a_i \ldots a_{j-1}$.

Now, we say that a span $[i,j\rangle$ of w is well-nested if $w[i,j\rangle$ is well-nested. Note that ε is well-nested, so $[i,i\rangle$ is a well-nested span for every i. For a position $k \in [1, n+1]$, we define the *current-level span* of k, $\mathsf{currlevel}(k)$, as the well-nested span $[j,k\rangle$ such that $j = \min\{j' \mid [j',k\rangle \text{ is well-nested}\}$. Note that $[k,k\rangle$ is always well-nested and thus $\mathsf{currlevel}(k)$ is well defined. We also identify the *lower-level span* of k, $\mathsf{lowerlevel}(k)$, defined as $\mathsf{lowerlevel}(k) = \mathsf{currlevel}(j-1) = [i, j-1\rangle$ whenever $\mathsf{currlevel}(k) = [j,k\rangle$ and $j > 1$. In contrast to $\mathsf{currlevel}(k)$, $\mathsf{lowerlevel}(k)$ is not always well defined given that it is "one level below" than $\mathsf{currlevel}(k)$, and this level may not exist. More concretely, for $\mathsf{currlevel}(k) = [j,k\rangle$ and $\mathsf{lowerlevel}(k) = [i, j-1\rangle$, these spans will look as follows:

$${}_{\mathtt{1}}a_1\,{}_{\mathtt{2}}a_2\,{}_{\mathtt{3}}\ldots {<}a_{i-1}\,{}_{\mathtt{i}}\overbrace{a_i \ldots a_{j-2}}^{\mathsf{lowerlevel}(k)}\,{}_{\mathtt{j-1}}{<}a_{j-1}\,{}_{\mathtt{j}}\overbrace{a_j \ldots a_{k-1}}^{\mathsf{currlevel}(k)}\downarrow_{\mathtt{k}}a_k \ldots\ {}_{\mathtt{n}}a_n\,{}_{\mathtt{n+1}}$$

As an example, consider the word ${}_{\mathtt{1}}(\,{}_{\mathtt{2}}(\,{}_{\mathtt{3}})\,{}_{\mathtt{4}}(\,{}_{\mathtt{5}}(\,{}_{\mathtt{6}})\,{}_{\mathtt{7}})\,{}_{\mathtt{8}})\,{}_{\mathtt{9}}$. The only well-nested spans besides the ones of the form $[i,i\rangle$ are $[1,9\rangle$, $[2,4\rangle$, $[2,8\rangle$, $[4,8\rangle$ and $[5,7\rangle$, therefore $\mathsf{currlevel}(8) = [2,8\rangle$, and $\mathsf{lowerlevel}(7) = [2,4\rangle$.

We are ready to explain the purpose of the determinization above. Let $w = a_1 a_2 \cdots a_n$ be a well-nested word and $\rho^{\det} = (S_1, \tau_1) \xrightarrow{a_1} \ldots \xrightarrow{a_{k-1}} (S_k, \tau_k)$ be the (partial) run of

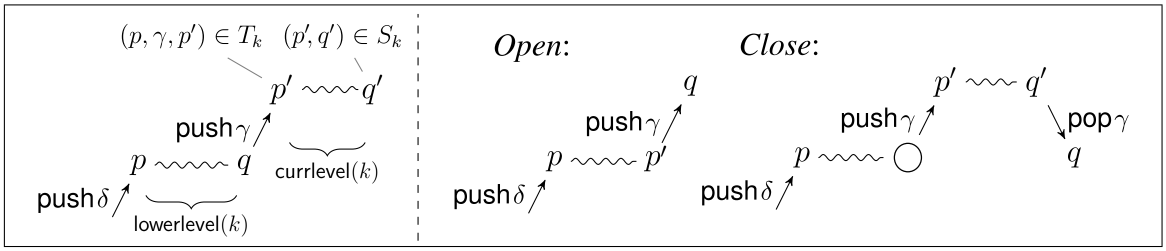


Figure 3.7. Left: An example run of some VPA $\mathcal{A}$ at step k. Right: Illustration of two nondeterministic runs for some VPA $\mathcal{A}$, as considered in the determinization process.

$\mathcal{A}^{\mathrm{det}}$ until some k. Furthermore, assume $\tau_k = T_k \cdot \tau$ for some $T_k \in \Gamma^{\mathrm{det}}$ and $\tau \in (\Gamma^{\mathrm{det}})^*$. The connection between ρ^{det} and the runs of $\mathcal{A}$ over $a_1 \dots a_{k-1}$ is given by the following invariants:

(a) $(p,q) \in S_k$ if, and only if, there exists a run $(q_1,\sigma_1) \xrightarrow{a_1} \dots \xrightarrow{a_{k-1}} (q_k,\sigma_k)$ of $\mathcal{A}$ over $a_1 \dots a_{k-1}$ such that $q_j = p$, $q_k = q$, and $\mathsf{currlevel}(k) = [j,k\rangle$.

(b) $(p,\gamma,q) \in T_k$ if, and only if, there exists a run $(q_1,\sigma_1) \xrightarrow{a_1} \dots \xrightarrow{a_{k-1}} (q_k,\sigma_k)$ of $\mathcal{A}$ over $a_1 \dots a_{k-1}$ such that $q_i = p$, $q_j = q$, $\sigma_k = \gamma\sigma$ for some σ, and $\mathsf{lowerlevel}(k) = [i, j-1\rangle$.

On one hand, (a) says that each pair $(p,q) \in S_k$ represents some non-deterministic run of $\mathcal{A}$ over w for which q is the k-th state, and p was visited on the step when the current symbol at the top of the stack was pushed. On the other hand, (b) says that $(p,\gamma,q) \in T_k$ represents some run of $\mathcal{A}$ over w for which γ is at the top of the stack, q was visited on the step when γ was pushed, and p was visited on the step when the symbol below γ was pushed (see Figure 3.7 (left)). More importantly, these conditions are exhaustive, that is, every run of $\mathcal{A}$ over $a_1 \dots a_{k-1}$ is represented by ρ^{det}.

By these two invariants, the correctness of $\mathcal{A}^{\mathrm{det}}$ easily follows, and the reader can get some intuition behind the construction of $\delta^{\mathrm{det}}(S, {<}a)$ and $\delta^{\mathrm{det}}(S,T,a{>})$ (see Figure 3.7 (right) for a graphical description). Indeed, the most important consequence of these two invariants is that a tuple $(q_j,q_k) \in S_k$ represents the interval of some run over $w[j,k\rangle$ with $\mathsf{currlevel}(k) = [j,k\rangle$ and the tuple $(q_i,\gamma,q_j) \in T_k$ represents the interval of some run over $w[i, j-1\rangle$ with $\mathsf{lowerlevel}(k) = [i, j-1\rangle$, i.e., the level below. In other words, the configuration (S_k, τ_k) of $\mathcal{A}^{\text{det}}$ forms a succinct representation of all the non-deterministic runs of $\mathcal{A}$. This is the starting point of our update algorithm, that we discuss next.

The streaming evaluation algorithm. In Algorithm 2, we present the update phase for solving the streaming version of ENUMVPANN. The main procedure is UPDATEPHASE, that receives an I/O-unambiguous VPAnn $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$ and a stream $\mathcal{S}$, reads the next (k-th) symbol and computes the set of outputs $[\![\mathcal{T}]\!](\mathcal{S}[1, k])$. More specifically, it constructs an ε-ECS $\mathcal{D}$ and a vertex v_{out} such that $\mathcal{L}_{\mathcal{D}}(v_{\text{out}}) = [\![\mathcal{T}]\!](\mathcal{S}[1, k])$ if $\mathcal{S}[1, k]$ is well-nested, and $\emptyset$ otherwise. After the UPDATEPHASE procedure is done, we can enumerate $\mathcal{L}_{\mathcal{D}}(v_{\text{out}})$ with output-linear delay by calling the enumeration phase, that is, by applying Theorem 3.4. An example execution and the resulting data structure are shown in Figure 3.8.

Towards this goal, in Algorithm 2 we make use of the following data structures. First, we use an ε-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$, nodes $v \in V$, and the operations add, union, and prod over $\mathcal{D}$ and v (see Section 3.5). For the sake of simplification, we overload the notation of these operators slightly so that if $v = \emptyset$, then $\mathsf{union}(\mathcal{D}, v, v') = \mathsf{union}(\mathcal{D}, v', v) = (\mathcal{D}, v')$. We use a constant-time-access map (which we will call hash table) S which indexes nodes v in $\mathcal{D}$ by pairs of states $(p, q) \in Q \times Q$. We denote the elements of S as "$(p, q) : v$" where (p, q) is the index and v is the content. Furthermore, we write $S_{p,q}$ to access the node v. We also use a stack T that stores hash tables: each element is a hash table that indexes vertices v in $\mathcal{D}$ by triples $(p, \gamma, q) \in Q \times \Gamma \times Q$. We assume that T has the standard stack methods push and pop where if $T = t_k \cdots t_1$, then $\mathsf{push}(T, t) = t\, t_k \cdots t_1$ and $\mathsf{pop}(T) = t_{k-1} \cdots t_1$. We write $\emptyset$ for denoting the empty stack or for checking if T is empty. Similarly to S, we use the notation $T_{p,\gamma,q}$ to access the nodes in the topmost hash table in T (i.e., T is a stack of hash tables). We assume that accessing a non-assigned index in these hash tables returns the empty set. All variables (e.g., S or T) are defined globally in Algorithm 2 and they can be accessed by any of the subprocedures. Since we use the RAM model (see Section 3.1), every operation over hash tables or stacks takes constant time.

Algorithm 2 builds the ε-ECS $\mathcal{D}$ incrementally, reading the stream $\mathcal{S}$ one letter at a time by calling `yield`$\mathcal{S}$ and keeping a counter k for the position of the current letter. For every $k \in [1, n+1]$, UPDATEPHASE builds the k-th iteration of table S and stack T, which we note as S^k and T^k, respectively. Before UPDATEPHASE is called for the first time, it runs INTIALIZE (lines 1-5) to set the initial values of k, $\mathcal{D}$, S, and T. We consider the initial S and T as the 1-st iteration, defined as $S^1 = \{(q,q) : v_\varepsilon \mid q \in I\}$ and $T^1 = \emptyset$ (i.e. the empty stack) where v_ε is a node in $\mathcal{D}$ such that $\mathcal{L}_\mathcal{D}(v_\varepsilon) = \{\varepsilon\}$ (lines 3-5).

In the k-th iteration, depending on whether the current letter is an open symbol or a close symbol, the OPENSTEP or CLOSESTEP procedures are called, updating S^{k-1} and T^{k-1} to S^k and T^k, respectively. More specifically, UPDATEPHASE adds nodes to $\mathcal{D}$ such that the nodes in S^k represent the runs over $w[j,k\rangle$ where $\mathsf{currlevel}(k) = [j,k\rangle$, and the nodes in the topmost table in T^k represent the runs over $w[i,j-1\rangle$ where $\mathsf{lowerlevel}(k) = [i,j-1\rangle$. Moreover, for a given pair (p,q), the node $S^k_{p,q}$ represents all runs over $w[j,k\rangle$ with $\mathsf{currlevel}(k) = [j,k\rangle$ that start on p and end on q. For a given triple (p,γ,q) the node $T^k_{p,\gamma,q}$ represents all runs over $w[i,j-1\rangle$ with $\mathsf{lowerlevel}(k) = [i,j-1\rangle$ that start on p, and end on q right after pushing γ onto the stack. Here, the intuition gained in the determinization of VPA is helpful. Indeed, table S^k and stack T^k are the mirror of the configuration (S_k, τ_k) of $\mathcal{A}^{\mathrm{det}}$ (recall invariants (a) and (b)).

Before formalizing the previous ideas, we will describe in more detail what the procedures OPENSTEP and CLOSESTEP exactly do. Recall that the operation $\mathsf{add}(\mathcal{D}, a)$ simply creates a node in $\mathcal{D}$ labeled as a; the operation $\mathsf{prod}(\mathcal{D}, v_1, v_2)$ returns a pair $(\mathcal{D}', v')$ such that $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_\mathcal{D}(v_1) \cdot \mathcal{L}_\mathcal{D}(v_2)$; and the operation $\mathsf{union}(\mathcal{D}, v_3, v_4)$ returns a pair $(\mathcal{D}', v')$ such that $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_\mathcal{D}(v_3) \cup \mathcal{L}_\mathcal{D}(v_4)$. To improve the presentation of the algorithm, we include a simple procedure called IFPROD (lines 20-26). Basically, this procedure receives

Algorithm 2 The update phase of the streaming evaluation algorithm for ENUMVPANN given an I/O-unambiguous VPAnn $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$ and a stream $\mathcal{S}$.

```
1:  procedure INITIALIZE($\mathcal{T}, \mathcal{S}$)
2:      $k \leftarrow 1, \mathcal{D} \leftarrow \emptyset$
3:      $(\mathcal{D}, v_\varepsilon) \leftarrow \mathsf{add}(\mathcal{D}, \varepsilon)$
4:      $S \leftarrow \{(q, q) : v_\varepsilon \mid q \in I\}$
5:      $T \leftarrow \emptyset$
6:
7:  procedure UPDATEPHASE($\mathcal{T}, \mathcal{S}$)
8:      $a \leftarrow \mathtt{yield}\mathcal{S}$
9:      if $a \in \Sigma^{<}$ then
10:         $\mathcal{D} \leftarrow$ OPENSTEP$(\mathcal{D}, a, k)$
11:     else if $a \in \Sigma^{>}$ then
12:         $\mathcal{D} \leftarrow$ CLOSESTEP$(\mathcal{D}, a, k)$
13:     $k \leftarrow k + 1$
14:     $v_{\text{out}} \leftarrow \emptyset$
15:     if $T = \emptyset$ then
16:         for each $p \in I, q \in F$ s.t. $S_{p,q} \neq \emptyset$ do
17:             $(\mathcal{D}, v_{\text{out}}) \leftarrow \mathsf{union}(\mathcal{D}, v_{\text{out}}, S_{p,q})$
18:     ENUMERATIONPHASE$(\mathcal{D}, v_{\text{out}})$
19:
20: procedure IFPROD($\mathcal{D}, v, b, k$)
21:     if $b = (a, \vartheta)$ then
22:         $(\mathcal{D}', v') \leftarrow \mathsf{add}(\mathcal{D}, (\vartheta, k))$
23:         $(\mathcal{D}', v') \leftarrow \mathsf{prod}(\mathcal{D}', v, v')$
24:     else
25:         $(\mathcal{D}', v') \leftarrow (\mathcal{D}, v)$
26:     return $(\mathcal{D}', v')$
27: procedure OPENSTEP($\mathcal{D}, {<}a, k$)
28:     $S' \leftarrow \emptyset$
29:     $T \leftarrow \mathsf{push}(T, \emptyset)$
30:     for $p \in Q$ and $(p', b, q, \gamma) \in \Delta$
31:                 with $b \in \{{<}a, ({<}a, \vartheta)\}$ do
32:         if $S_{p,p'} \neq \emptyset$ then
33:             if $S'_{q,q} = \emptyset$ then
34:                 $(\mathcal{D}, v_\varepsilon) \leftarrow \mathsf{add}(\mathcal{D}, \varepsilon)$
35:                 $S'_{q,q} \leftarrow v_\varepsilon$
36:             $v \leftarrow S_{p,p'}$
37:             $(\mathcal{D}, v) \leftarrow$ IFPROD$(\mathcal{D}, v, b, k)$
38:             $(\mathcal{D}, v) \leftarrow \mathsf{union}(\mathcal{D}, v, T_{p,\gamma,q})$
39:             $T_{p,\gamma,q} \leftarrow v$
40:     $S \leftarrow S'$
41:     return $\mathcal{D}$
42:
43: procedure CLOSESTEP($\mathcal{D}, a{>}, k$)
44:     $S' \leftarrow \emptyset$
45:     for $p, p' \in Q$ and $(q', b, \gamma, q) \in \Delta$
46:                 with $b \in \{a{>}, (a{>}, \vartheta)\}$ do
47:         if $S_{p',q'} \neq \emptyset$ and $T_{p,\gamma,p'} \neq \emptyset$ then
48:             $(\mathcal{D}, v) \leftarrow \mathsf{prod}(\mathcal{D}, T_{p,\gamma,p'}, S_{p',q'})$
49:             $(\mathcal{D}, v) \leftarrow$IFPROD$(\mathcal{D}, v, b, k)$
50:             $(\mathcal{D}, v) \leftarrow \mathsf{union}(\mathcal{D}, v, S'_{p,q})$
51:             $S'_{p,q} \leftarrow v$
52:     $T \leftarrow \mathsf{pop}(T)$
53:     $S \leftarrow S'$
54:     return $\mathcal{D}$
```

a node v, an element b which can be either an input symbol a, or a pair (a, ϑ) for some output symbol ϑ, and a position k, and computes $(\mathcal{D}', v')$ such that $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_{\mathcal{D}}(v) \cdot \{(\vartheta, k)\}$ if $b = (a, \vartheta)$, and $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_{\mathcal{D}}(v)$ otherwise.

In OPENSTEP, S^k is created (i.e. S'), and an empty table is pushed onto T^{k-1} to form T^k (line 29). Then, all nodes in S^{k-1} (i.e. S) are checked to see if the runs they represent can be extended with a transition in Δ (lines 30-32). If this is the case (line 33 onwards), a node v_ε with the ε-output is added in S^k to start a new level (lines 33-35). Then, if the transition had a non-empty output, the node $S^k_{p,p'}$ is connected with a new label node to form the node v (lines 36-37). This node is stored in $T^k_{p,\gamma,q}$, or united with the node that was already present there (lines 38-39).

In CLOSESTEP, S^k is initialized as empty (line 44). Then, the procedure looks for all of the valid ways to join a node in T^{k-1}, a node in S^{k-1}, and a transition in Δ to form a new node in S^k. More precisely, it looks for quadruples (p, γ, p', q') for which $T^{k-1}_{p,\gamma,p'}$ and $S^{k-1}_{p',q'}$ are defined, and there is a close transition that starts on q' that reads γ (lines 45-47). These nodes are joined and connected with a new label node if it corresponds (lines 48-49), and stored in $S^k_{p,q}$ or united with the node that was already present there (lines 50-51). Finally, the top of the stack T is popped after all tuples (p, γ, p', q') are checked (line 52).

As was already mentioned, in each step the construction of $\mathcal{D}$ follows the ideas of the determinization of a visibly pushdown automaton. As such, Figure 3.7 also aids to illustrate how the table S^k and the top of the stack T^k are constructed.

Example 3.2. *To illustrate the inner workings of the algorithm, we provide an example of a run with the VPAnn from Example 3.1 as input, and an input stream $\mathcal{S}$ whose first characters are $<<><>>$. At this point we remind the reader that the states of this VPAnn are named* p, q *and* r, *written in a serif-less font, and they should not be read as generic states. The execution consists in calling* INITIALIZE *over $\mathcal{T}$ and $\mathcal{S}$ and then calling* UPDATEPHASE *repeatedly six times. The resulting ECS $\mathcal{D}$ and node v_{out} will be shown to satisfy $\mathcal{L}_{\mathcal{D}}(v_{\mathsf{out}}) = [\![\mathcal{T}]\!](\mathcal{S}[1,6])$.*

Recall the three accepting runs of $\mathcal{T}$ over the nested word $<<><>>$:

$$\begin{array}{ll}
\rho_1: & \mathsf{p},\varepsilon \xrightarrow{(<,\triangleright)} \mathsf{q},\mathsf{Y} \xrightarrow{<} \mathsf{q},\mathsf{XY} \xrightarrow{>} \mathsf{q},\mathsf{Y} \xrightarrow{<} \mathsf{q},\mathsf{XY} \xrightarrow{>} \mathsf{q},\mathsf{Y} \xrightarrow{(>,\triangleleft)} \mathsf{r},\varepsilon \\
\rho_2: & \mathsf{p},\varepsilon \xrightarrow{<} \mathsf{p},\mathsf{X} \xrightarrow{(<,\triangleright)} \mathsf{q},\mathsf{YX} \xrightarrow{(>,\triangleleft)} \mathsf{r},\mathsf{X} \xrightarrow{<} \mathsf{r},\mathsf{XX} \xrightarrow{>} \mathsf{r},\mathsf{X} \xrightarrow{>} \mathsf{r},\varepsilon \\
\rho_3: & \mathsf{p},\varepsilon \xrightarrow{<} \mathsf{p},\mathsf{X} \xrightarrow{<} \mathsf{p},\mathsf{XX} \xrightarrow{>} \mathsf{p},\mathsf{X} \xrightarrow{(<,\triangleright)} \mathsf{q},\mathsf{YX} \xrightarrow{(>,\triangleleft)} \mathsf{r},\mathsf{X} \xrightarrow{>} \mathsf{r},\varepsilon
\end{array}$$

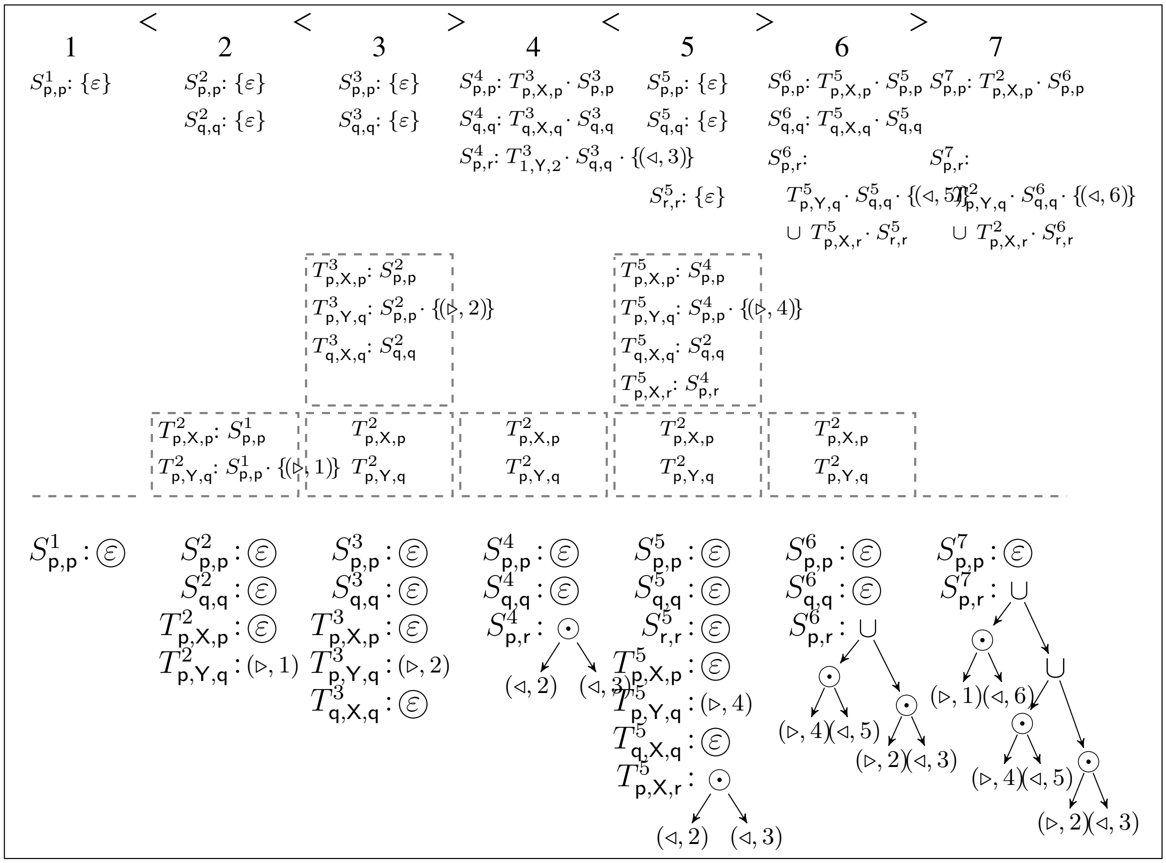

$<$ $<$ $>$ $<$ $>$ $>$
1 2 3 4 5 6 7
$S^1_{p,p}$: $\{\varepsilon\}$
$S^2_{p,p}$: $\{\varepsilon\}$
$S^2_{q,q}$: $\{\varepsilon\}$
$S^3_{p,p}$: $\{\varepsilon\}$
$S^3_{q,q}$: $\{\varepsilon\}$
$S^4_{p,p}$: $T^3_{p,X,p} \cdot S^3_{p,p}$
$S^4_{q,q}$: $T^3_{q,X,q} \cdot S^3_{q,q}$
$S^4_{p,r}$: $T^3_{1,Y,2} \cdot S^3_{q,q} \cdot \{(\triangleleft, 3)\}$
$S^5_{p,p}$: $\{\varepsilon\}$
$S^5_{q,q}$: $\{\varepsilon\}$
$S^5_{r,r}$: $\{\varepsilon\}$
$S^6_{p,p}$: $T^5_{p,X,p} \cdot S^5_{p,p}$
$S^6_{q,q}$: $T^5_{q,X,q} \cdot S^5_{q,q}$
$S^6_{p,r}$:
$T^5_{p,Y,q} \cdot S^5_{q,q} \cdot \{(\triangleleft, 5)\}$
$\cup\ T^5_{p,X,r} \cdot S^5_{r,r}$
$S^7_{p,p}$: $T^2_{p,X,p} \cdot S^6_{p,p}$
$S^7_{p,r}$:
$T^2_{p,Y,q} \cdot S^6_{q,q} \cdot \{(\triangleleft, 6)\}$
$\cup\ T^2_{p,X,r} \cdot S^6_{r,r}$
$T^3_{p,X,p}$: $S^2_{p,p}$
$T^3_{p,Y,q}$: $S^2_{p,p} \cdot \{(\triangleright, 2)\}$
$T^3_{q,X,q}$: $S^2_{q,q}$
$T^5_{p,X,p}$: $S^4_{p,p}$
$T^5_{p,Y,q}$: $S^4_{p,p} \cdot \{(\triangleright, 4)\}$
$T^5_{q,X,q}$: $S^2_{q,q}$
$T^5_{p,X,r}$: $S^4_{p,r}$
$T^2_{p,X,p}$: $S^1_{p,p}$
$T^2_{p,Y,q}$: $S^1_{p,p} \cdot \{(\triangleright, 1)\}$
$T^2_{p,X,p}$
$T^2_{p,Y,q}$
$S^1_{p,p}$: ε
$S^2_{p,p}$: ε
$S^2_{q,q}$: ε
$T^2_{p,X,p}$: ε
$T^2_{p,Y,q}$: $(\triangleright, 1)$
$S^3_{p,p}$: ε
$S^3_{q,q}$: ε
$T^3_{p,X,p}$: ε
$T^3_{p,Y,q}$: $(\triangleright, 2)$
$T^3_{q,X,q}$: ε
$S^4_{p,p}$: ε
$S^4_{q,q}$: ε
$S^4_{p,r}$: $\odot$
$(\triangleleft, 2)$ $(\triangleleft, 3)$
$S^5_{p,p}$: ε
$S^5_{q,q}$: ε
$S^5_{r,r}$: ε
$T^5_{p,X,p}$: ε
$T^5_{p,Y,q}$: $(\triangleright, 4)$
$T^5_{q,X,q}$: ε
$T^5_{p,X,r}$: $\odot$
$(\triangleleft, 2)$ $(\triangleleft, 3)$
$S^6_{p,p}$: ε
$S^6_{q,q}$: ε
$S^6_{p,r}$: $\cup$
$\odot$ $\odot$
$(\triangleright, 4)$ $(\triangleleft, 5)$
$(\triangleright, 2)$ $(\triangleleft, 3)$
$S^7_{p,p}$: ε
$S^7_{p,r}$: $\cup$
$\odot$ $\cup$
$(\triangleright, 1)$ $(\triangleleft, 6)$
$\odot$ $\odot$
$(\triangleright, 4)$ $(\triangleleft, 5)$
$(\triangleright, 2)$ $(\triangleleft, 3)$

Figure 3.8. Example of running Algorithm 2 over the VPAnn $\mathcal{T}$ from Figure 3.1 and a stream $\mathcal{S}$ such that $\mathcal{S}[1,6] = <<><>>$. The bottom part illustrates the state of $\mathcal{D}$, and the node from $\mathcal{D}$ that is stored at every new index at each step.

The call to INITIALIZE *is depicted in column 1. The table S is created and its only index is (p,p). This index includes the only node in $\mathcal{D}$, which is an ε-node. The stack T is also created and is empty at this point.*

The following columns represent the state of the indices and what is stored in each after each call to UPDATEPHASE*. The upper region shows the indices from S as they end up being defined at the end of each call; the middle region shows the state of the stack T in full, also at the end of each call (the stacks in the figure show the latest element on top; whenever the stack is empty, it is represented by a single horizontal line); the bottom region shows, in each column k, the nodes in the ECS $\mathcal{D}$ that are referenced directly by S^k*

and T^k. Note that the indices of each S^k are lost at the end of step $k-1$, yet the indices of T^k might be used in a later step, as is the case of T^2 at step 7.

To explain a bit further, in these regions, the indices are mapped to their respective sets, which are written as the logic of the algorithm defines them. Let us explain the definition of two particular indices to illustrate the constructions in more detail.

- *First is index $T^3_{\mathsf{p,Y,q}}$: Since it is first defined in step three, after an open symbol, it is built through lines 30-39. The only relevant transition for this index is* $(\mathsf{p}, (<, \triangleright), \mathsf{q}, \mathsf{Y})$. *After seeing that $S^2_{\mathsf{p,p}}$ is not empty, in this same iteration the algorithm builds the index $S^3_{\mathsf{q,q}}$ with an ε-node (lines 34,35), and after doing this, it builds a node v that represents the concatenation of $S^2_{\mathsf{p,p}}$ with the set* $\{(\triangleright, 2)\}$ *(line 37) – and since $S^2_{\mathsf{p,p}}$ contained only an ε-node, v ends up as a bottom node with label* $(\triangleright, 2)$. *This node then goes through a union with an empty index (line 38), so it remains unchanged, and then it is assigned to $T^3_{\mathsf{p,Y,q}}$ (line 39). In the end, the node in $\mathcal{D}$ that is assigned to the index is only this bottom node v, which is depicted in the bottom part of the figure.*
- *Now let us see index $S^7_{\mathsf{p,r}}$: This index is defined through lines 45-51. The relevant transitions in this case are* $(\mathsf{q}, (>, \triangleleft), \mathsf{Y}, \mathsf{r})$ *and* $(\mathsf{r}, >, \mathsf{X}, \mathsf{r})$. *Note how when the procedure goes through lines 48-49 for the first transition, it defines a node v that represents the concatenation of $T^2_{\mathsf{p,Y,q}}$, $S^6_{\mathsf{q,q}}$ and the set* $\{(\triangleleft, 6)\}$, *and when it goes through these lines for the second transition, v represents the concatenation of $T^2_{\mathsf{p,X,r}}$ and $S^6_{\mathsf{r,r}}$. It is worth noting that indices $T^2_{\mathsf{p,Y,q}}$ and $T^2_{\mathsf{p,X,r}}$ were defined back in step 2, remained in the stack during steps 3 through 6, and only now at step 7 are they used. The two mentioned sets are merged by the union operator in line 50. Finally, the index is assigned a union node that represents this merging, which can be also seen in the bottom part of the figure.*

To conclude the example, note how the only relevant index that is used in line 17 is $S^7_{\mathsf{p,r}}$. At the very end, the node v_{out} is equal to the union node stored at this index, which when enumerated will produce the set $[\![\mathcal{T}_1]\!](\mathcal{S}[1,6])$.

Correctness of the streaming evaluation algorithm. The way how the table S^k and the stack T^k are constructed in Algorithm 2 is formalized in the following result. Recall that a run of $\mathcal{T}$ over a well-nested word $w = a_1 \cdots a_n$ is a sequence of the form:

$$\rho = (q_1, \sigma_1) \xrightarrow{b_1} \ldots \xrightarrow{b_n} (q_{n+1}, \sigma_{n+1})$$

where each $b_i \in \{a_i, (a_i, \mathsf{o}_{i})\}$. Given a span $[i, j\rangle$, define a subrun of ρ as a subsequence $\rho[i, j\rangle = (q_i, \sigma_i) \xrightarrow{b_i} \ldots \xrightarrow{b_{j-1}} (q_j, \sigma_j)$. We also extend the function out to receive a subrun $\rho[i, j\rangle$ in the following way: $\mathsf{out}(\rho[i, j\rangle) = \mathsf{out}(\mathsf{o}_{i}, i) \cdot \ldots \cdot \mathsf{out}(\mathsf{o}_{j-1}, j - 1)$. Finally, define $\mathrm{Runs}(\mathcal{T}, w)$ as the set of all runs of $\mathcal{T}$ over w.

Lemma 3.2. *Let $\mathcal{T}$ be a VPAnn and $w = a_1 \cdots a_n$ be a well-nested word. While running the procedure* UPDATEPHASE *of Algorithm 2, for every $k \in [1, n + 1]$, every pair of states p, q and stack symbol γ the following hold:*

(i) *$\mathcal{L}_{\mathcal{D}}(S^k_{p,q})$ has exactly all sequences $\mathsf{out}(\rho[j, k\rangle)$ such that $\rho \in \mathrm{Runs}(\mathcal{T}, w[1, k\rangle)$, $\mathsf{currlevel}(k) = [j, k\rangle$, and $\rho[j, k\rangle$ starts on p and ends on q.*

(ii) *If $\mathsf{lowerlevel}(k)$ is defined, then $\mathcal{L}_{\mathcal{D}}(T^k_{p,\gamma,q})$ has exactly all sequences $\mathsf{out}(\rho[i, j\rangle)$ such that $\rho \in \mathrm{Runs}(\mathcal{T}, w[1, j\rangle)$, $\mathsf{lowerlevel}(k) = [i, j - 1\rangle$, and $\rho[i, j\rangle$ starts on p, ends on q, and the last symbol pushed onto the stack was γ.*

PROOF. We will prove the lemma by induction on k. The case $k = 1$ is trivial since $\mathsf{currlevel}(1) = [1, 1\rangle$, $S^1_{p,q}$ is empty and $\mathsf{lowerlevel}(1)$ is not defined. We assume that statements 1 and 2 of the lemma are true for $k - 1$ and below.

If $a_k \in \Sigma^<$, the algorithm proceeds into OPENSTEP to build S^k and T^k. Statement 1 can be proved trivially since $\mathsf{currlevel}(k) = [k, k\rangle$, similarly as for the base case. For statement 2 let $\mathsf{lowerlevel}(k) = [i, k - 1\rangle$, and consider a run $\rho \in \mathrm{Runs}(\mathcal{T}, w[1, k\rangle)$ such that $\rho[i, k\rangle$ starts on p and ends on q for some p, q and γ, and let p' be its second-to-last state. Since a_k is an open symbol, then the string $a_{i+1} \cdots a_{k-1}$ is well-nested, so it holds that $\mathsf{currlevel}(k - 1) = [i, k - 1\rangle$. Therefore, from our hypothesis it holds that $\mathcal{L}_{\mathcal{D}}(S^{k-1}_{p,p'})$ contains $\mathsf{out}(\rho[i, k - 1\rangle)$, and so, $\mathsf{out}(\rho[i, k\rangle)$ is included in $\mathcal{L}_{\mathcal{D}}(T^k_{p,\gamma,q})$ at some iteration of $T^k_{p,\gamma,q}$ at line 39. To show that every element in $\mathcal{L}_{\mathcal{D}}(T^k_{p,\gamma,q})$ corresponds to some run $\rho \in \mathrm{Runs}(\mathcal{T}, w[1,k\rangle)$, we note that the only step that modifies $T^k_{p,\gamma,q}$ is line 39, which is reached only when a valid subrun from i to k can be constructed.

If $a_k \in \Sigma^{>}$, the algorithm proceeds into CLOSESTEP to build S^k and T^k. Let $\mathsf{currlevel}(k) = [j,k\rangle$. In this case, statement 2 can be deduced directly from the hypothesis since $j < k$ and the table on the top of T^k is the same as T^j. To prove statement 1, note that since a_k is a close symbol it holds that $\mathsf{currlevel}(k-1) = [j', k-1\rangle$ and $\mathsf{lowerlevel}(k-1) = [j, j'-1\rangle$ for some j'. Consider a run $\rho \in \mathrm{Runs}(\mathcal{T}, w)$ such that $\rho[j,k\rangle$ starts on p, ends on q, and the last symbol pushed onto the stack is γ. This run can be subdivided into three subruns from p to p', from p' to q', and a transition from q' to q as it is illustrated in Figure 3.7 (Right). The first two subruns correspond to $\rho[j, j'+1\rangle$ and $\rho[j', k-1\rangle$, for which $\mathsf{out}(\rho[j,j'+1\rangle) \in \mathcal{L}_{\mathcal{D}}(T^{k-1}_{p,\gamma,q})$ and $\mathsf{out}(\rho[j',k-1\rangle) \in \mathcal{L}_{\mathcal{D}}(S^{k-1}_{p',q'})$. Therefore, $\mathsf{out}(\rho[j,k\rangle) \in \mathcal{L}_{\mathcal{D}}(S^k_{p,q})$ at some iteration of line 51. To show that every element in $S^k_{p,q}$ corresponds to some run $\rho \in \mathrm{Runs}(\mathcal{T}, w[1,k\rangle)$, note that the only line at which $S^k_{p,q}$ is modified are is line 51, which is reached only when a valid run from j to k has been constructed. □

Since w is well nested, then $\mathsf{currlevel}(|w|+1) = [1, |w|+1\rangle$, and so, the lemma implies that the nodes in $S^{|w|+1}$ represent all runs of $\mathcal{T}$ over w. Then, whenever $\mathcal{S}[1,k]$ is well-nested, the stack T is empty (i.e., $T = \emptyset$) and there may be something to enumerate (line 15). By taking the union of all pairs in S^{k+1} that represent accepting runs (as is done in lines 16-17), we can conclude the correctness of Algorithm 2.

Theorem 3.5. *Given a VPAnn $\mathcal{T}$ and a stream $\mathcal{S}$,* UPDATEPHASE$(\mathcal{T}, \mathcal{S})$ *fulfils the conditions of a streaming evaluation algorithm and, after reading the k-th symbol, produces a pair $(\mathcal{D}, v_{\mathrm{out}})$ such that $\mathcal{L}_{\mathcal{D}}(v_{\mathrm{out}}) = [\![\mathcal{T}]\!](\mathcal{S}[1,k])$.*

At this point, we address the fact that $\mathcal{D}$ needs to be duplicate-free in order to enumerate all the outputs from $(\mathcal{D}, v_{\mathrm{out}})$ without repetitions. This is guaranteed, essentially, by the fact that $\mathcal{T}$ is I/O-unambiguous. Indeed, the previous result holds even if $\mathcal{T}$ is not I/O-unambiguous. The next result guarantees that the output can be enumerated efficiently.

Lemma 3.3. *Let $\mathcal{T}$ be an I/O-unambiguous VPAnn. While running the* UPDATEPHASE *procedure of Algorithm 2, the ε-ECS $\mathcal{D}$ is duplicate-free at every step.*

PROOF. For the sake of simplification, assume that $\mathcal{T}$ is I/O-unambiguous on subruns as well. Formally, we extend the condition so that for every well-nested word w, span $[i, j\rangle$ and $\mu \in \Omega^*$, there exists only one run $\rho \in \text{Runs}(\mathcal{T}, w)$ such that $\mu = \mathsf{out}(\rho[i, j\rangle)$. Towards a contradiction, we assume that $\mathcal{D}$ is not duplicate-free. Therefore, at least one of these conditions must hold: (1) there is some union node v in $\mathcal{D}$ for which $Ł_{\mathcal{D}}(\ell(v))$ and $Ł_{\mathcal{D}}(r(v))$ are not disjoint, or (2) there is some product node v for which there are at least two ways to decompose some $\mu \in Ł_{\mathcal{D}}(v)$ in non-empty strings μ_1 and μ_2 such that $\mu = \mu_1 \cdot \mu_2$ and $\mu_1 \in Ł(\ell(v))$ and $\mu_2 \in Ł_{\mathcal{D}}(r(v))$.

Assume the first condition is true and let v be a union node that satisfies it, and let k be the step in which it was added to $\mathcal{D}$. If this node was added on OPENSTEP, then the node v represents a subset of the subruns defined in condition 1 of Lemma 3.2. Consider two different iterations of lines 38-39 on step k where two nodes v and v' were united for which there is an element $\mu \in Ł_{\mathcal{D}}(v) \cap Ł_{\mathcal{D}}(v')$. Since these nodes were assigned to $T_{p,\gamma,q}$ on different iterations, the states p' that were being considered must have been different. Therefore, if $\mathsf{lowerlevel}(k) = [i, j\rangle$, $\mu = \mathsf{out}(\rho[i, k\rangle) = \mathsf{out}(\rho'[i, k\rangle)$ for two runs ρ and ρ' where the $(k-1)$-th state is different. This violates the condition that $\mathcal{T}$ is I/O-unambiguous. If this node was added on CLOSESTEP, we can follow an analogous argument. Note that union nodes created on a prod operation are duplicate-free by construction (see Theorem 3.4).

Assume now that the second condition is true and let v be a node for which the condition holds and let k be the step where it was created. We note that this node could not have been created in OPENSTEP since the only step that creates product nodes is line 39, where v_λ has the label (ϑ, k), and $S_{p,p'}$ is connected to nodes that were created in a previous step, so all of the elements $\mu \in \text{Ł}(S_{p,p'})$ only contain pairs (ϑ, j) where $j < k$. We can follow a similar argument to prove that this node could not have been created in line 49 of CLOSESTEP. We now have that v was created in line 48 of OPENSTEP, and therefore $\ell(v) = T^{k-1}_{p,\gamma,q}$ and $r(v) = S^{k-1}_{p',q'}$ unless either of these indices were empty. However, that is not possible since we assumed that the step where v was created was k, and if either were empty, no node would have been created. Now let $\mu \in \text{Ł}(v)$ be such that there exist strings $\mu_1, \mu_1' \in \text{Ł}(T^{k-1}_{p,\gamma,q})$ and $\mu_2, \mu_2' \in \text{Ł}(S^{k-1}_{p',q'})$ such that $\mu = \mu_1\mu_2 = \mu_1'\mu_2'$ and $\mu_1 \neq \mu_1'$. Without loss of generality, let μ'' be the non-empty suffix in μ_1 such that $\mu_1'\mu'' = \mu_1$. Here we reach a contradiction since μ'' is a prefix of μ_2 and thus it must contain a pair (ϑ, j) such that and $j \in \mathsf{lowerlevel}(k)$ and $j \in \mathsf{currlevel}(k)$, which is not possible. □

The complexity of this algorithm can be easily deduced from the fact that the ε-ECS operations we use take constant time (Theorem 3.4). For a VPAnn $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$, in each of the calls to OPENSTEP, lines 32-39 perform a constant number of instructions, and they are visited at most $|Q||\Delta|$ times. In each of the calls to CLOSESTEP, lines 47-51 perform a constant number of instructions, and they are visited at most $|Q|^2|\Delta|$ times. Combined with Theorem 3.5, Lemma 3.3, and Theorem 3.4, this proves the main result in this chapter (i.e., Theorem 3.1).

Extension to prefixes of well-nested words. The aforementioned algorithm is described only for well-nested inputs. In this subsection, we will give the main ideas of how it can be extended to handle prefixes of well-nested words. First, we note that if Algorithm 2 were to be used over a prefix $w[1, k\rangle$ of a well-nested word $w \in \Sigma^{<*>}$, the resulting $\mathcal{D}$ in line 18 would represent only outputs from the span $[j, k\rangle = \mathsf{currlevel}(k)$. Then, the idea is to add a second stack T' to the algorithm, and the invariant will be that the topmost element in it contains the outputs that correspond to the span $[1, j\rangle$. To be a bit more precise, each element contains an extra one-dimensional table, indexed by states in Q, such that T'_q is the union of $T_{p,\gamma,q}$ for every state p and stack symbol γ. Recall that we are abusing notation by using T' to also represent the topmost element in the stack. This extension can be added to OpenStep with no extra cost, and in CloseStep we need only to add an extra pop to the topmost element in T'. The final change is to replace line 17 by the steps $(\mathcal{D}, v) \leftarrow \mathsf{prod}(\mathcal{D}, T'_p, S_{p,q})$ and $(\mathcal{D}, v_{\text{out}}) \leftarrow \mathsf{union}(\mathcal{D}, v_{\text{out}}, v)$, and of course eliminate the restriction that $p \in I$ in the **for** argument. For the sake of presentation, we omit a formal proof of correctness.

