PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

EXPRESSIVENESS AND COMPLEXITY ANALYSIS OF INFORMATION EXTRACTION LANGUAGES

FRANCISCO JOSÉ MATURANA SANGUINETI

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
CRISTIAN RIVEROS

Santiago de Chile, March 2017

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

EXPRESSIVENESS AND COMPLEXITY ANALYSIS OF INFORMATION EXTRACTION LANGUAGES

FRANCISCO JOSÉ MATURANA SANGUINETI

Members of the Committee:
CRISTIAN RIVEROS
MARCELO ARENAS
PABLO BARCELÓ
HERNÁN SANTA MARÍA

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, March 2017

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Professor Cristian Riveros, for his continued guidance and encouragement throughout my studies. Without his kindness, dedication, and great knowledge this work would have not been possible.

I would like to express my gratitude towards Professor Domagoj Vrgoč for his invaluable assistance and advice as well as his contributions to this work.

Many thanks to Professor Marcelo Arenas for being an incredible teacher and for his help, which played a key role in the research leading to this thesis.

I also wish to thank the Center for Semantic Web Research and its members for their financial support and for providing valuable instances to learn from accomplished researchers and share with them.

Last, but not least, I would like to thank my family for giving them their unconditional support throughout my life and nurturing in me the curiosity and aspiration that lead me to choose this path.

My master's degree studies, as well as the research described in this thesis, were partially funded by grant *Núcleo Milenio Centro de Investigación de la Web Semántica* number NC120004, and grant *Fondecyt de Iniciación* number 11150653.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Rule-based information extraction has lately received a fair amount of attention from the database community, with several languages appearing in the last few years. Although information extraction systems are intended to deal with semistructured data, all language proposals introduced so far are designed to output relations, thus making them incapable of handling incomplete information. Moreover, little is known about how these proposals compare to each other in terms of expressive power, and we do not have a deep understanding of the complexity of these languages.

To remedy the situation, in this thesis we study the expressiveness and complexity of information extraction languages under a unifying framework supporting incomplete information. To this end, we propose a general framework subsuming previously proposed languages and allowing us to handle mappings instead of relations. We then use this framework to compare different methods for extracting information defined in the past, and study their computational properties such as query enumeration, satisfiability and equivalence. As we show, no previously proposed method reigns supreme, however, by combining several characteristics of different approaches we can obtain an expressive information extraction formalism which can be efficiently implemented and used in practice.

Keywords: Information Extraction; Automata Theory; Formal Languages; Database Theory.

RESUMEN

Los lenguajes de Extracción de Información (EI) en base a reglas han recibido atención de parte de la comunidad de bases de datos últimamente, con varios lenguajes nuevos apareciendo en los últimos años. A pesar de que los sistemas de EI suelen procesar datos semi-estructurados, todos los lenguajes que se han propuesto hasta ahora están diseñados para producir relaciones y, por lo tanto, son incapaces de trabajar con información incompleta. Además, existe poco conocimiento acerca de cómo estas propuestas se comparan en términos de poder expresivo y complejidad.

Para remediar esto, esta tesis estudia la expresividad y complejidad de distintos lenguajes de EI a través de un marco teórico unificador con soporte para información incompleta. Con este fin, se propone un lenguaje que generaliza otras propuestas anteriores y que utiliza funciones parciales (también llamadas *mappings*) en lugar de relaciones. Luego utilizamos este lenguaje general para comparar distintos métodos de EI definidos en el pasado y estudiar sus propiedades computacionales, tales como: enumeración de consultas, satisfacibilidad y equivalencia. Como se muestra, ninguno de los métodos propuestos domina a los otros, sin embargo, combinando ciertas características de distintos enfoques se puede obtener un lenguaje para EI que es expresivo, puede implementarse eficientemente y puede ser utilizado en la práctica.

Palabras claves: Extracción de Información; Teoría de Automatas; Lenguajes Formales; Teoría de Bases de Datos.

1. INTRODUCTION

With the abundance of different formats arising in practice these days, there is a great need for methods extracting singular pieces of data from a variety of distinct files. This process, known as information extraction, or IE for short, is particularly prevalent in big corporations that manage systems of increasing complexity which need to incorporate data coming from different sources. As a result, a number of systems supporting the extraction of information from text-like data have appeared throughout the years (Cunningham, 2002; Krishnamurthy et al., 2008; Shen, Doan, Naughton, & Ramakrishnan, 2007), and the topic received a substantial coverage in research literature (see Kimelfeld, 2014 for a good survey).

Historically, there have been two main approaches to information extraction: the *statistical* approach utilizing machine-learning methods, and the *rule-based* approach utilizing traditional finite-language methods. In this thesis we focus on the rule-based approach to information extraction for two reasons. First, rule-based information extraction is founded on methods traditionally used by the database community (such as logic, automata, or datalog-based languages) and has therefore enjoyed a great amount of coverage in the database literature (Arenas, Maturana, Riveros, & Vrgoč, 2016; Fagin, Kimelfeld, Reiss, & Vansummeren, 2014, 2015; Freydenberger & Holldack, 2016). Second, as argued by Chiticariu, Li, and Reiss (2013); Kimelfeld (2014), due to their simplicity and ease of use, rule-based systems seem to be more prevalent in the industrial solutions.