3.7. Application: document spanners and extraction grammars

Liat Peterfreund (Peterfreund, 2021) proposed using extraction grammars to specify document spanners, which is the natural extension of regular spanners to a controlled form of recursion. Furthermore, she provided an enumeration algorithm for unambiguous functional extraction grammars that outputs the results with constant delay after quintic time preprocessing (i.e., in the document), later improved to cubic time (see Chapter 4). By restricting to the class of visibly pushdown extraction grammars, we can show a streaming enumeration algorithm with update-time that is independent of the document, and output-linear delay. We proceed by recalling the framework of document spanners and extraction grammars to define the class of visibly pushdown extraction grammars and state the main algorithmic result of the section.

We use the framework of extraction grammars, recently proposed in (Peterfreund, 2021), to specify document spanners. For $\mathcal{X} \subseteq \mathsf{Vars}$, recall that $\mathsf{C}_{\mathcal{X}} = \{\vdash^x, \dashv^x \mid x \in \mathcal{X}\}$ is the set of captures of $\mathcal{X}$. An *extraction context-free grammar*, or *extraction grammar* for short, is a tuple:

$$G = (\mathcal{X}, V, \Sigma, S, P)$$

such that $\mathcal{X} \subseteq \mathsf{Vars}$, V is a finite set of non-terminal symbols with $V \cap \mathsf{Vars} = \emptyset$, Σ is the alphabet of terminal symbols with $\Sigma \cap V = \emptyset$, $S \in V$ is the start symbol, and $P \subseteq V \times (V \cup \Sigma \cup \mathsf{C}_{\mathcal{X}})^*$ is a finite relation. In the literature, the elements of V are also referred to as “variables”, but we call them non-terminals to distinguish V from Vars. Each pair $(A, \alpha) \in P$ is called a production and we write it as $A \to \alpha$. The set of productions

P defines the (left) derivation relation:

$$\Rightarrow_G \;\subseteq\; (V \cup \Sigma \cup \mathsf{C}_{\mathcal{X}})^* \times (V \cup \Sigma \cup \mathsf{C}_{\mathcal{X}})^*$$

such that $wA\beta \Rightarrow_G w\alpha\beta$ iff $w \in (\Sigma \cup \mathsf{C}_{\mathcal{X}})^*$, $A \in V$, $\alpha, \beta \in (V \cup \Sigma \cup \mathsf{C}_{\mathcal{X}})^*$, and $A \to \alpha \in P$. We denote by $\Rightarrow_G^*$ the reflexive and transitive closure of $\Rightarrow_G$. Then the language defined by G is $\mathcal{L}(G) = \{w \in (\Sigma \cup \mathsf{C}_{\mathcal{X}})^* \mid S \Rightarrow_G^* w\}$. Naturally, each word $w \in \mathcal{L}(G)$ is a ref-word.

In order to define a spanner from G, we need to interpret ref-words as mappings (Freydenberger, 2019). The spanner $[\![G]\!]$ associated to an extraction grammar G is defined over any document $d \in \Sigma^*$ as:

$$[\![G]\!](d) \;=\; \{\, \mu^r \;\mid\; r \in \mathcal{L}(G),\; r \text{ is valid for } \mathcal{X},\; \text{and } \mathsf{plain}(r) = d \,\}.$$

There are two classes of extraction grammars that are relevant for our discussion. The first class of grammars is called functional extraction grammars. An extraction grammar G is *functional* if every $r \in \mathcal{L}(G)$ is valid for $\mathcal{X}$. In (Peterfreund, 2021) it was shown that for any extraction grammar G there exists an equivalent functional grammar G' (i.e. $[\![G]\!] = [\![G']\!]$). Non-functional grammars are problematic given that, even for regular spanners, their decision problems easily become intractable (Maturana, Riveros, & Vrgoc, 2018; Freydenberger, Kimelfeld, & Peterfreund, 2018). For this reason, from now on we restrict to functional extraction grammars without loss of expressive power. The second class of grammars is called unambiguous extraction grammars. An extraction grammar G is *unambiguous* if for every $r \in \mathcal{L}(G)$ there exists exactly one path from S to r in the graph $((V \cup \Sigma \cup \mathsf{C}_{\mathcal{X}})^*, \Rightarrow_G)$. In other words, there exists exactly one leftmost derivation.

We consider now a sub-class of extraction grammars for nested words. Let $\Sigma = (\Sigma^{<}, \Sigma^{>}, \Sigma^{|})$ be a structured alphabet. A *visibly pushdown extraction grammar* (VPEG) is a functional extraction grammar $G = (\mathcal{X}, V, \Sigma, S, P)$ in which $\Sigma = (\Sigma^{<}, \Sigma^{>}, \Sigma^{|})$ is a structured alphabet, and all the productions in P are of one of the following forms: (1) $A \to \varepsilon$; (2) $A \to aB$ such that $a \in \Sigma^{|} \cup \mathsf{C}_{\mathcal{X}}$ and $B \in V$; (3) $A \to {<}a\, B\, b{>}\, C$ such that ${<}a \in \Sigma^{<}$, $b{>} \in \Sigma^{>}$, and $B, C \in V$. Intuitively, rules $A \to aB$ allow producing arbitrary sequences of neutral symbols, where rules $A \to {<}a\, B\, b{>}\, C$ forces the word to be well-nested.

Visibly pushdown extraction grammars are a subclass of extraction grammars that works for well-nested documents. In fact, the reader can notice that the visibly pushdown restriction for extraction grammars is the analog counterpart of visibly pushdown grammars[3] introduced in (Alur & Madhusudan, 2004b). Therefore, one could expect VPEGs to be less expressive than extraction grammars. Interestingly, we can use Theorem 3.1 to give an efficient streaming enumeration algorithm for evaluating VPEG.

Before stating the main result of this section, we will specify the format in which the results are to be enumerated. In a similar fashion as in Section 3.3, we define the support of a $(\mathcal{X}, d)$-mapping μ, denoted by $\mathsf{supp}(\mu)$, as the set of positions mentioned in μ, namely,

$$\mathsf{supp}(\sigma) = \{i \mid \mu(x) = [i, j\rangle \text{ or } \mu(x) = [j, i\rangle \text{ for some } j \in [1, |d| + 1] \text{ and } x \in \mathcal{X}\}.$$

Let $\mathsf{supp}(\sigma) = \{i_1, \ldots, i_m\}$ such that $i_j < i_{j+1}$ for every $j < m$. Then, we define the encoding of μ as: $\mathsf{enc}(\mu) = (S_1, i_1) \ldots (S_m, i_m)$ where $S_i = \{\vdash^x \mid \mu(x) = [i, j\rangle \text{ for some } j\} \cup \{\dashv^x \mid \mu(x) = [j, i\rangle \text{ for some } j\}$. The enumeration algorithm we provide thus enumerates the encoding of every mapping from $[\![G]\!](d)$.

Theorem 3.6. *Fix a set of variables $\mathcal{X}$. The problem of, given a functional visibly pushdown extraction grammar $G = (\mathcal{X}, V, \Sigma, S, P)$ and a stream $\mathcal{S}$, enumerating all $(\mathcal{X}, \mathcal{S}[1, n])$-mappings of $[\![G]\!](d)$ can be solved with update-time $\mathcal{O}(2^{|G|^3})$, and output-linear delay. Furthermore, if G is restricted to also be unambiguous, then the problem can be solved with update-time $\mathcal{O}(|G|^3)$.*

PROOF. To link the model of visibly pushdown extraction grammars and visibly pushdown automata we define another class of automata based on the ideas in (Peterfreund,

[3] The definition of visibly pushdown grammars in (Alur & Madhusudan, 2004b) is slightly more complicated given that they consider nested words that are not necessary well-nested (see the discussion in Section 3.1).

2021). Let $\mathcal{A}$ be an *extraction visibly pushdown automaton* (EVPA) if $\mathcal{A} = (X, Q, \Sigma, \Gamma, \Delta, I, F)$ where X is a set of variables, Q is a set of states, $\Sigma = (\Sigma^{<}, \Sigma^{>}, \Sigma^{|})$ is a visibly pushdown alphabet, Γ is a stack alphabet, $\Delta \subseteq (Q \times \Sigma^{<} \times Q \times \Gamma) \cup (Q \times \Sigma^{>} \times \Gamma \times Q) \cup (Q \times (\Sigma^{|} \cup \mathsf{C}_X) \times Q)$, I is a set of initial states, and F is a set of final states. Note that this is a simple extension of VPA where neutral transitions are allowed to read neutral symbols or captures in X. We define the runs as we did for VPA, except the input in an EVPA is a ref-word $r \in (\Sigma \cup \mathsf{C}_X)^*$, and we say that $r \in \mathcal{L}(\mathcal{A})$ if and only if there is an accepting run of $\mathcal{A}$ on r. Furthermore, we say that $\mathcal{A}$ is functional if every $r \in \mathcal{L}(\mathcal{A})$ is valid for X, and $\mathcal{A}$ is unambiguous if for every ref-word $r \in (\Sigma \cup \mathsf{C}_X)^*$ there exists at most one accepting run of $\mathcal{A}$ over r. It is clear that this is a direct counterpart to visibly pushdown extraction grammars. Therefore, we can use the ideas in (Alur & Madhusudan, 2004b) to obtain a one-to-one conversion from one to another.

CLAIM 3.3. *For a given VPEG G there exists an EVPA $\mathcal{A}_G$ such that $\mathcal{L}(G) = \mathcal{L}(\mathcal{A}_G)$. Moreover, $\mathcal{A}_G$ is unambiguous iff G is unambiguous, and $\mathcal{A}_G$ can be constructed in time $\mathcal{O}(|G|)$.*

PROOF. Let $G = (X, V, \Sigma, S, P)$ be a VPEG. We construct an EVPA given by $\mathcal{A}_G = (X, Q, \Sigma, \Gamma, \Delta, I, F)$ such that $\mathcal{L}(G) = \mathcal{L}(\mathcal{A}_G)$ using an almost identical construction to the one in Theorem 6 of (Alur & Madhusudan, 2004b). The only differences arise from our structure being defined for well-nested words, which makes the construction a bit simpler, and from the case where a production is of the form $X \to aY$, for which we add the possibility that $a \in \mathsf{C}_X$. This construction provides one transition in Δ per production in P, and in some cases, it needs to check if a variable is *nullable* (see (Alur & Madhusudan, 2004b)). Checking if a single variable is nullable is costly, but by a constant number of traversals in P it is possible to check which variables in X are nullable or not, which can be done before building Δ. Therefore, this construction can be done in time $\mathcal{O}(|P|)$. Finally, $\mathcal{A}_G$ is unambiguous if and only if G is unambiguous, which is another consequence of Theorem 6 of (Alur & Madhusudan, 2004b). □

We define the spanner $[\![\mathcal{A}]\!]$ for a given EVPA $\mathcal{A}$ identically as in the definition of an extraction grammar. Note that from the proof it follows that if G is functional, then $\mathcal{A}_G$ is functional as well.

For the next part of the proof, assume that $\mathcal{A}_G$ is functional and unambiguous. We will show that for an EVPA $\mathcal{A}$ and stream $\mathcal{S}$, the set $[\![\mathcal{A}]\!](d)$, can be enumerated with output-linear delay and update-time $\mathcal{O}(|\mathcal{A}_G|^3)$, for $d = \mathcal{S}[1, n]$. Towards this goal, we will start with an unambiguous $\mathcal{A}_G = (X, Q, \Sigma, \Gamma, \Delta, I, F)$ and convert it into a VPAnn $\mathcal{T}_G$ with output symbol set 2^{C_X}, and then use our algorithm to enumerate the set $[\![\mathcal{T}_G]\!](d')$ where $d' = d\#$, using a dummy symbol $\#$. We will show that this construction is correct because $[\![\mathcal{T}_G]\!](d') = \{\mathsf{enc}(\mu) \mid \mu \in [\![A_G]\!](d)\}$.

Let $\mathcal{T}_G = (Q', \Sigma', \Gamma, \Omega, \Delta', I, F')$ where $Q' = Q \cup \{q_f\}$, $\Sigma' = (\Sigma^{<}, \Sigma^{>}, \Sigma^{|}_{\#})$ such that $\Sigma^{|}_{\#} = \Sigma^{|} \cup \{\#\}$, $\Omega = 2^{\mathsf{C}_X}$ and $F' = \{q_f\}$. To define Δ' we introduce a merge operation on a path over $\mathcal{A}_G$. This is defined for any non-empty sequence of transitions $t = (p_0, v_1, p_1)(p_1, v_2, p_2) \cdots (p_{m-1}, v_m, p_m) \in \Delta^*$ such that $v_i \in \mathsf{C}_X$ for $i \in [1, m]$. If these conditions hold, we say that t is a v-path ending in p_m. Let t be such a v-path and let $S = \{v_1, \ldots, v_m\}$. For ${<}a \in \Sigma^{<}$, and a transition $(p, {<}a, \gamma, q)$ such that $p = p_m$, we define $\mathsf{merge}(t, (p, {<}a, \gamma, q)) := (p_0, {<}a, S, \gamma, q)$. For $a{>} \in \Sigma^{>}$ and a transition $(p, a{>}, q, \gamma)$ such that $p = p_m$, we define $\mathsf{merge}(t, (p, a{>}, q, \gamma)) := (p_0, a{>}, S, q, \gamma)$. For $a \in \Sigma^{|}$ and a transition (p, a, q) such that $p = p_m$, we define $\mathsf{merge}(t, (p, a, q)) := (p_0, a, S, q)$. We now define Δ' as follows:

$$\begin{aligned}
\Delta' \;=\; & \big(\Delta \setminus (Q \times \mathsf{C}_X \times Q)\big) \cup \\
& \quad \{\mathsf{merge}(t, (p, {<}a, \gamma, q)) \mid \text{there is a v-path } t \in \Delta^* \text{ ending in } p \text{ and } (p, {<}a, \gamma, q) \in \Delta\} \cup \\
& \quad \{\mathsf{merge}(t, (p, a{>}, q, \gamma)) \mid \text{there is a v-path } t \in \Delta^* \text{ ending in } p \text{ and } (p, a{>}, q, \gamma) \in \Delta\} \cup \\
& \quad \{\mathsf{merge}(t, (p, a, q)) \mid \text{there is a v-path } t \in \Delta^* \text{ ending in } p \text{ and } (p, a, q) \in \Delta\} \cup \\
& \quad \{\mathsf{merge}(t, (p, \#, q_f)) \mid \text{there is a v-path } t \in \Delta^* \text{ ending in } p \text{ and } p \in F\}.
\end{aligned}$$

Since $\mathcal{A}_G$ is functional, the C_X-transitions in Δ define a DAG over Q, from which we deduce that Δ is well-defined. Let us make note that we will use the symbol ω to refer to an output of a VPAnn to avoid confusion with mappings.

Before proving the equivalence between these two structures, let us note that a v-path $t = (p_0, v_1, p_1)(p_1, v_2, p_2) \dots (p_{m-1}, v_m, p_m)$ can be translated directly into a subrun:

$$\rho_t \;=\; p_0 \xrightarrow{v_1} p_1 \xrightarrow{v_2} \dots \xrightarrow{v_m} p_m.$$

We indistinguishably apply the operation merge over v-paths or subruns of this form. Also, note that if $\mathcal{A}$ is unambiguous, and merge is done on a subrun of an accepting run, then the operation is reversible.

Let $\mu \in [\![\mathcal{A}_G]\!](d)$, and let $r \in \mathcal{L}(\mathcal{A}_G)$ such that r is valid for X, $\mathsf{plain}(r) = d$ and $\mu^r = \mu$. Let ρ be the accepting run of $\mathcal{A}_G$ over r, and consider a run ρ' that is obtained from ρ by applying merge over every maximal sequence t of variable transitions. From the construction of Δ', it can be seen that this is a valid and accepting run of $\mathcal{T}_G$ over d', and that $\mathsf{out}(\rho') = \mathsf{enc}(\mu)$. We conclude that $\{\mathsf{enc}(\mu) \mid \mu \in [\![A_G]\!](d)\} \subseteq [\![\mathcal{T}_G]\!](d')$.

Now, let $\omega = [\![\mathcal{T}_G]\!](d')$, and let ρ be an accepting run of $\mathcal{T}_G$ over d' with output ω. Consider the run ρ' that is obtained from ρ by applying the reverse of merge on every transition in it that contains an output. Since ρ was over $d \cdot \#$ and accepting, clearly ρ' is a valid run of $\mathcal{A}_G$ over d, and ends in a state in F. One can also check that the sequence of symbols in ρ forms a ref-word r for which $\mathsf{enc}(\mu^r) = \omega$ and that ρ is valid for X because it is accepting and $\mathcal{A}_G$ is functional. We conclude that $[\![\mathcal{T}_G]\!](d') \subseteq \{\mathsf{enc}(\mu) \mid \mu \in [\![\mathcal{A}_G]\!](d)\}$, and from the previous paragraph we obtain that these sets are equal.

To see that $\mathcal{T}_G$ is unambiguous, one simply needs to see that (1) every accepting run ρ of $\mathcal{T}_G$ has a unique counterpart ρ' of $\mathcal{A}_G$, as it was stated in the previous part of the proof, and (2) the merge operation has only one possible output. Therefore, if two runs ρ_1 and ρ_2 of $\mathcal{T}_G$ have the same output, their counterparts ρ_1' and ρ_2' of $\mathcal{A}_G$ satisfy $\rho_1' = \rho_2'$. Clearly, applying the merge operation over these runs renders ρ_1 and ρ_2, so we conclude that they are equal, and thus, $\mathcal{T}_G$ is unambiguous.

The size of Δ is bounded by the number of valid v-paths there could exist in $\mathcal{A}_G$. Recall that $\mathcal{A}_G$ is functional, and thus, every v-path in $\mathcal{A}_G$ contains at most one instance of each element in C_X. From this, it follows that the size of $\mathcal{T}_G$ is in $\mathcal{O}(|\Delta||2^{\mathsf{C}_X}|)$. Furthermore, since the transitions in Δ form a DAG over Q, each of these v-paths can be found by a single traversal over $\mathcal{A}_G$, so building $\mathcal{T}_G$ takes extra time $\mathcal{O}(|\Delta|)$.

By using the algorithm detailed in Section 3.6 we can enumerate the set $[\![\mathcal{T}_G]\!](d)$ with update-time $\mathcal{O}(|\mathcal{T}_G|^3)$ and output-linear delay. However, with a more fine-grained analysis of the algorithm, we note that the update-time is bounded by $|Q'|^2|\Delta'| \in \mathcal{O}(|Q|^2|\Delta||2^{\mathsf{C}_X}|)$. We modify the enumeration algorithm slightly so that for each output $\omega \in [\![\mathcal{T}_G]\!](d)$, there is an extra step of building the expected output in $[\![G]\!](d)$. We do this by checking ω symbol by symbol and building a mapping $\mu \in [\![G]\!](d)$, which can be done in time $\mathcal{O}(|\mu|)$, since clearly $|\omega| \leq |\mu|$. It follows that this enumeration can be done with update-time $\mathcal{O}(|G|^3)$ and output-linear delay.

Finally, we address the case where G is an arbitrary VPEG. The way we deal with this case is by determinizing the EVPA constructed in Claim 3.3. This can be done in time $\mathcal{O}(2^{|\mathcal{A}_G|})$. From here, we can follow the reasoning given for the unambiguous case to prove the statement. □

Note that, although the update-time of the algorithm is exponential in the size of the grammar, in terms of data complexity the update-time is constant. Furthermore, for the special case of unambiguous grammars the update-time even is polynomial. Unambiguous grammars are very common in parsing tasks (Aho, Sethi, & Ullman, 1986) and, thus, this restriction could be useful in practice.

3.8. Related Work

The problem of streaming query evaluation has been extensively studied in the last decades. Some work considered streaming verification, like schema validation (Segoufin & Vianu, 2002) or type-checking (Kumar, Madhusudan, & Viswanathan, 2007), where the output is true or false. Other proposals (Y. Chen, Davidson, & Zheng, 2006; Olteanu, Furche, & Bry, 2004; Josifovski, Fontoura, & Barta, 2005; Green, Gupta, Miklau, Onizuka, & Suciu, 2004; Olteanu, 2007) provided streaming algorithms for XPath or XQuery's fragments; however, extending them for reaching constant-delay enumeration seems unlikely. Furthermore, most of these works (Kumar et al., 2007; Gou & Chirkova, 2007; Gauwin et al., 2009b) assumed outputs of fixed size (i.e., tuples). People have also considered other aspects of streaming evaluation with outputs like earliest query answering (Gauwin et al., 2009b) or bounded delay (Gauwin, Niehren, & Tison, 2009a) (i.e., given the first visit of a node, find the earliest event that permits its selection). These aspects are orthogonal to the problem studied here. Another line of research is (Bar-Yossef et al., 2005, 2007), which presents space lower bounds for evaluating fragments of XPath or XQuery over streams. These works do not consider restrictions on the delay to give outputs.

Visibly pushdown automata (Alur & Madhusudan, 2004b) are a model usually used for streaming evaluation of boolean queries (Kumar et al., 2007). In (Filiot, Gauwin, Reynier, & Servais, 2019; Alur, Fisman, Mamouras, Raghothaman, & Stanford, 2020), the authors studied the evaluation of transducers built from visibly pushdown automata in a streaming fashion, but none of them saw enumeration problems. Other extensions (Gauwin, Niehren, & Roos, 2008) for streaming evaluation have been analyzed but restricted to fixed-size outputs, and constant delay was not included.

Constant-delay algorithms have been studied for several classes of query languages and structures (Segoufin, 2013), as we already discussed. In (Bagan, 2006a; Amarilli et al., 2017), researchers considered query evaluation over trees (i.e., a different representation for nested documents), but their algorithms are for offline evaluation, and it is not clear how to extend this algorithm for the online setting. This research is extended with updates in (Amarilli et al., 2019b), which can encode streams by inserting new data items to the left. However, their update-time is logarithmic, whereas our proposal can do it with constant time (i.e., in data complexity). Furthermore, to the best of our knowledge, it is unclear how to modify the work in (Amarilli et al., 2019b) to get constant update-time in our scenario.

Streaming evaluation with constant-delay enumeration was included in the context of dynamic query evaluation (Idris et al., 2017; Berkholz et al., 2017; Nikolic & Olteanu, 2018; Kara, Nikolic, Olteanu, & Zhang, 2020) or complex event processing (Grez et al., 2019; Grez & Riveros, 2020). In both cases, the input cannot encode nested documents, and their results do not apply.

4. ENUMERATION FOR ANNOTATED GRAMMARS

In this chapter, we introduce annotated grammars, an extension of context-free grammars which allows annotations on terminals. We study the enumeration problem for annotated grammars: fixing a grammar, and given a string as input, enumerate all annotations of the string that form a word derivable from the grammar.

The first result in this chapter is an algorithm for unambiguous annotated grammars, which preprocesses the input string in cubic time and enumerates all annotations with output-linear delay.

We then study how we can reduce the preprocessing time while keeping the same delay bound, by making additional assumptions on the grammar. Specifically, we present a class of grammars which only have one derivation shape for all outputs, for which we can enumerate with quadratic time preprocessing.

We also give classes that generalize regular spanners for which linear time preprocessing is enough.

Comparison to previous chapter. In this chapter, we re-use the enumeration data structure of Chapter 3, and we consider a transducer model in Section 4.4 that recaptures some of the already presented results. However, in our problem, we work with pushdown automata without a *visibility* guarantee. This poses new technical challenges: the underlying tree structure (i.e., the parse tree) is not known in advance and is generally not unique.

Outline of the paper. In Section 4.1 we give some basic definitions of the concepts we will use througout the paper. In Section 4.2, we describe our results for unambiguous grammars. In Section 4.3, we present a relaxation of unambiguous grammars that we call rigid grammars, and some results for this class. In Section 4.4, we describe our results for pushdown annotators. In Section 4.5, we detail how to use our results in the context of document spanners. We close our chapter in Section 4.6 by discussing related work.

4.1. Grammars and Annotators

Strings and annotations. Let Σ be a finite alphabet. We write Σ^* for the set of strings over Σ. The *length* of a string $w = w_1 \cdots w_n \in \Sigma^*$ is $|w| := n$. The string of length 0 is written ε. We write $u \cdot v$ or uv for the *concatenation* of $u, v \in \Sigma^*$.

Let Ω be a finite set of annotations. An *annotated string* is a string $\hat{w} \in (\Sigma \cup \Sigma \times \Omega)^*$. We denote strings by w and annotated strings by $\hat{w}$ when this avoids confusion. Intuitively, if $\hat{w} = \hat{w}_1 \cdots \hat{w}_n$, then $\hat{w}_i = (a, \vartheta) \in \Sigma \times \Omega$ means that the letter a at position i is annotated with ϑ(called an *annotated letter*) and $\hat{w}_i \in \Sigma$ means that there is no annotation at position i. Given an annotated string $\hat{w} = \hat{w}_1 \cdots \hat{w}_n$, we denote by $\mathsf{str}(\hat{w}) = \mathsf{str}(\hat{w}_1) \cdot \cdots \cdot \mathsf{str}(\hat{w}_n)$ the *unannotated string* of $\hat{w}$, i.e., $\mathsf{str}((a, \vartheta)) := a$ and $\mathsf{str}(a) := a$, and we denote by $\mathsf{ann}(\hat{w}) = \mathsf{ann}(\hat{w}_1, 1) \cdot \cdots \cdot \mathsf{ann}(\hat{w}_n, n)$ the *annotations* of $\hat{w}$, i.e., $\mathsf{ann}((a, \vartheta), i) := (\vartheta, i)$ and $\mathsf{ann}(a, i) := \varepsilon$. Note that $|\mathsf{str}(\hat{w})| = |w|$, but the length $|\mathsf{ann}(\hat{w})|$ of $\mathsf{ann}(\hat{w})$ can be much less than $|w|$.

Annotated grammars. A *context-free grammar* (CFG) over Σ is a tuple $G = (V, \Sigma, P, S)$, where V is a set of *nonterminals*, Σ is the alphabet (whose letters are called *terminals*), $S \in V$ is the *start symbol*, and P is a finite set of *rules* of the form $X \to \alpha$ where $X \in V$ and $\alpha \in (V \cup \Sigma)^*$. We assume that V and Σ are disjoint. In this chapter, we extend this definition to an *annotated (context-free) grammar* $\mathcal{G} = (V, \Sigma, \Omega, P, S)$, which is simply the CFG $(V, \Sigma \cup \Sigma \times \Omega, P, S)$. We use G to denote a CFG and $\mathcal{G}$ to denote an annotated grammar. The *terminals* of $\mathcal{G}$ are letters $a \in \Sigma$ and annotated letters $(a, \vartheta) \in \Sigma \times \Omega$.

We recall the semantics of a CFG $G = (V, \Sigma, P, S)$. Given a string $u \in \Sigma^*$, two strings $\gamma, \delta \in (V \cup \Sigma)^*$, and $X \in V$, we say that $uX\delta$ *produces* $u\gamma\delta$, denoted by $uX\delta \Rightarrow_G u\gamma\delta$, if P contains the rule $X \to \gamma$. We then say that $\alpha \in (V \cup \Sigma)^*$ *derives* $\beta \in (V \cup \Sigma)^*$, denoted by $\alpha \Rightarrow_{\mathcal{G}}^* \beta$ or just $\alpha \Rightarrow^* \beta$, if there is a sequence of strings $\alpha_1, \ldots, \alpha_m$ with $m \geq 1$ such that $\alpha = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_m = \beta$. We say that G *derives* $\alpha \in (V \cup \Sigma)^*$ if $S \Rightarrow^* \alpha$, and define the *language* $L(G)$ of G as the set of strings $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$.

Note that our derivations are leftmost derivations, which is standard for the unambiguity notions that we introduce afterwards. The *language of an annotated grammar* $\mathcal{G}$ is that of the underlying CFG on the alphabet of terminals $\Sigma \cup \Sigma \times \Omega$. In particular, $L(\mathcal{G})$ is a set of annotated strings.

The purpose of annotated grammars is to consider all possible annotations of an input unannotated string $w \in \Sigma^*$. Specifically, the *semantics* of an annotated grammar $\mathcal{G}$ is the function $[\![\mathcal{G}]\!]$ mapping each string $w \in \Sigma^*$ to the following (possibly empty) set of annotations: $[\![\mathcal{G}]\!](w) \;:=\; \{\mathsf{ann}(\hat{w}) \mid \hat{w} \in L(\mathcal{G}) \wedge \mathsf{str}(\hat{w}) = w\}$.

An *output* of evaluating $\mathcal{G}$ over w is just an element $\mu \in [\![\mathcal{G}]\!](w)$. Note that, in the case when $\Omega = \emptyset$, for all $w \in \Sigma^*$ we have $[\![\mathcal{G}]\!](w) = \emptyset$ if $w \notin L(\mathcal{G})$ and $[\![\mathcal{G}]\!](w) = \{\varepsilon\}$ if $w \in L(\mathcal{G})$. So, annotated grammars subsume CFGs. In Section 4.5, we show that they also subsume the extraction grammars of (Peterfreund, 2023), which implies that annotated grammars are more expressive than regular spanners (Florenzano et al., 2018; Amarilli et al., 2020), or even visibly pushdown transducers from Chapter 3.

Towards ensuring tractability, we call a CFG G *unambiguous* if for every $w \in L(G)$ there is a unique derivation of w by G. We call an annotated grammar $\mathcal{G}$ *unambiguous* if the underlying CFG over $\Sigma \cup \Sigma \times \Omega$ is unambiguous. Intuitively, this means that each output $\mu \in [\![\mathcal{G}]\!](w)$ can be produced in only one way. Remember that there are CFGs G with no unambiguous CFG G' *equivalent* to G (i.e., such that $L(G') = L(G)$), and it is undecidable to check whether an input CFG is unambiguous, or has an equivalent unambiguous CFG. The same is immediately true for annotated grammars.

Problem statement. The goal of this chapter is to study how to efficiently enumerate the annotations of an annotated grammar:

Input: An annotated grammar $\mathcal{G}$ and a string $s \in \Sigma^*$

Output: Enumerate the outputs of $[\![\mathcal{G}]\!](s)$

As it was stated, we work in the standard computational model of Random Access Machines (RAM) with logarithmic word size and uniform cost measure, having addition and subtraction as basic operations (Aho et al., 1974). The size of $\mathcal{G}$ is measured as the sum of rule lengths.

The ultimate goal of this chapter is to find enumeration algorithms to enumerate the outputs of annotated grammars with linear preprocessing and output-linear delay. However, as we will see, this goal is not always realistic, so we will initially settle for a higher processing time, i.e., quadratic or cubic, before presenting classes with linear preprocessing in data complexity. We present our first results towards this goal in this next section.

4.2. Unambiguous Grammars

In this section we start presenting our results and show a first algorithm to enumerate the outputs of an annotated grammar on an input string. The algorithm applies to any unambiguous annotated grammar, and ensures cubic-time preprocessing and output-linear delay in data complexity; in terms of combined complexity, the preprocessing is linear on the grammar. This improves the result by Peterfreund (Peterfreund, 2023), which had quintic-time preprocessing in data complexity.

The section is structured as follows. We present a general-purpose enumeration data structure called *enumerable sets*, which is the basis of our enumeration algorithms. We then introduce the *arity-two normal form* for annotated grammars, designed to ensure efficient enumeration, and which can be enforced in linear time. After this, we present our algorithm and state the main result in this chapter (Theorem 4.2). Last, we state a conditional data complexity lower bound.

Enumerable sets. The preprocessing phase of our enumeration algorithm builds data structures representing the set of outputs to enumerate. For this, we essentially re-use the ε-ECS data structure of Chapter 3, but for convenience we present them in a self-contained way for our context, and name them *enumerable sets*. We now define them and state that we can enumerate their contents with output-linear delay (Theorem 4.1). The enumeration phase of our algorithm simply enumerates the outputs of an enumerable set using this delay guarantee.

An *enumerable set* is a representation of a set of strings over some alphabet $\emptyset$. For our case, we want strings of $\emptyset^*$ to describe outputs, so $\emptyset$ consists of pairs of annotations with positions of the input string w, i.e., $\emptyset := \Omega \times \{1, \dots, |w|\}$.

The basic enumerable sets are:

- empty, the empty set;
- $\mathsf{singleton}(\varepsilon)$, the singleton set containing the empty string;
- $\mathsf{singleton}(x)$ for $x \in \emptyset$, the singleton set with the single-character string x.

Enumerable sets can be combined using operators to form more complex enumerable sets. The operators that we consider all take constant-time and are *fully-persistent* (Driscoll et al., 1986a). Specifically, given enumerable sets $\mathcal{D}_1$ and $\mathcal{D}_2$, combining them creates an enumerable set without modifying $\mathcal{D}_1$ and $\mathcal{D}_2$ (i.e., they can still be used in other operator applications). To make this possible, enumerable sets can share some components, e.g., some parts of the arguments $\mathcal{D}_1$ and $\mathcal{D}_2$ can be shared in memory, and the result can also have some parts that are shared with $\mathcal{D}_1$ and $\mathcal{D}_2$. This is similar, e.g., to persistent lists, where we can only extend a list by adding an element to its head: this does not modify the original list, and returns a new list sharing some memory with the original list.

The two operators to combine enumerable sets are:

- The *union* operator $\mathsf{union}(\mathcal{D}_1, \mathcal{D}_2)$ can be applied if the sets represented by $\mathcal{D}_1$ and $\mathcal{D}_2$ are disjoint, intuitively to avoid duplicates. It returns an enumerable set representing the union of these sets,
- The *product* operator $\mathsf{prod}(\mathcal{D}_1, \mathcal{D}_2)$ can be applied if there are no common letters in the strings of the sets represented by $\mathcal{D}_1$ and $\mathcal{D}_2$, i.e., if $\mathcal{D}_1$ (resp. $\mathcal{D}_2$) represents S_1 (resp. S_2) then the sets $\{x_1 \in \emptyset \mid x_1 \text{ occurs in some } w_1 \in S_1\}$

and $\{x_2 \in \emptyset \mid x_2 \text{ occurs in some } w_2 \in S_2\}$ are disjoint. Then, the operation returns an enumerable set $\mathcal{D}$ which represents the concatenations of the strings in $\mathcal{D}_1$ and in $\mathcal{D}_2$: formally, it represents $S_1 \cdot S_2 = \{w_1 \cdot w_2 \mid w_1 \in S_1, w_2 \in S_2\}$.

It is known that enumerable sets can be enumerated efficiently:

Theorem 4.1. *We can implement enumerable sets such that:*

- *The enumerable sets* empty, *$\mathsf{singleton}(\varepsilon)$, and $\mathsf{singleton}(x)$ for $x \in \emptyset$ can be built in constant time;*
- *The union and product operations can be implemented in constant time and in a fully persistent way;*
- *Given an enumerable set, we can enumerate the strings it represents with output-linear delay and memory usage linear in the number of instructions used to build it.*

We omit a formal proof given that it follows directly from Theorem 3.4.

Arity-two normal form. Having presented enumerable sets, we now present the normal form to enforce on annotated grammars. Our results could be shown using the commonly known Chomsky normal form (CNF), but we cannot always obtain an equivalent CNF of linear size from a grammar. For this reason, we use a variant of CNF, the *arity-two normal form* (2NF) (Lange & Leiß, 2009), which is intuitively like CNF but without disallowing rules of the form $X \to Y$ or $X \to \varepsilon$. Formally, we say that an annotated grammar $(V, \Sigma, \Omega, P, S)$ is in *arity-two normal form* (2NF) if the following hold:

- Every nonterminal X can derive some string, i.e., there exists $\hat{w} \in (\Sigma \cup \Sigma \times \Omega)^*$ such that $X \Rightarrow_{\mathcal{G}}^* \hat{w}$.
- Every nonterminal X can be reached from the start symbol S, i.e., there exists $\alpha, \beta \in (V \cup \Sigma \cup \Sigma \times \Omega)^*$ such that $S \Rightarrow_{\mathcal{G}}^* \alpha X \beta$.
- For every rule $X \to \alpha$ in P, we have $|\alpha| \leq 2$, and if $\alpha = 2$ then it consists of two nonterminals.

We can easily translate annotated grammars to 2NF, as in (Lange & Leiß, 2009):

PROPOSITION 4.1 ((Lange & Leiß, 2009)). *Given any annotated grammar $\mathcal{G}$, we can compute in linear time an annotated grammar $\mathcal{G}'$ in 2NF such that $\mathcal{G}$ and $\mathcal{G}'$ are equivalent. Furthermore, if $\mathcal{G}$ is unambiguous then $\mathcal{G}'$ is unambiguous as well.*

We omit a proof of this statement here since it follows from the reference. However, we will prove a stronger version of it in the following sections when further restrictions for grammars are introduced.

By Proposition 4.1, we assume that the input grammar $\mathcal{G}$ is in 2NF.

We also compute in linear time some more information about $\mathcal{G}$. First, we precompute which nonterminals are *nullable*, i.e., are such that $X \Rightarrow_{\mathcal{G}}^{*} \varepsilon$: if we have a rule $X \rightarrow \varepsilon$ then X is nullable, and if we have a rule $X \rightarrow Y$ where Y is nullable or $X \rightarrow YZ$ where Y and Z are nullable then X is also nullable. From this information, we further compute for each nonterminal Y a set $\mathcal{D}[Y]$ of all nonterminals X such that one of the following rules exist: a rule $X \rightarrow Y$, a rule $X \rightarrow YZ$ where Z is nullable, or a rule $X \rightarrow ZY$ where Z is nullable. We can clearly compute all of this in linear time (note that each rule contributes at most two entries to $\mathcal{D}$).

Second, as the grammar is assumed to be unambiguous, it also contains no *cycles*, i.e., there are no sequence of nonterminals $X_1 \ldots X_n$ such that for $1 \leq i < n$, $X_i \in \mathcal{D}[X_{i+1}]$ and $X_1 \in \mathcal{D}[X_n]$. Indeed, otherwise there would be infinitely many possible derivations of some string starting at X_1, contradicting the unambiguity of $\mathcal{G}$. Thus, we can sort the nonterminals of $\mathcal{G}$ in *topological order*, by which we mean that when $Y \in \mathcal{D}[X]$ then Y is enumerated after X. Intuitively, when we consider a nonterminal X, we want to be done with processing the nonterminals Y such that $X \rightarrow Y$ or $X \rightarrow YZ$ or $X \rightarrow ZY$ with Z nullable. This order can also be computed in linear time.

Enumeration algorithm. We now present the preprocessing phase of the enumeration algorithm, formalized as Algorithm 3 where the input string $w = a_1 \dots a_n$ is assumed nonempty.

The principle of the algorithm is the following:

PRINCIPLE 4.1. *For every triple of the form (i, j, X) with $1 \leq i < j \leq n+1$ and $X \in N$, the table cell $\mathbb{I}[i][j][X]$ will contain an enumerable set representing the annotations of the string $a_i \cdots a_{j-1}$ that can be derived from symbol X in the grammar.*

These sets are initialized to be empty. In lines 5–11 of the algorithm, the cells $\mathbb{I}[i][j][X]$ with $j - i = 1$ are initialized to consider derivations via "simple rules" of the form $X \to a$ or $X \to (a, \vartheta)$. (For now, ignore the role of the endIn table.) Note that the rules of the form $X \to \varepsilon$ are considered when defining $\mathcal{D}$ and not further examined by the algorithm. At the end, line 24 returns the enumerable set for the annotations of the entire string derivable from the start symbol, i.e., the outputs of $\mathcal{G}$ on w.

The main part of the algorithm consists in satisfying Principle 4.1 by adding the annotations corresponding to "complex" rules (i.e., of the form $X \to YZ$ or $X \to Y$). At the beginning of the algorithm the cells of the table $\mathbb{I}$ might lack some annotations corresponding to complex rules, but each cell will be considered complete at some point during the execution, at which point it will satisfy Principle 4.1 and will not be modified anymore. We define the order in which the cells are considered complete as follows: $(i, j, X) < (i', j', X')$ when $j < j'$ or $(j = j' \wedge i > i')$ or $(j = j' \wedge i = i' \wedge X < X')$ where we order nonterminals X and X' following the topological order from $\mathcal{D}$.

Consider the *complex derivations* starting from X of the string $a_i \cdots a_{j-1}$, i.e., those that begin with a complex rule. We will see here how to reflect them in $\mathbb{I}[i][j][X]$. There are two kinds of complex derivations. The first kind is the derivations where we first rewrite X to another nonterminal Z with a rule $X \to Z$, or by rewriting X to YZ or ZY but where Y is nullable and will be rewritten to ε. In these three cases, we have $X \in \mathcal{D}[Z]$.

Thus, we fill the index $\mathbb{I}[i][j][X]$ with the contents of $\mathbb{I}[i][j][Z]$, which is already complete, for $X \in \mathcal{D}[Z]$ (lines 15-16).

The second kind of complex derivation begins with a complex rule $X \rightarrow YZ$ where neither Y nor Z will be rewritten to ε. In this case, the set of annotations to add into $\mathbb{I}[i][j][X]$ using this rule is the union of products of all the $\mathbb{I}[i][k][Y]$ and $\mathbb{I}[k][j][Z]$ where $i < k < j$. We have $(i, k, Y) < (k, j, Z) < (i, j, X)$, so we can fill $\mathbb{I}[i][j][X]$ with the product of the contents of $\mathbb{I}[i][k][Y]$ and $\mathbb{I}[k][j][Z]$, at the moment where $\mathbb{I}[k][j][Z]$ is considered complete.

To summarize, from line 12 onwards, the algorithm considers the positions j in ascending order, and populates all cells $\mathbb{I}[i][j][X]$ so that they are complete. To do so, we consider the triples (k, j, Z) by increasing order in our sorting criterion, i.e., by decreasing k, then increasing Z in the order of the topological sort. Whenever we consider a cell, it is complete, and we consider its contributions to cells of the form $\mathbb{I}[i][j][X]$ with $i = k$ using complex rules of the first kind (lines 15-16), and if it is non-empty we consider how to combine it with a neighboring cell (which is also complete and non-empty) as we explained previously, adding the results to a cell $\mathbb{I}[i][j][X]$ with $i < k$ which is not yet complete (lines 17–23).

We now explain the optimization involving the set endIn. It is not necessary to achieve the cubic running time of this section, but is required for the quadratic bound in Section 4.3. The optimization is that, when processing the triple (k, j, Z) and the rule $X \rightarrow YZ$, we do not test all the possible cells $\mathbb{I}[i][k][Y]$, but only those that are non-empty. Indeed, if $\mathbb{I}[i][k][Y]$ is empty, then the concatenation of $\mathbb{I}[i][k][Y]$ with $\mathbb{I}[k][j][Z]$ is also empty. Thus, we maintain the list $\mathsf{endIn}[k][Y]$ of all the i's to consider with $i < k$, i.e., those such that $\mathbb{I}[i][k][Y]$ is non-empty. We initialize this list to be empty, add i to $\mathsf{endIn}[k][Y]$ whenever $\mathbb{I}[i][k][Y]$ becomes non-empty (at line 11 in the base case, or at line 21 before adding to an empty cell for the first time). Then, we only consider the indices i of this list to combine $\mathbb{I}[i][k][Y]$ with another cell.

We now argue that our algorithm is correct, and in particular that **(i)** we satisfy Principle 4.1; that **(ii)** all the unions are disjoint, and that **(iii)** all the products involve enumerable sets on disjoint alphabets. One can establish **(i)** by showing by induction over cells that the invariant is correct when each cell is considered complete by our algorithm (and the cell is not changed afterwards). Knowing **(i)**, the first violation of **(ii)** would witness that the same annotation of some factor $a_i \cdots a_{j-1}$ can be derived in two different ways from a nonterminal X, contradicting unambiguity, so there are no violations of **(ii)**. For **(iii)**, we simply observe that, by **(i)**, $\mathbb{I}[i][j][X]$ only contains pairs of the form (ϑ, k) for some $i \leq k < j$, so we can indeed perform the product of $\mathbb{I}[i][k][Y]$ and $\mathbb{I}[k][j][Z]$.

This establishes that the algorithm is correct. Now, the running time of the preprocessing phase of the algorithm is clearly in $\mathcal{O}(n^3|\mathcal{G}|)$, because (1) the endIn lists are of size $\mathcal{O}(n)$ at most, and (2) the consideration of all $Z \in N$ and $X \in \mathcal{D}[Z]$ is in $\mathcal{O}(|\mathcal{G}|)$: every $X \in \mathcal{D}[Z]$ corresponds to a rule, so the consideration of all $Z \in N$ and rules in CRule$[Z]$ is in $\mathcal{O}(|\mathcal{G}|)$. The enumeration phase is then simply that of Theorem 4.1. Hence, we have shown that enumeration for unambiguous annotated grammars can be achieved with cubic time preprocessing and output-linear delay:

Theorem 4.2. *Given an unambiguous annotated grammar $\mathcal{G}$ and an input string w, we can enumerate $[\![\mathcal{G}]\!](w)$ with preprocessing in $\mathcal{O}(|w|^3 \cdot |\mathcal{G}|)$ (hence cubic in data complexity), and output-linear delay (independent from w or $\mathcal{G}$). The memory usage is in $\mathcal{O}(|w|^3 \cdot |\mathcal{G}|)$.*

Lower bounds. We cannot show a lower bound that matches the complexity of our algorithm, but we can prove that we cannot achieve a preprocessing time better than the time to test whether a string is accepted by a CFG, which is essentially the complexity of Boolean matrix multiplication (Abboud et al., 2018):

PROPOSITION 4.2. *Let ω be the smallest value such that we can multiply two Boolean $n \times n$ matrices in time $\mathcal{O}(n^{\omega+o(1)})$. Then for any $c > 0$ there is an unambiguous annotated*

Algorithm 3 Preprocessing phase: given a 2NF unambiguous annotated grammar $\mathcal{G} = (N, \Sigma, \Omega, P, S)$ and a non-empty string $w = a_1 \cdots a_n$, compute an enumerable set representing $[\![\mathcal{A}]\!](w)$.

```
I ← an array $(n+1) \times (n+1) \times N$ initialized with empty
endIn ← an array $(n+1) \times N$ initialized with empty lists
CRule ← an array such that CRule$[Z] = \{X \to YZ \in P\}$
$\mathcal{D}$ ← an array as described in the presentation of 2NF
**for** $1 \leq i \leq n$ **do**
    **if** rule $(X \to a_i)$ in $P$ **then**
        $\mathbb{I}[i][i+1][X] \leftarrow \mathsf{union}(\mathbb{I}[i][i+1][X], \mathsf{singleton}(\varepsilon))$
    **for** rule $(X \to (a_i, \vartheta))$ in $P$ **do**
        $\mathbb{I}[i][i+1][X] \leftarrow \mathsf{union}(\mathbb{I}[i][i+1][X], \mathsf{singleton}((\vartheta, i)))$
    **if** $\mathbb{I}[i][i+1][X] \neq$ empty **then**
        endIn$[i+1][X]$.append($i$)
**for** $j = 1$ **to** $n+1$ **do**
    **for** $k = j-1$ **downto** $1$ **do**
        **for** nonterminal $Z \in N$ in topological order **do**
            **for** nonterminal $X \in \mathcal{D}[Z]$ **do**
                $\mathbb{I}[k][j][X] \leftarrow \mathsf{union}(\mathbb{I}[k][j][X], \mathbb{I}[k][j][Z])$
            **if** $\mathbb{I}[k][j][Z] \neq$ empty **then**
                **for** rule $(X \to YZ)$ in CRule$[Z]$ **do**
                    **for** $i \in$ endIn$[k][Y]$ **do**
                        **if** $\mathbb{I}[i][j][X] =$ empty **then**
                            endIn$[j][X]$.append($i$)
                        $\mathbb{I}[i][j][X] \leftarrow \mathsf{union}(\mathbb{I}[i][j][X],$
                                    $\mathsf{prod}(\mathbb{I}[i][k][Y], \mathbb{I}[k][j][Z]))$
**return** $\mathbb{I}[1][n+1][S]$
```

grammar $\mathcal{G}$ such that, given an input string w, enumerating $[\![\mathcal{G}]\!](w)$ with output-linear delay requires a preprocessing time of $\Omega(|w|^{\omega - c})$.

PROOF. We know from (Abboud et al., 2018) that for any $c > 0$, there exists a fixed grammar $\mathcal{G}$ such that determining whether a string w is derived by $\mathcal{G}$, cannot be solved in time $\mathcal{O}(|w|^{\omega - c})$, unless the conjecture in graph algorithms mentioned in (Abboud et al., 2018) is false.

We will see that this conditional lower bound translates to unambiguous annotated grammars. Indeed, we will show that for each grammar $\mathcal{G}$ there exists an unambiguous annotated grammar $\mathcal{G}'$ such that w is derived by $\mathcal{G}$ if and only if $[\![\mathcal{G}']\!](w)$ is non-empty. Therefore after the preprocessing of w for $\mathcal{G}'$, we know in constant time whether w is derived by $\mathcal{G}$ which proves that the preprocessing of $\mathcal{G}'$ on w requires $\mathcal{O}(|w|^{\omega-c})$ time, assuming the conjecture is true.

Now let us show how to translate a grammar $\mathcal{G}$ into an unambiguous annotated grammar $\mathcal{G}'$. This can be challenging, because $\mathcal{G}$ is not necessarily unambiguous: for this reason we need to define $\mathcal{G}'$ intuitively by adding annotations that disambiguate the various possible derivations of $\mathcal{G}$, to guarantee that the result is unambiguous. As this is cumbersome to do on grammars, we use the correspondence between annotated grammars and pushdown annotators (Proposition 4.7), shown later in the article.

In this proof, we will use the notion of *pushdown automata* (PDA); see Definition 4.2 for the formal definition. Let us consider a PDA $\mathcal{P}$ which is equivalent to $\mathcal{G}$. As is standard with PDAs, we can change the given definition to suppose without loss of generality that no transition in $\mathcal{P}$ is an ε-transition. Specifically, we consider PDAs in a slightly different model where transitions are of the form $(q_1, a, s_1, q_2, s_2) \in Q \times \Sigma \times \Gamma^+ \times Q \times \Gamma^+$: such a transition means that in state q_1, when the top stack symbols are s_1 and the next letter to read is a, the automaton can read the letter, move to state q_2 and replace s_1 by s_2 on the stack. We create our unambiguous PDAnn $\mathcal{P}'$ from $\mathcal{P}$ by replacing each transition $t = (q_1, a, s_1, q_2, s_2)$ to a set of transitions that first pop the symbols of s_1 from the stack, then reads a, then pushes the symbols of s_2 onto the stack. The first state of this transition is q_1, the last state is q_2 but we make sure that each of the intermediate states are unique to t. Furthermore, the transition that reads the letter a outputs a symbol unique to the transition t. Therefore, by construction there is a bijection between runs of $\mathcal{P}$ and runs of $\mathcal{P}'$ and the PDAnn $\mathcal{P}'$ is unambiguous because the run used for each output can be retrieved from that output.

We conclude by using Proposition 4.7 to obtain an equivalent annotated grammar $\mathcal{G}'$, which is also unambiguous. Thus, we know that on any unannotated string w, the set $[\![\mathcal{G}']\!](w)$ is empty if $\mathcal{G}$ does not derive w, and non-empty if it does. Thus, we know that, if we assume the conjecture is true, we cannot determine in $\mathcal{O}(|w|^{\omega-c})$ whether $[\![\mathcal{G}']\!](w)$ is empty or not. But if we have an algorithm to enumerate $[\![\mathcal{G}']\!](w)$ with output-linear delay, as any output has size $\mathcal{O}(|w|)$ in $|w|$, we can do this with a complexity linear in $|w|$ which is that of the preprocessing of the enumeration algorithm. Thus, we conclude that the preprocessing conditionally requires $\Omega(|w|^{\omega-c})$ time.

□

4.3. Rigid Grammars

In the previous section, we have shown how to enumerate the output of unambiguous annotated grammars on strings, with output-linear delay and cubic preprocessing in the input string. This algorithm has two drawbacks: it requires us to impose that the grammar is unambiguous, and the cubic preprocessing may be expensive.

In this section, we introduce a new class of annotated grammars, called *rigid grammars*. Rigid grammars do not need to be unambiguous, but as we will show a rigid grammar can always be converted to additionally impose unambiguity. The point of rigid grammars is that we can show a quadratic bound on Algorithm 3 for them.

We first define rigid grammars in this section. We then state that we can impose unambiguity for rigid grammars, and derive some consequences about their expressiveness and the complexity of recognizing them. Last, we show a quadratic bound on the preprocessing time for output-linear enumeration for such grammars, and explain why a better bound would be challenging to achieve.

4.3.1. Definitions

Rigid grammars. We first define the restricted notion of grammars that we study. Consider an annotated grammar $\mathcal{G} = (N, \Sigma, \Omega, P, S)$, and a string $\gamma \in (\Sigma \cup (\Sigma \times \Omega) \cup N)^*$ of nonterminals and of terminals which may carry an annotation in Ω. We will be interested in the *shape* of γ, written $\mathsf{shape}(\gamma)$: it is the string over $\{0, 1\}$ obtained by replacing every nonterminal of N in γ by 1 and replacing all terminals (annotated or not) by 0: note that $|\mathsf{shape}(\gamma)| = |\gamma|$.

We then say that an annotated grammar $\mathcal{G}$ is *rigid* if for every string $w \in \Sigma^*$, all derivations from the start symbol S of $\mathcal{G}$ to an annotated string $\hat{w}$ of w have the same sequence of shapes (remember that we only consider leftmost derivations). Formally, there exists a sequence $s_1, \ldots, s_k \in \{0, 1\}^*$ depending only on w such that for any derivation $S = \alpha_1 \Rightarrow_{\mathcal{G}} \alpha_2 \Rightarrow_{\mathcal{G}} \ldots \Rightarrow_{\mathcal{G}} \alpha_m = \hat{w}$ with $\mathsf{str}(\hat{w}) = w$, we have $m = k$ and $\mathrm{shape}(\alpha_i) = s_i$ for all $1 \leq i \leq k$.

Intuitively, the sequence of shapes of a derivation describes the skeleton of the corresponding derivation tree. Thus, a rigid annotated grammar is one where, for each unannotated string, all derivation trees for all annotations of the string are isomorphic (ignoring the labels of nonterminals and the annotation of terminals).

Now we restate Proposition 4.1 while including the angle of rigid grammars.

PROPOSITION 4.3 (Proposition 4.1). *Given any annotated grammar $\mathcal{G}$, we can compute in linear time an annotated grammar $\mathcal{G}'$ in 2NF such that $\mathcal{G}$ and $\mathcal{G}'$ are equivalent. Furthermore, if $\mathcal{G}$ is unambiguous (resp. rigid) then $\mathcal{G}'$ is unambiguous (resp. rigid) as well.*

PROOF. **Conditions 1 and 2: removing useless nonterminals**. We first perform a linear-time exploration from the terminals to mark the nonterminals X that can derive some string of terminals. The base case is if a nonterminal X has a rule $X \to \alpha$ where α only consists of terminals (in particular $\alpha = \varepsilon$), then we mark it. The induction is that whenever a nonterminal X has a rule $X \to \alpha$ where α only consists of terminals and of marked nonterminals, then we mark X. At the end of this process, it is clear that any nonterminal that is not known to derive a string of terminals indeed does not derive any string, because any derivation of a string of terminals from a nonterminal X would witness that all nonterminals in this derivation, including X, should have been marked, which is impossible. Hence, we can remove the nonterminals that are not marked without changing the language or successful derivations of the grammar, and satisfy condition 1 in linear time.

Second, we perform a linear-time exploration from the start symbol S to mark the nonterminals X that can be reached in a derivation from S. The base case is that S is marked. The induction is that whenever a nonterminal Y occurs in the right-hand side of a rule having X as its left-hand side, and X is marked, then we mark Y. At the end of the process, if a nonterminal X is not marked, then indeed there is no derivation from S that produces a string featuring X, as otherwise it would witness that X is marked, which is impossible. Hence, we can again remove the nonterminals that are not marked, the grammar and successful derivations are again unchanged, and we satisfy condition 2 in linear time.

As the transformations here only remove nonterminals and rules that cannot appear in a derivation, they clearly preserve unambiguity as well as rigidity.

Condition 3: shape of rules. We first ensure that every right-hand side of a rule is of size ≤ 2. Given the annotated grammar $\mathcal{G}$, for every rule $X \to \alpha$ where $|\alpha| > 2$, letting $\alpha = \alpha_1 \cdots \alpha_n$, we introduce $n-2$ fresh nonterminals $X_{\alpha,1}, \ldots, X_{\alpha,n-2}$, and replace the rule by the following: $X \to \alpha_1 X_{\alpha,1}$, $X_{\alpha,1} \to \alpha_2 X_{\alpha,2}$, ..., $X_{\alpha,n-2} \to \alpha_{n-1}\alpha_n$.

We make sure that the right-hand side of rules of size 2 consist only of nonterminals by introducing fresh intermediate nonterminals whenever necessary, which rewrite to the requisite terminal.

It is then clear that the result satisfies condition 3, and that there is a one-to-one correspondence between derivations in the original grammar and derivations in the rewritten grammar. To see this, note that there is an obvious one-to-one function which maps derivations from the original grammar into derivations in the new grammar, and that there is a slightly more involved function which receives a derivation in the new grammar, and builds a derivation in the original grammar by following the steps detailed above (and using the fact that each fresh nonterminal is associated to exactly one rule), which is also one-to-one. We conclude that the original grammar is unambiguous if and only if the new grammar is unambiguous.

The last point to check is that the arity-2 transformation preserves rigidity, i.e., if the original annotated grammar is rigid then so is the image of the transformation. Let X be some symbol of the original grammar $\mathcal{G}$, and $w \in \Sigma^*$ be a string. Let us show that all derivations from the corresponding symbol X' of the rewritten grammar $\mathcal{G}'$ have same shape. We do so by induction on the length of w and then on the topological order on nonterminals. The base case of w of length 0 is clear: the possible derivations are sequences of applications of rules of the form $Y \to Z$ in a sequence of some fixed length, followed by a rule of the form $Y \to \varepsilon$, and what can happen in the rewritten grammar is the same.

For the inductive case, as $\mathcal{G}$ is rigid, we know that there must be one fixed profile $\pi \in \{0, 1\}^k$ such that all derivations of w from X start by the application of a rule $X \to \alpha$ where α corresponds to profile π, i.e., it has length k and its i-th character is a nonterminal or terminal according to the value of the i-th bit of π. Otherwise the existence of two different right-hand-side profiles would contradict rigidity. Furthermore, by considering the possible sub-derivations from α_1 (including the empty derivation if α_1 is a terminal), we know that α_1 derives some fixed prefix of w and that all such derivations have the same sequence of profiles; otherwise we would witness a contradiction to rigidity. By applying the same argument successively to $\alpha_2, \ldots, \alpha_k$, we deduce that there must be a partition of $w = w_1 \cdots w_k$ such that, in all derivations of w from X, the derivation applies a rule with right-hand having profile π to produce some string $\alpha_1 \cdots \alpha_k$, and then each α_i derives an annotation of w_i and for each i all possible derivations of some annotation of w_i by some i-th element in the right-hand size of such a rule has the same sequence of profiles.

As the string is nonempty we know that $k > 0$. Further, if $k = 1$ then X and the productions involving X were not rewritten so we immediately conclude either with the case of a rule $X \to \tau$ for a terminal τ or by induction hypothesis on the nonterminals in the topological order for the case of a rule of the form $X \to Y$. Hence, we assume that $k \geq 2$.

We know by induction that, in the rewritten grammar, the derivation from X will start by rewriting X to $Y_1 X_{\alpha,1}$, the $X_{\alpha,1}$ being itself rewritten to $Y_2 X_{\alpha,2}$, and so on, for some right-hand size α of a rule $X \to \alpha$ having profile π. Clearly each Y_i will have to derive an annotation of the w_i in the partitioning of w, as a derivation following a different partitioning would witness a derivation in the original grammar that contradicts rigidity. Now, the profile π indicates if each Y_i is a nonterminal of the initial grammar or a fresh nonterminal introduced to rewrite to a terminal. In the latter case, there is no possible deviation in profiles. In the former case, we conclude by induction hypothesis that each Y_i derives annotations of its w_i that all have the same profile, and we conclude that all derivations in the rewritten grammar indeed have the same profile, concluding the proof. □

Rigidity vs unambiguity. Unambiguity and rigidity for annotated grammars seem incomparable: unambiguity imposes that every annotation is produced by only one derivation, whereas rigidity imposes that all derivations across all annotations have the same shape (but the same annotation may be obtained multiple times).

However, it turns out that, on rigid grammars, we can impose unambiguity without loss of generality: all rigid grammars can be converted to equivalent rigid and unambiguous grammars.

Theorem 4.3. *For any rigid grammar $\mathcal{G}$ we can build an equivalent rigid and unambiguous grammar $\mathcal{G}'$. The transformation runs in exponential time, i.e., time $\mathcal{O}(2^{|\mathcal{G}|^c})$ for some $c > 0$.*

PROOF. In this proof, we will use the notion of PDAnn introduced in Section 4.4, and we will use Proposition 4.7, which is also stated in Section 4.4.

To prove Theorem 4.3, we introduce a general-purpose normal form on PDAnn, where, intuitively, the only choices that can be made during a run are between the *types* of transition to apply.

Definition 4.1. *A PDAnn $\mathcal{P}$ is* deterministic-modulo-profile *if it satisfies the following conditions:*

(i) *for each state p there is at most one push transition that starts on p, formally* $|\{q, \gamma \in Q \times \Gamma \mid (p, q, \gamma) \in \Delta\}| \leq 1$

(ii) *for each state p and stack symbol γ there is at most one pop transition that starts on p, γ, formally* $|\{q \in Q \mid (p, \gamma, q) \in \Delta\}| \leq 1$

(iii) *for each state p, letter a, and output $\vartheta \in \Omega$, there is at most one read-write transition that starts on p, a, ϑ, formally, we have* $|\{q \in Q \mid (p, (a, \vartheta), q) \in \Delta\}| \leq 1$.

(iv) *for each state p and letter a, there is at most one read transition that starts on p, a, formally* $|\{q \in Q \mid (p, a, q) \in \Delta\}| \leq 1$.

Lemma 4.1. *Let $\mathcal{P}$ be a PDAnn. We can build an equivalent PDAnn $\mathcal{P}'$ which is deterministic-modulo-profile. The transformation takes exponential time, i.e., time $\mathcal{O}(2^{|\mathcal{P}|^c})$ for some $c > 0$.*

Further, on any string w, there is an accepting run of $\mathcal{P}$ on w with profile π iff there is an accepting run of $\mathcal{P}'$ on w with the same profile.

Proof. The proof is similar to the determinization of visibly pushdown automata (see Section 3.6, also Proposition 3.2).

Given $\mathcal{P} = (Q, \Sigma, \Omega, \Gamma, \Delta, q_0, F)$, we build $\mathcal{P}' = (Q', \Sigma, \Omega, \Gamma', \Delta', S_I, F')$ as follows. We build $Q' = 2^{Q \times Q}$, intuitively denoting a set of pairs of states (p, q) of $\mathcal{P}$ such that $\mathcal{P}$ can be at state q at this point if it was at state p when the topmost stack symbol was pushed. We build $\Gamma' = 2^{Q \times \Gamma \times Q}$, intuitively specifying the sets of possible stack symbols and remembering the state just after the previous stack symbol was pushed and the state just after that symbol was pushed. We build $S_I = \{(q_0, q_0)\}$, meaning that initially we are at the initial state q_0 and were here when the stack was initialized. We build $F' = \{S \mid (q_0, q) \in S \text{ for some } q \in F\}$, meaning that we accept when $\mathcal{P}$ reaches a final state and we were at the initial state when the stack was initialized. Let Δ' be defined as follows:

- The (unique) push transition from a state $S \in Q'$ makes $\mathcal{P}'$ push a stack symbol S' and move to a state T, intuitively defined as follows. For every pair (p, p') of S and push transition $(p', q, \gamma) \in \Delta$ in the original PDAnn, we can move to state (q, q) and push on the stack the symbol (p, γ, q). The stack symbol S' is the set of all possible stack symbols that can be pushed in this way, and T is the set of all possible states that can be reached in this way.

  Formally, for every $S \in Q'$ we include (S, S', T) in Δ', where:

$$T = \{(p, \gamma, q) \mid (p, p') \in S \text{ and } (p', q, \gamma) \in \Delta \text{ for some } p, p', q \in Q, \gamma \in \Gamma\},$$

$$S' = \{(q, q) \mid (p, p') \in S \text{ and } (p', q, \gamma) \in \Delta \text{ for some } p, p', q \in Q, \gamma \in \Gamma\}$$

- The (unique) pop transition from a state $S \in Q'$ and topmost stack symbol $T \in \Gamma'$ makes $\mathcal{P}'$ move to a state T' intuitively defined as follows. For every pair (p', q') of S, we consider all triples (p, γ, p') of the topmost stack symbol T, and if the original PDAnn had a pop transition $(q', \gamma, q) \in \Delta$, then we can pop the topmost stack symbol and go to the state (p, q). The new state T' is the set of all pairs (p, q) that can be reached in this way.

  Formally, for every $(S, T) \in Q' \times \Gamma'$ we include (S, T, S') in Δ', where:

$$S' = \{(p, q) \mid (p, \gamma, p') \in T \text{ and } (p', q') \in S \text{ and } (q', \gamma, q) \in \Delta \text{ for some } p, p', q, q' \in Q, \gamma \in \Gamma\},$$

- The (unique) read-write transition from a state $S \in Q'$ on a letter $a \in \Sigma$ and output ⟜ $\in \Omega$ makes $\mathcal{P}'$ move to a state S' intuitively defined as follows: we consider all pairs (p, p') in S and all transitions from p' with a and ⟜in $\mathcal{P}$ to some state q, and move to all possible pairs (p', q).

Formally, for every $(S, a, \multimap) \in Q' \times \Sigma \times \Omega$ we include $(S, (a, \multimap), S')$ in Δ', where:

$$S' = \{(p, q) \mid (p, p') \in S \text{ and } (p', (a, \multimap), q) \in \Delta \text{ for some } p, p', q \in Q\}.$$

- The (unique) read transition from a state $S \in Q'$ on a letter $a \in \Sigma$ makes $\mathcal{P}'$ move to a state S' intuitively defined as follows: we consider all pairs (p, p') in S and all transitions from p' with a in $\mathcal{P}$ to some state q, and move to all possible pairs (p', q).
  Formally, for every $(S, a) \in Q' \times \Sigma$ we include (S, a, S') in Δ', where:

$$S' = \{(p, q) \mid (p, p') \in S \text{ and } (p', a, q) \in \Delta \text{ for some } p, p', q \in Q\}.$$

It is clear by definition that $\mathcal{P}'$ is deterministic-modulo-profile, and it is clear that the running time of the construction satisfies the claimed time bound.

We now show that $\mathcal{P}$ and $\mathcal{P}'$ are equivalent.

Now, for the forward direction, let us first assume without loss of generality that whenever $\mathcal{P}$ makes a push transition then the stack symbol that it pushes is annotated with the state reached just after the push. Then we will show that every instantaneous description that can be reached by $\mathcal{P}$ can be reached by $\mathcal{P}'$ by induction on the run. Specifically, we show by induction on the length of the run ρ the following claim: if $\mathcal{P}$ has a run ρ on a string w that produces μ from an initial state $q_0 \in T$ to an instantaneous description $(q, i), \alpha$, with $\alpha = \gamma_0 p_0, \ldots, \gamma_m p_m$ being the sequence of the stack symbols and states annotating them, then $\mathcal{P}'$ has a run ρ' on w from S_I to an instantaneous description $(S, i), \alpha'$ with $\alpha' = T_0 \ldots T_m$ such that T_0 contains (q_0, γ_0, p_0), T_1 contains (p_0, γ_1, p_1), ..., T_m contains (p_{m-1}, γ_m, p_m) and S contains (p_m, q); further ρ and ρ' have the same profile.

The base case of an empty run on a string is immediate: if $\mathcal{P}$ has an empty run from an initial state q_0, then it reaches the instantaneous description with $(q_0, 0)$ and the empty stack, and then $\mathcal{P}'$ then has an empty run reaching the instantaneous description $(S, 0)$ with the empty stack and S indeed contains (q_0, q_0).

For the induction case, assume that $\mathcal{P}$ has a non-empty run ρ_+ on a string w that produces μ. First, write ρ_+ as a run ρ followed by one single transition of $\mathcal{P}$. We know $\mathcal{P}$ has a run ρ on w which produces μ from an initial state q_0 to an instantaneous description $(q, i), \alpha$, with $\alpha = \gamma_0 p_0, \ldots, \gamma_m p_m$. By the induction hypothesis, we know that $\mathcal{P}'$ has a run ρ' on w from (q_0, q_0) to an instantaneous description $(S, i), \alpha'$ with $\alpha' = T_0 \ldots T_m$ such that T_0 contains (q_0, γ_0, p_0), T_1 contains (p_0, γ_1, p_1), ..., T_m contains (p_{m-1}, γ, p_m) and S contains (p_m, q); and ρ' and ρ have the same profile. We now distinguish on the type of the transition used to extend ρ to ρ_+.

If that transition is a read-write transition $(q, (a, \vartheta), q')$, we consider the read-write transition of $\mathcal{P}'$ labeled with (a, ϑ) from T, and call S' the state that $\mathcal{P}'$ reaches. As $(p_m, q) \in S$ and $(q, (a, \vartheta), q') \in \Delta$, we know that $(p_m, q') \in S'$. Thus, $\mathcal{P}'$ can read (a, ϑ) and reach a suitable state S' and position $i + 1$ and the stacks are unchanged so the claim is proven.

If that transition is a read transition (q, a, q'), we follow an analogous reasoning.

If that transition is a push transition (q, q', γ), the position of $\mathcal{P}$ is unchanged and the new stack is extended by γ annotated with state q'. Consider the push transition of $\mathcal{P}'$ from q, and call S' the state reached and $T = T_{m+1}$ the stack symbol that is pushed. As $(p_m, q) \in S$ and $(q, q', \gamma) \in \Delta$, we know that T contains (p_m, γ, q'), and S' contains (q', q'), which is what we needed to show.

If that transition is a pop transition, (q, γ_m, q'), the position of $\mathcal{P}$ is unchanged and the topmost stack symbol is removed. Consider the topmost stack symbol T_m and the transition of $\mathcal{P}'$ that pops it from S, and call S' the state that we reach. We know that S contains (p_m, q) and T_m contains (p_{m-1}, γ_m, p_m) and $(q, \gamma_m, q') \in \Delta$, so S' contains (p_{m-1}, q'), which is what we needed to show.

Note that, in all four cases, the profile of ρ_+ and ρ'_+ is the same, because this was true of ρ and ρ', and the type of transition done to extend ρ' to ρ'_+ is the same as the type of transition done to extend ρ to ρ_+.

The inductive claim is therefore shown, and thus if $\mathcal{P}$ has a run ρ on some string w that produces μ starting at some initial state q_0 and ending at state q, then $\mathcal{P}'$ has a run ρ' on w which produces μ and ending at a state of the form (q_0, q) for q_0 and having same profile. Thus, if ρ is accepting then q is final for $\mathcal{P}$ and (q_0, q) is final for $\mathcal{P}'$ so ρ' is accepting. This concludes the forward implication.

We now show the backward implication, and show it again by induction, again assuming that $\mathcal{P}$ annotates the symbols of its stack with the state reached just after pushing them. We show by induction on the length of a run ρ' the following claim: if $\mathcal{P}'$ has a run ρ' on a string w that produces $\mu$from its initial state to an instantaneous description $(S, i), \alpha'$ with $\alpha' = T_0, \ldots, T_m$ being the sequence of the stack symbols, then for any choice of elements $(q_0, \gamma_0, p_0) \in T_0$, $(p_0, \gamma_1, p_1) \in T_1$, ..., $(p_{m-1}, \gamma_m, p_m) \in T_m$ and $(p_m, q) \in S$ it holds that $\mathcal{P}$ has a run ρ on w producing μ from some initial state q_0 to the instantaneous description $(q, i), \alpha$ with $\alpha = \gamma_0 q_0, \ldots, \gamma_m q_m$ (writing next to each stack symbol the state that annotates it), and ρ' and ρ have the same profile.

The base case of an empty run on a string is again immediate: if $\mathcal{P}'$ has an empty run from its initial state, then it reaches the instantaneous description with $(S_I, 0)$ and empty stack, and then $\mathcal{P}$ has an empty run from any initial state q_0 to q_0 so that indeed S_I contains (q_0, q_0).

For the induction case, assume that $\mathcal{P}'$ has a non-empty run ρ'_+ on w which produces μ, We write again ρ'_+ as a run ρ' followed by one single transition of $\mathcal{P}'$. We know $\mathcal{P}'$ has a run ρ' on w which produces μ from the initial state S_I to an instantaneous description $(S, i), \alpha'$, with $\alpha = T_0 \ldots T_m$. By the induction hypothesis, we know that for any choice of elements $(q_0, \gamma_0, p_0) \in T_0$, $(p_0, \gamma_1, p_1) \in T_1$, ..., $(p_{m-1}, \gamma_m, p_m) \in T_m$ and $(p_m, q) \in S$, then $\mathcal{P}$ has a run ρ on w which produces μ from some initial state q_0 to the instantaneous description $(q, i), \alpha$ with $\alpha = \gamma_0 q_0, \ldots, \gamma_m q_m$, and ρ and ρ' have the same profile. We now distinguish on the type of transition used to extend ρ' to ρ'_+.

If the last transition is a read-write transition $(S, (a, \multimap), S')$ with S' defined as in the construction, consider any choice of $(q_0, \gamma_0, p_0) \in T_0$, $(p_0, \gamma_1, p_1) \in T_1$, ..., $(p_{m-1}, \gamma_m, p_m) \in T_m$ and $(p_m, q') \in S'$, and then there must be some state p'' such that $(p'', (a, \multimap), q) \in \Delta$ and $(p_m, p'') \in S$. Using the induction hypothesis but picking $(p_m, p'') \in S$, we obtain a run ρ of $\mathcal{P}$ on w which produces μ, with the correct stack and ending at position i on state p'', which we can extend by the read transition $(p'', (a, \multimap), q)$ to reach state q at position $i + 1$ without touching the stack, proving the result.

If the last transition is a read transition (S, a, S') with S' defined as in the construction, we follow an analogous reasoning.

If the last transition is a push transition (S, S', T) with T defined as in the construction, consider any choice of $(q_0, \gamma_0, p_0) \in T_0$, $(p_0, \gamma_1, p_1) \in T_1$, ..., $(p_{m-1}, \gamma_m, p_m) \in T_m$, $(p_m, \gamma_{m+1}, p_{m+1}) \in T_{m+1}$ and $(p_{m+1}, q') \in S'$. We know that we must have $q' = p_{m+1}$, and that there must be some state p'' and push transition $(p'', p_{m+1}, \gamma_{m+1})$ and pair (p_m, p'') in S. Using the induction hypothesis but picking $(p_m, p'') \in S$, we obtain a run ρ of $\mathcal{P}$ on w which produces μ with topmost stack symbol γ_m, ending at state p'', which we can extend with the push transition $(p'', p_{m+1}, \gamma_{m+1})$ to obtain the desired stack and reach state $p_{m+1} = q'$, proving the result.

If the last transition is a pop transition (S, T, S') with S' defined as in the construction, consider any choice of $(q_0, \gamma_0, p_0) \in T_0$, $(p_0, \gamma_1, p_1) \in T_1$, ..., $(p_{m-2}, \gamma_{m-1}, p_{m-1}) \in T_{m-1}$, and $(p_{m-1}, q) \in S'$. We know that there is a pair $(p', q') \in S$ and a triple (p_{m-1}, γ_m, p') in T_m and a pop transition (q', γ_m, q) in Δ. Applying the induction hypothesis, we get a run ρ of $\mathcal{P}$ on w which produces μ and with topmost stack symbol γ_m annotated with state p' and ending at state q'. The pop transition (q', γ_m, q) allows us to extend this run to reach state q and remove the topmost stack symbol, while the rest of the stack is correct, proving the result.

Again, we have ensured that ρ is extended to ρ_+ with the same transition as the transition used to extend ρ' to ρ'_+, ensuring that ρ_+ and ρ'_+ have same profile. This concludes the proof of the backward induction, ensuring that if $\mathcal{P}'$ has a run from S_I to some final state S reading a string w and producing μ, and having (q_0, q_f) with $q_\mathrm{f} \in F$ in S, then $\mathcal{P}$ has a run reading w which produces μ going from q_0 to the final state q_f. This concludes the backward implication and completes the proof. □

We can now show Theorem 4.3 via Proposition 4.7, using also the notion of *profiled PDAnn* defined in Section 4.4:

Let $\mathcal{G}$ be a rigid annotated grammar. Using Proposition 4.7, we transform it in polynomial time to a profiled PDAnn $\mathcal{P}$. Using Lemma 4.1, we build in exponential time an equivalent PDAnn $\mathcal{P}'$ satisfying the conditions of the lemma.

We know that $\mathcal{P}'$ is still profiled. Indeed, if we assume by contradiction that there is a string w on which $\mathcal{P}'$ has two accepting runs with different profiles, then by the last condition of Lemma 4.1, the same is true of $\mathcal{P}$, contradicting the fact that $\mathcal{P}$ is profiled.

Now, we claim that $\mathcal{P}'$ is necessarily also unambiguous. To see why, consider two accepting runs ρ and ρ' of $\mathcal{P}'$ on some string w. Since $\mathcal{P}'$ is profiled, ρ and ρ' must have the same profile. But now, the conditions of Lemma 4.1 ensure that, knowing the input string w and profile, the runs ρ and ρ' are completely determined. Specifically, this is an immediate induction on the run. The base case is that there is only one initial state, so both ρ and ρ' must have the same initial state. Now, assuming by induction that the runs so far are identical and have the same stack, there are three cases. First, if the profile tells us that both runs make a push transition, the symbol pushed and state reached are determined by the last states of the runs so far, which are identical by inductive hypothesis. Second, if the profile tells us that both runs make a read-write transition (or read transition), the state reached is determined by the input and output symbols (or just the input symbol), and by the last states of the run so far, which are identical by inductive hypothesis. Third, if the profile tells us that both runs make a pop transition, the state reaches is determined by the last state of the run so far, and the topmost symbol of the stack, which are identical by inductive hypothesis. This concludes the inductive proof.

Thus, for any two accepting runs ρ and ρ' on the string w which produce the same output, they must identical. Thus, $\mathcal{P}'$ is unambiguous. We use Theorem 4.7 to transform $\mathcal{P}'$ back into an annotated grammar, which is still rigid and unambiguous, and equivalent to the original rigid annotated grammar $\mathcal{G}$. The overall complexity of the transformation is in $\mathcal{O}((2^{(|\mathcal{G}|^c)^{c'}})^{c''})$ for some $c, c', c'' > 0$, so it is in $\mathcal{O}(2^{|\mathcal{G}|^d})$ for some $d > 0$ overall, and the time complexity is as stated. $\square$

Expressiveness of rigid grammars. Armed with Theorem 4.3, we study what is the expressive power of rigid grammars. For this, let us first go back to the setting without annotations. Theorem 4.3 tells us that for (unannotated) CFGs the rigidity requirement is equivalent to the usual unambiguity requirement: each accepted word has a unique derivation. Now, for the case of an annotated grammar $\mathcal{G}$, rigidity additionally imposes the requirement that all annotations of an input string have the same parse tree. In particular, the language of the strings where $\mathcal{G}$ accepts some annotation must be recognizable by a rigid (unannotated) CFG, hence an unambiguous CFG (by Theorem 4.3). Formally:

PROPOSITION 4.4. *For a rigid grammar $\mathcal{G}$, let L' be the set of strings with nonempty output, i.e., $L' = \{w \mid [\![\mathcal{G}]\!](w) \neq \emptyset\}$. Then L' is recognized by an unambiguous CFG.*

PROOF. This proof is based on extending the definitions of unambiguity and rigidness of annotated grammars over unannotated context-free grammars. Indeed, an unambiguous annotated grammar with an empty output set is just an unambiguous CFG, and a rigid annotated grammar with an empty output set is a CFG for which every derivation of a given string $w \in \Sigma^*$ has the same shape.

Consider the (unannotated) grammar $\mathcal{G}'$ obtained from $\mathcal{G}$ by removing all annotations on terminals, and making $\Omega = \emptyset$. It can be seen that $L(\mathcal{G}') = L'$ since for each string w, if $w \in L(\mathcal{G}')$, then there is at least one $\hat{w} \in L(\mathcal{G})$ with $\mathsf{str}(\hat{w}) = w$ and vice versa. Now, we claim that $\mathcal{G}'$ is *rigid*, by extending the notion onto CFGs in the obvious way. To see this, consider a string $w \in L(\mathcal{G}')$; all derivations of w by $\mathcal{G}'$ correspond to derivations by $\mathcal{G}$ of some $\hat{w}$ such that $\mathsf{str}(\hat{w}) = w$. Because $\mathcal{G}$ is rigid, all these derivations have the same shape. Now, using Theorem 4.3, we can compute a rigid and unambiguous grammar $\mathcal{G}''$ recognizing the same language over Σ^* as $\mathcal{G}'$, i.e., L'. But as L' is a language without output, the unambiguity of $\mathcal{G}''$ actually means that $\mathcal{G}''$ is an unambiguous CFG. Hence, L' is recognized by an unambiguous grammar, concluding the proof. □

This yields concrete examples of languages (on the empty annotation alphabet) that cannot be recognized by a rigid annotated grammar, e.g., inherently ambiguous context-free languages such as $L_\mathrm{a} = \{a^i b^j c^k \mid i, j, k \geq 1 \wedge (i = j \vee j = k)\}$ on $\{a, b, c\}^*$ (Maurer, 1969). Proposition 4.4 also implies that we cannot decide if the language of an annotated grammar can be expressed instead by a rigid grammar, or if an annotated grammar is rigid:

PROPOSITION 4.5. *Given an unannotated grammar $\mathcal{G}$, it is undecidable to determine whether $\mathcal{G}$ is rigid, and it is undecidable to determine whether there is some equivalent rigid grammar $\mathcal{G}'$.*

PROOF. We first show the undecidability of checking if an annotated grammar has an equivalent rigid annotated grammar:

CLAIM 4.1. *Consider the problem, given an annotated grammar $\mathcal{G}$, of determining whether there exists some equivalent rigid annotated grammar equivalent to $\mathcal{G}$. This problem is undecidable.*

PROOF. We reduce from the problem of deciding whether the language L_2 of an input (unannotated) context-free grammar $\mathcal{G}_2$ can be recognized by an unambiguous context-free grammar: this task is known to be undecidable (Ginsburg & Ullian, 1966). Consider $\mathcal{G}_2$ as an annotated grammar (with empty annotations). Let us show that L_2 can be recognized by a rigid annotated grammar iff it can be recognized by an unambiguous context-free grammar, which concludes. For the forward direction, if L_2 can be recognized by an unambiguous context-free grammar, then that grammar is in particular rigid. For the backward direction, if L_2 can be recognized by a rigid grammar, then Proposition 4.4 implies that L_2 can also be recognized by an unambiguous context-free grammar. Thus, we have showed that the (trivial) reduction is correct. □

We next show that it is undecidable to check if an input annotated grammar is rigid:

CLAIM 4.2. *Consider the problem, given an annotated grammar $\mathcal{G}$, of determining whether it is rigid. This problem is undecidable.*

PROOF. We adapt the standard proof of undecidability (Chomsky & Schützenberger, 1959, Ambiguity Theorem 2) for the problem of deciding, given an input unannotated grammar $\mathcal{G}$, if it is unambiguous. The reduction is from the Post Correspondence Problem (PCP), which is undecidable: we are given as input sequences $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$ of strings over some alphabet Σ, and we ask whether there is a non-empty sequence of indices $i_1, \ldots, i_m$ of integers in $[1, n]$ such that $\alpha_{i_1} \ldots \alpha_{i_m} = \beta_{i_1} \ldots \beta_{i_m}$. Given the input sequences $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$ to the PCP, we consider the alphabet $\Sigma' = \Sigma \cup \{1, \ldots, n\}$, and we consider the CFG having nonterminals S, S_1, and S_2, start symbol S, and rules $S \rightarrow S_1$, $S \rightarrow S_2$, $S_1 \rightarrow \varepsilon$, $S_2 \rightarrow \varepsilon$, and for each $1 \leq i \leq n$ the productions $S_1 \rightarrow \alpha_i S_1 i$ and $S_2 \rightarrow \beta_i S_2 i$.

We claim that this grammar is ambiguous iff there is a solution to the Post correspondence problem. Indeed, given any solution $\alpha_{i_1} \cdots \alpha_{i_m} = \beta_{i_1} \ldots \beta_{i_m}$, considering the string $\alpha_{i_1} \cdots \alpha_{i_m} i_m \cdots i_1 = \beta_{i_1} \ldots \beta_{i_m} i_m \cdots i_1$, we can parse it with one derivation featuring S_1 and one derivation featuring S_2. Conversely, if we can parse a string $w \in \Sigma^*$ with two different derivations, we know that there cannot be two different derivations featuring S_1. Indeed, reading the string from right to left uniquely identifies the possible derivations from S_1. The same argument applies to derivations featuring S_2. Hence, if the grammar is ambiguous, then there is exactly one derivation featuring S_1 and exactly one derivation featuring S_2. These two derivations can be used to find a solution to the Post correspondence problem.

We now adapt this proof to show the undecidability of rigidity. We say that an input to the PCP is *trivial* if there is i such that $\alpha_i = \beta_i$. We can clearly decide in linear time, given the input to the PCP, if it is trivial. Hence, the PCP is also undecidable in the case where the PCP is non-trivial. Now, when doing the reduction above on a PCP instance that is not trivial, we observe that two derivations of the same string can never have the same sequences of shapes. Indeed, if we have two derivations of the same string, then as we explained one must feature S_1 and the other must feature S_2, and they give a solution $\alpha_{i_1} \cdots \alpha_{i_m} = \beta_{i_1} \ldots \beta_{i_m}$ to the PCP. Assume by contradiction that both derivations have the same sequences of shapes. Then, it means that we have $\left|\alpha_{i_j}\right| = \left|\beta_{i_j}\right|$ for every $1 \leq j \leq m$. In particular we have $|\alpha_{i_1}| = |\beta_{i_1}|$, and so we know that $\alpha_{i_1} = \beta_{i_1}$ and the PCP instance was trivial, a contradiction.

Hence, let us reduce from the PCP on non-trivial instances to the problem of deciding whether an input annotated grammar is not rigid. Given a non-trivial PCP instance, we construct $\mathcal{G}$ as above, but seeing it as an annotated grammar with no outputs. Then $\mathcal{G}$ is not rigid iff there is a string w such that the empty annotation of w has two derivations that do not have the same sequence of shapes. But this is equivalent to $\mathcal{G}$ being unambiguous when seen as a CFG. Indeed, for the forward direction, if $\mathcal{G}$ has two such derivations on a string w then clearly w witnesses that $\mathcal{G}$ is ambiguous when seen as a CFG. Conversely, if $\mathcal{G}$ is ambiguous when seen as a CFG, we have explained in the previous paragraph that the two derivations must have different sequences of shapes, so $\mathcal{G}$ is not rigid. Hence, we conclude that there is a solution to the input non-trivial PCP instance iff $\mathcal{G}$ is not rigid. This establishes that the problem is undecidable and concludes the proof. □

The proof follows from Claims 4.1 and 4.2. □

These undecidability results make rigid grammars less appealing, but note that our enumeration algorithm for such grammars applies in particular to decidable grammar classes which are designed to ensure rigidity. For instance, this would be the case of grammars arising from visibly pushdown automata, which we discuss in more detail in the next section.

Enumeration algorithm. We now give our algorithm with quadratic preprocessing time for rigid grammars. Given a rigid grammar, we first make it unambiguous if necessary, using Theorem 4.3, in exponential time in the input grammar. The result is a rigid and unambiguous annotated grammar. Now, we transform it in 2NF like in Section 4.3: this takes linear time, preserves unambiguity, and one can check that it also preserves rigidity.

Armed with our rigid and unambiguous grammar $\mathcal{G}$ in 2NF, we can simply use Algorithm 3 to construct a data structure allowing us to enumerate the outputs with output-linear delay. But we now claim that Algorithm 3 runs in time $\mathcal{O}(|\mathcal{G}| \cdot |w|^2)$ because $\mathcal{G}$ is rigid.

For this, we study for every nonterminal X and pair $1 \leq i \leq j \leq n+1$ how many times we can consider the cell $\mathbb{I}[i][j][X]$ in lines 20–23. Whenever we consider it, we witness the existence of a complex rule $X \to YZ$ and a value k such that $\mathbb{I}[i][k][Y]$ and $\mathbb{I}[k][j][Z]$ are nonempty (the first is because $i \in \mathsf{endIn}[k][Z]$). Thus, we witness a derivation from X of some annotation of the string $a_i \cdots a_{j-1}$ that starts with a rule $X \to YZ$ where Y derives some annotation of the string $a_i \cdots a_{k-1}$ and Z derives some annotation of the string $a_k \cdots a_{j-1}$. We now claim that, for (i, j, X), the rigidity of the grammar ensures that there is only one such value k. Indeed, assume by contradiction that we have two rules $X \to YZ$ and $X \to Y'Z'$ and two values $i \leq k < k' \leq j$ such that Y and Y' respectively derive some annotation of the strings $a_i \cdots a_{k-1}$ and $a_i \cdots a_{k'-1}$, and Z and Z' respectively derive some annotation of the strings $a_k \cdots a_{j-1}$ and $a_{k'} \cdots a_{j-1}$. Then once we are done rewriting Y and all the nonterminals that it generates in the first derivation, we obtain a different shape from what we obtain after rewriting Y' and all the nonterminals it generates in the second derivation, contradicting the rigidity of the grammar.

Thus, whenever we consider the cell $\mathbb{I}[i][j][X]$ in lines 21–23, it is for one value of k which is unique for (i, j, X), and we thus consider the cell once at most for every complex rule of the grammar with X as left-hand-side. Thus, we consider the cells of $\mathbb{I}[i][j]$ at most $|\mathcal{G}|$ times in total. As there are $\mathcal{O}(n^2)$ pairs (i,j), this ensures that the total running time of the innermost for loop (lines 19–23), and that of the entire algorithm, is indeed in $\mathcal{O}(|\mathcal{G}| \cdot |w|^2)$:

Theorem 4.4. *Given a rigid annotated grammar $\mathcal{G}$ and an input string w, we can enumerate $[\![\mathcal{G}]\!](w)$ with preprocessing in $\mathcal{O}(|w|^2)$ data complexity and output-linear delay (independent from w or $\mathcal{G}$). The combined complexity of the preprocessing is $\mathcal{O}(2^{|\mathcal{G}|^c} \cdot |w|^2)$ for some $c > 0$, or $\mathcal{O}(|\mathcal{G}| \cdot |w|^2)$ if $\mathcal{G}$ is additionally assumed to be unambiguous.*

Optimality. We now turn to the question of whether the quadratic preprocessing time for rigid grammars is optimal. For this, we notice that the parsing of (unannotated) unambiguous grammars can be performed in quadratic time, but the question of finding a better algorithm was open as of 2012 (Schmitz, 2012). Now, this is a special case of our problem, because an unannotated unambiguous grammar is in particular a rigid and unambiguous annotated grammar, and enumerating the outputs of an unannotated grammar just means deciding in constant time after the preprocessing whether the input unannotated string is accepted or not. Thus:

PROPOSITION 4.6. *Any algorithm to enumerate the accepted outputs of a rigid annotated grammar can be used to test if an input string is accepted by an unambiguous unannotated grammar, with same complexity as that of the preprocessing phase.*

PROOF. Assume we have such an algorithm $\mathcal{A}$. Consider a procedure which receives an unambiguous unannotated CFG $\mathcal{G}$ and an input string w, converts $\mathcal{G}$ into an annotated grammar $\mathcal{G}'$ with empty output set. Since $\mathcal{G}'$ is unambiguous and rigid, we can run $\mathcal{A}$ over $\mathcal{G}'$ and w. If $w \in L(\mathcal{G})$, then $[\![\mathcal{G}']\!](w) = \{\varepsilon\}$, and if $w \notin L(\mathcal{G})$, then $[\![\mathcal{G}']\!](w) = \emptyset$. Thus, after the preprocessing phase of $\mathcal{A}$ we need only to wait a constant amount of time to see if the string ε is given as output, or none is. We conclude that this procedure solves the problem with the same complexity as the preprocessing phase of $\mathcal{A}$. □

For this reason, we leave open the question of whether a better than quadratic preprocessing time can be achieved in this case.

4.4. Pushdown Annotators

We have presented an enumeration algorithm for annotated grammars that achieves quadratic-time preprocessing and output linear delay on rigid annotated grammars. We now study whether the bound can be improved even further to achieve linear-time preprocessing and output-linear delay, which is the best possible data complexity bound in our model.

To achieve this, it is natural to look for a class of grammars having some "deterministic" behavior. Unfortunately, grammars are not convenient for this purpose, and so we move to the equivalent model of pushdown automata. We thus introduce *pushdown annotators* and show that they are equally expressive to annotated grammars. We present syntactic restrictions on pushdown annotators that ensure quadratic-time preprocessing, similarly to rigid grammars. Then, we propose additional deterministic conditions on pushdown annotators that allow for linear-time preprocessing.

Pushdown annotators. A *pushdown annotator* (PDAnn) is a tuple $\mathcal{P} = (Q, \Sigma, \Omega, \Gamma, \Delta, q_0, F)$ where Q is a finite set of *states*, Σ is the alphabet, Ω is a finite set of annotations, Γ is a finite set of *stack symbols*, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ are the *final states*. We assume that the set Γ of stack symbols is disjoint from $(\Sigma \cup \Sigma \times \Omega)$. Finally, Δ is a finite set of *transitions* that are of the following kinds:

- *Read-write transitions* of the form $(p, (a, \vartheta), q) \in Q \times (\Sigma \times \Omega) \times Q$, meaning that, if the next letter of the string is a, the annotator can go from states p to q while reading that letter and writing the annotation ϑ;
- *Read-only transitions* of the form $(p, a, q) \in Q \times \Sigma \times Q$, meaning that the annotator can go from p to q while reading a;

- *Push transitions* of the form $(p, q, \gamma) \in Q \times (Q \times \Gamma)$, meaning that the annotator can go from p to q while pushing the symbol γ on the stack;
- *Pop transitions* of the form $(p, \gamma, q) \in (Q \times \Gamma) \times Q$, meaning that, if the topmost symbol of the stack is γ, the annotator can go from p to q while removing this topmost symbol γ.

We now give the semantics of PDAnns. Fix a string $w = w_1 \cdots w_n \in \Sigma^*$. A *configuration* of $\mathcal{P}$ over w is a pair $C = (q, i) \in Q \times [0, n]$ of the current state and position in w. An *instantaneous description* of $\mathcal{P}$ is a pair (C, α) where C is a configuration and $\alpha \in \Gamma^*$ describes the stack. A *run* of $\mathcal{P}$ over w is a sequence:

$$\rho \ := \ (C_0, \alpha_0) \xrightarrow{t_1} (C_1, \alpha_1) \xrightarrow{t_2} \ \ldots \ \xrightarrow{t_m} (C_m, \alpha_m) \tag{4.1}$$

such that $C_0 = (q_0, 0)$ and $\alpha_0 = \varepsilon$, each t_k is a transition in Δ, and for each $k \in [1, m]$ the following hold:

- if t_k is a read-write transition $(p, (a, \vartheta), q)$ or a read-only transition (p, a, q), then $\alpha_k = \alpha_{k-1}$, $C_{k-1} = (p, i-1)$, $C_k = (q, i)$ and $a = a_i$ for some $i \in [1, n]$;
- if t_k is a push transition (p, q, γ), then $\alpha_k = \alpha_{k-1}\gamma$ and for some $i \in [1, n]$, $C_{k-1} = (p, i)$, $C_k = (q, i)$; and
- if t_k is a pop transition (p, γ, q), then $\alpha_{k-1} = \alpha_k\gamma$ and for some $i \in [1, n]$, $C_{k-1} = (p, i)$, $C_k = (q, i)$.

We say that ρ is *accepting* if $(C_m, \alpha_m) = ((q_f, n), \varepsilon)$ for some $q_f \in F$. We define the *annotation* of ρ as $\mathsf{ann}(\rho) = \mathsf{ann}(t_1, C_1) \cdot \cdots \cdot \mathsf{ann}(t_m, C_m)$ such that $\mathsf{ann}(t, C) = \varepsilon$ if t is a push, pop, or a read-only transition (p, a, q), and $\mathsf{ann}(t, C) = (i, \vartheta)$ if t is a read-write transition $(p, (a, \vartheta), q)$ and $C = (q, i)$. Finally, we define the function $[\![\mathcal{P}]\!]$ that maps any $w \in \Sigma^*$ to its set of outputs:

$$[\![\mathcal{P}]\!](w) = \{\mathsf{ann}(\rho) \mid \rho \text{ is an accepting run of } \mathcal{P} \text{ over } w\}.$$

Similarly to annotated grammars, we say that $\mathcal{P}$ is *unambiguous* if for every $w \in \Sigma^*$ and output μ, there exists at most one accepting run ρ of $\mathcal{P}$ over w such that $\mathsf{ann}(\rho) = \mu$.

One can alternatively see a PDAnn as a *pushdown transducer* (Berstel, 2013), which is the standard way to extend automata to have an output. However, an important difference is that a PDAnn concisely represents outputs by only writing the annotations and their positions: this can be much smaller than the input string, and cannot easily be encoded as a transducer on a finite alphabet. For instance, where a PDAnn can produce an output such as $(2, \propto), (5, \propto')$, a transducer would either write $\propto\propto'$ (losing the position information) or $␣\propto␣␣\propto'$ (whose length is always linear in the input) for a special symbol $␣$.

Profiled PDAnns and annotated grammars. To define the analogue of rigid annotated grammars on PDAnn, we will study the *stack profile* (or simply *profile*) of PDAnn runs, which is informally the sequence of all stack heights. Formally, let $\mathcal{P}$ be a PDAnn, w be a string, and consider a run ρ of $\mathcal{T}$ over w like in (†). The *profile* π of ρ is the sequence $\pi := |\alpha_0|, \ldots, |\alpha_m|$. We then introduce *profiled* PDAnns by requiring that all accepting runs of the PDAnn on an input string have the same profile (no matter their output). Formally, we say that a PDAnn $\mathcal{P}$ is *profiled* if, for every string w, all accepting runs of $\mathcal{T}$ over w have the same profile.

As usual for context-free grammars and pushdown automata, the formalisms of annotated grammars and PDAnn have the same expressive power. We call two annotated grammars $\mathcal{G}$ and $\mathcal{G}'$ *equivalent* if they define the same functions, i.e., $[\![\mathcal{G}]\!] = [\![\mathcal{G}']\!]$, and extend this notion to PDAnn in the expected way. We then have:

PROPOSITION 4.7. *Annotated grammars and PDAnn are equally expressive. Specifically, for any annotated grammar $\mathcal{G}$, we can build an equivalent PDAnn $\mathcal{P}$ in polynomial time, and vice versa. Further, $\mathcal{G}$ is unambiguous (resp., rigid) iff $\mathcal{P}$ is unambiguous (resp., profiled).*

PROOF. In this proof, we will need to use the standard notion of a *pushdown automaton* (PDA), whose definition has been omitted so far. We give it here:

Definition 4.2. *A* pushdown automaton *(PDA) is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, q_0, F)$, where Q is a finite set of* states*, Σ is the alphabet, Γ is a finite alphabet of* stack symbols*, $q_0 \in Q$ is the* initial state*, $F \subseteq Q$ are the* final states*. We assume that Γ is disjoint from Σ. Further, Δ is a finite set of* transitions *of the following kind:*

- Read transitions *of the form $(p, a, q) \in Q \times \Sigma \times Q$, meaning that the automaton can go from state p to state q while reading the letter a;*
- Push transitions *of the form $(p, q, \gamma) \in Q \times Q \times \Gamma$, meaning that the automaton can go from state p to state q while pushing the symbol γ on the stack;*
- Pop transitions *of the form $(p, \gamma, q) \in Q \times \Gamma \times Q$, meaning that, if the topmost symbol of the stack is γ, the automaton can go from p to q while removing this topmost symbol γ.*

We omit the definition of the semantics of PDAs, which are standard, and allow us to define the *language* $L(\mathcal{A})$ accepted by a PDA. It is also well-known that CFGs and PDAs have the same expressive power, i.e., given a CFG G, we can build in polynomial time a PDA $\mathcal{A}$ which is *equivalent* in the sense that $L(G) = L(\mathcal{A})$, and vice-versa.

We will also need to use the standard notion of a *deterministic* PDA (with acceptance by final state). Formally:

Definition 4.3. *Let $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, q_0, F)$ be a PDA. For $p \in Q$, we define the* next-transitions *of p as the set $\Delta(p)$ of all transitions in Δ that start on p, i.e., $\Delta(p) = \{(p, x, y) \mid (p, x, y) \in \Delta\}$. We say that a PDA $\mathcal{A}$ is* deterministic *if for every state $q \in Q$, one of the following conditions hold:*

(a) *$\Delta(q) \subseteq Q \times \Sigma \times Q$ and $|\{q' \mid (q, a, q') \in \Delta\}| \leq 1$ for each $a \in \Sigma$. Informally, all applicable transitions are read transitions, and there is at most one such applicable transition for each letter.*

(b) $\Delta(q) \subseteq Q \times (Q \times \Gamma)$*, and* $|\Delta(q)| \leq 1$*. Informally, all applicable transitions are push transitions, and there is at most one such transition from* q*.*

(c) $\Delta(q) \subseteq (Q \times \Gamma) \times Q$ *and* $|\{q' \mid (q, \gamma, q')\}| \leq 1$ *for each* $\gamma \in \Gamma$*. Informally, all applicable transitions from* q *are pop transitions, and there is at most one such applicable transition for each stack symbol.*

It is clear that the definition ensures that, on every input string w, a deterministic PDA $\mathcal{A}$ has at most one run accepting w, so that we can check in linear time in $\mathcal{A}$ and w if $w \in L(\mathcal{A})$. Further, it is known that deterministic PDAs are strictly less expressive than general PDAs.

In order to prove the statement of Proposition 4.7, let us first give the formal definitions needed for the result. We say that two annotated grammars $\mathcal{G}$ and $\mathcal{G}'$ are *equivalent* if they define the same functions, i.e., $[\![\mathcal{G}]\!] = [\![\mathcal{G}']\!]$. We define equivalence in the same way for two PDAnns, or for an annotated grammar and a PDAnn.

We first show one direction:

CLAIM 4.3. *For any annotated grammar* $\mathcal{G}$*, we can build an equivalent PDAnn* $\mathcal{P}$ *in polynomial time. Further, if* $\mathcal{G}$ *is unambiguous then so is* $\mathcal{P}$*. Moreover, if* $\mathcal{G}$ *is rigid, then* $\mathcal{P}$ *is profiled.*

PROOF. This is a standard transformation. Let $\mathcal{G} = (V, \Sigma, \Omega, P, S)$. We build a PDAnn $\mathcal{P} = (Q, \Sigma, \Omega, \Gamma, \Delta, q_0, F)$ as follows: For every rule $X \to \alpha$ of $\mathcal{G}$ and position $0 \leq i \leq |\alpha|$, the PDAnn $\mathcal{P}$ has a state (X, α, i) in Q, plus a special state q_0 which is the only initial and only final state. Also, $\Gamma = Q$. The set Δ has the following transitions:

- A push transition $(q_0, (S, \alpha, 0), q_0)$ and a pop transition $((S, \alpha, |\alpha|), q_0, q_0)$, for every rule $S \to \alpha$.
- For each state $(X, \alpha, |\alpha|)$ for a production $X \to \alpha$, a pop transition reading a state from the stack and moving to that state.

- For each state (X, α, i) where the $(i+1)$-th element of α (numbered from 1) is a nonterminal Y, for every rule $Y \to \beta$, a pop transition pushing $(X, \alpha, i+1)$, and moving to $(Y, \beta, 0)$.
- For each state (X, α, i) where the $(i+1)$-th element of α (numbered from 1) is a terminal τ, a read transition moving to $(X, \alpha, i+1)$ reading the symbol of τ and outputting the annotation of τ (if any).

We will show a function that maps a given leftmost derivation $S \Rightarrow_{\mathcal{G}} \gamma_1 \Rightarrow_{\mathcal{G}} \ldots \Rightarrow_{\mathcal{G}} \gamma_m = \hat{w}$ into a run in $\mathcal{P}$. To do this, we convert is sequence of productions into a sequence of strings which has the same size as the run (minus one). These strings serve as an intermediate representation of both the derivation and the run. The process is essentially to simulate the run in $\mathcal{P}$.

- First, we reduce the derivation into a sequence of productions $X_1 \Rightarrow \gamma_1, X_2 \Rightarrow \gamma_2, \ldots, X_m \Rightarrow \gamma_m$ which uniquely defines the derivation.
- The alphabet in which we represent strings that produce other strings include two special markers $\downarrow$ and $\uparrow$.
- We start on the string $\downarrow S \uparrow$.
- If the current string is $\hat{u} \downarrow X\beta$, and it is the i-th one that has reached a string of this form, then it must hold that $X = X_i$. We follow it by $\hat{u} \downarrow \gamma_i \uparrow \beta$.
- If the current string is $\hat{u} \downarrow \tau\beta$, for some terminal τ, we follow it by $\hat{u}\tau \downarrow \beta$.
- If the current string is $\hat{u} \downarrow\uparrow \beta$, then we follow it by $\hat{u} \downarrow \beta$.
- If the current string is $\hat{u} \downarrow$, there is no follow up.

Interestingly, this function is completely reversible, since to obtain a sequence of productions from a sequence of strings in this model, all we need to do is to remove the markers $\downarrow$ and $\uparrow$ and eliminate the duplicate strings that appear. We will borrow the name splain to talk about the function which receives a string and returns one which deletes all markers. It is obvious that the resulting derivation is the original one.

Furthermore, and more interestingly, we can extend the function shape to receive one of these strings and return a string in the alphabet $\{0, 1, \downarrow, \uparrow\}$. For two derivations that have the same shape, the resulting sequences have the same shape as well.

This sequence of strings represents a run in $\mathcal{P}$ almost verbatim, and we only need to adapt it into a sequence of pushes, pops and reads: We make a run ρ which starts on q_0, pushes $(X, \gamma_1, 0)$ to the stack, and moves to the state $(X, \gamma_1, 0)$. This pairs exactly to the strings $\downarrow S$ and $\downarrow \gamma_1 \uparrow$, which are the first two in the sequence. Then, we read the sequence of strings in order. If the current string is $\hat{u} \downarrow X\beta$, and this is the i-th time a string of this form is seen, then the current state must be $(Y, \alpha_1 X_i \alpha_2, k)$, where $|\alpha_1| = k$; we push $(Y, \alpha_1 X_i \alpha_2, k+1)$ onto the stack, and move to the state $(X_i, \gamma_i, 0)$. If the current string is $\hat{u} \downarrow a\beta$ for some $a \in \Sigma$, and the current state is (X, γ, k), we read τ, and move to the state (X, γ, k). If the current string is $\hat{u} \downarrow (a, \vartheta)\beta$ for some $a \in \Sigma$, and the current state is (X, γ, k), we read a, output ϑ, and move to the state (X, γ, k). If the current string is $\hat{u} \downarrow\uparrow \beta$, we pop the topmost state from the stack and we move into that state. It is straightforward to see that this run represents exactly the leftmost derivation $S \Rightarrow^*_{\mathcal{G}} \hat{w}$, and that for each annotated string $\hat{w} \in L(\mathcal{G})$ if and only if there is a run of $\mathcal{P}$ over w that produces $\mu = \mathsf{ann}(\hat{w})$ as output.

This function is also reversible. Consider a run of $\mathcal{P}$ over a string w which produces μ as output. This run must start on q_0, and then push q_0 and move onto a state $(S, \alpha, 0)$ for some rule $S \to \alpha$. Thus, our first two strings in the sequence are $\downarrow S \uparrow$ and $\downarrow \alpha \uparrow\uparrow$. If the current state is (X, α, k) and the next transition is to push $(X, \alpha, k+1)$ onto the stack to move into the state $(Y, \gamma, 0)$, then the current string is of the form $\hat{u} \downarrow X\beta$, so we follow it by the string $\hat{u} \downarrow \gamma \uparrow \beta$. If the next transition is a pop, then the current string is $\hat{u} \downarrow\uparrow \beta$, so we follow it by $\hat{u} \downarrow \beta$. If the current transition is a read, then the current string is $\hat{u} \downarrow a\beta$ for $a \in \Sigma$, so we follow it by $\hat{u}\tau \downarrow \beta$. If the current transition is a read-write, then the current string is $\hat{u} \downarrow (a, \vartheta)\beta$ for $(a, \vartheta) \in \Sigma \times \Omega$, so we follow it by $\hat{u}(a, \vartheta) \downarrow \beta$. It can easily be seen that using the original function over this resulting sequence would give the original sequence back. We point that these two reversible functions mean that there is a one to one correspondence between derivations of $S \Rightarrow^*_{\mathcal{G}} \hat{w}$ and accepting runs of $\mathcal{P}$ over w with output $\mu = \mathsf{ann}(\hat{w})$.

Similarly to the observation we made before, we notice that if we start on a sequence in the intermediate model, the profile of the resulting run ρ is fully given by the shape of the sequence (at each step, the size of the stack will be equal to the number of markers $\uparrow$ present in the string).

Now assume that $\mathcal{G}$ is unambiguous. Seeing that $\mathcal{P}$ is unambiguous as well comes straightforwardly from the fact that the functions presented above are bijective.

Assume now that $\mathcal{G}$ is rigid. Let w be an unannotated string and consider two runs ρ_1 and ρ_2 of $\mathcal{P}$ over w which output μ_1 and μ_2 respectively. Convert these two runs into sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ in the intermediate model. Note that if we convert these two sequences into derivations $S \Rightarrow^*_{\mathcal{G}} \mu_i(w)$, they will have the same shape. We can apply the functions above to obtain the two runs ρ_1 and ρ_2 back, and note that they have the same profile. We conclude that if $\mathcal{G}$ is rigid, then $\mathcal{P}$ is profiled. □

We then show another direction:

CLAIM 4.4. *For any PDAnn $\mathcal{P}$, we can build an equivalent annotated grammar $\mathcal{G}$ in polynomial time. Further, if $\mathcal{P}$ is unambiguous then so is $\mathcal{G}$. Moreover, if $\mathcal{P}$ is profiled, then $\mathcal{G}$ is rigid.*

PROOF. This is again a standard transformation. We first transform the input PDAnn $\mathcal{P} = (Q, \Sigma, \Omega, \Gamma, \Delta, q_0, F)$ to accept by empty stack, i.e., to accept iff the stack is empty. To do this, we build an equivalent PDAnn $\mathcal{P}' = (Q', \Sigma, \Omega, \Gamma', \Delta', q_0', F')$ where $Q' = Q \cup \{q_0', q_e, q_f\}$, $\Gamma' = \Gamma \cup \{\gamma_0\}$, $F = \{q_f\}$, and we add the following transitions to Δ to obtain Δ': A push transition (q_0', q_0, γ_0), a pop transition (q, γ_0, q_f) for every $q \in F$ (for runs that accept at a point where the stack is already empty), plus a pop transition (q, γ, q_e) for any other $\gamma \in \Gamma$, and a pop transition (q_e, q_0, q_f).

This clearly ensures that there is a bijection between the accepting runs of $\mathcal{P}$ and those of $\mathcal{P}'$: given an accepting run ρ of $\mathcal{P}$, the bijection maps it to an accepting run of $\mathcal{P}'$ by extending it with a push transition at the beginning, and pop transitions at the end. Further, all accepting runs in $\mathcal{P}'$ now finish with an empty stack, more specifically a run is accepting iff it finishes with an empty stack.

Now, we can perform the transformation. The nonterminals of the grammar are triples of the form (q, γ, q') for states q and q' and a stack symbol γ. Intuitively, (p, γ, q') will derive the strings that can be read by the PDAnn starting from state p, reaching some other state q with the same stack, not seeing the stack at all in the process, and then popping γ to reach q'.

The production rules are the following:

- A rule $S \to (q_0, \gamma_0, q_f)$.
- A rule $(p, \gamma, q') \to (q, \gamma', r)(r, \gamma, q')$ for every nonterminal (p, γ, q'), push transition $(p, q, \gamma') \in \Delta$ and state r.
- A rule $(p, \gamma, q) \to \varepsilon$ for each pop transition $(p, \gamma, q) \in \Delta$.
- A rule $(p, \gamma, q') \to (a, \eth)(q, \gamma, q')$ for each read-write transition $(p, (a, \eth), q)$, and a rule $(p, \gamma, q') \to a(q, \gamma, q')$ for each read transition (p, a, q), for each nonterminal (p, γ, q').

As we did in Proposition 4.3, we will show a function which receives an accepting run ρ over w in $\mathcal{P}$ with output μ and outputs a leftmost derivation $S = \alpha_1 \Rightarrow_{\mathcal{G}} \alpha_2 \Rightarrow_{\mathcal{G}} \ldots \Rightarrow_{\mathcal{G}} \alpha_m = \mu(w)$. The way we do this is quite straightforward: There is a one-to-one correspondence between snapshots in the run to each α_i. Indeed, it can be seen that $\alpha_i = \hat{u}(q_1, \gamma_1, q_2)(q_2, \gamma_2, q_3) \ldots (q_k, \gamma_k, q_f)$ for some string $\hat{u} \in (\Sigma \cup (\Sigma \times \Omega))^*$, some states $q_1, \ldots, q_k$ and stack symbols $\gamma_1, \ldots, \gamma_k$. Moreover, the i-th stack in the run is equal to $\gamma_1 \gamma_2 \ldots \gamma_k$, whereas each state q_j is the first state that is reached after popping the respective γ_{j-1}. We see that this function is fully reversible, as each production corresponds unequivocally to a transition in particular. This implies that $\mathcal{P}$ is unambiguous if, and only if $\mathcal{G}$ is unambiguous.

For the next part of the proof, we bring attention to the fact that there are exactly four possible shapes on the right sides of the rules in $\mathcal{G}$. Each of these directly map to some type of transition, be it the initial push transition (q_0', q_0, γ_0), a different push transition, a pop transition, or a read (or read-write) transition. To be precise, these shapes are the strings 1, 11, ε and 01 respectively. From here it can be easily seen that, while comparing a run ρ to is respective derivation $S \Rightarrow_{\mathcal{G}}^{*} \hat{w}$, each production in the run immediately tells which type of transition was taken, and each transition in the run immediately tells which rule (and therefore, rule shape) was used. Therefore, each derivation shape maps to exactly one stack profile and vice versa, from which we conclude that $\mathcal{P}$ is profiled if, and only if, $\mathcal{G}$ is rigid. □

The proof now follows from Claims 4.3 and 4.4, □

Let us now study the enumeration for PDAnns. We know that the problem for unambiguous PDAnns can be solved via Proposition 4.7 with cubic-time preprocessing in data complexity and output-linear delay (with Theorem 4.2). We know that profiled PDAnns can be made unambiguous (via Proposition 4.7 and Theorem 4.3) and so that we can solve enumeration for them in quadratic-time preprocessing in data complexity and output-linear delay (using Theorem 4.4). We now show that, if we are given a profile of an unambiguous PDAnn $\mathcal{P}$ on an input string w, we can use it as a guide to enumerate with linear preprocessing and output-linear delay the set $[\![\mathcal{P}]\!]_{\pi}(w)$ of annotations for that profile, i.e., all $\mathsf{ann}(\rho)$ such that ρ is an accepting run with profile π of $\mathcal{P}$ over w. Formally:

Lemma 4.2. *Given an unambiguous PDAnn $\mathcal{P}$, there exists an enumeration algorithm that receives as input a string w and a profile π of $\mathcal{P}$ over w, and enumerates $[\![\mathcal{P}]\!]_{\pi}(w)$ with output-linear delay after linear-time preprocessing in data complexity.*

PROOF. We will show a linear-time reduction to enumeration for an unambiguous VPAnn (see Chapter 3).

The theorem we use can be stated as follows:

Theorem 4.5. *(Theorem 3.1) There is an algorithm that receives an unambiguous VPAnn $\mathcal{T} = (Q, \hat{\Sigma}, \Gamma, \Omega, \Delta, q_0, F)$ and an input string w, and enumerates the set $[\![\mathcal{T}]\!](w)$ with output-linear delay after a preprocessing phase that takes $\mathcal{O}(|Q|^2 \cdot |\Delta| \cdot |w|)$ time.*

The rest of the proof will consist on showing a linear-time reduction from the problem of enumerating the set $[\![\mathcal{P}]\!]_\pi(w)$ for an unambiguous PDAnn $\mathcal{P}$ and input string w to the problem of enumerating the set $[\![\mathcal{T}]\!](w')$ for an unambiguous VPAnn $\mathcal{T}$, and input string w'.

Let $\mathcal{P} = (Q, \Sigma, \Omega, \Gamma, \Delta, q_0, F)$ and let $w \in \Sigma^*$ be an input string. Consider the structured alphabet $\hat{\Sigma} = (\{<\}, \{>\}, \Sigma)$ for some $<, > \notin \Sigma$. Assume $\pi = \pi_1, \ldots, \pi_m$. We construct a well-nested string $w' = b_1 \cdots b_{m-1}$ where $b_i = <$ if $\pi_i > \pi_{i+1}$, $b_i = <$ if $\pi_i < \pi_{i+1}$, and $b_i = w_j$ otherwise, where i is the j-th index in which $\pi_i = \pi_{i+1}$. We also build a table Ind such that $\mathsf{Ind}(i) = j$ for each of the indices in the third case. We build a VPAnn $\mathcal{T} = (Q, \hat{\Sigma}, \Gamma, \Omega, \Delta', I, F)$ where $I = \{q_0\}$ and we get Δ' by replacing every push transition $(p, q, \gamma) \in \Delta$ by $(p, <, q, \gamma)$ and every pop transition $(p, \gamma, q) \in \Delta$ by $(p, >, q, \gamma)$. Note that read and read-write transitions are untouched.

Let w be an input string, and let μ be an output. Consider the output μ' which is obtained by shifting the indices in μ to those that correspond in w'. We argue that for each run ρ of $\mathcal{P}$ over w with profile π which produces μ, there is exactly one run ρ' of $\mathcal{T}$ over w' which produces μ', and vice versa. We see this by a straightforward induction argument on the size of the run. This immediately implies that for each output $\mu \in [\![\mathcal{T}]\!](w)$ there exists exactly one output $\mu' \in [\![\mathcal{P}]\!]_\pi(w)$, which has its indices shifted as we mentioned. The algorithm then consists on simulating the procedure from Theorem 4.5 over $\mathcal{T}$ and w', and before producing an output μ, we replace the indices to the correct ones following the table Ind. The time bounds are unchanged since the table Ind has linear size in m, and replacing the index on some output μ can be done linearly on $|\mu|$. We conclude that there is algorithm that enumerates the set $[\![\mathcal{P}]\!]_\pi(w)$ with output-linear delay after a preprocessing that takes $\mathcal{O}(|Q|^2 \cdot |\Delta| \cdot |\pi|)$ time. □

This result implies that we could achieve linear-time enumeration over profiled PDAnn if we could easily discover their (unique) profile. We achieve this in *deterministically profiled PDAnns*.

Deterministically profiled PDAnn. Let $\mathcal{P} = (Q, \Sigma, \Omega, \Gamma, \Delta, q_0, F)$ be a PDAnn. We say that a PDAnn $\mathcal{P}$ is *deterministically profiled* if, for any string $w \in \Sigma^*$, for any two partial runs ρ and ρ' of $\mathcal{P}$ over w with the same length, ρ and ρ' have the same profile.

The relationship between deterministically profiled PDAnns and deterministic pushdown automata is similar to the relationship between profiled PDAnns and unambiguous pushdown automata (the latter relationship was stated as Proposition 4.4 in the context of grammars).

Specifically:

PROPOSITION 4.8. *For a deterministically profiled PDAnn $\mathcal{P}$, let L' be the set of strings with nonempty output, i.e., $L' = \{w \mid [\![\mathcal{P}]\!](w) \neq \emptyset\}$. Then L' is recognized by a deterministic pushdown automaton.*

PROOF. For this result, we use the notion of PDA (Definition 4.2) and deterministic PDA (Definition 4.3) that were presented inside a previous proof.

As we have done in previous proofs, the strategy consists on starting with a profiled-deterministic PDAnn $\mathcal{P}$, and building a PDAnn $\mathcal{P}'$ by eliminating the output symbols from each transition. This PDAnn behaves almost identically to a pushdown automaton $\mathcal{A}$ in the sense that if $w \in L(\mathcal{A})$, then $[\![\mathcal{P}]\!](w) = \{\varepsilon\}$, and that if $w \notin L(\mathcal{A})$ then $[\![\mathcal{P}]\!](w) = \emptyset$. Whenever this holds, we say that the PDAnn and the pushdown automaton are *equivalent*. It is simple to see that for this $\mathcal{A}$ it holds that $L(\mathcal{A}) = L'$. To conclude the proof, we must show that $\mathcal{A}$ can be made deterministic. Without loss of generality, we remove from $\mathcal{A}$ all *inaccessible states*, i.e., all states for which there is no run that goes to the state.

First, we will prove that the PDAnn $\mathcal{P}'$ is profiled-deterministic. Let w be a string in Σ^* and let ρ_1' and ρ_2' be two partial runs of $\mathcal{P}'$ over w with the same profile, and with last configurations (q, i) and (q', i). There clearly exist partial runs ρ_1 and ρ_2 of $\mathcal{P}$ over w with the same profile, which can be obtained by replacing each transition by one of the transitions in $\mathcal{P}$ it was replaced by. Since $\mathcal{P}$ is profile-deterministic, then one of the following must hold in $\mathcal{P}$: (1) $\Delta(q) \cup \Delta(q') \subseteq Q \times (\Sigma \cup \Sigma \times \Omega) \times Q$, i.e., all transitions from q and q' are read or read-write transitions; (2) $\Delta(q) \cup \Delta(q') \subseteq Q \times (Q \times \Gamma)$, i.e., all transitions from q and q' are push transitions; or (3) $\Delta(q) \cup \Delta(q') \subseteq (Q \times \Gamma) \times Q$, i.e., all transitions from q and q' are pop transitions. Note that if (2) or (3) hold, then in the new PDAnn $\mathcal{P}$ the condition holds again in $\mathcal{P}'$ trivially since none of the transitions in $\Delta(q)$ and $\Delta(q')$ was changed. Moreover, if (1) holds, then it can be seen that all of the transitions that belonged in $Q \times (\Sigma \times \Omega) \times Q$ now belong in $Q \times \Sigma \times Q$, which also leaves the condition unchanged in $\mathcal{P}'$. We conclude that $\mathcal{P}'$ is profiled-deterministic.

The next step is to use Lemma 4.1 from $\mathcal{P}'$ to obtain an equivalent PDAnn $\mathcal{P}''$ which is deterministic-modulo-profile. We will argue that if we start with $\mathcal{P}'$, which was profiled-deterministic, then the resulting $\mathcal{P}''$ is equivalent to a pushdown automaton $\mathcal{A}'$ which is also deterministic. Let w be an input string in Σ^* and let ρ'' be a partial run of $\mathcal{A}$ over w with last configuration (S, i) and with topmost symbol on the stack T. Let us recall what $\mathcal{P}''$ being deterministic-modulo-profile entails that the following conditions hold:

(i) There is at most one push transition that starts on S; formally, we have:

$$|\{S', T \in Q'' \times \Gamma'' \mid (S, S', T) \in \Delta\}| \leq 1.$$

(ii) There is at most one pop transition that starts on S, T; formally, for each γ, we have:

$$|\{S' \in Q \mid (S, \gamma, S') \in \Delta\}| \leq 1.$$

(iii) For each letter a, and output $\vartheta \in \Omega$, there is at most one read-write transition that starts on S, a, ϑ; formally, we have

$$|\{S' \in Q'' \mid (S, (a, \vartheta), S') \in \Delta''\}| \leq 1.$$

(iv) For each letter a, there is at most one read transition that starts on S, a; formally, we have:

$$|\{q \in Q'' \mid (S, a, S') \in \Delta''\}| \leq 1.$$

We will show that at most one of these conditions holds. Recall that in the transformation, the states of $\mathcal{P}''$ are sets which contain pairs of states $(p, q) \in Q' \times Q'$, and the stack symbols are triples $(p, \gamma, q) \in Q' \times \Gamma' \times Q'$. Now, recall the claim that was proven in the lemma, on the backwards direction:

If $\mathcal{P}''$ has a run ρ'' on a string w, producing output μ, from its initial state to an instantaneous description $(S, i), \alpha'$ with $\alpha' = T_0, \ldots, T_m$ being the sequence of the stack symbols, then for any choice of elements $(q_0, \gamma_0, p_0) \in T_0$, $(p_0, \gamma_1, p_1) \in T_1$, ..., $(p_{m-1}, \gamma_m, p_m) \in T_m$ and $(p_m, q) \in S$ it holds that $\mathcal{P}'$ has a run ρ' on w producing output μ from some initial state q_0 to the instantaneous description $(q, i), \alpha$ with $\alpha = \gamma_0 q_0, \ldots, \gamma_m q_m$ (writing next to each stack symbol the state that annotates it), and ρ'' and ρ' have the same profile.

Since $\mathcal{P}'$ is profiled-deterministic, then each run ρ'^{+} which continues ρ' by one step must have the same shape. This implies that exactly one of the following conditions must hold:

- The last transition in ρ'^{+} is a read or read-write transition. Therefore, all transitions from q are either read or read-write transitions.
- The last transition in ρ'^{+} is a push transition. Therefore, all transitions from q are pop transitions.
- The last transition in ρ'^{+} is a pop transition. Therefore, all transitions from q are pop transitions.

Assume bullet point 1 holds. Note that there are no read-write transitions in $\mathcal{P}'$ so there are only read transitions. From here, we prove that only (4) is true simply by inspecting the transformation in the lemma; if (1) held, then there would be a push transition from q in $\mathcal{P}'$, if (2) held then there would be a pop transition from q and γ in $\mathcal{P}'$, and (3) never holds. Now, assume bullet point 2 holds. From here, we prove that only (1) can be true; if (2) held, then there would be a pop transition from q and γ in $\mathcal{P}'$, if (4) held, then there would be a read transition from q in $\mathcal{P}'$, and again, (3) is never true. Lastly, assume bullet point 3 holds. From here, we prove that only (2) can be true; if (1) held, then there would be a push transition from q in $\mathcal{P}'$, if (4) held, then there would be a read transition from q in $\mathcal{P}'$, and yet again, (3) is never true. We conclude that from the 4 points, at most one of these can be true at the same time.

Now we prove that the equivalent PDA $\mathcal{A}$ is deterministic. Let q be a state of $\mathcal{A}$. As all states of $\mathcal{A}$ are accessible, pick ρ'' to be a run that reaches state q. We have argued that at most one of the points in the list above is true of $\mathcal{P}'$, and it cannot be point (3). Now, we see that (a) is equivalent to (4), that (b) is equivalent to (1) and (c) is equivalent to (2). Since only one of the conditions among (1), (2) or (4) can be true, the same holds for (a), (b) and (c), from which we conclude that $\mathcal{A}$ is deterministic. This completes the proof. □

This result gives a concrete picture of the expressive power of deterministically profiled PDAnn $\mathcal{A}$, i.e., as acceptors they are more powerful than the class of *visibly pushdown automata* (Alur & Madhusudan, 2004a), where each alphabet letter must have a specific effect on the profile. Deterministically profiled PDAnn are also reminiscent of the *height-determinism* notion introduced for pushdown automata (Nowotka & Srba, 2007), but extend this with the support of annotations.

Deterministically profiled PDAnn are designed to ensure that they have only one profile (i.e., they are profiled), and further that their unique profile can be constructed in linear time:

PROPOSITION 4.9. *A deterministically profiled PDAnn $\mathcal{P}$ is always profiled, and given a string w, the unique profile of accepting runs of $\mathcal{P}$ over w can be computed in linear time in w.*

PROOF. Consider a deterministically profiled PDAnn $\mathcal{P}$. To prove that it is profiled, consider an input string $w \in \Sigma^*$. We will prove by a simple induction argument that any two runs of $\mathcal{P}$ over w have the same profile. The base case is trivial since the run is of length 0, and the profile up to now is composed simply of the stack size 0. Assume now that for each pair of runs ρ and ρ' of $\mathcal{P}$ over w of size k, that they have the same profile. We will show that for every pair of runs ρ_1 and ρ_2 over w of size $k+1$, they have the same profile as well. Note that the runs ρ_1^- and ρ_2^- that are obtained by removing the last step have the same profile, by the hypothesis. From the definition of deterministically profiled it can be directly seen that if (1) the last transition in ρ_1 is a read or read-write transition, then for the runs ρ_1^- and ρ_2^-, the only choices are read or read-write transitions, from which we deduce that the last transition in ρ_2 is a read or read-write transition as well, if (2) the last transition in ρ_1 is a push transition, then for the run ρ_1^- and ρ_2^- the only choice are push transitions, and therefore the last transition in ρ_2 has to be a push transition as well, and if (3) the last transition in ρ_1 is a pop transition, then for ρ_1^- and ρ_2^- the only choices are pop transitions, so the last transition in ρ_2 must be a pop transition as well. We obtain that ρ_1 and ρ_2 have the same profile, and from the induction argument, we conclude that $\mathcal{P}$ is profiled.

Now, consider a deterministically profiled PDAnn $\mathcal{P}$ and an input string w. We will prove that the unique profile of accepting runs of $\mathcal{P}$ over w can be computed in linear time in $|w|$. The way we do this is by using the pushdown automaton $\mathcal{A}$ that was constructed in Proposition 4.8. By inspecting the proof, it can be seen that the unique profile of $\mathcal{P}$ over w is maintained throughout the construction. Indeed, the first construction simply removes the output symbols, which does not affect the profile, and the second construction has an invariant that keeps the profile intact as well. Therefore, by running the automaton $\mathcal{A}$ over w, and storing the stack sizes at each step, we obtain a profile π which is exactly the same profile of the accepting runs of $\mathcal{P}$ over w. To finish the proof, we only need to argue that this profile has linear size on $|w|$ (from a data complexity perspective). This follows from the fact that any run of a deterministic pushdown automaton $\mathcal{A}$ over a string w has $\mathcal{O}(f(\mathcal{A}) \times |w|)$ length, for some computable function f. This can be seen from a counting argument: (1) There is a maximum stack size k that can be reached in an accepting run of $\mathcal{P}$ over w from an empty stack through ε-transitions, which is given by the number of states in $\mathcal{A}$. Otherwise, there are two configurations which are reachable from one another in a way such the stack, as it was at the first configuration, is not seen. This implies that there is a loop, and since $\mathcal{A}$ is deterministic, $\mathcal{A}$ does not accept w. (2) From a given stack, the maximum numbers of steps that can be taken without reading from w, and without seeing the topmost symbol on the stack is given by the number of possible stacks of size k. (3) Between a read (or read-write) transition and the next one, the maximum height difference is k, and if we move out of a read (or read-write) transition with a certain stack, from (2) we can see that we can only do a fixed number of steps before consuming some symbol from this stack, and therefore, the number of steps is bounded by a factor depending on $\mathcal{A}$ multiplied by the size of the stack up until this point, which is linear on the number of symbols in w read so far. We conclude that w' has size linear on w, from a data complexity point of view. □

Together with Lemma 4.2, this yields:

Corollary 4.1. *Let $\mathcal{P}$ be a deterministically profiled PDAnn. Then for every string w the set $[\![\mathcal{P}]\!](w)$ can be enumerated with output-linear delay after linear-time preprocessing in data complexity.*

4.5. Application: Document Spanners

We have presented our enumeration results for annotated grammars and pushdown annotators. We conclude the chapter by applying them to the standard context of *document spanners* (Fagin et al., 2015) and to the *extraction grammars* recently introduced in (Peterfreund, 2023). We omit a formal introduction of extraction grammars as they were defined in Section 3.7.

It can be noted that extraction grammars are like annotated grammars but with variable operations that describe span endpoints (whereas our annotations are arbitrary), and that are expressed as separate variable operation characters (not annotations of existing letters).

We can now formally define the *equivalence* between an extraction grammar and an annotated grammar. To do so, we first explain how we can translate mappings to annotations:

Definition 4.4 (Output associated to a mapping)**.** *Given a set $\mathcal{X}$ of variables, the corresponding set of annotations $\Omega_{\mathcal{X}}$ will be the powerset of $\mathsf{C}_{\mathcal{X}}$. Now, given a mapping η on a document d and variables $\mathcal{X}$ to an annotation, we let $\mathcal{I} = \bigcup_{x \in \mathcal{X}} \{i, j \mid \eta(x) = [i, j\rangle\}$ be the set of indices which appear in some span of η. Further, for each $k \in \mathcal{I}$, let $S_k = \{\vdash_x \mid \exists j.\, \eta(x) = [k, j\rangle\} \cup \{\dashv_x \mid \exists i.\, \eta(x) = [i, k\rangle\}$. We now define the output $\mathsf{out}(\eta) = (i_1, S_{i_1}) \dots (i_m, S_{i_m})$ where $\mathcal{I} = \{i_1, \dots, i_m\}$ and $i_1 < \cdots < i_m$, namely, we group the captures for each position as a set and use this set as the annotation. Note that the largest index that appears in the annotation can be $|d| + 1$ because of the range of spans.*

Comparing both formalisms. Given an extraction grammar $\mathcal{H}$ on alphabet Σ and with variables $\mathcal{X}$, we say that it has an *equivalent annotated grammar* $\mathcal{G}$ if $\mathcal{G}$ is over the set of annotations $\Omega_{\mathcal{X}}$ and over the alphabet $\Sigma \cup \{\#\}$ for $\#$ a fresh symbol, and if for every document $d \in \Sigma^*$ and every mapping η of d over $\mathcal{X}$, we have $\eta \in [\![\mathcal{H}]\!](d)$ iff $\mathsf{out}(\eta) \in [\![\mathcal{G}]\!](d \cdot \#)$. The $\#$-symbol at the end is used because of the difference in the indexing of spans (from 1 to $|d| + 1$) and annotations (from 1 to d).

We show that every extraction grammar has an equivalent annotation grammar in this sense, and the translation further preserves unambiguity:

PROPOSITION 4.10. *Given any extraction grammar $\mathcal{H}$ with k variables, we can build an equivalent annotated grammar $\mathcal{G}$ in time $\mathcal{O}(9^{3k} \cdot |\mathcal{H}|^2)$. Moreover, if $\mathcal{H}$ is unambiguous then so is $\mathcal{G}$.*

PROOF. Recall that, in the statement of this result, the formal notion of an *equivalent annotated grammar* is the one defined above. Recall also the formal definition of *ref-words* (see Section 2.2).

Let r be a ref-word in $\Sigma \cup \mathsf{C}_{\mathcal{X}}$ and let $\hat{w}$ be an annotated string in $\Sigma \cup (\Sigma \times \Omega_{\mathcal{X}})$. We say that r and $\hat{w}$ are *equivalent* if $\mathsf{plain}(r \cdot \#) = \mathsf{str}(\hat{w})$ and $\mathsf{out}(\eta^r) = \mathsf{ann}(\hat{w})$. For example, the ref-word $r_1 = \vdash_x \mathtt{a}\mathtt{a} \dashv_x \vdash_y \mathtt{b}\mathtt{b} \dashv_y \mathtt{b}$ is equivalent to $\hat{w}_1 = (\mathtt{a}, \{\vdash_x\})\, \mathtt{a}\, (\mathtt{b}, \{\dashv_x, \vdash_y\})\, \mathtt{b}\, \mathtt{b}\, (\mathtt{b}, \{\dashv_y\})\, \#$.

The overall strategy of this proof is going to be to construct an annotated grammar $\mathcal{G}$ in a way such that for every ref-word $r \in L(\mathcal{H})$ there exists an equivalent annotated string $\hat{w} \in L(\mathcal{G})$, and vice versa. It is clear that this implies that $\mathcal{G}$ and $\mathcal{H}$ are equivalent.

The way we build $\mathcal{G}$ will look like we are "pushing" the variable operations to the next terminal to the right. We will do this process one variable operation at a time.

First, we need to define an intermediate model between those of extraction grammars and annotated grammars. We define *extraction grammars with annotations* as a straightforward extension of extraction grammars which allow annotations on terminals that are not variable operations. For the set of variables $\mathcal{X}$, recall from Section 2.2 that we define the variable operations of $\mathcal{X}$ by $\mathsf{C}_{\mathcal{X}} = \{\vdash_x, \dashv_x \mid x \in \mathcal{X}\}$, and recall from Definition 4.4 that we define $\Omega_{\mathcal{X}} = 2^{\mathsf{C}_{\mathcal{X}}}$. An *extraction grammar with annotations* is a tuple $\mathcal{F} = (V, \Sigma, \mathcal{X}, P, S)$ where V is a finite set of nonterminal symbols, Σ is an alphabet and $\mathcal{X}$ is a set of variables, such that V, Σ, $\Sigma \times \Omega_{\mathcal{X}}$, and $\mathsf{C}_{\mathcal{X}}$ are pairwise disjoint, P is a finite set of rules of the form $A \to \alpha$ with $A \in V$ and $\alpha \in (V \cup \Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}) \cup \mathsf{C}_{\mathcal{X}})^*$, and $S \in V$ is the start symbol. As in the other models, the semantic of extraction grammars is defined through derivations. Specifically, the set P defines the (left) derivation relation

$\Rightarrow_{\mathcal{F}} \subseteq (V \cup \Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}) \cup \mathsf{C}_{\mathcal{X}})^* \times (V \cup \Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}) \cup \mathsf{C}_{\mathcal{X}})^*$ such that $\hat{u}A\beta \Rightarrow_{\mathcal{F}} \hat{u}\alpha\beta$ iff $\hat{u} \in (\Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}) \cup \mathsf{C}_{\mathcal{X}})^*$, $A \in V$, $\alpha, \beta \in (V \cup \Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}) \cup \mathsf{C}_{\mathcal{X}})^*$, and $A \to \alpha \in P$. We denote by $\Rightarrow_{\mathcal{F}}^*$ the reflexive and transitive closure of $\Rightarrow_{\mathcal{F}}$. Then the language defined by $\mathcal{F}$ is $L(\mathcal{F}) = \{\hat{w} \in (\Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}) \cup \mathsf{C}_{\mathcal{X}})^* \mid S \Rightarrow_{\mathcal{F}}^* \hat{w}\}$. In addition, we assume that no string $\hat{w}$ in $L(\mathcal{F})$ has a variable operation as its last symbol.

An extraction grammar with annotations generates strings over $\Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}) \cup \mathsf{C}_{\mathcal{X}}$, which we now refer to as *annotated ref-words*, and each annotated ref-word defines an output. We will define the semantics of extraction grammars with annotations recursively by using the semantics of annotated grammars as a starting point, that is, by extending the function ann to receive strings over $\Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}) \cup \mathsf{C}_{\mathcal{X}}$. In particular, for an annotated ref-word $\hat{r} \in (\Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}))^*$ the result of $\mathsf{ann}(\hat{r})$ stays the same. For a string $\hat{r} = \hat{u}\kappa a\hat{v} \in (\Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}) \cup \mathsf{C}_{\mathcal{X}})^*$ where $\kappa \in \mathsf{C}_{\mathcal{X}}$ and $a \in \Sigma$, we define $\mathsf{ann}(\hat{r}) = \mathsf{ann}(\hat{u}(a, \{\kappa\})\hat{v})$, and for a string $\hat{r} = \hat{u}\kappa(a, \eth)\hat{v} \in (\Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}) \cup \mathsf{C}_{\mathcal{X}})^*$ where $\eth \in \Omega_{\mathcal{X}}$, we define $\mathsf{ann}(\hat{r}) = \mathsf{ann}(\hat{u}(a, \eth \cup \{\kappa\})\hat{v})$.

Further, we extend the function str to receive strings over $\Sigma \cup (\Sigma \times \Omega_{\mathcal{X}}) \cup \mathsf{C}_{\mathcal{X}}$ as $\mathsf{str}(\hat{r}) = \mathsf{str}(\mathsf{plain}(\hat{r}))$. Therefore, for an extraction grammar with annotations $\mathcal{F}$ and a string $w \in \Sigma^*$ we define the function $[\![\mathcal{F}]\!]$ as: $[\![\mathcal{F}]\!](w) := \{\mathsf{ann}(\hat{r}) \mid \hat{r} \in L(\mathcal{F}) \wedge \mathsf{str}(\hat{r}) = w\}$. We maintain the notions of equivalency between annotated grammars, extraction grammars, and extraction grammars with annotations. Likewise, we define equality between annotated strings, ref-words and annotated ref-words in the obvious way. Further, we note that any extraction grammar $\mathcal{H} = (V, \Sigma, \mathcal{X}, P, S)$ is equivalent to the extraction grammar with annotations $\mathcal{F} = (V', \Sigma \cup \{\#\}, \mathcal{X}, P', S')$ where $V' = V \cup \{S'\}$ for some $S' \notin V$, and $P' = P \cup \{S' \to S\#\}$. It is obvious that $\mathcal{F}$ is unambiguous if and only if $\mathcal{H}$ is unambiguous.

We proceed as follows. First we convert $\mathcal{H}$ into a functional extraction grammar. As detailed in Peterfreund's work (Peterfreund, 2023, Propositions 10 and 12), this takes running time $\mathcal{O}(3^{2k}|\mathcal{H}|^2)$, and if the initial grammar is unambiguous then so is the resulting grammar. Note that this implies that every ref-word r which is derivable from S contains each variable operation at most once. Hence we can build an equivalent extraction grammar with annotations $\mathcal{F} = (V, \Sigma, \mathcal{X}, P, S)$ using the technique above. We then convert $\mathcal{F}$ into a version of CNF which is slightly more restrictive than arity-two normal form: We allow rules of the form $X \to YZ$, $X \to \varepsilon$ and $X \to \tau$, for nonterminals X, Y and Z and a terminal τ, but rules of the form $X \to Y$ are not permitted. Converting to this formalism can be done in linear time in $|\mathcal{F}|$ while preserving unambiguity, e.g., by transforming rules of the form $X \to Y$ to $X \to EY$ for some fresh nonterminal E with a rule $E \to \varepsilon$, and otherwise applying our result on arity-2 normal form (Proposition 4.1). We pick an order over the variable operations in $\mathsf{C}_{\mathcal{X}}$ and for each $\kappa \in \mathsf{C}_{\mathcal{X}}$ we do the following:

Define a function proc_κ that receives an annotated ref-word $\hat{r} \in (\Sigma \cup (\Sigma \times 2^{\mathsf{C}_{\mathcal{X}}}) \cup \mathsf{C}_{\mathcal{X}})^*$ and:

(i) if $\hat{r} \in (\Sigma \cup (\Sigma \times 2^{\mathsf{C}_{\mathcal{X}}}) \cup (\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\}))^*$, then $\mathsf{proc}_\kappa(\hat{r}) = \hat{r}$,

(ii) if $\hat{r} = \hat{u}\kappa\beta a\hat{v}$ for some $\hat{u}, \hat{v} \in (\Sigma \cup (\Sigma \times 2^{\mathsf{C}_{\mathcal{X}}}) \cup (\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\}))^*$, $\beta \in (\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\})^*$ and $a \in \Sigma$, then $\mathsf{proc}_\kappa(\hat{r}) = \hat{u}\kappa\beta(a, \{\kappa\})\hat{v}$,

(iii) if $\hat{r} = \hat{u}\kappa\beta(a, T)\hat{v}$, with $T \subseteq \mathsf{C}_{\mathcal{X}} \setminus \{\kappa\}$, then $\mathsf{proc}_\kappa(\hat{r}) = \hat{u}\kappa\beta(a, T \cup \{\kappa\})\hat{v}$, and

(iv) proc_κ is undefined in any other case.

It is straightforward to see that the annotated ref-words $\hat{r}$ and $\hat{t} = \mathsf{proc}_\kappa(\hat{r})$ are equivalent whenever $\mathsf{proc}_\kappa(\hat{r})$ is defined.

We build an extraction grammar with annotations $\mathcal{F}' = (V', \Sigma, \mathcal{X}, P', S')$ where $V' = V_{\mathsf{out}} \cup V_{\mathsf{in}} \cup V_{\mathsf{left}} \cup V_{\mathsf{mid}} \cup V_{\mathsf{right}} \cup \{S'\}$, and $V_{\mathsf{scr}} = \{A^{\mathsf{scr}} \mid A \in P\}$ for $\mathsf{scr} \in \{\mathsf{out}, \mathsf{in}, \mathsf{left}, \mathsf{mid}, \mathsf{right}\}$, and P' is defined by the following rules:

- For each rule $S \to AB$ in P, we add the rules $S' \to A^{\mathsf{out}}B^{\mathsf{out}}$, $S' \to A^{\mathsf{in}}B^{\mathsf{out}}$ and $S' \to A^{\mathsf{left}}B^{\mathsf{right}}$ to P'.
- For each rule $A \to BC$, we add the rules $A^{\mathsf{out}} \to B^{\mathsf{out}}C^{\mathsf{out}}$, $A^{\mathsf{in}} \to B^{\mathsf{in}}C^{\mathsf{out}}$, $A^{\mathsf{in}} \to B^{\mathsf{out}}C^{\mathsf{in}}$, $A^{\mathsf{in}} \to B^{\mathsf{left}}C^{\mathsf{right}}$, $A^{\mathsf{left}} \to B^{\mathsf{out}}C^{\mathsf{left}}$, $A^{\mathsf{left}} \to B^{\mathsf{left}}C^{\mathsf{mid}}$, $A^{\mathsf{mid}} \to B^{\mathsf{mid}}C^{\mathsf{mid}}$, $A^{\mathsf{right}} \to B^{\mathsf{right}}C^{\mathsf{out}}$ and $A^{\mathsf{right}} \to B^{\mathsf{mid}}C^{\mathsf{right}}$ to P'.

- For each rule $A \to a$, $a \in \Sigma$, we add $A^{\mathsf{out}} \to a$ and $A^{\mathsf{right}} \to (a, \{\kappa\})$.
- For each rule $A \to (a, T)$, $a \in \Sigma$ and $T \subseteq \mathsf{C}_{\mathcal{X}} \setminus \{\kappa\}$, we add $A^{\mathsf{out}} \to (a, T)$ and $A^{\mathsf{right}} \to (a, T \cup \{\kappa\})$.
- For each rule $A \to \kappa'$, $\kappa' \in \mathsf{C}_{\mathcal{X}} \setminus \{\kappa\}$, we add the rules $A^{\mathsf{out}} \to \kappa'$ and $A^{\mathsf{mid}} \to \kappa'$.
- For each rule $A \to \kappa$, we add the rule $A^{\mathsf{left}} \to \varepsilon$.
- For each rule $A \to \varepsilon$, we add the rules $A^{\mathsf{out}} \to \varepsilon$ and $A^{\mathsf{mid}} \to \varepsilon$.

Let $\hat{\Sigma}_\kappa = \Sigma \cup (\Sigma \times 2^{\mathsf{C}_{\mathcal{X}}}) \cup \mathsf{C}_{\mathcal{X}} \setminus \{\kappa\}$. For each nonterminal $A \in V$ these hold:

$$\begin{aligned}
L(A^{\mathsf{out}}) &= \{\hat{w} \mid A \Rightarrow^*_{\mathcal{F}} \hat{w},\ \text{where } \hat{w} \in \hat{\Sigma}^*_\kappa\} \\
L(A^{\mathsf{in}}) &= \{\mathsf{proc}_\kappa(\hat{w}) \mid A \Rightarrow^*_{\mathcal{F}} \hat{w},\ \text{where } \hat{w} = \hat{u}\kappa\beta a\hat{v} \text{ or } \hat{w} = \hat{u}\kappa\beta(a,T)\hat{v}, \\
&\qquad \hat{u}, \hat{v} \in \hat{\Sigma}^*_\kappa, \beta \in (\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\})^*, a \in \Sigma, T \subseteq \mathsf{C}_{\mathcal{X}} \setminus \{\kappa\}\} \\
L(A^{\mathsf{left}}) &= \{\hat{w}\beta \mid A \Rightarrow^*_{\mathcal{F}} \hat{w}\kappa\beta,\ \text{where } \hat{w} \in \hat{\Sigma}^*_\kappa, \beta \in (\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\})^*\} \\
L(A^{\mathsf{right}}) &= \{\beta(a, \{\kappa\})\hat{w} \mid A \Rightarrow^*_{\mathcal{F}} \beta a\hat{w},\ \text{where } \hat{w} \in \hat{\Sigma}^*_\kappa, \beta \in (\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\})^*\} \cup \\
&\quad \{\beta(a, T \cup \{\kappa\})\hat{w} \mid A \Rightarrow^*_{\mathcal{F}} \beta(a,T)\hat{w}, \text{where } \hat{w} \in \hat{\Sigma}^*_\kappa, \beta \in (\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\})^*, T \subseteq \mathsf{C}_{\mathcal{X}} \setminus \{\kappa\}\} \\
L(A^{\mathsf{mid}}) &= \{\beta \mid A \Rightarrow^*_{\mathcal{F}} \beta,\ \text{where } \beta \in (\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\})^*\}
\end{aligned}$$

These equalities are given without proof since they are not used, and are just for illustrating the idea behind the proof.

For the rest of our proof we will represent derivations $X \Rightarrow^* \delta$ as the sequence of productions $X_1 \Rightarrow \gamma_1, X_2 \Rightarrow \gamma_2, \ldots, X_m \Rightarrow \gamma_m$, where $X_1 = X$ and $\gamma_m = \delta$, which uniquely determines the derivation by doing it in the leftmost way. We use this representation to state exactly how derivations in $\mathcal{F}$ are translated to derivations in $\mathcal{F}'$ and vice versa.

Another notion we need to address is how, in a given derivation $X \Rightarrow^* \delta$, instances of nonterminals are located with respect to each other. By an *instance of a nonterminal* (or just *instance*), we mean an X_i along with some specific derivation $X_i \Rightarrow \gamma_i$ in the sequence. For some instances X_i and X_j, we say that X_i is a *descendant* of X_j if $X_i = X_j$, or if $Y \Rightarrow X_iZ$, or $Y \Rightarrow ZX_i$ for some Y which is a descendant of X_j. We say that X_i is *to the left* of X_j (or X_j is *to the right* of X_i) if there is a derivation $X \Rightarrow YZ$ in the sequence such that X_i is a descendant of Y and X_j is a descendant of Z.

We note a few things in our construction: (1) Each $X \in V_{\mathsf{out}}$ only produces terminals (which do not include κ) and nonterminals in V_{out}; furthermore, every rule $X \to YZ$ in P is copied into P' as $X^{\mathsf{out}} \to Y^{\mathsf{out}}Z^{\mathsf{out}}$. (2) Nonterminals in V_{in} do not produce any terminals or ε directly, so they need to derive into some $X \in V_{\mathsf{in}}$ and some $Y \in V_{\mathsf{right}}$ to derive some string. (3) As with V_{out}, each $X \in V_{\mathsf{mid}}$ only produces terminals in $\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\}$ and nonterminals in V_{mid}. (4) Each $X \in V_{\mathsf{left}}$ (resp. V_{right}) produces exactly one nonterminal $X' \in V_{\mathsf{left}}$ (resp. V_{right}), or ε (resp. (a, T) for some $a \in \Sigma$ and $T \subseteq \mathsf{C}_{\mathcal{X}}$ such that $\kappa \in T$); this, as a consequence, means that on each derivation from $\mathcal{F}'$ where the first production is not $S' \Rightarrow X^{\mathsf{out}}Y^{\mathsf{out}}$ there is exactly one derivation $X \Rightarrow \varepsilon$ such that $X \in V_{\mathsf{left}}$ (resp., exactly one derivation $X \Rightarrow (a, T)$, such that $X \in V_{\mathsf{right}}$).

From point (1) we see that each annotated ref-word $\hat{r} \in L(\mathcal{F})$ such that $\hat{r} \in \hat{\Sigma}^*_\kappa$ (this is, which does not mention κ at all) can be derived by S' starting by $S' \Rightarrow A^{\mathsf{out}}B^{\mathsf{out}}$, $S' \Rightarrow a$, $S' \Rightarrow \varepsilon$ or $S' \Rightarrow \kappa'$.

On the other hand, each annotated ref-word $\hat{r} \in L(\mathcal{F}')$ which does not have κ on any annotation set was necessarily derived through rules of the form $X^{\mathsf{out}} \to Y^{\mathsf{out}}Z^{\mathsf{out}}$ which correspond to the rule $X \to YZ$ in P, so we deduce that $\hat{r} \in L(\mathcal{F})$.

We shall now prove that for any string $\hat{r} = \hat{u}\kappa\beta a\hat{v}$, or $\hat{r} = \hat{u}\kappa\beta(a, T)\hat{v}$, where $\hat{u}, \hat{v} \in \hat{\Sigma}^*_\kappa$, $\beta \in \mathsf{C}_{\mathcal{X}} \setminus \{\kappa\}$, $a \in \Sigma$ and $T \subseteq \mathsf{C}_{\mathcal{X}} \setminus \{\kappa\}$ such that $\hat{r} \in L(\mathcal{F})$, it holds that $\mathsf{proc}(\hat{r}) \in$

$L(\mathcal{F}')$. W.l.o.g., let $\hat{r} = \hat{u}\kappa\beta a\hat{v}$ and consider some leftmost derivation of $\hat{r}$ from $\mathcal{F}$:

$$\begin{aligned} S &\Rightarrow^*_{\mathcal{F}} \hat{u}_1 A\delta \\ &\Rightarrow_{\mathcal{F}} \hat{u}_1 BC\delta \\ &\Rightarrow^*_{\mathcal{F}} \hat{u}_1\hat{u}_2\kappa\beta_1 C\delta \\ &\Rightarrow^*_{\mathcal{F}} \hat{u}_1\hat{u}_2\kappa\beta_1\beta_2 a\hat{v}_1\delta \\ &\Rightarrow^*_{\mathcal{F}} \hat{u}_1\hat{u}_2\kappa\beta_1\beta_2 a\hat{v}_1\hat{v}_2 = \hat{r}, \end{aligned}$$

where we have that $\beta = \beta_1\beta_2$, $\hat{u} = \hat{u}_1\hat{u}_2$ and $\hat{v} = \hat{v}_1\hat{v}_2$. Note that this is an arbitrary derivation, and we are merely identifying these nonterminals A, B and C. We also identify the nonterminals D, which produces κ, and E, which produces a. For the rest of the current part of the proof, we only refer to the *instances* of these nonterminals. Using this, we build a derivation from $\mathcal{F}'$ step by step:

(i) We have $S' \Rightarrow^*_{\mathcal{F}'} \hat{u}_1 A^{\mathsf{in}}\delta'$, where δ' is obtained by replacing each nonterminal X in δ by X^{out}. We get this by starting with the derivation $S \Rightarrow^*_{\mathcal{F}} \hat{u}_1 A\delta$, and replacing $X \Rightarrow_{\mathcal{F}} YZ$ by $X^{\mathsf{in}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{in}}Z^{\mathsf{out}}$ if A is a descendant of Y, by $X^{\mathsf{in}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{out}}Z^{\mathsf{in}}$ if Z is, or by $X^{\mathsf{out}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{out}}Z^{\mathsf{out}}$ if none is. We also replace each $X \Rightarrow_{\mathcal{F}} \tau$, for $\tau \in \Sigma \cup (\Sigma \times 2^{\mathsf{C}_{\mathcal{X}}}) \cup \mathsf{C}_{\mathcal{X}} \setminus \{\kappa\} \cup \{\varepsilon\}$, by $X^{\mathsf{out}} \Rightarrow_{\mathcal{F}'} \tau$. If $A = S$, we replace A it by S'.

(ii) We have the rule $A^{\mathsf{in}} \to B^{\mathsf{left}}C^{\mathsf{right}}$ which was added to P'.

(iii) We have $B^{\mathsf{left}} \Rightarrow^*_{\mathcal{F}'} \hat{u}_2\beta_1$. We get this by starting from $B \Rightarrow^*_{\mathcal{F}} \hat{u}_2\kappa\beta_1$, and we replace $X \Rightarrow_{\mathcal{F}} YZ$ by $X^{\mathsf{left}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{left}}Z^{\mathsf{mid}}$ if E is a descendant of D, by $X^{\mathsf{left}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{out}}Z^{\mathsf{left}}$ if Z is, by $X^{\mathsf{out}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{out}}Z^{\mathsf{out}}$ if X is to the left of D, and by $X^{\mathsf{mid}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{mid}}Z^{\mathsf{mid}}$ if it is to the right. We also replace $X \Rightarrow_{\mathcal{F}} \tau$ by $X^{\mathsf{out}} \Rightarrow_{\mathcal{F}'} \tau$ if X is to the left of D, and by $X^{\mathsf{mid}} \Rightarrow_{\mathcal{F}'} \tau$ if it is to the right. Lastly, we replace $D \Rightarrow_{\mathcal{F}} \kappa$ by $D^{\mathsf{left}} \Rightarrow_{\mathcal{F}'} \varepsilon$.

(iv) We have $C^{\mathsf{right}} \Rightarrow^*_{\mathcal{F}'} \beta_2(a, \{\kappa\})\hat{v}_1$. We get this by starting from $C \Rightarrow^*_{\mathcal{F}} \beta_2 a\hat{v}_1$, and we replace $X \Rightarrow_{\mathcal{F}} YZ$ by $X^{\mathsf{right}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{mid}}Z^{\mathsf{right}}$ if E is descendant of Z, or by $X^{\mathsf{right}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{right}}Z^{\mathsf{out}}$ if Y is, by $X^{\mathsf{mid}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{mid}}Z^{\mathsf{mid}}$ if X is to the left of

E, and by $X^{\mathsf{right}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{right}} Z^{\mathsf{right}}$ if it is to the right. We also replace $X \Rightarrow_{\mathcal{F}} \tau$ by $X^{\mathsf{mid}} \Rightarrow_{\mathcal{F}'} \tau$ if X is to the left of E, and by $X^{\mathsf{out}} \Rightarrow_{\mathcal{F}'} \tau$ if it is to the right. Lastly, we replace $E \Rightarrow_{\mathcal{F}} a$ by $E^{\mathsf{right}} \Rightarrow_{\mathcal{F}'} (a, \{\kappa\})$.

(v) We have $\delta' \Rightarrow^{*}_{\mathcal{F}'} \hat{v}_2$, which we obtain from $\delta \Rightarrow^{*}_{\mathcal{F}} \hat{v}_2$ by replacing each $X \Rightarrow_{\mathcal{F}} YZ$ by $X^{\mathsf{out}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{out}} Z^{\mathsf{out}}$, and each $X \Rightarrow_{\mathcal{F}} \tau$ by $X^{\mathsf{out}} \Rightarrow_{\mathcal{F}'} \tau$.

In the end, we get the following leftmost derivation from $\mathcal{F}'$:

$$\begin{aligned} S &\Rightarrow^{*}_{\mathcal{F}'} \hat{u}_1 A^{\mathsf{in}} \delta' \text{ (or } \hat{u}_1 S' \delta') \\ &\Rightarrow_{\mathcal{F}'} \hat{u}_1 B^{\mathsf{left}} C^{\mathsf{right}} \delta' \\ &\Rightarrow^{*}_{\mathcal{F}'} \hat{u}_1 \hat{u}_2 \beta_1 C^{\mathsf{right}} \delta' \\ &\Rightarrow^{*}_{\mathcal{F}'} \hat{u}_1 \hat{u}_2 \beta_1 \beta_2 (a, \{\kappa\}) \hat{v}_1 \delta' \\ &\Rightarrow^{*}_{\mathcal{F}'} \hat{u}_1 \hat{u}_2 \beta_1 \beta_2 (a, \{\kappa\}) \hat{v}_1 \hat{v}_2 = \mathsf{proc}_{\kappa}(\hat{r}), \end{aligned}$$

which proves that $\mathsf{proc}(\hat{r}) \in L(\mathcal{F}')$.

We will prove that for every $\hat{s} = \hat{u}(a, T)\hat{v} \in L(\mathcal{F}')$ where $\kappa \in T$ there is $\hat{r}$ such that $\mathsf{proc}_{\kappa}(\hat{r}) = \hat{s}$ in a similar way. We argue that any leftmost derivation that produces $\hat{s}$ has the following form:

$$\begin{aligned} S' &\Rightarrow^{*}_{\mathcal{F}'} \hat{u}_1 A^{\mathsf{in}} \delta_1 \;\text{ (or } \hat{u}_1 S' \delta_1) \\ &\Rightarrow_{\mathcal{F}'} \hat{u}_1 B^{\mathsf{left}} C^{\mathsf{right}} \delta_1 \\ &\Rightarrow^{*}_{\mathcal{F}'} \hat{u}_1 \hat{u}_2 D^{\mathsf{left}} \delta_2 C^{\mathsf{right}} \delta_1 \\ &\Rightarrow_{\mathcal{F}'} \hat{u}_1 \hat{u}_2 \delta_2 C^{\mathsf{right}} \delta_1 \\ &\Rightarrow^{*}_{\mathcal{F}'} \hat{u}_1 \hat{u}_2 \beta_1 C^{\mathsf{right}} \delta_1 \\ &\Rightarrow^{*}_{\mathcal{F}'} \hat{u}_1 \hat{u}_2 \beta_1 \beta_2 (a, T) \hat{v}_1 \delta_1 \\ &\Rightarrow^{*}_{\mathcal{F}'} \hat{u}_1 \hat{u}_2 \beta_1 \beta_2 (a, T) \hat{v}_1 \hat{v}_2 = \hat{s}, \end{aligned}$$

where $\delta \in V^{*}_{\mathsf{out}}$, $\delta' \in V^{*}_{\mathsf{mid}}$, $\beta_1, \beta_2 \in (\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\})^{*}$, $\hat{u} = \hat{u}_1 \hat{u}_2$ and $\hat{v} = \hat{v}_1 \hat{v}_2$. The reasoning goes as follows:

- We know that $S' \Rightarrow^*_{\mathcal{F}'} \hat{u}\beta(a,T)\hat{v} \in L(\mathcal{F}')$. If $X \Rightarrow_{\mathcal{F}'} (a,T)$ and $\kappa \in T$, then $X \in V_{\mathsf{right}}$.
- From the way $\mathcal{F}'$ was built, there must be a production $X \Rightarrow_{\mathcal{F}'} YZ$ in $S' \Rightarrow^*_{\mathcal{F}'} \hat{s}$ such that $X \in V_{\mathsf{in}}$ (or $X = S'$), $Y \in V_{\mathsf{left}}$ and $Z \in V_{\mathsf{right}}$, as it is the only way to derive a nonterminal in V_{right}. Let A^{in} (or S'), B^{left} and C^{right} be these X, Y and Z respectively.
- Seeing the rules in P' we note that every string of terminals that is derivable from B^{left} is of the form $\hat{w}\beta$, where $\hat{w} \in \hat{\Sigma}_\kappa$ and $\beta \in (\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\})^*$. Furthermore, this string satisfies that there is a production $X \Rightarrow_{\mathcal{F}'} \varepsilon$ for some $X \in V_{\mathsf{left}}$ such that this ε is exactly at the left of where β begins. Let $\hat{u}_2$ be this $\hat{w}$, let D^{left} be this X, and let β_1 be this β.
- Likewise, we note that C^{right} always derives a string of terminals of the form $\beta(a',T')\hat{w}$ for some $\beta \in (\mathsf{C}_{\mathcal{X}} \setminus \{\kappa\})^*$ and $\hat{w} \in \hat{\Sigma}_\kappa$. Let β_2 be this β and let $\hat{v}_1$ be this $\hat{w}$.
- Lastly, let $S' \Rightarrow^*_{\mathcal{F}'} \hat{u}_1 A^{\mathsf{in}}\delta_1$ (or $\hat{u}_1 S'\delta_1$) be the one that derives $\hat{s}$. From the rules in P', we note that δ_1 is composed solely of nonterminals in V_{out}.

An important point that can be seen from this reasoning is that for each instance $X \neq S'$ that appears in the derivation $S' \Rightarrow^*_{\mathcal{F}'} \hat{s}$, we can deduce the set V_{scr} for which $X \in V_{\mathsf{scr}}$, among the options $\mathsf{scr} \in \{\mathsf{out}, \mathsf{in}, \mathsf{left}, \mathsf{mid}, \mathsf{right}\}$, by seeing its position in the derivation. To be precise, this is given from how X relates to the instances $D^{\mathsf{left}} \Rightarrow \varepsilon$, and to $E^{\mathsf{right}} \Rightarrow (a,T)$, for the nonterminal $E^{\mathsf{right}} \in V_{\mathsf{right}}$ that satisfies this. (1) If X is to the left of D^{left}, then $X \in V_{\mathsf{out}}$, (2) if D^{left} is a descendant of X, but E^{right} is not, then $X \in V_{\mathsf{left}}$, (3) if X is to the right of D^{left}, and is to the left of E^{right}, then $X \in V_{\mathsf{mid}}$, (4) if E^{right} is a descendant of X, but D^{left} is not, then $X \in X_{\mathsf{right}}$, (5) if X is to the right of E^{right}, then $X \in V_{\mathsf{out}}$, and (6) if both D^{left} and E^{right} are descendants of X, then $X \in V_{\mathsf{in}}$. We bring attention to the fact that in this paragraph we referred only to the instances of D^{left} and E^{right} on the derivations mentioned above.

Another, more important point, is this reasoning gives us the derivation presented above. This derivation is translated into the following derivation in $\mathcal{F}$:

$$\begin{aligned}
S &\Rightarrow^*_{\mathcal{F}} \hat{u}_1 A \delta'_1 \ \text{(or } \hat{u}_1 S \delta'_1\text{)} \\
&\Rightarrow_{\mathcal{F}} \hat{u}_1 B C \delta'_1 \\
&\Rightarrow^*_{\mathcal{F}} \hat{u}_1 \hat{u}_2 D \delta'_2 C \delta'_1 \\
&\Rightarrow_{\mathcal{F}} \hat{u}_1 \hat{u}_2 \kappa \delta'_2 C \delta'_1 \\
&\Rightarrow^*_{\mathcal{F}} \hat{u}_1 \hat{u}_2 \kappa \beta_1 C \delta'_1 \\
&\Rightarrow_{\mathcal{F}} \hat{u}_1 \hat{u}_2 \kappa \beta_1 \beta_2 (a, T \setminus \{\kappa\}) \hat{v}_1 \delta'_1, \quad \text{ or} \\
&\qquad \hat{u}_1 \hat{u}_2 \kappa \beta_1 \beta_2 a \hat{v}_1 \delta'_1, \\
&\Rightarrow^*_{\mathcal{F}} \hat{u}_1 \hat{u}_2 \kappa \beta_1 \beta_2 (a, T \setminus \{\kappa\}) \hat{v}_1 \hat{v}_2 = \hat{r}, \quad \text{ or} \\
&\qquad \hat{u}_1 \hat{u}_2 \kappa \beta_1 \beta_2 a \hat{v}_1 \hat{v}_2 = \hat{r},
\end{aligned}$$

Where δ'_1 and δ'_2 are obtained by replacing each X^{mid} by X in δ_1 and δ_2, respectively. It is direct to see that this is a valid derivation since for every production $X^{\mathsf{x}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{y}} Z^{\mathsf{z}}$ there exists a valid production $X \Rightarrow_{\mathcal{F}} YZ$, for any $\mathsf{x}, \mathsf{y}, \mathsf{z} \in \{\mathsf{in}, \mathsf{out}, \mathsf{left}, \mathsf{mid}, \mathsf{right}\}$. Furthermore, for the production $D^{\mathsf{left}} \Rightarrow_{\mathcal{F}'} \varepsilon$ there exists $D \Rightarrow_{\mathcal{F}} \kappa$, and for $C^{\mathsf{right}} \Rightarrow_{\mathcal{F}'} (a, T)$ there exists $C \Rightarrow_{\mathcal{F}} (a, T \setminus \{\kappa\})$ if $T \neq \{\kappa\}$, and $C \Rightarrow_{\mathcal{F}} a$ if $T = \{\kappa\}$. Further, note that $\mathsf{proc}_\kappa(\hat{r}) = \hat{s}$. We conclude that $\hat{r} \in L(\mathcal{F})$ for some $\hat{r}$ such that $\mathsf{proc}_\kappa(\hat{r}) = \hat{s}$.

From the arguments above, we obtain that for each annotated ref-word $\hat{r} \in L(\mathcal{F})$ there exists an equivalent annotated ref-word $\hat{t} \in L(\mathcal{F})$, given by $\hat{t} = \mathsf{proc}_\kappa(\hat{r})$. Furthermore, we showed that for each annotated ref-word $\hat{t} \in L(\mathcal{F}')$ there exists an equivalent $\hat{r} \in L(\mathcal{F})$. This implies that $\mathcal{F}$ and $\mathcal{F}'$ are equivalent.

Now, assume that $\mathcal{F}$ is unambiguous. We will prove that $\mathcal{F}'$ is unambiguous as well. Consider an annotated ref-word $\hat{t} \in L(\mathcal{F}')$ and consider two sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ which define the derivation $S' \Rightarrow^*_{\mathcal{F}'} \hat{t}$. We showed above how to translate these sequences into sequences $\mathcal{S}'_1$ and $\mathcal{S}'_2$ which define the derivation $S \Rightarrow^*_{\mathcal{F}} \hat{r}$, for some $\hat{r}$ such that $\hat{t} = \mathsf{proc}_{\kappa}(\hat{r})$. Since $\mathcal{F}$ is unambiguous, these sequences are equal. Assume now that $\mathcal{S}_1$ and $\mathcal{S}_2$ are not equal, but since their translations into $\mathcal{F}$ are the same, then it must be that for some production $X \Rightarrow_{\mathcal{F}} YZ$ or $X \Rightarrow_{\mathcal{F}} \tau$, there must be two productions $X^{\mathsf{x}_1} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{y}_1} Z^{\mathsf{z}_1}$ and $X^{\mathsf{x}_2} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{y}_2} Z^{\mathsf{z}_2}$, or $X^{\mathsf{x}_1} \Rightarrow_{\mathcal{F}'} \tau$ and $X^{\mathsf{x}_2} \Rightarrow_{\mathcal{F}'} \tau$ at the same position, for some $(\mathsf{x}_1, \mathsf{y}_1, \mathsf{z}_1) \neq (\mathsf{x}_2, \mathsf{y}_2, \mathsf{z}_2)$. We note that this is not possible since we argued that for a given derivation $S' \Rightarrow^*_{\mathcal{F}'} \hat{r}$, the set in which each nonterminal instance belongs, among V_{out}, V_{in}, V_{left}, V_{mid}, V_{right}, is fixed by its relation to certain instances of D^{left} and E^{right}. We conclude that $\mathcal{F}'$ is unambiguous.

Assume $\mathcal{F}'$ is unambiguous. We will prove that $\mathcal{F}$ is unambiguous as well. Likewise, consider an annotated ref-word $\hat{r} \in L(\mathcal{F})$, and consider two sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ which define the derivation $S \Rightarrow^*_{\mathcal{F}} \hat{t}$. We showed above how to convert these sequences into $\mathcal{S}'_1$ and $\mathcal{S}'_2$ which define the derivation $S' \Rightarrow^*_{\mathcal{F}'} \mathsf{proc}_{\kappa}(\hat{r})$. Since $\mathcal{F}'$ is unambiguous, then it must hold that $\mathcal{S}'_1 = \mathcal{S}'_2$. Note that the translation we showed consisted in replacing productions of the form $X \Rightarrow_{\mathcal{F}} YZ$ by $X^{\mathsf{x}} \Rightarrow_{\mathcal{F}'} Y^{\mathsf{y}} Z^{\mathsf{z}}$, $X \Rightarrow_{\mathcal{F}} \kappa$ by $X^{\mathsf{left}} \Rightarrow_{\mathcal{F}'} \varepsilon$, $X \Rightarrow_{\mathcal{F}} a$ by $X^{\mathsf{right}} \Rightarrow_{\mathcal{F}} (a, \{\kappa\})$ (or $X \Rightarrow_{\mathcal{F}} (a, T)$ by $X^{\mathsf{right}} \Rightarrow_{\mathcal{F}'} (a, T \cup \{\kappa\})$) for a single fixed production, and $X \Rightarrow_{\mathcal{F}} \tau$ by $X^{\mathsf{x}} \Rightarrow_{\mathcal{F}'} \tau$ in any other case, for some $\mathsf{x}, \mathsf{y}, \mathsf{z} \in \{\mathsf{out}, \mathsf{in}, \mathsf{left}, \mathsf{mid}, \mathsf{right}\}$. Therefore, the sequence S_1 from which S'_1 was obtained is uniquely defined, from which we deduce that $S_1 = S_2$, and we conclude that $\mathcal{F}$ is unambiguous.

At the end of the procedure, we obtain an extraction grammar with annotations $\mathcal{F}^\dagger = (V^\dagger, \Sigma, \mathcal{X}, P^\dagger, S^\dagger)$ such that there are no rules of the form $X \to \kappa$ in $P^\dagger$, for any $\kappa \in \mathsf{C}_{\mathcal{X}}$. From this, we obtain the annotated grammar $\mathcal{G} = (V^\dagger, \Sigma, \Omega_{\mathcal{X}}, P^\dagger, S^\dagger)$ which is equivalent to $\mathcal{H}$. Furthermore, $\mathcal{G}$ is unambiguous if and only if $\mathcal{H}$ is unambiguous.

With respect to the running time of building $\mathcal{G}$, note that in each iteration of the algorithm, by starting on an extraction grammar with annotations $\mathcal{F}$ with a set of rules P, the resulting $\mathcal{F}'$ has a set of rules P' with a size of $9|P|$. Since this step is repeated twice for each variable $x \in \mathcal{X}$ (once for each variable operation), the total running time is $\mathcal{O}(9^{2|\mathcal{X}|}(3^{2|\mathcal{X}|}|\mathcal{H}|^2)) = \mathcal{O}(9^{3|\mathcal{X}|}|\mathcal{H}|^2)$. □

Hence, our formalism of annotated grammars captures that of extraction grammars. Unfortunately, the translation is exponential, intuitively because Ω must cover all possible sets of variable operations. We note that, in exchange for this, annotated grammars are strictly more *expressive*: each output can annotate an arbitrary number of positions in the string (e.g., every other character), unlike extraction grammars whose mappings have a fixed number of variables.

We complete Proposition 4.10 to give more intuition about the conciseness of extraction grammars vs annotated grammars, and the difference in expressiveness.

We first give a simple example to show that, with our notion of equivalence, we may indeed need an exponential number of symbols in the annotation set, implying that extraction grammars are in some cases exponentially more concise:

Example 4.1. *Consider the following functional extraction grammar $\mathcal{H}$ with n variables $x_1, \ldots, x_n$ and alphabet $\{a\}$:*

$$\begin{array}{rrcl} \mathcal{H}: & A_1 & \rightarrow & \vdash_{x_1}\dashv_{x_1} A_2 \ \mid \vdash_{x_1} A_2 \dashv_{x_1} \\ & A_2 & \rightarrow & \vdash_{x_2}\dashv_{x_2} A_3 \ \mid \vdash_{x_2} A_3 \dashv_{x_2} \\ & & \vdots & \\ & A_n & \rightarrow & \vdash_{x_n}\dashv_{x_n} a \ \mid \vdash_{x_n} a \dashv_{x_n} \end{array}$$

For the document a, this extraction grammar will output all possible combinations depending on whether $\dashv_{x_i}$ is at the beginning or end of a for each $i \leq n$. Thus, an equivalent annotated grammar will need to consider all possible subsets of $\{\dashv_x \mid x \in \mathcal{X}\}$ as possible annotations of the character a, which will require an exponential number of rules.

We then illustrate why annotated grammars are in fact *strictly* more expressive: in addition to capturing all extraction grammars (Proposition 4.10), annotated grammars can express functions that do not correspond to an annotation grammar.

Example 4.2. *Consider a singleton annotation set* $\Omega = \{\vartheta\}$*, a singleton alphabet* $\Sigma = \{a\}$*, and the annotated grammar with start symbol* S *and production* $S \rightarrow a(a, \vartheta)S|a|\varepsilon$*. For each string of* Σ^**, it produces one output where every other character is annotated. This cannot be expressed by an extraction grammar, as such a grammar fixes a finite set* $\mathcal{X}$ *of variables independently from the input document, and each variable is mapped to only one span.*

Enumeration for extraction grammars. As extraction grammars can be rewritten to annotated grammars in an unambiguity-preserving way (Proposition 4.10), we can derive from Theorem 4.2 an enumeration result for unambiguous extraction grammars with cubic preprocessing time in data complexity.

Theorem 4.6. *Given an unambiguous extraction grammar* $\mathcal{H}$ *with* k *variables and a string* s*, we can enumerate the mappings of* $[\![\mathcal{H}]\!](s)$ *with preprocessing time* $\mathcal{O}(9^{3k} \cdot |\mathcal{H}|^2 \cdot |s|^3)$ *(hence, cubic in* $|s|$*), and with output-linear delay (independent from* s*,* k*, or* $\mathcal{H}$*).*

In data complexity, this improves over the result of (Peterfreund, 2023) for unambiguous extraction grammars, whose preprocessing time is $\mathcal{O}(9^{2k} \cdot |\mathcal{H}|^2 \cdot |s|^5)$, i.e., our data complexity is cubic instead of quintic. We leave to future work a study of enumeration results for restricted classes of extraction grammars via Theorems 4.2 and 4.4.

4.6. Related Work

We have explained how our work is set in the context of document spanners (Fagin et al., 2015), and in particular of enumeration results for regular spanners (Florenzano et al., 2018; Amarilli et al., 2019c). A recent survey of much of this literature can be found in Peterfreund's PhD thesis (Peterfeund, 2019). The most related work to ours is the more recent introduction of extraction grammars by Peterfreund (Peterfreund, 2023), which we already discussed.

There are also some other extensions of regular spanners that are reminiscent of CFGs, e.g., core spanners (featuring equality) or generalized core spanners (with difference) already introduced in (Fagin et al., 2015), or Datalog evaluated over regular spanners as in (Peterfreund, ten Cate, Fagin, & Kimelfeld, 2019). However, to our knowledge, there are no known constant-delay enumeration algorithms in these contexts.

Our study of enumeration for annotated grammars is also reminiscent of enumeration results for queries over trees expressed as tree automata. An algorithm for this was given by Bagan (Bagan, 2006b) with linear-time preprocessing and constant-delay in data complexity, for deterministic tree automata, and this was extended in (Amarilli, Bourhis, Mengel, & Niewerth, 2019d) to nondeterministic automata. However, this is again more restricted: evaluating a tree automaton on a tree amounts to evaluating a *visibly pushdown* automaton over a string representation of the tree, which is again more restrictive than general context-free grammars.

5. ENUMERATION ON SLP-COMPRESSED DOCUMENTS

In this chapter, we study the problem of enumerating results from a query over a compressed document. The model we use for compression are straight-line programs (SLPs), which are defined by a context-free grammar that produces a single string. For our queries, we use a model called Annotated Automata, an extension of regular automata that allows annotations on letters. This model extends the notion of regular spanners as it allows arbitrarily long outputs.

The main result in this chapter is an algorithm that evaluates such a query by enumerating all results with output-linear delay after a preprocessing phase which takes linear time on the size of the SLP, and cubic time over the size of the automaton. We achieve this through a persistent data structure named Enumerable Compact Sets with Shifts which guarantees output-linear delay under certain restrictions. These results imply constant-delay enumeration algorithms in the context of regular spanners.

Further, we use an extension of annotated automata which utilizes succinctly encoded annotations to save an exponential factor from previous results that dealt with constant-delay enumeration over variable-set automata. Lastly, we extend our results to allow complex document editing while maintaining the constant delay guarantee.

Outline of the chapter. In Section 5.1 we introduce the setting and its corresponding enumeration problem. In Section 5.2, we present our data structure for storing and enumerating the outputs, and in Section 5.3 we show the evaluation algorithm. Section 5.4 offers the application of the algorithmic results to document spanners, plus an extension for compressed annotation schemes, and Section 5.5 shows how to extend these results to deal with complex document editing.

5.1. Setting and main problem of the chapter

In this section, we present the setting and state the main result of the chapter. First, we define straight-line programs, which we will use for the compressed representation of input documents. Then we introduce the definition of annotated automaton, an extension of regular automata to produce outputs. We use annotated automata as our computational model to represent queries over documents. By combining both formalisms, we state the main enumeration problem and the main technical result.

SLP-compression. Recall that a *context-free grammar* is a tuple $G = (N, \Sigma, R, S_0)$, where N is a non-empty set of non-terminals, Σ is finite alphabet, $S_0 \in N$ is the start symbol and $R \subseteq N \times (N \cup \Sigma)^+$ is the set of rules. As a convention, the rule $(A, w) \in R$ will be written as $A \to w$, and we will call Σ and N the set of terminal and non-terminal symbols, respectively. A context-free grammar $S = (N, \Sigma, R, S_0)$ is a *straight-line program* (SLP) if R is a total function from N to $(N \cup \Sigma)^+$ and the directed graph $(N, \{(A, B) \mid (A, w) \in R \text{ and } B \text{ appears in } w\})$ is acyclic. For every $A \in N$, let $R(A)$ be the unique $w \in (N \cup \Sigma)^+$ such that $(A, w) \in R$, and for every $a \in \Sigma$ let $R(a) = a$. We extend R to a morphism $R^* : (N \cup \Sigma)^* \to \Sigma^*$ recursively such that $R^*(d) = d$ when d is a document, and $R^*(\alpha_1 \ldots \alpha_n) = R^*\big(R(\alpha_1) \cdot \ldots \cdot R(\alpha_n)\big)$, where $\alpha_i \in (N \cup \Sigma)$ for every $i \leq n$. By our definition of SLP, $R^*(A)$ is in Σ^+, and uniquely defined for each $A \in N$. Then we define the document encoded by S as $\mathrm{doc}(S) = R^*(S_0)$.

Example 5.1. *Let $S = (N, \Sigma, R, S_0)$ be an SLP with $N = \{S_0, A, B\}$, $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{r}\}$, and $R = \{S_0 \to A\mathsf{r}BABA, A \to \mathsf{ba}, B \to A\mathsf{ra}\}$. We then have that $\mathrm{doc}(A) = \mathsf{ba}$, $\mathrm{doc}(B) = \mathsf{bara}$ and $\mathrm{doc}(S) = \mathrm{doc}(S_0) = \mathsf{barbarababaraba}$, namely, the string represented by S.*

We define the size of an SLP $S = (N, \Sigma, R, S_0)$ as $|S| = \sum_{A \in N} |R(A)|$, namely, the sum of the lengths of the right-hand sides of all rules. It is important to note that an SLP S can encode a document $\mathrm{doc}(S)$ such that $|\mathrm{doc}(S)|$ is exponentially larger with respect to $|S|$. For this reason, SLPs stay as a commonly used data compression scheme (Storer

& Szymanski, 1982; Kieffer & Yang, 2000; Rytter, 2002; Claude & Navarro, 2011), and they are often studied particularly because of their algorithmic properties; see (Lohrey, 2012) for a survey. In this chapter, we consider SLP compression to represent documents and use the formalism of annotated automata for extracting relevant information from the document.

Annotated automata. An *annotated automaton* (AnnA for short) is a finite state automaton where we label some transitions with annotations. Formally, it is a tuple $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, I, F)$ where Q is a state set, Σ is an input alphabet, Ω is an output alphabet, $I \subseteq Q$ and $F \subseteq Q$ are the initial and final set of states, respectively, and

$$\Delta \;\subseteq\; \underbrace{Q \times \Sigma \times Q}_{\text{read transitions}} \;\cup \underbrace{Q \times (\Sigma \times \Omega) \times Q}_{\text{read-annotate transitions}}$$

is the transition relation, which contains *read transitions* of the form $(p, a, q) \in Q \times \Sigma \times Q$, and *read-annotate transitions* of the form $(p, (a, \vartheta), q) \in Q \times (\Sigma \times \Omega) \times Q$.

Similarly to transducers (Berstel, 2013), a symbol $a \in \Sigma$ is an input symbol that the machine reads and $\vartheta \in \Omega$ is an output symbol that indicates what the machine prints in an output tape. A run ρ of $\mathcal{A}$ over a document $d = a_1 a_2 \ldots a_n \in \Sigma^*$ is a sequence of the form:

$$\rho \;:=\; q_1 \xrightarrow{b_1} q_2 \xrightarrow{b_2} \ldots \xrightarrow{b_{n+1}} q_{n+1}$$

such that $q_1 \in I$ and, for each $i \in \{1, \ldots, n\}$, it holds that either $b_i = a_i$ and $(q_i, a_i, q_{i+1}) \in \Delta$, or $b_i = (a_i, \vartheta)$ and $(q_i, (a_i, \vartheta), q_{i+1}) \in \Delta$. We say that ρ is accepting if $q_{n+1} \in F$.

We define the *annotation* of ρ as $\mathsf{ann}(\rho) = \mathsf{ann}(b_1) \cdot \ldots \cdot \mathsf{ann}(b_n)$ such that $\mathsf{ann}(b_i) = (\vartheta, i)$ if $b_i = (a, \vartheta)$, and $\mathsf{ann}(b_i) = \varepsilon$ otherwise, for each $i \in \{1, \ldots, n\}$. Given an annotated automaton $\mathcal{A}$ and a document $d \in \Sigma^*$, we define the set $[\![\mathcal{A}]\!](d)$ of all outputs of $\mathcal{A}$ over d as:

$$[\![\mathcal{A}]\!](d) \;=\; \{\mathsf{ann}(\rho) \mid \rho \text{ is an accepting run of } \mathcal{A} \text{ over } d\}.$$

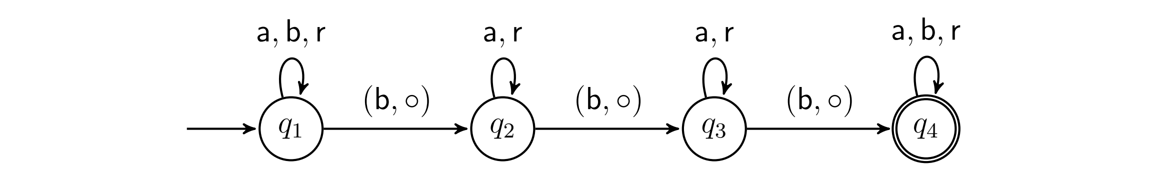


Figure 5.1. Example of an annotated automaton.

Note that each output in $[\![\mathcal{A}]\!](d)$ is a sequence of the form $(\vartheta_1, i_1) \dots (\vartheta_k, i_k)$ for some $k \leq n$ where $i_1 < \dots < i_k$ and each (ϑ_j, i_j) means that position i_j is annotated with the symbol ϑ_j.

Example 5.2. *Consider an AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, I, F)$ where $I = \{q_1\}$ $Q = \{q_1, q_2, q_3, q_4\}$, $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{r}\}$, $\Omega = \{\circ\}$, and $F = \{q_4\}$. We define Δ as the set of transitions that are depicted in Figure 5.1. For the document $d =$ $\mathsf{barbarababaraba}$ the set $[\![A]\!](d)$ contains the strings $(\circ, 1)(\circ, 4)(\circ, 8)$, $(\circ, 4)(\circ, 8)(\circ, 10)$ and $(\circ, 8)(\circ, 10)(\circ, 14)$. Intuitively, $\mathcal{A}$ selects triples of* b *which are separated by characters other than* b.

Annotated automata are the natural regular counterpart of annotated grammars introduced in Chapter 4. Moreover, it is the generalization and simplification of similar automaton formalisms introduced in the context of information extraction (Fagin et al., 2015; Peterfreund, 2023), complex event processing (Grez, Riveros, Ugarte, & Vansummeren, 2021; Grez & Riveros, 2020), and enumeration in general (Bourhis, Grez, Jachiet, & Riveros, 2021). In Section 5.4, we show how we can reduce the automaton model of document spanners, called a variable-set automaton, into a (succinctly) annotated automaton, generalizing the setting in (Schmid & Schweikardt, 2021).

Similar to other automata models, the notion of an unambiguous automaton will be crucial for us to remove duplicate runs for the same output. Specifically, we say that an AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, I, F)$ is *unambiguous* if for every $d \in \Sigma^*$ and every $w \in [\![\mathcal{A}]\!](d)$ there is exactly one accepting run ρ of $\mathcal{A}$ over d such that $w = \mathsf{ann}(\rho)$. On the other hand, we say that $\mathcal{A}$ is *deterministic* if Δ is a partial function of the form $\Delta : (Q \times \Sigma \cup Q \times (\Sigma \times \Omega)) \to Q$. Note that every deterministic AnnA is always unambiguous. The definition of unambiguous is in line with the notion of unambiguous annotated grammar (see Chapter 4, similar notions are also present in Chapter 3), and determinism with the idea of I/O-determinism used in (Florenzano et al., 2020; Bourhis et al., 2021; Grez et al., 2021). As usual, one can easily show that for every AnnA $\mathcal{A}$ there exists an equivalent deterministic AnnA (of exponential size) and, therefore, an equivalent unambiguous AnnA (see (Florenzano et al., 2020; Bourhis et al., 2021; Grez et al., 2021) for a proof of this result).

Lemma 5.1. *For every annotated automaton $\mathcal{A}$ there exists a deterministic annotated automaton $\mathcal{A}'$ such that $[\![\mathcal{A}]\!](d) = [\![\mathcal{A}']\!](d)$ for every $d \in \Sigma^*$.*

Regarding the expressive power of annotated automata, we note that they have the same expressive power as MSO formulas with monadic second-order free variables. We refer the reader to Chapter 3 for an analogous result in the context of nested documents. In fact, the equivalence between these models can be obtained as a corollary of Proposition 1 in Chapter 3. Finally, by Lemma 5.1 we do not loose expressive power if we restrict to the class unambiguous annotated automata, since we can convert every annotated automata into an deterministic one with an exponential cost in the size of the initial automaton.

Main result. We are interested in the problem of evaluating annotated automata over an SLP-compressed document, namely, to enumerate all the annotations over the document represented by an SLP. Formally, we define the main evaluation problem of this chapter as follows. Let $\mathcal{C}$ be any class of AnnA (e.g., unambiguous AnnA).

Problem:	$\text{SLP}\textsc{Enum}[\mathcal{C}]$
Input:	An AnnA $\mathcal{A} \in \mathcal{C}$ and an SLP S
Output:	Enumerate $[\![\mathcal{A}]\!](\text{doc}(S))$

As established, we assume here the computational model of Random Access Machines (RAM) with uniform cost measure and addition and subtraction as basic operations. Further, as it is commonly done on algorithms over SLPs and other compression schemes, we assume that the registers in the underlying RAM-model allow for constant-time arithmetical operations over positions in the *uncompressed* document (i.e., they have $\mathcal{O}(\log|\operatorname{doc}(S)|)$ size).

The notion of output-linear delay is a refinement of the better-known constant-delay bound, which requires that each output has a constant size (i.e., with respect to the input). Since even the document encoded by an SLP can be of exponential length, it is more reasonable in our setting to use the output-linear delay guarantee.

The following is the main technical result of this chapter.

Theorem 5.1. *Let $\mathcal{C}$ be the class of all unambiguous AnnAs. Then one can solve the problem* $\textsc{SLPEnum}[\mathcal{C}]$ *with linear preprocessing time and output-linear delay. Specifically, there exists an enumeration algorithm that runs in $|\mathcal{A}|^3 \times |S|$-preprocessing time and output-linear delay for enumerating $[\![\mathcal{A}]\!](\operatorname{doc}(S))$ given an unambiguous AnnA $\mathcal{A}$ and an SLP S.*

We dedicate the rest of the chapter on presenting the enumeration algorithm of Theorem 5.1. In Section 5.3 we explain the preprocessing phase of the algorithm. Before that, in the following section, we explain how *Enumerable Compact Sets with Shifts* work, which is the data structure that we use to store the outputs during the preprocessing phase.

5.2. Enumerable compact sets with shifts

In this section, we present the data structure, called *Enumerable Compact Sets with Shifts*, that will be used to compactly store the outputs of evaluating an annotated automaton over a straight-line program. This structure extends Enumerable Compact Sets (ECS) introduced in Chapter 3 (which were, in turn, strongly inspired by the work in (Amarilli et al., 2017, 2019a)). Indeed, people have also used ECS extensions in (Bucchi, Grez, Quintana, Riveros, & Vansummeren, 2022). This new version extends ECS by introducing a shift operator, which allows to succinctly move all outputs' positions with a single call.

Although the shift nodes require a revision of the complete ECS model, it simplifies the evaluation algorithm in Section 5.3 and achieves output-linear delay for enumerating all the outputs. For completeness of presentation, this section goes through all main details of ECS, as it was done in Chapter 3, and how to modify them with shifts.

The structure. Let Ω be an output alphabet such that Ω has no elements in common with $\mathbb{Z}$ or $\{\cup, \odot\}$ (i.e., $\Omega \cap \mathbb{Z} = \emptyset$ and $\Omega \cap \{\cup, \odot\} = \emptyset$). We define an *Enumerable Compact Set with Shifts* (Shift-ECS) as a tuple:

$$\mathcal{D} \;=\; (\Omega, V, \ell, r, \lambda)$$

such that V is a finite sets of nodes, $\ell\colon V \to V$ and $r\colon V \to V$ are the *left* and *right* partial functions, and $\lambda\colon V \to \Omega \cup \mathbb{Z} \cup \{\cup, \odot\}$ is a labeling function. We assume that $\mathcal{D}$ forms an acyclic graph, namely, the induced graph $(V, \{(v, \ell(v)), (v, r(v)) \mid v \in V\})$ is acyclic. Further, for every node $v \in V$, $\ell(v)$ is defined iff $\lambda(v) \in \mathbb{Z} \cup \{\cup, \odot\}$, and $r(v)$ is defined iff $\lambda(v) \in \{\cup, \odot\}$. Notice that, by definition, nodes labeled by Ω are bottom nodes in the acyclic structure formed by $\mathcal{D}$, and nodes labeled by $\mathbb{Z}$ or $\{\cup, \odot\}$ are inner nodes. Here, $\mathbb{Z}$-nodes are unary operators (i.e., $r(\cdot)$ is not defined over them), and $\cup$-nodes or $\odot$-nodes are binary operators. Indeed, we say that $v \in V$ is a *bottom node* if $\lambda(v) \in \Omega$, a *product node* if $\lambda(v) = \odot$, a *union node* if $\lambda(v) = \cup$, and a *shift node* if $\lambda(v) \in \mathbb{Z}$. Finally, we define the size of $\mathcal{D}$ as $|\mathcal{D}| = |V|$.

The outputs retrieved from a Shift-ECS are strings of the form $(\vartheta_1, i_1)(\vartheta_2, i_2)\ldots(\vartheta_\ell, i_\ell)$, where $\vartheta_j \in \Omega$ and $i_j \in \mathbb{Z}$. To build them, we use the *shifting function* $\mathsf{sh} : (\Omega \times \mathbb{Z}) \times \mathbb{Z} \to (\Omega \times \mathbb{Z})$ such that $\mathsf{sh}((\vartheta, i), s) = (\vartheta, i + s)$. We extend this function to strings over $\Omega \times \mathbb{Z}$ such that $\mathsf{sh}((\vartheta_1, i_1)\ldots(\vartheta_\ell, i_\ell), s) = (\vartheta_1, i_1 + s)\ldots(\vartheta_\ell, i_\ell + s)$ and to set of strings such that $\mathsf{sh}(L, s) = \{\mathsf{sh}(w, s) \mid w \in L\}$ for every $L \subseteq (\Omega \times \mathbb{Z})^*$.

Each node $v \in V$ of a Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ defines a set of output strings. Specifically, we associate a set of strings $[\![\mathcal{D}]\!](v)$ recursively as follows. For any two sets of strings L_1 and L_2, define $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$. Then:

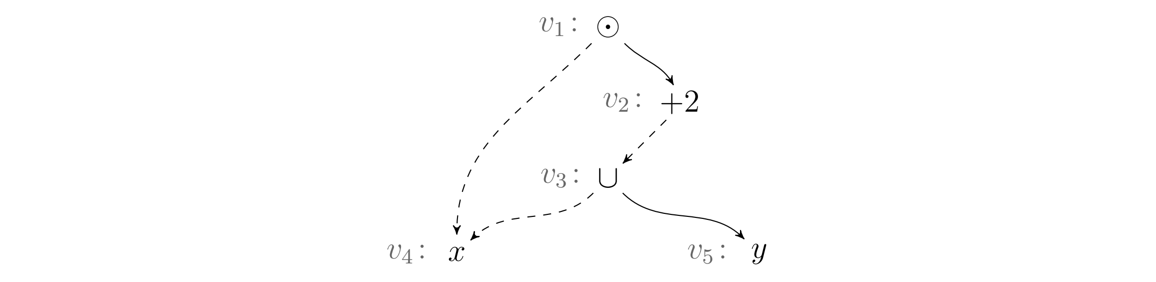


Figure 5.2. An example of a Shift-ECS with output alphabet $\{x, y\}$. We use dashed and solid edges for the left and right partial functions, respectively.

- if $\lambda(v) = \vartheta \in \Omega$, then $[\![\mathcal{D}]\!](v) = \{(\vartheta, 1)\}$;
- if $\lambda(v) = \cup$, then $[\![\mathcal{D}]\!](v) = [\![\mathcal{D}]\!](\ell(v)) \cup [\![\mathcal{D}]\!](r(v))$;
- if $\lambda(v) = \odot$, then $[\![\mathcal{D}]\!](v) = [\![\mathcal{D}]\!](\ell(v)) \cdot [\![\mathcal{D}]\!](r(v))$; and
- if $\lambda(v) \in \mathbb{Z}$, then $[\![\mathcal{D}]\!](v) = \mathsf{sh}([\![\mathcal{D}]\!](\ell(v)), \lambda(v))$.

Example 5.3. *Suppose $\Omega = \{x, y\}$. Consider the Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ where $V = \{v_1, v_2, v_3, v_4, v_5\}$, $\ell(v_1) = v_4$, $r(v_1) = v_2$, $\ell(v_2) = v_3$, $\ell(v_3) = v_4$, $r(v_3) = v_5$, $\lambda(v_1) = \odot$, $\lambda(v_2) = +2$, $\lambda(v_3) = \cup$, $\lambda(v_4) = x$ and $\lambda(v_5) = y$. We show an illustration of this Shift-ECS in Figure 5.2. One can easily check that the sets of words $[\![\mathcal{D}]\!]$ associated to each node are:*

$$
\begin{aligned}
[\![\mathcal{D}]\!](v_4) &= \{(x, 1)\} \\
[\![\mathcal{D}]\!](v_5) &= \{(y, 1)\} \\
[\![\mathcal{D}]\!](v_3) &= \{(x, 1), (y, 1)\} \\
[\![\mathcal{D}]\!](v_2) &= \{(x, 3), (y, 3)\} \\
[\![\mathcal{D}]\!](v_1) &= \{(x, 1)(x, 3), (x, 1)(y, 3)\}.
\end{aligned}
$$

The enumeration algorithm. Given that every node of a Shift-ECS represents a set of strings, we are interested in enumerating them with output-linear delay. Specifically, we focus on the following problem. Let $\mathcal{C}$ be a class of Enumerable Compact Sets with Shifts.

Problem: ShiftECSEnum$[\mathcal{C}]$
Input: a Shift-ECS $\mathcal{D} \in \mathcal{C}$ and a node v of $\mathcal{D}$
Output: Enumerate $[\![\mathcal{D}]\!](v)$.

The plan then is to provide an enumeration algorithm with output-linear delay for ShiftECSEnum$[\mathcal{C}]$ and some helpful class $\mathcal{C}$. A reasonable strategy to enumerate the set $[\![\mathcal{D}]\!](v)$ is to do a traversal on the structure while accumulating the shift values in the path to each leaf. However, to be able to do this without repetitions and output-linear delay, we need to guarantee two conditions: first, that one can obtain every output from $\mathcal{D}$ in only one way and, second, union and shift nodes are *close* to an output node (i.e., a bottom node or a product node), in the sense that we can always reach them in a bounded number of steps. To ensure that these conditions hold, we impose two restrictions on an ECS.

(i) We say that $\mathcal{D}$ is *duplicate-free* if $\mathcal{D}$ satisfies the following two properties: (1) for every union node v it holds that $[\![\mathcal{D}]\!](\ell(v))$ and $[\![\mathcal{D}]\!](r(v))$ are disjoint, and (2) for every product node v and for every $w \in [\![\mathcal{D}]\!](v)$, there exists a unique way to decompose $w = w_1 \cdot w_2$ such that $w_1 \in [\![\mathcal{D}]\!](\ell(v))$ and $w_2 \in [\![\mathcal{D}]\!](r(v))$.

(ii) We define the notion of *k-bounded* Shift-ECS as follows. Given a Shift-ECS $\mathcal{D}$, define the (left) output-depth of a node $v \in V$, denoted by $\mathsf{odepth}_{\mathcal{D}}(v)$, recursively as follows: $\mathsf{odepth}_{\mathcal{D}}(v) = 0$ whenever $\lambda(v) \in \Omega$ or $\lambda(v) = \odot$, and $\mathsf{odepth}_{\mathcal{D}}(v) = \mathsf{odepth}_{\mathcal{D}}(\ell(v)) + 1$ whenever $\lambda(v) \in \{\cup\} \cup \mathbb{Z}$. Then, for $k \in \mathbb{N}$ we say that $\mathcal{D}$ is k-bounded if $\mathsf{odepth}_{\mathcal{D}}(v) \leq k$ for all $v \in V$.

Proposition 5.1. *Fix $k \in \mathbb{N}$. Let $\mathcal{C}_k$ be the class of all duplicate-free and k-bounded Shift-ECSs. Then one can solve the problem* ShiftECSEnum$[\mathcal{C}_k]$ *with output-linear delay and without preprocessing (i.e. constant preprocessing time).*

Proof. Let $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ be a duplicate-free and k-bounded Shift-ECS. The algorithm that we present is a depth-first traversal of the DAG, done in a recursive fashion

Algorithm 4 Enumeration over a node u from some ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$.

```
procedure ENUMERATE($v$)
    $\tau \leftarrow$ CREATE($v$)
    while $\tau$.NEXT $=$ true do
        $\tau$.PRINT($0$)

▷ Bottom node iterator $\tau_\Omega$
procedure CREATE($v$) ▷ Sup. $\lambda(v) \in \Omega$
    $u \leftarrow v$
    hasnext $\leftarrow$ true

procedure NEXT
    if hasnext $=$ true then
        hasnext $\leftarrow$ false
        return true
    return false

procedure PRINT($s$)
    print : $(\lambda(u), 1 + s)$

▷ Product node iterator $\tau_\odot$
procedure CREATE($v$) ▷ Sup. $\lambda(v) = \odot$
    $u \leftarrow v$
    $\tau_\ell \leftarrow$ CREATE($\ell(u)$)
    $\tau_\ell$.NEXT
    $\tau_r \leftarrow$ CREATE($r(u)$)

procedure NEXT
    if $\tau_r$.NEXT $=$ false then
        if $\tau_\ell$.NEXT $=$ false then
            return false
        $\tau_r \leftarrow$ CREATE($r(u)$)
        $\tau_r$.NEXT
    return true

procedure PRINT($s$)
    $\tau_\ell$.PRINT($s$)
    $\tau_r$.PRINT($s$)
▷ Union/$\mathbb{Z}$ node iterator $\tau_{\cup/\mathbb{Z}}$
procedure CREATE($v$)
                        ▷ Sup. $\lambda(v) \in \{\cup\} \cup \mathbb{Z}$
    St $\leftarrow$ push(St, $(v, 0)$ )
    St $\leftarrow$ TRAVERSE(St)
    $\tau \leftarrow$ CREATE(top(St).$u$)

procedure NEXT
    if $\tau$.NEXT $=$ false then
        St $\leftarrow$ pop(St)
        if length(St) $= 0$ then
            return false
        $(u, s) \leftarrow$ top(St)
        if $\lambda(u) \in \{\cup\} \cup \mathbb{Z}$ then
            St $\leftarrow$ TRAVERSE(St)
        $\tau \leftarrow$ CREATE($u$)
        $\tau$.NEXT
    return true

procedure PRINT($s$)
    $(u, s') \leftarrow$ top(St)
    $\tau$.PRINT($s + s'$)

procedure TRAVERSE(St)
    while $\lambda$(top(St).$u$) $\in \{\cup\} \cup \mathbb{Z}$ do
        $(u, s) \leftarrow$ top(St)
        St $\leftarrow$ pop(St)
        if $\lambda(u) \in \mathbb{Z}$ then
            $v \leftarrow \ell(u)$
            $s' \leftarrow s + \lambda(u)$
            St $\leftarrow$ push(St, $(v,\ s')$)
        else
            St $\leftarrow$ push(St, $(r(u), s)$ )
            St $\leftarrow$ push(St, $(\ell(u), s)$ )
    return St
```

to ensure that after retrieving some output w, the next one w' can be printed in $O(k \cdot (|w| + |w'|))$ time. The entire procedure is detailed in Algorithm 4.

To simplify the presentation of the algorithm, we use an *iterator interface* that, given a node v, it contains all information and methods to enumerate the outputs $[\![\mathcal{D}]\!](v)$. Specifically, an iterator τ must implement the following three methods:

$$\text{CREATE}(v) \rightarrow \tau \qquad \tau\,.\text{NEXT} \rightarrow b \qquad \tau\,.\text{PRINT}(s) \rightarrow \emptyset$$

where v is a node, b is either **true** or **false**, and $\emptyset$ means that the method does not return an output. The first method, CREATE, receives a node v and creates an iterator τ of the type of v. We will implement three types of iterators, one for bottom nodes (τ_Ω), one for product nodes ($\tau_\odot$), and one for union and $\mathbb{Z}$-nodes together ($\tau_{\cup/\mathbb{Z}}$). The second method, τ .NEXT, moves the iterator to the next output, returning **true** if, and only if, there is an output to print. Then the last method, τ .PRINT, receives an integer value s, and writes the current output pointed by τ to the output registers after shifting the output by s. We assume that, after creating an iterator τ, one must first call τ .NEXT to move to the first output before printing. Furthermore, if τ .NEXT outputs **false**, then the behavior of τ .PRINT is undefined. Note that one can call τ .PRINT several times, without calling τ .NEXT, and the iterator will write the same output each time in the output registers.

Assume we can implement the iterator interface for each type. Then the procedure ENUMERATE(v) in Algorithm 4 (lines 1-4) shows how to enumerate the set $[\![\mathcal{D}]\!](v)$ by using an iterator τ for v. In the following, we show how to implement the iterator interface for each type and how the size of the next output bounds the delay between two outcomes.

We start by presenting the iterator τ_Ω for a bottom node v (lines 6-18), called a *bottom node iterator*. We assume that each τ_Ω has internally two values, denoted by u and $\mathrm{hasnext}$, where u is a reference to v and $\mathrm{hasnext}$ is a boolean variable. The purpose of a bottom node iterator is only to print $(\lambda(u), 1 + s)$ for some shift s. For this goal, when we create τ_Ω, we initialize u equal to v and $\mathrm{hasnext} = \mathtt{true}$ (lines 8-9). Then, when we call τ_Ω .NEXT for the first time, we swap $\mathrm{hasnext}$ from **true** to **false** and output **true** (i.e., there is one output ready to be returned). Then any following call to τ_Ω .NEXT will be false (lines 11-15). Finally, the τ_Ω .PRINT writes the pair $(\lambda(u), 1 + s)$ to the output registers (lines 17-18). Here, we assume the existence of a method **print** on the RAM model for writing the next entry to the output registers.

For a product node, we present a *product node iterator* $\tau_{\odot}$ in Algorithm 4 (lines 20-37). This iterator receives a product node v with $\lambda(v) = \odot$ and stores a reference of v, called u, and two iterators τ_ℓ and τ_r, for iterating through the left and right nodes $\ell(u)$ and $r(u)$, respectively. The CREATE method initializes u with v, creates the iterators τ_ℓ and τ_r, and calls τ_ℓ.NEXT to be prepared for the first call of $\tau_{\odot}$ .NEXT (lines 21-25). The purpose of $\tau_{\odot}$ .NEXT is to fix one output for the left node $\ell(u)$ and iterate over all outputs of $r(u)$ (lines 27-33). When we stop enumerating all outputs of $[\![\mathcal{D}]\!](r(u))$, we move to the next output of τ_ℓ, and iterate again over all $[\![\mathcal{D}]\!](r(u))$ (lines 29-32). For printing, we recursively call first the printing method of τ_ℓ, and then the one of τ_r (lines 35-37).

The most involved case is the *union/$\mathbb{Z}$ node iterator* $\tau_{\cup/\mathbb{Z}}$ (lines 38-72). This iterator receives a node v of one of two types, either a union node, or a $\mathbb{Z}$-node. It keeps a *stack* St and an iterator τ. The elements in the stack are pairs (u, s) where u is a node and s is an integer. We assume the standard implementation of a stack with the native methods push, pop, top, and length: the first three define the standard operations over stacks, and length counts the elements in a stack. The purpose of the stack is to perform a *depth-first-search* traversal of all union and $\mathbb{Z}$ nodes below v, reaching all possible output nodes u such that there is a path of only union and $\mathbb{Z}$ nodes between v and u. At every point, if an element (u, s) is in the stack, then s is equal to the sum of all $\mathbb{Z}$ nodes in the path from v to u. If the top node of St is a pair (u, s) such that u is an output node, then τ is an iterator for u, which enumerates all their outputs. If $p = (u, s)$, we will use the notation $p.u$ to refer to u.

In order to perform the *depth-first-search* traversal of union and $\mathbb{Z}$ nodes, we use the auxiliary method TRAVERSE(St) (lines 61-72). While the node u at the top of St is a union or a shift node, we pop the top pair (u, s) from St. If u is a shift node (lines 65-68), we push the pair (v, s') in the stack where v is the node $\ell(u)$ pointed by u and s' is the sum of the current shift s with the shift $\lambda(u)$. Otherwise, if u is a union node (lines 69-71), we first

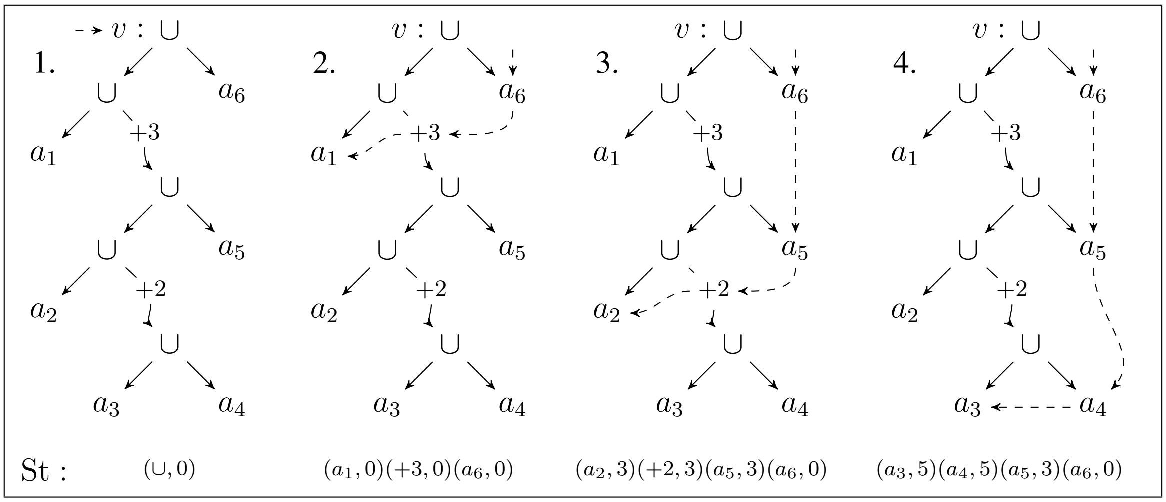


Figure 5.3. Evolution of the stack St (written on the bottom and represented by dashed arrows) for an iterator over the node v in the figure. The underlying ECS is made of union nodes, two $\mathbb{Z}$ nodes, and six bottom nodes. The first figure is St after calling $\mathrm{St} \leftarrow \mathsf{push}(\mathrm{St}, (v, 0))$, the second is after calling $\mathrm{St} \leftarrow \textsc{Traverse}(\mathrm{St})$. The last two figures represent successive calls to $\mathsf{pop}(\mathrm{St}), \mathrm{St} \leftarrow \textsc{Traverse}(\mathrm{St})$.

push the right pair $(r(u), s)$ followed by the left pair $(\ell(u), s)$ into the stack. The while-loop will eventually reach an output node at the top of the stack and end. It is important to note that $\textsc{traverse}(\mathrm{St})$ takes $\mathcal{O}(k)$ steps, given that the ECS is k-bounded. Then if k is fixed, the TRAVERSE procedure takes constant time. In Figure 5.3, we illustrate the evolution of a stack St inside a union node iterator when we call $\textsc{traverse}(\mathrm{St})$ several times.

The methods of a union/$\mathbb{Z}$ node iterator $\tau_{\cup/\mathbb{Z}}$ are then straightforward. For CREATE (lines 39-43), we push $(v, 0)$ and then traverse St, finding the first leftmost output node from v (lines 41-42). Then we build the iterator τ of this output node for being ready to start enumerating their outputs (line 43). For NEXT, we consume all outputs by calling τ .NEXT (line 46). When there are no more outputs, we pop the top node from St and check if the stack is empty or not (lines 47-48). If this is the case, there are no more outputs and we output **false**. Otherwise, if St is non-empty but the top pair (u, s) of St contains a union node, then we apply the TRAVERSE method for finding the leftmost output node from u (lines 50-52). When the procedure is done, we know that the node in the top pair is an output node, and then we create an iterator and move to its first output (lines 53-54). For PRINT(s), we see the pair (u, s') at the top, where we remind that s' represents the sum of all $\mathbb{Z}$ nodes on the way to u (line 58), and u is assumed to be an output node. Then, we call the print method of τ which is ready to write the current output, and over which we add the value $s + s'$ (line 59).

In order to prove the correctness of the enumeration procedure, one can verify that ENUMERATE(v) in Algorithm 4 enumerates all the outputs in the set $[\![\mathcal{D}]\!](v)$ one by one, and without repetitions, which follows from the fact that $\mathcal{D}$ is duplicate-free. To bound the delay between outputs, the fact that $\mathcal{D}$ is k-bounded implies that the delay is bounded by $\mathcal{O}(k \cdot |w_0|)$ if w_0 is the first output, or $O(k \cdot (|w| + |w'|))$ if w and w' are the previous and next outputs, respectively. Specifically:

- CREATE(v) takes time $\mathcal{O}(k \cdot |w_0|)$,
- NEXT takes time $\mathcal{O}(k \cdot |w_0|)$ for the first call, and $\mathcal{O}(k \cdot (|w| + |w'|))$ for the next call, and
- PRINT(s) takes time $\mathcal{O}(k \cdot |w'|)$ where w' is the current output to be printed.

Overall, ENUMERATE(v) in Algorithm 4 requires $O(k \cdot (|w| + |w'|))$ delay to write the next output w' in the output register, after printing the previous output w.

We end by pointing out that the existence of an enumeration algorithm $\mathcal{E}$ with delay $O(k \cdot (|w| + |w'|))$ between any consecutive outputs w and w', implies the existence of an enumeration algorithm $\mathcal{E}'$ with output-linear delay as defined in Section 5.1. We start noting that k is a fixed value and then the delay of $\mathcal{E}$ only depends on $|w| + |w'|$. For depending only on the next output w', one can perform the following strategy for $\mathcal{E}'$: start by running $\mathcal{E}$, enumerate the first output w_0, advance $k \cdot |w_0|$ more steps of $\mathcal{E}$, and stop. Then continue running $\mathcal{E}$, enumerate the next output w_1, advance $k \cdot |w_1|$ more steps, and stop[1]. By repeating this enumeration process, one can verify that the delay between the

[1] If advancing $k \cdot |w_1|$ more steps requires printing part of the next outputs $w_2, w_3, \ldots$, we could store these outputs in some temporary registers of the RAM model to retrieve them later.

i-th output w_i and the $(i+1)$-th output w_{i+1} is $O(|w_{i+1}|)$. Therefore, $\mathcal{E}'$ has output-linear delay. □

Operations. The next step is to provide a set of operations that allow extending a Shift-ECS $\mathcal{D}$ in a way that maintains k-boundedness. Fix a Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$. Then for any ♌$\in \Omega$, $v_1, \ldots, v_4, v \in V$ and $k \in \mathbb{Z}$, we define the operations:

$$\begin{aligned} \mathsf{add}(\text{♌}) &\to v' \qquad & \mathsf{prod}(v_1, v_2) &\to v' \\ \mathsf{union}(v_3, v_4) &\to v' \qquad & \mathsf{shift}(v, k) \quad &\to v' \end{aligned}$$

such that $[\![\mathcal{D}]\!](v') := \{(\text{♌}, 1)\}$; $[\![\mathcal{D}]\!](v') := [\![\mathcal{D}]\!](v_1) \cdot [\![\mathcal{D}]\!](v_2)$; $[\![\mathcal{D}]\!](v') := [\![\mathcal{D}]\!](v_3) \cup [\![\mathcal{D}]\!](v_4)$; and $[\![\mathcal{D}]\!](v') := \mathsf{sh}([\![\mathcal{D}]\!](v), k)$, respectively. Here we assume that the union and prod respect properties (1) and (2) of a duplicate-free Shift-ECS, namely, $[\![\mathcal{D}]\!](v_3)$ and $[\![\mathcal{D}]\!](v_4)$ are disjoint and, for every $w \in [\![\mathcal{D}]\!](v_1) \cdot [\![\mathcal{D}]\!](v_2)$, there exists a unique way to decompose $w = w_1 \cdot w_2$ such that $w_1 \in [\![\mathcal{D}]\!](v_1)$ and $w_2 \in [\![\mathcal{D}]\!](v_2)$.

Strictly speaking, each operation above should receive as input the data structure $\mathcal{D}$, and output a fresh node v' plus a new data structure $\mathcal{D}' = (\Omega, V', \ell', r', \lambda')$ such that $\mathcal{D}'$ is an extension of $\mathcal{D}$, namely, $\mathsf{obj} \subseteq \mathsf{obj}'$ for every $\mathsf{obj} \in \{V, \ell, r, \lambda\}$ and $v' \in V' \setminus V$. Note that we assume that each operation can only extend the data structure with new nodes and that old nodes are immutable after each operation. For simplification, we will not explicitly refer to $\mathcal{D}$ on the operations above, although they modify $\mathcal{D}$ directly by adding new nodes.

To define the above operations, we impose further restrictions on the structure below the operations' input nodes to ensure k-boundedness. Towards this goal, we introduce the notion of *safe nodes*. We say that a node $v \in V$ is *safe* if v is a shift node and either $\ell(v)$ is an output node (i.e., a bottom or product node), or $u = \ell(v)$ is an union node, $\mathsf{odepth}_{\mathcal{D}}(u) = 1$, and $r(u)$ is a shift node with $\mathsf{odepth}_{\mathcal{D}}(r(u)) \leq 2$. In other words, v is safe if it is a shift node over an output node or over a union node with an output on the left and a shift node on the right, whose output depth is less or equal to 2. The trick then is to show that all operations over Shift-ECSs receive only safe nodes and always output safe nodes. As we will see, safeness will be enough to provide a light structural restriction on the operations' input nodes in order to maintain k-boundedness after each operation.

Next, we show how to implement each operation assuming that every input node is safe. In fact, the cases of add and shift are straightforward. For $\mathsf{add}(\vartheta) \to v'$ we extend $\mathcal{D}$ with two fresh nodes v' and u such that $\lambda(u) = \vartheta$, $\lambda(v') = 0$, and $\ell(v') = u$. In other words, we hang a fresh 0-shift node v' over a fresh ϑ-node u, and output v'. For $\mathsf{shift}(v, k) \to v'$, add the fresh node v' to $\mathcal{D}$, and set $\ell(v') = \ell(v)$ and $\lambda(v') = \lambda(v) + k$. One can easily check that in both cases the node v' represents the desired set, is safe, and k-boundedness is preserved.

To show how to implement $\mathsf{prod}(v_1, v_2) \to v'$, recall that v_1 and v_2 are safe and, in particular, both are shift nodes. Then we need to extend $\mathcal{D}$ with fresh nodes v', v'', and v''' such that $\ell(v') = v''$, $\ell(v'') = \ell(v_1)$, $r(v'') = v'''$, $\ell(v''') = \ell(v_2)$, $\lambda(v') = \lambda(v_1)$, $\lambda(v'') = \odot$ and $\lambda(v''') = \lambda(v_2) - \lambda(v_1)$. Figure 5.4(a) shows a diagram of this gadget. One can easily check that v' represents the product of v_1 and v_2, v' is safe, and the new version of $\mathcal{D}$ is k-bounded whenever $\mathcal{D}$ is also k-bounded.

The last operation is $\mathsf{union}(v_3, v_4) \to v'$. The strategy is then to prove that if v_3 and v_4 are safe nodes, then we can implement the operator and produce a safe node v'. Let us define v' as follows:

- If at least one among $\ell(v_3)$ and $\ell(v_4)$ is an output node, assume without loss of generality that it is $\ell(v_3)$. We extend $\mathcal{D}$ with nodes v', v'' and v''', where $\ell(v') = v''$, $\ell(v'') = \ell(v_3)$, $r(v'') = v'''$, $\ell(v''') = \ell(v_4)$, $\lambda(v') = \lambda(v_3)$, $\lambda(v'') = \cup$ and $\lambda(v''') = \lambda(v_4) - \lambda(v_3)$. This construction is identical to the prod construction shown in Figure 5.4(a), except replacing v_1 and v_2 by v_3 and v_4 respectively, and $\lambda(v'')$ from $\odot$ to $\cup$.
- When both $u_3 = \ell(v_3)$ and $u_4 = \ell(v_4)$ are union nodes, let $k_1' = \lambda(r(u_3))$ and let $k_2' = \lambda(r(u_4))$. We extend $\mathcal{D}$ with fresh nodes $v', v_1', v_2', v_3', v_4', v_5'$ and v_6'. Define

$\lambda(v') = k_1$, $\lambda(v_1') = \cup$, $\lambda(v_2') = k_2 - k_1$, $\lambda(v_3') = \cup$, $\lambda(v_4') = k_1 + k_1' - k_2$, $\lambda(v_5') = \cup$ and $\lambda(v_6') = k_2 + k_2' - k_1 - k_1'$. Then define $\ell(v') = v_1'$, $\ell(v_1') = \ell(u_3)$, $r(v_1') = v_2'$, $\ell(v_2') = v_3'$, $\ell(v_3') = \ell(u_4)$, $r(v_3') = v_4'$, $\ell(v_4') = v_5'$, $\ell(v_5') = \ell(r(u_3))$, $r(v_5') = v_6'$ and $\ell(v_6') = \ell(r(u_4))$. We show an illustration of this gadget in Figure 5.4(b).

We can prove that the last construction has several interesting properties. First, one can check that $[\![\mathcal{D}]\!](v') = [\![\mathcal{D}]\!](v_3) \cup [\![\mathcal{D}]\!](v_4)$ since each shift value is constructed so that the accumulated shift value from v' to each node remains unchanged. Thus, the semantics is well-defined. Second, union can be computed in constant time in $|\mathcal{D}|$ given that we only need to add a fixed number of fresh nodes. Furthermore, the produced node v' is safe, even though some of the new nodes are not necessarily safe. Finally, the new $\mathcal{D}$ is 3-bounded whenever $\mathcal{D}$ is 3-bounded. This is straightforward to see for the case when $\ell(v_3)$ (or $\ell(v_4)$) is an output node. To see this for the second case, we first have to notice that $\ell(u_3)$ and $\ell(u_4)$ are output nodes, and that $\mathsf{odepth}(\ell(r(u_3))) \leq 1$ and $\mathsf{odepth}(\ell(r(u_4))) \leq 1$. We can check the depth of each node going from the bottom to the top: $\mathsf{odepth}(v_6') \leq 2$, $\mathsf{odepth}(v_5') \leq 2$, $\mathsf{odepth}(v_4') \leq 3$, $\mathsf{odepth}(v_3') \leq 1$, $\mathsf{odepth}(v_2') \leq 2$, $\mathsf{odepth}(v_1') \leq 1$ and $\mathsf{odepth}(v') \leq 2$.

By the previous discussion, if we start with a Shift-ECS $\mathcal{D}$ which is 3-bounded (in particular, empty) and we apply the add, prod, union and shift operators between safe nodes (which also produce safe nodes), then the result is 3-bounded as well. Furthermore, the data structure is fully-persistent (Driscoll, Sarnak, Sleator, & Tarjan, 1986b): for every node v in $\mathcal{D}$, $[\![\mathcal{D}]\!](v)$ is immutable after each operation. Finally, by Proposition 5.1, the result can be enumerated with output-linear delay.

Theorem 5.2. *The operations* add, prod, union *and* shift *take constant time and are fully persistent. Furthermore, if we start from an empty Shift-ECS $\mathcal{D}$ and apply these operations over safe nodes, the result node v' is always a safe node and the set $[\![\mathcal{D}]\!](v)$ can be enumerated with output-linear delay (without preprocessing) for every node v.*

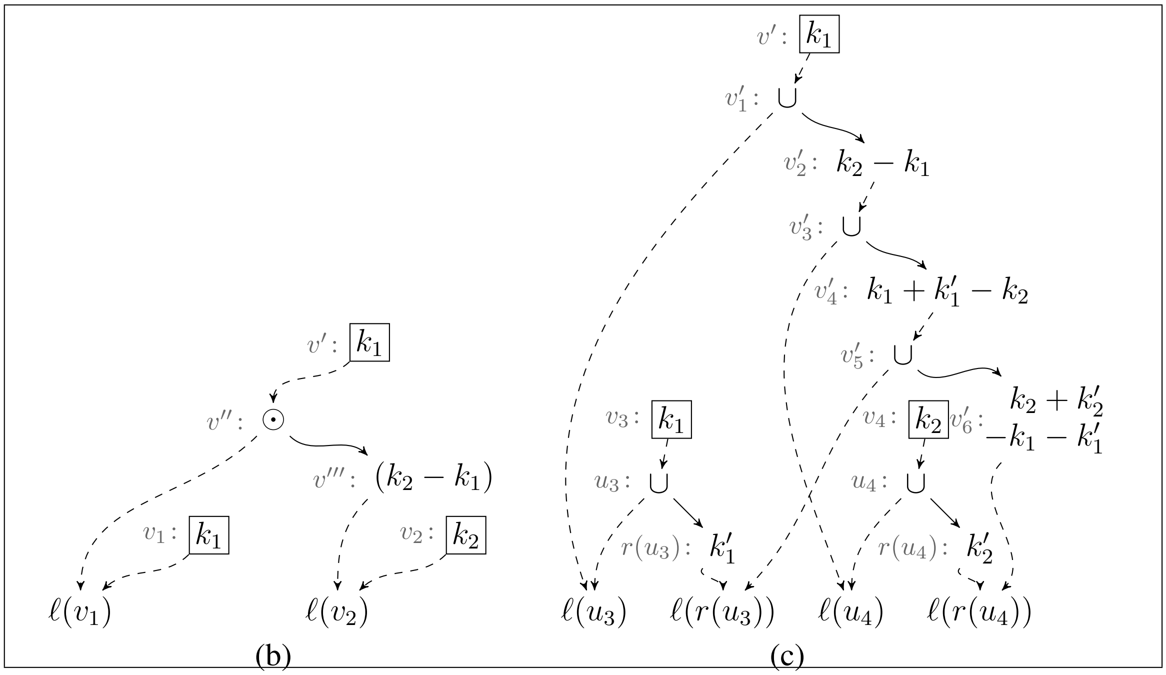


Figure 5.4. (a) Gadget for $\mathsf{prod}(\mathcal{D}, v_1, v_2, k)$. (b) Gadget for $\mathsf{union}(\mathcal{D}, v_3, v_4)$. We use dashed and solid edges for the left and right mappings, respectively. Node names are in grey at the left of each node. Nodes in square boxes are the input and output nodes of each operation.

The empty- and ε-nodes. The last step of constructing our model of Shift-ECS is the inclusion of two special nodes that produce the empty set and the empty string, called empty- and ε-nodes, respectively.

We start with the empty node, which is easier to incorporate into a Shift-ECS. Consider a special node $\bot$ and include it on every Shift-ECS $\mathcal{D}$, such that $[\![\mathcal{D}]\!](\bot) = \emptyset$. Then extend the operations prod, union, and shift accordingly to the empty set, namely, $\mathsf{prod}(v_1, v_2) \to \bot$ whenever v_1 or v_2 is equal to $\bot$, $\mathsf{union}(v, \bot) = \mathsf{union}(\bot, v) \to v$, and $\mathsf{shift}(\bot, k) \to \bot$ for every nodes v_1, v_2, v, and $k \in \mathbb{Z}$. It is easy to check that one can include the $\bot$-node into Shift-ECSs without affecting the guarantees of Theorem 5.2.

The other special node is the ε-node. Let ε denote a special node, included on every Shift-ECS $\mathcal{D}$, such that $[\![\mathcal{D}]\!](\varepsilon) = \{\varepsilon\}$. With these new nodes in a Shift-ECS, we need to revise our notions of output-depth, duplicate-free, and k-boundedness to change the enumeration algorithm, and to extend the operations add, prod, union, and shift over so-called ε-safe nodes (i.e., the extension of safe nodes with ε).

First, we add the condition that for any Shift-ECS $\mathcal{D}$ the ε-node is either parentless (and thus disconnected to the rest of the structure), or if it has any parents, then they must be parentless union nodes, and the ε-node must be the left child of each. We call this the *ε-condition*. We will see that any node that might be the result of applying the operations add, prod, union and shift is equivalent to a node in a Shift-ECS $\mathcal{D}$ that satisfies this.

The notions of output-depth, duplicate-free, and k-boundedness are mostly unchanged in this setting: If v is the ε-node, then $\mathsf{odepth}(v) = 0$, and if v is a union node with the ε-node as its left child, $\mathsf{odepth}(v) = 1$. No other node has the ε-node as a descendant, so the definition of output-depth remains the same; and the definitions of duplicate-free and k-bounded remain unchanged as well. By assuming the ε-condition, one can see that enumeration remains identical as before, except now the ε-node might be seen once as the left child of the root. Then we can state the following as a corollary of Proposition 5.1:

Corollary 5.1. *Fix $k \in \mathbb{N}$. Let $\mathcal{C}_k$ be the class of all duplicate-free and k-bounded Shift-ECSs that satisfy the ε-condition. Then one can solve the problem* $\textsc{ShiftECSEnum}[\mathcal{C}_k]$ *with output-linear delay and without preprocessing (i.e. constant preprocessing time).*

We now define the notion of *ε-safe* nodes. These are nodes v which satisfy one of three conditions: (1) v is the ε-node, (2) v is safe and none of its descendants is the ε-node, and (3) v is a union node, its left child is the ε-node, and its right child is a safe node for which none of its descendants is the ε-node. To guide the constructions, we call these conditions 1, 2 and 3. These conditions will allow us to use the constructions already given for prod, union, and shift over safe nodes, such as node v, in the case of condition 2, or $r(v)$, in the case of condition 3. The rest of this proof will be extending these operations to work over ε-safe nodes, and check that they can be done in constant time, return an ε-safe node, maintain the ε-condition in the Shift-ECS, and satisfy the specifications of each operation (i.e., that for the resulting node v' the set $[\![\mathcal{D}]\!](v')$ contains what it should).

We define the operation add the same as in a Shift-ECS without ε. The operation $\mathsf{add}(\varepsilon)$ is not defined since the ε-node is always part of $\mathcal{D}$.

The prod case is somewhat more involved. First, assume that some node among v_1 and v_2 is the ε-node (condition 1). Without loss of generality, assume that $v_1 = \varepsilon$. If v_2 is the ε-node as well, we simply return v_1; and if v_2 is not the ε-node (conditions 2 or 3), we simply return v_2. The requirements of the construction follow trivially. Now, assume that both v_1 and v_2 satisfy condition 2. For this case we simply return $\mathsf{prod}(v_1, v_2)$ as it was defined for a Shift-ECS without ε. Lastly, assume at least one node among v_1 and v_2 satisfies condition 3 and none of them are the ε-node. These cases are depicted in Figure 5.5. We describe them formally as follows, assuming that v' is the output and using the operations union and prod as they were defined for a Shift-ECS without ε.

- If v_1 satisfies condition 2 and v_2 satisfies condition 3, define $v'' \leftarrow \mathsf{prod}(v_1, r(v_2))$ and $v' \leftarrow \mathsf{union}(v_1, v'')$.
- If v_1 satisfies condition 3 and v_2 satisfies condition 2, define $v'' \leftarrow \mathsf{prod}(r(v_1), v_2)$ and $v' \leftarrow \mathsf{union}(v'', v_2)$.
- If both v_1 and v_2 satisfy condition 3, define $v'' \leftarrow \mathsf{prod}(r(v_1), r(v_2))$, $v_3 \leftarrow \mathsf{union}(v'', r(v_2))$, and $v_4 \leftarrow \mathsf{union}(r(v_1), v_3)$. Lastly, define $\lambda(v') = \cup$, $\ell(v') = \varepsilon$, and $r(v') = v_4$.

One can verify that these constructions work as expected, namely, $[\![\mathcal{D}']\!](v) = [\![\mathcal{D}]\!](v_1) \cdot [\![\mathcal{D}]\!](v_2)$. Also, if v_1 and v_2 are ε-safe, then v' is ε-safe as well. Also, each construction does a fixed number of steps, so they take constant time.

Similarly, we address the operation $\mathsf{union}(v_3, v_4) \to v'$ by considering each case separately.

- Assume v_3 is the ε-node (condition 1). (1) If v_4 also is the ε-node, then we simply return $v' \leftarrow \varepsilon$. (2) If v_3 satisfies condition 2, let v' be a union node such that $\ell(v') = \varepsilon$ and $r(v') = v_4$. It can be seen that v' satisfies condition 3 and that its ε-safe. (3) If v_4 satisfies condition 3, we simply return $v' \leftarrow v_4$.

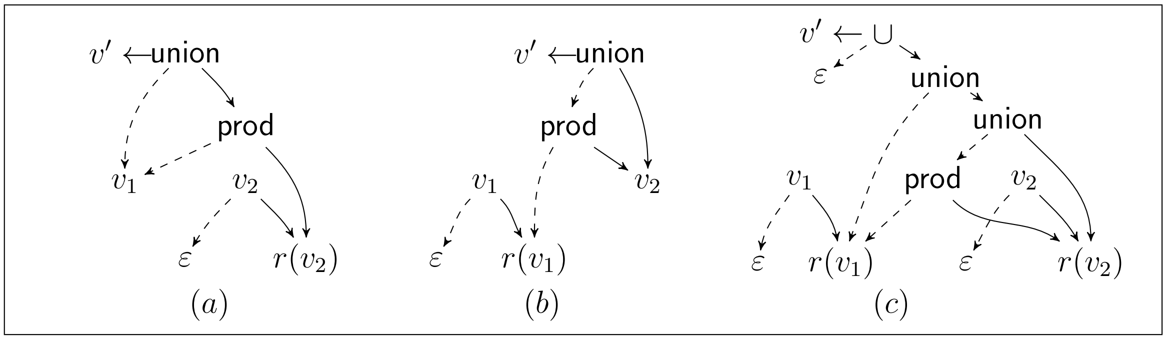


Figure 5.5. Gadgets for prod as defined for a Shift-ECS with the ε-node.

- Assume v_3 satisfies condition 2. (1) If v_4 is the ε-node, we return $v' \leftarrow \mathsf{union}(v_4, v_3)$ using the construction for the case (2) in the previous item. (2) If v_4 satisfies condition 2, then we return $v' \leftarrow \mathsf{union}(v_3, v_4)$ as it was defined for Shift-ECSs without ε. (3) If v_4 satisfies condition 3, let $v'' \leftarrow \mathsf{union}(v_3, r(v_4))$, and let v' be a union node such that $\ell(v') = \varepsilon$ and $r(v) = v''$. Again, it can be seen that v' is ε-safe.
- Assume v_3 satisfies condition 3. (1) If v_4 is the ε-node, we simply return $v' \leftarrow \varepsilon$. (2) If v_4 satisfies condition 2, we follow the construction given in case (3) of the previous item. (3) If v_4 satisfies condition 3, let $v'' \leftarrow \mathsf{union}(r(v_3), r(v_4))$ and let v' be a union node such that $\ell(v') = \varepsilon$ and $r(v) = v''$. It can be seen that v' is ε-safe.

Clearly, in each of these cases above we have that $[\![\mathcal{D}]\!](v') = [\![\mathcal{D}]\!](v_3) \cup [\![\mathcal{D}]\!](v_4)$, and if v_3 and v_4 are ε-safe, then v' is ε-safe as well. In addition, they all take constant time and in each case the new version of $\mathcal{D}$ satisfies the ε-condition.

The operation $\mathsf{shift}(v, k)$ is defined as follows. If v is the ε-node, we return $v' \leftarrow \varepsilon$. If v satisfies condition 2, we return $v' \leftarrow \mathsf{shift}(v, k)$ as it was defined for Shift-ECSs without ε. If v satisfies condition 3, let $v'' \leftarrow \mathsf{shift}(r(v), k)$ and let v' be a union node such that $\ell(v') = \varepsilon$ and $r(v) = v''$. It can be seen that v' is ε-safe.

By the previous constructions, we can conclude that the operations add, prod, union and shift, when done over ε-safe nodes, can be done in constant time, return an ε-safe node, maintain the ε-condition in the Shift-ECS, and satisfy the specifications of each operation. Furthermore, by virtue of Corollary 5.1, the set $[\![D]\!](v)$ can be enumerated with output-linear delay for each node v in $\mathcal{D}$.

Theorem 5.3. *The operations* add, prod, union *and* shift *over Shift-ECS extended with empty- and ε-nodes take constant time. Furthermore, if we start from an empty Shift-ECS $\mathcal{D}$ and apply* add, prod, union, *and* shift *over ε-safe nodes, the resulting node v' is always an ε-safe node, and the set $[\![\mathcal{D}]\!](v)$ can be enumerated with output-linear delay without preprocessing for every node v.*

For the rest of the chapter, we assume that a Shift-ECS is a tuple $\mathcal{D} = (\Omega, V, \ell, r, \lambda, \perp, \varepsilon)$ where we define $\Omega, V, \ell, r, \lambda$ as before, and $\perp, \varepsilon \in V$ are the empty and ε nodes, respectively. Further, we assume that ℓ, r, and λ are extended accordingly, namely, $\ell(v)$ and $r(v)$ are not defined whenever $v \in \{\perp, \varepsilon\}$, and $\lambda : V \to \Omega \cup \mathbb{Z} \cup \{\cup, \odot, \perp, \varepsilon\}$ such that $\lambda(v) = \perp$ ($\lambda(v) = \varepsilon$) iff $v = \perp$ ($v = \varepsilon$, respectively).

5.3. Evaluation of annotated automata over SLP-compressed strings

This section shows our algorithm for evaluating an annotated automaton over an SLP-compressed document. This evaluation is heavily inspired by the preprocessing phase in (Schmid & Schweikardt, 2021), as it primarily adapts the algorithm to the Shift-ECS data structure. In a nutshell, we keep matrices of Shift-ECS nodes, where each matrix represents the outputs of all partial runs of the annotated automaton over fragments of the compressed strings. We extend the operations of Shift-ECS over matrices of nodes, which will allow us to compose matrices, and thus compute sequences of compressed strings. Then the algorithm proceeds in a dynamic programming fashion, where matrices are computed bottom-up for each non-terminal symbol. Finally, the matrix that corresponds to the start symbol of the SLP will contain all the outputs. The result of this process is that each matrix entry succinctly represents an output set that can enumerated with output-linear delay.

Matrices of nodes. The main ingredient for the evaluation algorithm are matrices of nodes for encoding partial runs of annotated automata. To formalize this notion, fix an unambiguous AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, I, F)$ and a Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda, \bot, \varepsilon)$. We define a *partial run* ρ of $\mathcal{A}$ over a document $d = a_1 a_2 \dots a_n \in \Sigma^*$ as a sequence

$$\rho \;:=\; p_1 \xrightarrow{b_1} \dots \xrightarrow{b_n} p_{n+1}$$

such that $p_1 \in Q$, and for each $i \in [1, n]$ either $b_i = a_i$ and $(q_i, a_i, q_{i+1}) \in \Delta$, or $b_i = (a_i, \vartheta)$ and $(q_i, (a_i, \vartheta), q_{i+1}) \in \Delta$. Additionally, we say that the partial run ρ is from state p to state q if $p_1 = p$ and $p_{n+1} = q$. In other words, partial runs are almost equal to runs, except they can start and end at any states p and q, respectively.

For the algorithm, we use the set of all $Q \times Q$ matrices where entry $M[p, q]$ is a node in V for every $p, q \in Q$. Each node $M[p, q]$ represents all annotations of partial runs from state p to state q, which can be enumerated with output-linear delay by Theorem 5.3. Further, $M[p, q] = \bot$ represents that there is no run, and $M[p, q] = \varepsilon$ that there is a single run without outputs (i.e., a run that produces the ε output).

To combine matrices over $\mathcal{D}$-nodes, we define two operations. The first operation is the *matrix multiplication* over the semiring $(2^{(\Omega \times \mathbb{Z})^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$ but represented over $\mathcal{D}$. Formally, let $Q = \{q_1, \dots, q_m\}$ with $m = |Q|$. Then, for two $m \times m$ matrices M_1 and M_2, we define $M_1 \otimes M_2$ such that for every $p, q \in Q$:

$$(M_1 \otimes M_2)[p, q] := \mathsf{union}_{i=1}^{m} \Big(\mathsf{prod}\big(M_1[p, q_i], M_2[q_i, q]\big)\Big)$$

where $\mathsf{union}_{i=1}^{m} E_i \;:=\; \mathsf{union}(\dots \mathsf{union}(\mathsf{union}(E_1, E_2), E_3) \dots, E_m)$. That is, the node $(M_1 \otimes M_2)[p, q]$ represents the set $\bigcup_{i=1}^{m} \big([\![\mathcal{D}]\!](M_1[p, q_i]) \cdot [\![\mathcal{D}]\!](M_2[q_i, q]) \big)$.

The second operation for matrices is the extension of the *shift operation*. Formally, $\mathsf{shift}(M, k)[p, q] := \mathsf{shift}(M[p, q], k)$ for a matrix M, $k \in \mathbb{Z}$, and $p, q \in Q$. Since each operation over $\mathcal{D}$ takes constant time, overall multiplying M_1 with M_2 takes time $\mathcal{O}(|Q|^3)$ and shifting M by k takes time $\mathcal{O}(|Q|^2)$.

Algorithm 5 The enumeration algorithm of an unambiguous AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, I, F)$ over an SLP $S = (N, \Sigma, R, S_0)$.

```
 1: procedure EVALUATION($\mathcal{A}$, $S$)
 2:     Initialize $\mathcal{D}$ as an empty Shift-ECS
 3:     NONTERMINAL($S_0$)
 4:     $v \leftarrow \bot$
 5:     for each $p \in I, q \in F$ do
 6:         $v \leftarrow$ union($v, M_{S_0}[p, q]$)
 7:     ENUMERATE($v, \mathcal{D}$)
 8: procedure TERMINAL($a$)
 9:     $M_a \leftarrow \{[p, q] \to \bot \mid p, q \in Q\}$
10:     for each $(p, (a, \vartheta), q) \in \Delta$ do
11:         $M_a[p, q] \leftarrow$ union($M_a[p, q]$, add($\vartheta$))
12:     for each $(p, a, q) \in \Delta$ do
13:         $M_a[p, q] \leftarrow$ union($M_a[p, q]$, $\varepsilon$)
14:     $\mathrm{len}_a \leftarrow 1$
15: procedure NONTERMINAL($X$)
16:     $M_X \leftarrow \{[p, q] \to \bot \mid p, q \in Q, p \neq q\} \cup$
                $\{[p, q] \to \varepsilon \mid p, q \in Q, p = q\}$
17:     $\mathrm{len}_X \leftarrow 0$
18:     for $i = 1$ to $|R(X)|$ do
19:         $Y \leftarrow R(X)[i]$
20:         if $M_Y$ is not defined then
21:             if $Y \in \Sigma$ then
22:                 TERMINAL($Y$)
23:             else
24:                 NONTERMINAL($Y$)
25:         $M_X \leftarrow M_X \otimes$ shift($M_Y, \mathrm{len}_X$)
26:         $\mathrm{len}_X \leftarrow \mathrm{len}_X + \mathrm{len}_Y$
```

The algorithm. We present the evaluation algorithm for the SLPENUM problem in Algorithm 5. As expected, the main procedure EVALUATION receives as input an unambiguous annotated automaton $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, I, F)$ and an SLP $S = (N, \Sigma, R, S_0)$, and enumerates all outputs in $[\![\mathcal{A}]\!](\mathrm{doc}(S))$. To simplify the notation, in Algorithm 5 we assume that $\mathcal{A}$ and S are globally defined, and we can access them in any subprocedure. Similarly, we use a Shift-ECS $\mathcal{D}$, and matrix M_X and integer len_X for every $X \in N \cup \Sigma$, which can globally be accessed at any place as well.

The main purpose of the algorithm is to compute M_X and len_X recursively. On one hand, M_X is a $Q \times Q$ matrix where each node entry $M_X[p, q]$ represents all outputs of partial runs from p to q. On the other hand, len_X is the length of the string $R^*(X)$ (i.e., the string produced from X). Both M_X and len_X start undefined, and we compute them recursively, beginning from the non-terminal symbol S_0 and by calling the method NONTERMINAL(S_0) (line 3). After M_{S_0} was computed, we can retrieve the set $[\![\mathcal{A}]\!](S)$ by taking the union of all partial run's outputs from an initial state $p \in I$ to a state $q \in F$, and storing it in node v (lines 4-6). Finally, we can enumerate $[\![\mathcal{A}]\!](S)$ by enumerating all outputs represented by v (line 7).

The workhorses of the evaluation algorithm are procedures NONTERMINAL and TERMINAL in Algorithm 5. The former computes matrices M_X recursively whereas the latter is in charge of the base case M_a for a terminal $a \in \Sigma$. For computing the base case, we can start with M_a with all entries equal to the empty node $\perp$ (line 9). Then if there exists a read-annotate transition $(p, (a, \text{ծ}), q) \in \Delta$, we add an output node ծ to $M_a[p, q]$, by making the union between the current node at $M_a[p, q]$ with the node $\mathtt{add}(\text{ծ})$ (line 11). Also, if a read transition $(p, a, q) \in \Delta$ exists, we do the same but with the ε-node (line 13). Finally, we set the length of len_a to 1, and we have covered the base case.

For the recursive case (i.e., procedure NONTERMINAL(X)), we start with a sort of "identity matrix" M_X where all entries are set up to the empty-node except the ones where $p = q$ that are set up to the ε-node, and the value $\mathrm{len}_X = 0$ (lines 16-17). Then we iterate sequentially over each symbol Y of $R(X)$, where we use $R(X)[i]$ to denote the i-th symbol of $R(X)$ (lines 18-19). If M_Y is not defined, then we recursively compute TERMINAL(Y) or NONTERMINAL(Y) depending on whether Y is in Σ or not, respectively (lines 20-24). The matrix M_Y is memorized (by having the check in line 20 to see if it is defined or not) so we need to compute it at most once. After we have retrieved M_Y, we can compute all outputs for $R(X)[1] \ldots R(X)[i]$ by multiplying the current version of M_X (i.e., the outputs of $R(X)[1] \ldots R(X)[i-1]$), with the matrix M_Y shifted by the current length len_X (line 25). Finally, we update the current length of X by adding len_Y (line 26).

Theorem 5.4. *Algorithm 5 enumerates the set $[\![\mathcal{A}]\!](S)$ correctly for every unambiguous AnnA $\mathcal{A}$ and every SLP-compressed document S, with output-linear delay and after a preprocessing phase that takes time $\mathcal{O}(|\mathcal{A}| + |S| \times |Q|^3)$.*

PROOF. The correctness of the algorithm will follow after proving that for each M_X, for an $X \in \Sigma \cup N$ that is reachable from S_0, the set $[\![\mathcal{D}]\!](M_X[p, q])$ contains the annotations of all partial runs of $\mathcal{A}$ over $\mathrm{doc}(X)$ that start on p and end on q. First, let $X = a \in \Sigma$. This case can be quickly verified by inspecting the procedure TERMINAL(a), as for each index, the set $[\![\mathcal{D}]\!](M_a[p, q])$ contains ε, and the annotations ծ in the pair $(\text{ծ}, 1)$ whenever they correspond. Now, let X be a non-terminal and let $R(X) = \alpha$. Let $\alpha' Y$ be a nonempty prefix of α where Y is its last symbol. Note that for the empty string, a run of size one $\rho = q$ is always valid for any $q \in Q$, so we can safely consider that the set M_X, as declared in line 16 stores the correct nodes for the runs that read the empty string, and start and end on the same node. The proof will follow inductively by proving that M_X, at the iteration $i = |\alpha' Y|$ of line 18, satisfies that $[\![\mathcal{D}]\!](M_X[p,q])$ contains all partial annotations from a run of $\mathcal{A}$ over $\mathrm{doc}(\alpha' Y)$ that starts on p and ends on q; that is, after assuming that in line 24, the set $[\![\mathcal{D}]\!](M_X[p,q])$ contained all partial annotations from a run of $\mathcal{A}$ over $\mathrm{doc}(\alpha)$ that starts on p and ends on q. We have assumed that the nodes in M_Y store all annotations properly, so the inductive hypothesis is proven from the definition of the operator $\otimes$, as any possible annotation that should be added can be described by two partial runs, one that starts on p, reads $\mathrm{doc}(\alpha)$ and ends on q', and a second that starts on q', reads $\mathrm{doc}(Y)$ and ends on q, so it is properly added to the $[\![\mathcal{D}]\!](M_X[p,q])$ as the operator $\otimes$ does consider this q'. Plus, it can be seen that the positions that were obtained from M_Y are properly shifted when added to M_X by assuming the len_X variable is storing the correct length. From this reasoning, we can conclude that the set $[\![\mathcal{D}]\!](M_X[p,q])$ contains exactly the annotations from $[\![A]\!](\mathrm{doc}(\alpha))$.

To see that the operators add, union, prod and shift are done following all assumptions, we would like to note that union and prod are always called by respecting the duplicate-free property. However, this is not necessarily true, at least for the union case since $\mathcal{A}$ is allowed to be ambiguous in partial runs that do not form part of an accepting run. For simplicity, all statements in the rest of the proof are assumed to be said with respect to indices $M_X[p,q]$ which are reached when building an output string. In other words, that there exists an accepting run $\rho = p_1 \xrightarrow{b_1} \ldots \xrightarrow{b_n} p_{n+1}$ such that $p = p_\ell$, $q = p_r$ with $\ell \leq r$, and $a_1 \ldots a_{\ell-1} X a_r \ldots a_n$ is reachable from S_0 by following production rules in R. In the case of prod, proving the duplicate-free property is straightforward, since it is only done in line 25, and the positions on the left side (i.e., the ones taken from M_X) are all less or equal than the value len_X has at that point, so by shifting the values from M_Y by this amount, the sets from either side do not contain any annotation in common. In the case of union, one can check by inspection that in lines 11 and 13, this is done properly, since transitions do not appear more than once, and one can check that in line 25, if any of the calls to union done inside the $\otimes$ operation is done with duplicates, this implies that there is some annotation that appears in two distinct calls, since each of the union calls is done over a different index in the matrices M_Y and M_X, contradicting the assumption that $\mathcal{A}$ is unambiguous. Therefore, we conclude that the operators add, union, prod and shift are done properly, and the statement of the theorem follows.

Regarding performance, the main procedure calls NonTerminal or Terminal at most once for every symbol. After making all calls to Terminal, each transition in Δ is seen exactly once, and NonTerminal takes time at most $\mathcal{O}(|R(X)| \times |Q|^3)$ not taking into account the calls inside. Overall, the preprocessing time is $\mathcal{O}(|\mathcal{A}| + |S| \times |Q|^3)$. □

We want to finish by noticing that, contrary to (Schmid & Schweikardt, 2021), our evaluation algorithm does not need to modify the grammar S into Chomsky's normal form (CNF) since we can evaluate $\mathcal{A}$ over S directly. Although passing S into CNF can be done in linear time over S (Schmid & Schweikardt, 2021), this step can incur an extra cost, which we can avoid in our approach.

5.4. Applications in regular spanners

It was already shown in Chapter 4 that working with annotations directly and then providing a reduction from a spanner query to an annotation query is sometimes more manageable. In this section we will do just that: starting from a document-regular spanner pair $(d, \mathcal{M})$, we will show how to build a document-annotated automaton pair $(d', \mathcal{A})$ such that $\mathcal{M}(d) = [\![\mathcal{A}]\!](d')$. Although people have studied various models of regular spanners in the literature, we will focus here on sequential variable-set automata (VA) (Fagin et al., 2015) and sequential extended VA (Florenzano et al., 2020). The latter is essentially the model that the work of Schmid and Schweikardt used in their results (Schmid & Schweikardt, 2021). We present the models in this order, but we present first the reduction from the latter as it can be done into AnnA directly. In the second half of the section we reduce the former to *succinctly* annotated automata, an extension of AnnA that allows output symbols to be stored concisely. These reductions imply constant-delay enumeration for the spanner tasks.

Variable-set automata. A *variable-set automaton* (VA for short) is a tuple given by $\mathcal{A} = (Q, \Sigma, \mathcal{X}, \Delta, I, F)$ where Q is a set of states, $I, F \subseteq Q$, and Δ consists of *read transitions* $(p, a, q) \in Q \times \Sigma \times Q$ and *variable transitions* $(p, \vdash^x, q)$ or $(p, \dashv^x, q)$ where $p, q \in Q$ and $x \in \mathcal{X}$. The symbols $\vdash^x$ and $\dashv^x$ are referred to as *variable markers* of x, where $\vdash^x$ is *opening* and $\dashv^x$ is *closing*. Given a document $d = a_1 \ldots a_n \in \Sigma^*$ a configuration of $\mathcal{A}$ is a pair (q, i) where $q \in Q$ and $i \in [1, n+1]$. A run ρ of $\mathcal{A}$ over d is a sequence:

$$\rho \;:=\; (q_1, i_1) \xrightarrow{\sigma_1} (q_2, i_2) \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_m} (q_{m+1}, i_{m+1})$$

where $i_1 = 1$, $i_{m+1} = n + 1$, and for each $j \in [1, m]$, $(q_j, \sigma_j, q_{j+1}) \in \Delta$ and either (1) $\sigma_j = a_{i_j}$ and $i_{j+1} = i_j + 1$, or (2) $\sigma_j \in \{\vdash^x, \dashv^x \mid x \in \mathcal{X}\}$ and $i_{j+1} = i_j$. We say that ρ is *accepting* if $q_{m+1} \in F$ and that it is *valid* if variables are non-repeating, and they are opened and closed correctly. If ρ is accepting and valid, we define the mapping μ^ρ which maps $x \in \mathcal{X}$ to the span $[u, v\rangle$ if, and only if, there exist $j, k \in [1, m]$ such that $i_j = u, i_k = v$, and $\sigma_j = \vdash^x$ and $\sigma_k = \dashv^x$. We say that $\mathcal{A}$ is *sequential* if every accepting run is also valid. Finally, define the document spanner $[\![\mathcal{A}]\!]$ as the function:

$$[\![\mathcal{A}]\!](d) \;=\; \{\mu^\rho \mid \rho \text{ is an accepting and valid run of } \mathcal{A} \text{ over } d\}.$$

Like in AnnAs, we say $\mathcal{A}$ is *unambiguous* if for each mapping $\mu \in [\![\mathcal{A}]\!](d)$ there is exactly one accepting run ρ of $\mathcal{A}$ over d such that $\mu^\rho = \mu$.

Extended VA. An *extended variable-set automaton* (or eVA for short) is a tuple $\mathcal{A} = (Q, \Sigma, \mathcal{X}, \Delta, I, F)$ where Q, I, F are defined as in VA, Δ is a set consisting of *letter transitions* (p, a, q) where $a \in \Sigma$ and $p, q \in Q$ or *extended variable transitions* (p, S, q) where $S \subseteq \{\vdash^x, \dashv^x \mid x \in \mathcal{X}\}$ and S is non-empty; and $F \subseteq Q$. A run ρ over a document

$d = a_1 \ldots a_n$ is a sequence:

$$\rho = q_1 \xrightarrow{S_1} p_1 \xrightarrow{a_1} q_2 \xrightarrow{S_2} p_2 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_{n+1} \xrightarrow{S_{n+1}} p_{n+1}$$

where each S_i is a (possibly empty) set of markers, for each $i \in [1, n]$, $(p_i, a_i, q_{i+1}) \in \Delta$, and for each $i \in [1, n+1]$, if S_i is not empty, then $(q_i, S_i, p_i) \in \Delta$, and $p_i = q_i$, otherwise. We say that a run is accepting if $p_{n+1} \in F$. Also, we say that a run ρ is *valid* if variables are opened and closed in a correct manner: every marker $\vdash^x$ and $\dashv^x$ must appear at most once among the sets $S_1 \ldots S_{n+1}$; if one of them appears, the other does as well; and if $\vdash^x \in S_i$ and $\dashv^x \in S_j$ then it holds that $i \leq j$. For a valid run ρ, we define the mapping μ^ρ that maps x to $[i, j\rangle$ iff $\vdash^x \in S_i$ and $\dashv^x \in S_j$. The spanner $[\![\mathcal{A}]\!]$ is defined identically as for VA. The definitions of sequential and unambigous eVA are the same as well.

To motivate the reduction from sequential eVA to annotated automata, consider a document $d = \texttt{aab}$, and a run over d of some (unspecified) eVA with variable set $\mathcal{X} = \{x, y\}$:

$$\rho = q_1 \xrightarrow{\emptyset} q_1 \xrightarrow{\texttt{a}} q_2 \xrightarrow{\{\vdash^x, \dashv^x, \vdash^y\}} p_2 \xrightarrow{\texttt{a}} q_3 \xrightarrow{\emptyset} q_3 \xrightarrow{\texttt{b}} q_4 \xrightarrow{\{\dashv^y\}} p_4$$

This run defines the mapping μ which assigns $\mu(x) = [2, 2\rangle$ and $\mu(y) = [2, 4\rangle$. To translate this run to the annotated automata model, first we append an end-of-document character to d, and then "push" the marker sets one transition to the right. We then obtain a possible run of an annotated automaton with output set $\Omega = 2^{\{\vdash^x, \dashv^x | x \in \mathcal{X}\}}$ over the document $d' = \texttt{aab\#}$:

$$\rho' = q_1' \xrightarrow{\texttt{a}} q_2' \xrightarrow{(\texttt{a}, \{\vdash^x, \dashv^x, \vdash^y\})} q_3' \xrightarrow{\texttt{b}} q_4' \xrightarrow{(\texttt{\#}, \{\dashv^y\})} q_5'$$

The annotation of this run would then be $(2, \{\vdash^x, \dashv^x, \vdash^y\})(4, \{\dashv^y\})$, from where the mapping μ can be extracted directly. The reduction from extended VA into annotated automata operates in a similar fashion: the read transitions are kept, and for each pair of transitions $(p, S, q), (q, a, r)$ in the former, a transition $(p, (a, S), r)$ is added to the latter.

The equivalence between mappings and annotations is formally defined as follows: for some document d, a mapping μ from $\mathcal{X}$ to spans in d is equivalent to an annotation

$w = (S_1, i_1) \dots (S_m, i_m)$ if, and only if, for every $j \in [1, m]$:

$$S_j \;=\; \{\vdash^x \mid \mu(x) = [i_j, k\rangle\} \,\cup\, \{\dashv^x \mid \mu(x) = [k, i_j\rangle\}.$$

PROPOSITION 5.2. *For any unambiguous sequential extended VA $\mathcal{A}$ with state set Q and transition set Δ, there exists an AnnA $\mathcal{A}'$ with $\mathcal{O}(|Q| \times |\Delta|)$ transitions such that each mapping $\mu \in [\![A]\!](d)$ is equivalent to some unique $w \in [\![\mathcal{A}']\!](d\#)$ and vice versa, for every document d.*

PROOF. Let $\mathcal{A} = (Q, \Sigma, \mathcal{X}, \Delta, I, F)$ be an unambiguous sequential eVA. We build an annotated automaton $\mathcal{A}' = (Q', \Sigma \cup \{\#\}, \Omega, \Delta', I, F')$ as follows. Define $Q' = Q \cup \{q^*\}$ for some fresh state q^*, and $F' = \{q^*\}$. Further, define $\Omega = 2^{\{\vdash^x, \dashv^x \mid x \in \mathcal{X}\}}$ and:

$$\begin{aligned}\Delta' = \{&(p, a, q) \mid a \in \Sigma \text{ and } (p, a, q) \in \Delta\} \,\cup \\ &\{(p, (a, S), q) \mid (p, S, q'), (q', a, q) \in \Delta \text{ for some } q' \in Q\} \,\cup \\ &\{(p, \#, q^*) \mid p \in F\} \,\cup \\ &\{(p, (\#, S), q^*) \mid (p, S, q) \in \Delta \text{ for some } q \in F\}.\end{aligned}$$

To see the equivalence between $\mathcal{A}$ and $\mathcal{A}'$, let $d = a_1 \dots a_n$ be a document over Σ, and let ρ be an accepting run of $\mathcal{A}$ over d of the form:

$$\rho = q_1 \xrightarrow{S_1} p_1 \xrightarrow{a_1} q_2 \xrightarrow{S_2} p_2 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_{n+1} \xrightarrow{S_{n+1}} p_{n+1}$$

We define ρ' as the following sequence:

$$\rho' = q_1 \xrightarrow{b_1} q_2 \xrightarrow{b_2} \cdots \xrightarrow{b_n} q_{n+1} \xrightarrow{b_{n+1}} q^*$$

where $b_i = (a_i, S_i)$ if S_i is not empty, and $b_i = a_i$ otherwise, for each $i \leq n$. We define $b_{n+1} = (\#, S_{n+1})$ if S_{n+1} is not empty, and $b_{n+1} = \#$ otherwise. Since $\mathcal{A}$ is sequential, ρ is a valid run which defines a mapping $\mu^\rho \in [\![\mathcal{A}]\!](d)$. We can straightforwardly check that μ^ρ is equivalent to $\mathsf{ann}(\rho')$. It can also be seen directly from the construction that ρ' is a run from $\mathcal{A}'$ over $d\#$, and since ρ' is uniquely defined from ρ, we conclude that for every document d each $\mu \in [\![\mathcal{A}]\!](d)$ is equivalent to some unique $w \in [\![\mathcal{A}']\!](d\#)$.

To see the equivalence on the opposite direction, consider an accepting run ρ' of $\mathcal{A}'$ over $d\#$ as above, where each b_i, for $i \in [1, n]$, might be either a_i or a pair (a_i, S). From the construction, it can be seen that if $b_i = a_i$, there exists a transition $(q_i, a_i, q_{i+1}) \in \Delta$. Instead, if $b_i = (a_i, S)$, there exist transitions $(q_i, S, q'), (q', a_i, q_{i+1}) \in \Delta$ for some $q' \in Q$. Also, if $b_{n+1} = \#$ then $q_{n+1} \in F$, and if $b_{n+1} = (\#, S)$ there exists $(q_{n+1}, S, q') \in \Delta$ for some $q' \in F$. We define ρ as a run of $\mathcal{A}$ over d built by replacing each transition in ρ' by the corresponding transition(s) in $\mathcal{A}$. We note first that this run is accepting and valid, and since $\mathcal{A}$ is unambiguous, ρ must be uniquely defined. Indeed, when replacing $(q_i, (a_i, S), q_{i+1})$, we know there exist transitions (q_i, S, q') and (q', a_i, q_{i+1}) in Δ. Furthermore, this q' must be unique, otherwise we could define a different accepting run that defines the same mapping. We see that $\mathsf{ann}(\rho')$ is equivalent to μ^ρ, so we conclude that each annotation $w \in [\![\mathcal{A}]\!](d)$ is equivalent to some unique mapping $\mu \in [\![\mathcal{A}]\!](d)$.

To see that $\mathcal{A}'$ is unambiguous, consider towards a contradiction two different accepting runs ρ'_1 and ρ'_2 that retrieves the same annotation. Let i be such that the i-th states in ρ'_1 and ρ'_2 are different, and note it holds that $1 \leq i \leq n+1$ since $\mathcal{A}'$ has a unique final state q^*. By the previous discussion, we can build runs ρ_1 and ρ_2 of $\mathcal{A}$ over d that also define the same mapping. Furthermore, they differ at index $2i - 1$ (the index at which q_i is in the ρ written above), which is not possible since $\mathcal{A}$ is unambiguous. □

Combining Proposition 5.2 and Theorem 5.1, we get a constant-delay algorithm for evaluating an unambiguous sequential extended VA over a document, proving the extension of the result in (Schmid & Schweikardt, 2021). Notice that the result in (Schmid & Schweikardt, 2021) is for *deterministic* VA, where here we generalize this result for the unambiguous case, plus the constant delay.

Succinctly annotated automata. For the algorithmic result of sequential (non-extended) VA, we need an extension to annotated automata which features succinct representations of sets of annotations.

A *succinct enumerable representation scheme* (SERS) is a tuple:

$$\mathcal{S} = (\mathcal{R}, \Omega, |\cdot|, \mathcal{L}, \mathcal{E})$$

made of an infinite set of representations $\mathcal{R}$, and an infinite set of annotations Ω. It includes a function $|\cdot|$ that indicates, for each $r \in \mathcal{R}$ and ᘐ $\in \Omega$, the sizes $|r|$ and $|$ᘐ$|$, i.e., the number of units needed to store r and ᘐin the underlying computational model (e.g., the RAM model). The function $\mathcal{L}$ maps each element $r \in \mathcal{R}$ to some finite non-empty set $\mathcal{L}(r) \subseteq \Omega$. Lastly, there is an algorithm $\mathcal{E}$ which enumerates the set $\mathcal{L}(r)$ with output-linear delay for every $r \in \mathcal{R}$. Intuitively, a SERS provides us with representations to encode sets of annotations. Moreover, there is the promise of the enumeration algorithm $\mathcal{E}$ where we can recover all the annotations with output-linear delay. This representation scheme allows us to generalize the notion of annotated automaton for encoding an extensive set of annotations in the transitions.

Fix a SERS $\mathcal{S} = (\mathcal{R}, \Omega, |\cdot|, \mathcal{L}, \mathcal{E})$. A Succinctly Annotated Automaton over $\mathcal{S}$ (sAnnA for short) is a tuple $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, I, F)$ where all sets are defined like in AnnA, except that in Δ read-annotate transitions are of the form $(p, (a, r), q) \in Q \times (\Sigma \times \mathcal{R}) \times Q$. That is, transitions are now annotated by a representation r which encodes *sets* of annotations in Ω. For a read-annotate transition $\delta = (p, (a, r), q)$, we define its size as $|\delta| = |r| + 1$ and for a read transition $\delta = (p, a, q)$ we define its size as $|\delta| = 1$. A run ρ over a document $d = a_1 \ldots a_n$ is also defined as a sequence:

$$\rho := q_1 \xrightarrow{b_1} q_2 \xrightarrow{b_2} \ldots \xrightarrow{b_n} q_{n+1}$$

with the same specifications as in AnnA with the difference that it either holds that $b_i = a_i$, or $b_i = (a_i, r)$ for some representation r. We now define the *set of annotations* of ρ as: $\mathsf{ann}(\rho) = \mathsf{ann}(b_1, 1) \cdot \ldots \cdot \mathsf{ann}(b_n, n)$ such that $\mathsf{ann}(b_i, i) = \{($ᘐ$, i) \mid$ ᘐ $\in \mathcal{L}(r)\}$ if

$b_i = (a, r)$, and $\mathsf{ann}(b_i, i) = \{\varepsilon\}$ otherwise. The set $[\![\mathcal{A}]\!](d)$ is defined as the union of sets $\mathsf{ann}(\rho)$ for all accepting runs ρ of $\mathcal{A}$ over d. We say that $\mathcal{A}$ is unambiguous if for every document d and every annotation $w \in [\![\mathcal{A}]\!](d)$ there exists only one accepting run ρ of $\mathcal{A}$ over d such that $w \in \mathsf{ann}(\rho)$. Finally, we define the size of Δ as $|\Delta| = \sum_{\delta \in \Delta} |\delta|$, and the size of $\mathcal{A}$ as $|\mathcal{A}| = |Q| + |\Delta|$.

This annotated automata extension allows for representing output sets more compactly. Moreover, given that we can enumerate the set of annotations with output-linear delay, we can compose it with Theorem 5.1 to get an output-linear delay algorithm for the whole set.

Theorem 5.5. *Fix a SERS $\mathcal{S}$. There exists an enumeration algorithm that, given an unambiguous sAnnA $\mathcal{A}$ over $\mathcal{S}$ and an SLP S, it runs in $|\mathcal{A}|^3 \times |S|$-preprocessing time and output-linear delay for enumerating* $[\![\mathcal{A}]\!](\mathrm{doc}(S))$.

For this proof, we will use an extension of Shift-ECSs called *succinct Shift-ECSs*. A succinct Shift-ECSs is a tuple $\tilde{\mathcal{D}} = (\mathcal{S}, V, \ell, r, \lambda, \bot, \varepsilon)$, similar to Shift-ECS with the difference that the output set has been replaced by representations from a SERS $\mathcal{S}$. The set of strings associated to a node $v \in \tilde{D}$ is defined as follows: If $\lambda(v) = r \in \mathcal{R}$, where $\mathcal{R}$ is the set of representations in $\mathcal{S}$, then $[\![\tilde{D}]\!](v) = \mathcal{L}(r) \times \{1\}$. The rest of the sets $[\![\tilde{D}]\!](v)$ for union, product, and shift nodes v remain unchanged, and so are the notions of duplicate-free, k-bounded and ε-safe nodes, along with the operations for add, prod, union and shift. We will extend Proposition 5.3 for this data structure as follows:

PROPOSITION 5.3. *The operations* prod, union *and* shift *over succinct Shift-ECSs extended with empty- and ε-nodes take constant time, and the operation* add *with a representation r as argument takes time $\mathcal{O}(|r|)$. Furthermore, if we start from an empty succinct Shift-ECS $\mathcal{D}$ and apply* add, prod, union, *and* shift *over ε-safe nodes, the resulting node v' is always an ε-safe node, and the set $[\![\mathcal{D}]\!](v)$ can be enumerated with output-linear delay without preprocessing for every node v.*

PROOF. For this proof, we need to be a bit more specific regarding the enumeration algorithm $\mathcal{E}$ of an SERS $\mathcal{S}$. Precisely, we assume that $\mathcal{E}$ has an associated function `yield` and a constant c, and its procedure is to receive r and then allow up to $|\mathcal{L}(r)|$ calls to `yield`. Each of these calls produces a different annotation $\vartheta \in \mathcal{L}(r)$ and takes time at most $c \cdot (|\vartheta| + 1)$ for some fix constant c. Each of these annotations carries a flag `end` which is true if, and only if, it is the last output of the set. We also assume that a sequence of $|\mathcal{L}(r)|$ calls to `yield` might happen again with the same time bounds after the last call had set `end` to true.

We start this proof by re-stating that every definition in the Shift-ECS data structure is unchanged in its succinct version. The only difference is the definition of $[\![\tilde{\mathcal{D}}]\!](v)$ for a bottom node $v \in \tilde{\mathcal{D}}$.

First, we shall prove a version of Proposition 5.1 in this model, namely, that if a succinct Shift-ECS $\tilde{\mathcal{D}}$ is duplicate-free and k-bounded, then the set $[\![\tilde{\mathcal{D}}]\!](v)$ can be enumerated with output-linear delay for every $v \in \mathcal{D}$. We do this by adapting the proof of Proposition 5.1 to handle this new type of bottom node. Let $\mathcal{S} = (\mathcal{R}, \Omega, |\cdot|, \mathcal{L}, \mathcal{E})$ be the SERS associated to $\tilde{\mathcal{D}}$ and let c' be the constant associated to $\mathcal{E}$.

The main idea is to modify iterator τ_Λ from Algorithm 4 so that it stores, besides a value u and a flag $\mathrm{hasnext}$, an annotation ϑ. The iterator fully makes use of the fact that the `yield` procedure from $\mathcal{E}$ retreives an output, evolves the internal iterator so that the next call produces the next output, and also says at each point if the current output was the last one from the set or not. The overall strategy is then to call `yield` in NEXT, store the retrieved output in ϑ, and in PRINT, print whatever is currently stored in ϑ. More precisely, the CREATE procedure initializes $\mathcal{E}$ so that the next time it calls `yield` it prints the first output in the set $\mathcal{L}(r)$, and initializes $\mathrm{hasnext}$ to **true**. NEXT first checks if $\mathrm{hasnext}$ is set to **true**, and returns **false** if it is not the case; otherwise it calls `yield`, stores the output in ϑ, and returns **true**. If the output of `yield` is `end`, then it set $\mathrm{hasnext}$ to **false** and returns **false**. Finally, PRINT with input s simply prints the pair $(\vartheta, s + 1)$. It can be seen that the methods follow the necessary specifications to ensure that the correctness of the algorithm still holds for this version of the data structure.

To show the time bounds, we bring attention to the fact that the only difference in the algorithm time-wise is the time spent printing each pair (ϑ, i), as now this takes $c\cdot(|\vartheta|+1)$. Since Algorithm 4 was proven to have delay $O(k \cdot (|w| + |w'|))$ to write output w' after writing w for non-succinct Shift-ECS, we state that the same delay holds for this version of the algorithm since the time added is $O(|w|)$. This implies output-linear delay by following the same reasoning than the proof of Proposition 5.1.

The rest of the proof pertains the operations add, prod, union and shift. It is not hard to see that they can be kept unchanged and maintain all the conditions in the statement, with the sole exception of $\mathsf{add}(r)$, which now adds a representation r to the structure and takes $|r|$ time. This concludes the proof. □

PROOF OF THEOREM 5.5. The algorithm that we give to prove the statement is exactly Algorithm 5 line-by-line, with the sole exception of lines 10-11, which now read:

```
for each $(p, (a, r), q) \in \Delta$ do
    $M_a[p, q] \leftarrow \mathsf{union}(M_a[p, q],\ \mathsf{add}(r))$
```

The rest of the proof follows from the reasoning of the proof of Theorem 5.4, and by noticing that the definition of the set $[\![\tilde{D}]\!](v) = \mathcal{L}(r) \times \{1\}$ satisfies what is expected for an index $M_a[p, q]$. That is, that for every partial run ρ of $\mathcal{A}$ over a that starts on p and end on q, all annotations from the set $\mathsf{ann}(\rho)$ are included. The fact that the operations are done duplicate-free also follows from the fact that $\mathcal{A}$ is assumed to be unambiguous. □

The purpose of sAnnA is to encode sequential VA succinctly. Indeed, as shown in (Florenzano et al., 2020), representing sequential VA by extended VA has an exponential blow-up in the number of variables that cannot be avoided. Therefore, the reduction from Proposition 5.2 cannot work directly. Instead, we can use a Succinctly Annotated

Automaton over some specific SERS to translate every sequential VA into the annotation world efficiently.

PROPOSITION 5.4. *There exists an SERS $\mathcal{S}$ such that for any unambiguous sequential VA $\mathcal{A}$ with state set Q and transition set Δ there exists a sAnnA $\mathcal{A}$ over $\mathcal{S}$ of size $\mathcal{O}(|Q| \times |\Delta|)$ such that for every document d, each mapping $\mu \in [\![\mathcal{A}]\!](d)$ is equivalent to some unique $w \in [\![\mathcal{A}]\!](d\#)$ and vice versa. Furthermore, the number of states in $\mathcal{A}$ is in $\mathcal{O}(|Q|)$.*

PROOF. The SERS that we will consider are Enumerable Compact Sets as they were presented in Chapter3, defined over the output set $\Omega = \{\vdash^x, \dashv^x \mid x \in \mathcal{X}\}$, with the sole difference that the sets of outputs stored in a node no longer contain strings in Ω^*, but subsets of Ω instead. This difference is merely technical and has no influence in the time bounds of an ECS, as long as it is guaranteed to be duplicate-free (called *unambiguous* in Chapter 3).

First we assume that all transitions in $\mathcal{A}$ are reachable from some $q \in I$, and all of them reach an accepting state. We know that in the VA $\mathcal{A}$, the graph induced by the variable transitions is a DAG, otherwise it would not be sequential. We start with an empty ECS $\mathcal{D}$ and define a matrix $K[p, q]$, that first starts with the empty node in each index except the indices in the diagonal (those that satisfy $p = q$), which have the ε node. The idea is that at the end of this algorithm, $\mathcal{L}_{\mathcal{D}}(K[p, q])$ contains all sets of variable markers that can be seen in a path from p to q that does not contain a letter transition. We build this matrix by iterating over the variable transitions in $\mathcal{A}$ following some topological order of the DAG, starting from a root. For each variable transition (p, V, q), let $u \leftarrow \mathsf{add}(V)$ and for each $p' \in Q$ that can reach p in the DAG assign $K[p', q] \leftarrow \mathsf{union}(K[p', q], \mathsf{prod}(K[p', p], u))$. The time of this algorithm is $|Q| \times |\mathcal{A}|$.

After doing this, we perform a construction analogous to the one done in Proposition 5.2, in which we replace the extended variable transitions (p, S, q) by transitions (p, r, q) where $r = K[p, q]$, which happens every time $K[p, q]$ is not the empty node. The proof follows from this result as well. □

By Proposition 5.4 and Theorem 5.5 we prove the extension of the output-linear delay algorithm for unambiguous sequential VA.

5.5. Constant-delay preserving complex document editing

In this section, we show that the results obtained by Schmid and Schweikardt (Schmid & Schweikardt, 2022) regarding enumeration over document databases and complex document editing still hold, maintaining the same time bounds in doing these edits, but allowing output-linear delay. We also include a refinement of the result for whenever the edits needed are limited to the concatenation of two documents. To be precise, we will give an overview of the following theorem.

Theorem 5.6. *Let $D = \{d_1, \ldots, d_m\}$ be a document database that is represented by an SLP S in normal form. Let $\mathcal{A}_1, \ldots, \mathcal{A}_k$ be unambiguous sequential variable-set automata. When given the query data structures for S and $\mathcal{A}_1, \ldots, \mathcal{A}_k$, and a CDE-expression φ over D, we can construct an extension S' of S and new query data structures for S' and $\mathcal{A}_1, \ldots, \mathcal{A}_k$, and a new non-terminal $\tilde{A}$ of S', such that* $\mathrm{doc}(\tilde{A}) = \textsf{\textit{eval}}(\varphi)$.

- *If φ contains operations other than* **concat***, we require S to be strongly balanced. Then, S' is also strongly balanced, and this construction can be done $\mathcal{O}(k \cdot |\varphi| \cdot \log |d^*|)$ time in data-complexity where $|d^*| = |\max_\varphi(D)|$.*
- *If φ only contains* **concat***, then this can be done in $\mathcal{O}(k \cdot |\varphi|)$ time in data-complexity.*

*Afterwards, upon input of any $d \in$ **docs**(S') (represented by a non-terminal of S') and any $i \in [1, m]$, the set $[\![\mathcal{A}_i]\!](d)$ can be enumerated with constant-delay.*

The version of this result shown in (Schmid & Schweikardt, 2022) had extended VA instead of VA, and logarithmic delay instead of constant-delay. We dedicate the rest of this section to define the concepts we have not yet introduced in Theorem 5.6, and show how the techniques presented in (Schmid & Schweikardt, 2022) allow us to obtain this result.

Normal form, balanced and rootless SLPs. We define a *rootless SLP* as a triple $S = (N, \Sigma, R)$, where N is a set of non-terminals, Σ is the set of terminals, and R is a set of rules. Rootless SLPs are defined as SLPs with the difference that there is no starting symbol, and thus $\mathrm{doc}(S)$ is not defined. Instead, we define $\mathrm{doc}(A)$ for each $A \in N$ as $\mathrm{doc}(A) = R^*(A)$. We say that S is in *Chomsky normal form* (or just normal form) if every rule in R has the form $A \to a$ or $A \to BC$, where $a \in \Sigma$ and $A, B, C \in N$. Also, we say that S is *strongly balanced* if for each rule $A \to BC$, the value $\mathsf{ord}(B) - \mathsf{ord}(C)$ is either -1, 0 or 1, where $\mathsf{ord}(X)$ is the maximum distance from X to any terminal in the derivation tree.

Document Databases. A *document database* over Σ is a finite collection $D = \{d_1, \ldots, d_m\}$ of documents over Σ. Document databases are represented by a rootless SLP as follows. For an SLP $S = (N, \Sigma, R)$, let $\mathsf{docs}(S) = \{\mathrm{doc}(A) \mid A \in N\}$ be the set of documents represented by S. The rootless SLP S is a representation for a document database D if $D \subseteq \mathsf{docs}(S)$.

For a document database D, it is assumed that a rootless SLP S that represents D is in normal form and, for the effects of the first bullet point of Theorem 5.6, strongly balanced. It is also assumed that for each nonterminal A for which its rule has the form $A \to BC$, the values $|\mathrm{doc}(A)|$, $\mathsf{ord}(A)$ and nonterminals B and C are accessible in constant time. All these values can be precomputed with a linear-time pass over S. We call S along with constant-time access to these values *the basic data structure for* S.

Complex Document Editing. As in (Schmid & Schweikardt, 2022), given a document database $D = \{d_1, \ldots, d_m\}$ our goal is to create new documents by a sequence of text-editing operations. Here we introduce the notion of a CDE-expression over D, which is defined by the following syntax:

$$\varphi := \; d_\ell, \ell \in [1, m] \mid \mathsf{concat}(\varphi, \varphi) \mid \mathsf{extract}(\varphi, i, j) \mid \mathsf{delete}(\varphi, i, j) \mid \\ \mathsf{insert}(\varphi, \varphi, k) \mid \mathsf{copy}(\varphi, i, j, k)$$

where the values i, j are valid *positions*, and k is a valid *gap*. The semantics of these operations, called *basic operations*, works as follows:

$$\begin{aligned}
\mathsf{concat}(d, d') &= d\cdot d' & \mathsf{insert}(d, d', k) &= d[1, k\rangle\cdot d'\cdot d[k, |d| + 1\rangle \\
\mathsf{extract}(d, i, j) &= d[i, j + 1\rangle & \mathsf{delete}(d, i, j) &= d[1, i\rangle\cdot d[j + 1, |d| + 1\rangle \\
\mathsf{copy}(d, i, j, k) &= \mathsf{insert}(d, d[i, j + 1\rangle, k)
\end{aligned}$$

We write $\mathsf{eval}(\varphi)$ for the document obtained by evaluating φ on D according to these semantics. For an operation $\mathsf{extract}(\varphi, i, j)$, $\mathsf{delete}(\varphi, i, j)$, $\mathsf{insert}(\varphi, \psi, k)$, or $\mathsf{copy}(\varphi, i, j, k)$, i, j are valid positions if $i, j \in [1, |\mathsf{eval}(\varphi)|]$, and k is a valid gap if $k \in [1, |\mathsf{eval}(\varphi)| + 1]$. We define $|\varphi|$ as the number of basic operations in φ. For adding these new documents in the database we will use the notion of extending a rootless SLP. A rootless SLP $S' = (N', \Sigma, R')$ is called an extension of S if S' is in normal form, $N \subseteq N'$, and $R'(A) = R(A)$ for every $A \in N$. In this context, we call $N' \setminus N$ the *set of new non-terminals*. We define the *maximum intermediate document size* $|\max_\varphi(D)|$ induced by a CDE-expression φ on a document database D as the maximum size of $\mathsf{eval}(\psi)$ for any sub-expression ψ of φ (i.e., any substring ψ of φ that matches the CDE syntax).

Having defined most of the concepts mentioned in Theorem 5.6, we can re-state the following Theorem from (Schmid & Schweikardt, 2022), which will be instrumental in the final proof.

Theorem 5.7 ((Theorem 4.3 in (Schmid & Schweikardt, 2022))). *Let D be a document database represented by a strongly balanced rootless SLP S in normal form. When given the basic data structure for S and a CDE-expression φ over D, we can construct a strongly balanced extension S' of S, along with its basic data structure, and a non-terminal $\tilde{A}$ of S' such that $\mathrm{doc}(\tilde{A}) = \mathsf{eval}(\varphi)$. This construction takes time $\mathcal{O}\big(|\varphi| \cdot \log(|\max_\varphi(D)|)\big)$. In particular, the number of new non-terminals $|N' \setminus N|$ is in $\mathcal{O}\big(|\varphi| \cdot \log(|\max_\varphi(D)|)\big)$.*

For the second bullet point in Theorem 5.6, we use the following observation, which comes from the fact that applying concat to an SLP amounts to adding a single production.

OBSERVATION 5.1. *Let D be a document database represented by a rootless SLP S in normal form. Given the basic data structure for S and a CDE-expression φ over D which only mentions* concat*, we can construct an extension S' of S, along with its basic data structure, and a nonterminal $\tilde{A}$ of S' such that $\mathrm{doc}(\tilde{A}) = \textsf{eval}(\varphi)$. This construction takes time $\mathcal{O}(|\varphi|)$. In particular, the number of new non-terminals $|N' \setminus N|$ is in $\mathcal{O}(|\varphi|)$.*

The query data structure. The structure we will use is the one produced in Theorem 5.5. This structure is built by an algorithm that receives an SLP S, an unambiguous sAnnA $\mathcal{A}$, and produces a succinct Shift-ECS $\mathcal{D}$ indexed by the matrices M_A, for each non-terminal A in S. These matrices store nodes $v = M_A[p, q]$ such that $[\![\mathcal{D}]\!](v)$ contains all partial annotations from a path of $\mathcal{A}$ which starts p, ends in q, and reads the string $\mathrm{doc}(A)$. Note that, although the algorithm receives a "rooted" SLP, it can be adapted quite easily to rootless SLPs by adding a node v_A for each non-terminal A in S, built as $v_A = \textsf{union}_{q \in F}(M_A[q_0, q])$ (the same construction that was done for S_0 in the algorithm).

We define *the query data structure for S and $\mathcal{A}$* as the mentioned succinct Shift-ECS along with constant-time access to every index $M_A[p, q]$ for states p and q and non-terminal A. Note that for each A it holds that $[\![\mathcal{D}]\!](v_A) = [\![\mathcal{A}]\!](\mathrm{doc}(A))$. In particular, if S represents a document database D, then for each $d \in D$ there is a v in $\mathcal{D}$ for which $[\![\mathcal{D}]\!](v) = [\![\mathcal{A}]\!](d)$. Recall that for every node $v \in \mathcal{D}$, the set $[\![\mathcal{D}]\!](v)$ can be enumerated with output-linear delay.

Lemma 5.2. *Let S be an SLP in normal form and an extension S' of S with new non-terminals $\tilde{N} = N' \setminus N$. Also, let $\mathcal{A}$ be an unambiguous sAnnA and assume we are given the query data structure for S and $\mathcal{A}$, and the basic data structure for S'. We can construct the query data structure for S' and $\mathcal{A}$ in $\mathcal{O}(|\mathcal{A}|^3 \cdot |\tilde{N}|)$ time.*

PROOF. Let $\mathcal{D}$ be the succinct Shift-ECS associated to the query data structure for S and $\mathcal{A}$ and let $\tilde{R} = R' \setminus R$ be the set of new rules in S'. Consider the DAG that is induced by this set. We can go over these new rules $\tilde{A} \to BC$ in a bottom-up fashion, starting from the ones where $B, C \in N$, and define the nodes $M_{\tilde{A}}[p, q] = (M_B \otimes \textsf{shift}(M_C, |\mathrm{doc}(B)|))[p, q]$

for each $p, q \in Q$. Then, we build the nodes $v_{\tilde{A}}$ for each new non-terminal $\tilde{A}$ as defined. The new nodes are created by application of the Shift-ECS operations add, union, prod and shift which create the succinct Shift-ECS $\mathcal{D}'$ that defines the query data structure for S' and $\mathcal{A}$. The procedure takes $\mathcal{O}(|Q|^3 \cdot |\tilde{N}|)$ time. □

We finally have all the machinery to prove Theorem 5.6, extending the results shown in (Schmid & Schweikardt, 2022) for sequential VA and with output-linear delay.

Proof of Theorem 5.6. Let us first reduce the variable-set automata $\mathcal{A}_1, \ldots \mathcal{A}_k$ to sAnnAs $\mathcal{A}'_1, \ldots, \mathcal{A}'_k$ using the construction of Proposition 5.4. Note, however, that this reduction requires the input document to be modified as well. This can be solved by adding a non-terminal $A_\#$ for each $A \in N$, and a rule $A_\# \to AH$, where H is a new non-terminal with the rule $H \to \#$. Then, in the query data structure for S and $\mathcal{A}'$, the nodes v_A are defined over the matrices $M_{A_\#}$ instead. That way, when the user chooses a document $d \in \mathsf{docs}(S)$ and a variable set automata $\mathcal{A}_i$, she can be given the set $[\![\mathcal{A}'_i]\!](d\#)$ as output. Note that this has no influence in the time bounds given for the edit so far, except for a factor that is linear in $|\tilde{N}|$.

We can see now that the result follows due to Theorem 5.7, Observation 5.1, and Lemma 5.2. The fact that for each $d \in \mathsf{docs}(S')$ the set $[\![\mathcal{A}]\!](d)$ can be enumerated with output-linear delay follows from the definition of the query data structure for S' and $\mathcal{A}$. □

6. CONCLUSIONS AND FUTURE WORK

In this thesis, we developed a framework based on MSO to handle enumeration queries – called the *Annotators* framework. As a proof of concept, we used it to build three different models that perform output-linear enumeration: (1) MSO queries over nested documents, (2) context-free grammar queries over strings, and (3) regular queries over SLP-compressed documents. These results are given with several extensions: (1) allows one-pass-streaming computation over the document, (2) is given for three different fragments of context-free grammars, each with tight complexity bounds, and (3) are shown to be compliant with other useful frameworks that work on compressed documents. Furthermore, the work improves on other known results in the literature, especially from the perspective of document spanners.

We attribute these achievements to two main factors:

The use of an intuitive model for the query instances, namely, the Annotator framework. While document spanners have been studied extensively and with reasonable success, the task of enumerating span assignments can be simplified, in our opinion, a lot by forgetting about the spans themselves. Annotated automata and their derivatives can be seen as a version of document spanners in which a ever-present difficulty, which is having multiple symbols per document position, is completely avoided.

Using a modular data structure which is almost fully independent to the query language to represent the enumerable output data. Although the algorithm itself that builds the structure can be derived from already known results for circuits, the Enumerable Compact Sets and Enumerable Compact Sets with Shifts data structures streamline the building process significantly. The differences between our framework and previous algorithms for circuits are most patent in Chapter 3, as it is used to update the complete data structure in constant (data-independent) time.

The shown results also leave open future work in different avenues of improvement. We will list them by organizing them by the three main chapters.

Regarding the work detailed in Chapter 3, one direction of future work is to find a streaming evaluation algorithm with polynomial update-time for non-deterministic VPAnn (i.e., in the size of the VPAnn). In (Amarilli et al., 2019a), the authors provided a poly-time offline algorithm for non-deterministic word annotators (called vset automata). They extended this result to trees in (Amarilli et al., 2019b). One could use these techniques in Algorithm 2; however, it is unclear how to extend ECS to eliminate duplicates in a natural way.

Regarding space resources, another direction is to find an "instance optimal" streaming evaluation algorithm for VPAnn. As we mentioned, this problem generalizes the weak evaluation problem stated in (Segoufin & Vianu, 2002), given that it also considers the space to represent the output compactly.

Finally, it would be interesting to explore practical implementations. We believe that the data structure and algorithm presentation are well-suited for this and that they leave space for pertinent optimizations.

In Chapter 4 we presented our formalism of annotated grammars and our results on the efficient enumeration of all annotations of an input string. Our results achieve output-linear delay, and cubic-time preprocessing if the grammar is unambiguous, quadratic-time if it is rigid, and linear-time for profiled-deterministic PDAnns.

The main question left open in the chapter is that of the precise complexity of this task, depending on the grammar formalism. For instance, can we improve the $\mathcal{O}(n^3)$ algorithm to match the complexity of Valiant's parser? For which grammar classes can we extend the linear-time preprocessing approach? We believe, however, that an absolute classification is out of reach, given that classifying the fine-grained complexity of parsing is still open to a large extent even in the case of unannotated CFGs.

Given that the enumeration task is at least as difficult as parsing, it would be interesting to know about a tighter lower bound for enumeration in some fragment that is not immediately derivable from the parsing task itself.

With regards to Chapter 5, one natural direction for future work is to study which other compression schemes allow output-linear delay enumeration for evaluating annotated automata. To the best of our knowledge, the only model for compressed data in which spanner evaluation has been studied is SLPs. However, other models (such as some based on run-length encoding) allow better compression rates and might be more desirable results in practice.

Regarding the Shift-ECS data structure, it would be interesting to see how further one could extend the data structure while still allowing output-linear delay enumeration. Another aspect worth studying is whether there are enumeration results in other areas that one can improve using Shift-ECS.

Another aspect that might be improvable in Shift-ECS, or even the original ECS data structure, is the fact that output-linear delay allows an enumeration scheme where the user waits $O(|w|)$ time to see a single symbol of w. Is there more fine-grained enumeration bound which produces each output symbol in an output string with constant-delay?

One more direction is seeing if the extending the concisely annotated automata model complex models of computation, such as Tree Automata or Pushdown Automata allows for further improvements in known results related to spanner evaluation.

Lastly, it would be interesting to study whether one can apply fast matrix multiplication techniques to Algorithm 5 to improve the running time to sub-cubic time in the number of states.

REFERENCES

Abboud, A., Backurs, A., & Williams, V. V. (2018). If the current clique algorithms are optimal, so is Valiant's parser. *SIAM J. Comput.*, *47*(6).

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). *The design and analysis of computer algorithms*. Addison-Wesley.

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). Compilers, principles, techniques. *Addison wesley*, *7*(8), 9.

Altınel, M., & Franklin, M. J. (2000). Efficient filtering of XML documents for selective dissemination of information. In *Vldb* (pp. 53–64).

Alur, R., Fisman, D., Mamouras, K., Raghothaman, M., & Stanford, C. (2020). Streamable regular transductions. *Theor. Comput. Sci.*, *807*, 15–41.

Alur, R., & Madhusudan, P. (2004a). Visibly pushdown languages. In *Stoc.*

Alur, R., & Madhusudan, P. (2004b). Visibly pushdown languages. In *Proceedings of the 36th annual ACM symposium on theory of computing, chicago, il, usa, june 13-16, 2004* (pp. 202–211).

Amarilli, A., Bourhis, P., Jachiet, L., & Mengel, S. (2017). A circuit-based approach to efficient enumeration. In *Icalp* (Vol. 80, pp. 111:1–111:15).

Amarilli, A., Bourhis, P., Mengel, S., & Niewerth, M. (2019a). Constant-delay enumeration for nondeterministic document spanners. In *Icdt* (pp. 22:1–22:19).

Amarilli, A., Bourhis, P., Mengel, S., & Niewerth, M. (2019b). Enumeration on trees with tractable combined complexity and efficient updates. In *Pods* (pp. 89–103).

Amarilli, A., Bourhis, P., Mengel, S., & Niewerth, M. (2019c). Constant-delay enumeration for nondeterministic document spanners. In *icdt.*

Amarilli, A., Bourhis, P., Mengel, S., & Niewerth, M. (2019d). Enumeration on trees with tractable combined complexity and efficient updates. In *pods.*

Amarilli, A., Bourhis, P., Mengel, S., & Niewerth, M. (2020). Constant-delay enumeration for nondeterministic document spanners. *TODS.*

Arenas, M., Croquevielle, L. A., Jayaram, R., & Riveros, C. (2019). Efficient logspace classes for enumeration, counting, and uniform generation. In *Pods* (pp. 59–73).

Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (2002). Models and issues in data stream systems. In *Sigmod* (pp. 1–16).

Bagan, G. (2006a). MSO queries on tree decomposable structures are computable with linear delay. In *Csl* (pp. 167–181).

Bagan, G. (2006b). MSO queries on tree decomposable structures are computable with linear delay. In *Csl.*

Bagan, G., Durand, A., & Grandjean, E. (2007). On acyclic conjunctive queries and constant delay enumeration. In *Csl* (pp. 208–222).

Bar-Yossef, Z., Fontoura, M., & Josifovski, V. (2005). Buffering in query evaluation over XML streams. In *Pods* (pp. 216–227).

Bar-Yossef, Z., Fontoura, M., & Josifovski, V. (2007). On the memory requirements of XPath evaluation over XML streams. *J. Comput. Syst. Sci.*, *73*(3), 391–441.

Barloy, C., Murlak, F., & Paperman, C. (2021). Stackless processing of streamed trees. In *Pods.*

Berkholz, C., Gerhardt, F., & Schweikardt, N. (2020). Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, *7*(1), 4–33.

Berkholz, C., Keppeler, J., & Schweikardt, N. (2017). Answering conjunctive queries under updates. In *Pods* (pp. 303–318).

Berstel, J. (2013). *Transductions and context-free languages*. Springer-Verlag.

Boucher, C., Gagie, T., I, T., Köppl, D., Langmead, B., Manzini, G., ... Rossi, M. (2021). PHONI: streamed matching statistics with multi-genome references. In A. Bilgin, M. W. Marcellin, J. Serra-Sagristà, & J. A. Storer (Eds.), *31st data compression conference, DCC 2021, snowbird, ut, usa, march 23-26, 2021* (pp. 193–202). IEEE.

Bourhis, P., Grez, A., Jachiet, L., & Riveros, C. (2021). Ranked enumeration of MSO logic on words. In *Icdt* (Vol. 186, pp. 20:1–20:19).

Bourhis, P., Reutter, J. L., & Vrgoc, D. (2020). JSON: Data model and query languages. *Inf. Syst.*, *89*, 101478.

Brahem, M., Zeitouni, K., & Yeh, L. (2020). ASTROIDE: A unified astronomical big data processing engine over spark. *IEEE Trans. Big Data*, *6*(3), 477–491.

Bucchi, M., Grez, A., Quintana, A., Riveros, C., & Vansummeren, S. (2022). CORE: a complex event recognition engine. *Proc. VLDB Endow.*, *15*(9), 1951–1964.

Caralp, M., Reynier, P., & Talbot, J.-M. (2015). Trimming visibly pushdown automata. *Theor. Comput. Sci.*, *578*, 13–29.

Chen, S., & Lai, C. (2023). Patent litigation prediction using machine learning approaches. In C. Stephanidis, M. Antona, S. Ntoa, & G. Salvendy (Eds.), *HCI international 2023 posters - 25th international conference on human-computer interaction, HCII 2023, copenhagen, denmark, july 23-28, 2023, proceedings, part V* (Vol. 1836, pp. 389–395). Springer.

Chen, Y., Davidson, S. B., & Zheng, Y. (2006). An efficient XPath query processor for XML streams. In *Icde* (p. 79).

Chirkova, R., & Yang, J. (2012). Materialized views. *Found. Trends Databases*, *4*(4), 295–405.

Chomsky, N., & Schützenberger, M. P. (1959). The algebraic theory of context-free languages. In *Studies in logic and the foundations of mathematics* (Vol. 26).

Claude, F., & Navarro, G. (2011). Self-indexed grammar-based compression. *Fundam. Informaticae*, *111*(3), 313–337.

Courcelle, B. (2009). Linear delay enumeration and monadic second-order logic. *Discret. Appl. Math.*, *157*(12).

Driscoll, J. R., Sarnak, N., Sleator, D. D., & Tarjan, R. E. (1986a). Making data structures persistent. In *Stoc*.

Driscoll, J. R., Sarnak, N., Sleator, D. D., & Tarjan, R. E. (1986b). Making data structures persistent. In *Stoc* (pp. 109–121).

Durand, A., & Grandjean, E. (2007). First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. Comput. Log.*, *8*(4), 21.

Fagin, R., Kimelfeld, B., Reiss, F., & Vansummeren, S. (2015). Document spanners: A formal approach to information extraction. *J. ACM*, *62*(2), 12:1–12:51.

Fang, X., Wu, H., Jing, J., Meng, Y., Yu, B., Yu, H., & Zhang, H. (2024). NSEP: early fake news detection via news semantic environment perception. *Inf. Process. Manag.*, *61*(2), 103594.

Filiot, E., Gauwin, O., Reynier, P., & Servais, F. (2019). Streamability of nested word transductions. *LMCS*, *15*(2).

Filiot, E., Raskin, J., Reynier, P., Servais, F., & Talbot, J. (2018). Visibly pushdown transducers. *JCSS*, *97*, 147–181.

Florenzano, F., Riveros, C., Ugarte, M., Vansummeren, S., & Vrgoc, D. (2018). Constant delay algorithms for regular document spanners. In *Pods*.

Florenzano, F., Riveros, C., Ugarte, M., Vansummeren, S., & Vrgoc, D. (2020). Efficient enumeration algorithms for regular document spanners. *ACM Trans. Database Syst.*, *45*(1), 3:1–3:42.

Fortuna, P., Soler Company, J., & Wanner, L. (2021). How well do hate speech, toxicity, abusive and offensive language classification models generalize across datasets? *Inf. Process. Manag.*, *58*(3), 102524.

Freydenberger, D. D. (2019). A logic for document spanners. *Theory Comput. Syst.*, *63*(7), 1679–1754.

Freydenberger, D. D., Kimelfeld, B., & Peterfreund, L. (2018). Joining extractions of regular expressions. In *Pods* (pp. 137–149).

Gauwin, O., Niehren, J., & Roos, Y. (2008). Streaming tree automata. *Inf. Process. Lett.*, *109*(1), 13–17.

Gauwin, O., Niehren, J., & Tison, S. (2009a). Bounded delay and concurrency for earliest query answering. In *Lata* (Vol. 5457, pp. 350–361).

Gauwin, O., Niehren, J., & Tison, S. (2009b). Earliest query answering for deterministic nested word automata. In *Fct* (Vol. 5699, pp. 121–132).

Ginsburg, S., & Ullian, J. (1966). Ambiguity in context free languages. *JACM*, *13*(1).

Gou, G., & Chirkova, R. (2007). Efficient algorithms for evaluating XPath over streams. In *Sigmod* (pp. 269–280). ACM.

Green, T. J., Gupta, A., Miklau, G., Onizuka, M., & Suciu, D. (2004). Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, *29*(4), 752–788.

Grez, A., & Riveros, C. (2020). Towards streaming evaluation of queries with correlation in complex event processing. In *Icdt* (Vol. 155, pp. 14:1–14:17).

Grez, A., Riveros, C., & Ugarte, M. (2019). A formal framework for complex event processing. In *Icdt* (pp. 5:1–5:18).

Grez, A., Riveros, C., Ugarte, M., & Vansummeren, S. (2021). A formal framework for complex event recognition. *ACM Trans. Database Syst.*, *46*(4), 1–49.

Idris, M., Ugarte, M., & Vansummeren, S. (2017). The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *Sigmod* (pp. 1259–1274).

ITU. (2023). Facts and figures 2023 - internet use. `https://www.itu.int/itu-d/reports/statistics/2023/10/10/ff23-internet-use/`. (Accessed: 2024-02-20)

Jerrum, M., Valiant, L. G., & Vazirani, V. V. (1986). Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.*, *43*, 169–188.

Josifovski, V., Fontoura, M., & Barta, A. (2005). Querying XML streams. *VLDB J.*, *14*(2), 197–210.

Kara, A., Nikolic, M., Olteanu, D., & Zhang, H. (2020). Trade-offs in static and dynamic evaluation of hierarchical queries. In *Pods* (pp. 375–392).

Kieffer, J. C., & Yang, E. (2000). Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, *46*(3), 737–754.

Kumar, V., Madhusudan, P., & Viswanathan, M. (2007). Visibly pushdown automata for streaming XML. In *Www* (pp. 1053–1062).

Lange, M., & Leiß, H. (2009). To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Informatica Didactica*, *8*(2009).

Li, W., Shi, X., Huang, D., Shen, X., Chen, J., Kobayashi, H. H., … Shibasaki, R. (2023). Predlife: Predicting fine-grained future activity patterns. *IEEE Trans. Big Data*, *9*(6), 1658–1669.

Liang, W., Cao, J., Chen, L., Wang, Y., Wu, J., Beheshti, A., & Tang, J. (2023). Crime prediction with missing data via spatiotemporal regularized tensor decomposition. *IEEE Trans. Big Data*, *9*(5), 1392–1407.

Libkin, L. (2004). *Elements of finite model theory* (Vol. 41). Springer.

Lohrey, M. (2012). Algorithmics on SLP-compressed strings: A survey. *Groups Complex. Cryptol.*, *4*(2), 241–299.

Maturana, F., Riveros, C., & Vrgoc, D. (2018). Document spanners for extracting incomplete information: Expressiveness and complexity. In *Pods* (pp. 125–136).

Maurer, H. A. (1969). A direct proof of the inherent ambiguity of a simple context-free language. *JACM*, *16*(2).

Netflix. (2012). Netflix recommendations: Beyond the 5 stars (part 1). `https://netflixtechblog.com/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429`. (Accessed: 2024-02-20)

Nikolic, M., & Olteanu, D. (2018). Incremental view maintenance with triple lock factorization benefits. In *Sigmod* (pp. 365–380).

Nowotka, D., & Srba, J. (2007). Height-deterministic pushdown automata. In *Mfcs*.

Olteanu, D. (2007). SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. Knowl. Data Eng.*, *19*(7), 934–949.

Olteanu, D., Furche, T., & Bry, F. (2004). An efficient single-pass query evaluator for XML data streams. In *Sac* (pp. 627–631).

Olteanu, D., & Závodný, J. (2015). Size bounds for factorised representations of query results. *ACM TODS*, *40*(1), 2:1–2:44.

Peterfeund, L. (2019). *The complexity of relational queries over extractions from text* (Doctoral dissertation, Technion). Retrieved from `http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2019/PHD/PHD-2019-10.pdf`

Peterfreund, L. (2021). Grammars for document spanners. In *Icdt* (pp. 7:1–7:18).

Peterfreund, L. (2023). Enumerating grammar-based extractions. *Discrete Applied Mathematics*, *341*, 372-392.

Peterfreund, L., ten Cate, B., Fagin, R., & Kimelfeld, B. (2019). Recursive programs for document spanners. In *Icdt.*

Pin, J. (Ed.). (2021). *Handbook of automata theory*. European Mathematical Society Publishing House, Zürich, Switzerland.

Rytter, W. (2002). Application of lempel-ziv factorization to the approximation of grammar-based compression. In *Cpm* (Vol. 2373, pp. 20–31).

Schmid, M. L., & Schweikardt, N. (2021). Spanner evaluation over SLP-compressed documents. In *Pods* (pp. 153–165).

Schmid, M. L., & Schweikardt, N. (2022). Query evaluation over SLP-represented document databases with complex document editing. In *Pods* (pp. 79–89).

Schmitz, S. (2012). *Can all unambiguous grammars be parsed in linear time?* Theoretical Computer Science Stack Exchange. Retrieved from `https://cstheory.stackexchange.com/q/10504` (Version: 2012-03-02)

Segoufin, L. (2013). Enumerating with constant delay the answers to a query. In *Icdt* (pp. 10–20).

Segoufin, L., & Vianu, V. (2002). Validating streaming XML documents. In *Pods* (pp. 53–64).

Shalem, M., & Bar-Yossef, Z. (2008). The space complexity of processing XML twig queries over indexed documents. In *Icde* (pp. 824–832).

Similarweb. (2024). Top websites ranking. `https://www.similarweb.com/top-websites/`. (Accessed: 2024-02-20)

Storer, J. A., & Szymanski, T. G. (1982). Data compression via textual substitution. *J. ACM*, *29*(4), 928–951.

ten Cate, B., & Marx, M. (2007). Navigational XPath: calculus and algebra. *SIGMOD Record*, *36*(2), 19–26.

Torunczyk, S. (2020). Aggregate queries on sparse databases. In *Pods* (pp. 427–443).

Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, *23*(3), 337–343.