Generally, most rule-based information extraction frameworks deploy some form of regular-like expressions as the core mechanism for extracting the data (Arenas et al., 2016; Califf & Mooney, 1999; Fagin et al., 2015). Perhaps the best representative of this approach are the regular expressions with capture variables introduced by Fagin et al. (2015), called *regex formulas*, which form the basis of IBM's commercial IE tool SystemT (Krishnamurthy et al., 2008). In this setting, documents being processed are viewed as strings, which is a natural assumption for a wide variety of formats in use today (e.g. plain text, CSV files, JSON documents, etc.). The pieces of information we want to extract are captured by *spans*,

which are simply intervals inside the string representing our document (that is, spans are defined by the starting point and the ending point of some substring). Regex formulas are now used to parse these strings, with variables storing *spans*.

Once spans have been extracted using regular-like expressions, most IE frameworks allow combining them and controlling their structure through a variety of different methods. For instance, Fagin et al. (2015) permit manipulating spans extracted by regex formulas using algebraic operations, while Arenas et al. (2016) and Shen et al. (2007) deploy datalog-like programs to obtain a more general way of defining relations over spans. Unlike regex, which specify variable filters explicitly inside formulas, Arenas et al. (2016) use a more visual approach implemented through *extraction rules*. These permit constraining the shape of a span captured inside a variable separately.

The way that most frameworks view information extraction is by defining a relation over spans. For example, in regex formulas of Fagin et al. (2015) all variables must be assigned in order to produce an output tuple, and a similar thing happens with the rule-based language of Arenas et al. (2016). However, in practice we are often working with documents which have missing information or optional parts, and would therefore like to maximise the amount of information we extract from a document. To illustrate this, consider a CSV file[1] containing land registry records about buying and selling property. In Table 1.1 we give a few rows of such a document, where "␣" represents space and "↲" the new line symbol. Some sellers in this file have an additional field which contains the amount of tax they paid when selling the property. If we are extracting information about sellers (for instance their names) from such a file, we would then like to also include the tax information when it is available. However, if we are set on extracting relations, this will not be possible, since all variables need to be assigned in order to produce an output.

[1]CSV, or comma separated values, is a simple table-like format storing information separated by commas and new lines.

Table 1.1. Part of a CSV document containing information about buying and selling property.

```
Seller: John, ID75↲
Buyer: Marcelo, ID832, P78↲
Seller: Mark, ID7, $35,000↲
⋮
```

Apart from the inability to capture partial or incomplete information, some further shortcomings of previous approaches are that defining the semantics of extraction expressions cannot be done in a fully declarative way (Fagin et al., 2015), or that they assign arbitrary spans to variables when these are not matched against the document (Arenas et al., 2016). Furthermore, not much is known about how these approaches compare in terms of expressive power, and apart from some preliminary studies on their computational properties (Arenas et al., 2016; Freydenberger & Holldack, 2016), we do not have a good understanding of how difficult it is to evaluate these languages, nor of the complexity of their main static tasks.

In order to overcome these issues, we propose a general framework subsuming several previous approaches to IE. For this, we consider core extraction mechanisms such as variants of regular expressions and automata, and extend them in order to accommodate incompleteness. We start with regex formulas of Fagin et al. (2015) as the base, and redefine their semantics in such a way that they output *mappings* in place of relations, as it was done for the SPARQL query language in the Semantic Web context (Pérez, Arenas, & Gutierrez, 2009). This allows us to capture optional parts of documents, such as in the example from Table 1.1, since our expression will output a mapping that binds an extra variable to the tax data only when the latter is present in the document (otherwise we only extract the name of the seller). Furthermore, the generality of mappings allows us to have a simple declarative semantics of regex formulas. We also extend the automata models of Fagin et al. (2015) to this new setting and show how they can capture the extraction expressions under the new semantics.

Next, we study how the formal IE frameworks introduced previously compare in terms of expressive power. Here we start with regex formulas of Fagin et al. (2015) and span regular expressions of Arenas et al. (2016), since these form the information extraction base of the two frameworks. We prove that both languages are special cases of our definition[2], with the latter being less expressive. We then analyse extraction rules of Arenas et al. (2016) showing how these can be used to capture regex formulas and also provide a method for simplifying such rules.

After this we consider computational aspects of base IE formalisms, starting with the combined complexity of evaluating extraction expressions over documents. Here we isolate a decision problem which, once solved efficiently, would allow us to enumerate all mappings an expression outputs when matched to a document. Since the size of the answer is potentially exponential here, our objective is to obtain an *incremental polynomial time algorithm* (Johnson, Yannakakis, & Papadimitriou, 1988), that is, an enumeration algorithm that takes polynomial time between each output. As we show, this is generally not possible, but we do isolate a well-behaved fragment, called *sequential regex*, which properly includes *functional regex* introduced by Fagin et al. (2015). We also analyze the evaluation problem parametrized by the number of variables and show that the problem is *fixed parameter tractable* (Flum & Grohe, 2006) for all expressions and automata models studied in this thesis.

Then, we study static analysis of these languages, focusing on satisfiability and containment. While satisfiability is NP-hard for unrestricted languages, the sequentiality restriction we introduce when studying evaluation allows us to solve the problem efficiently. On the other hand, containment is bound to be PSPACE-hard, since all of our IE formalisms contain regular expressions, with a matching upper bound giving us completeness for the class. Since one way to lower this bound for regular languages is to consider deterministic models, we show how determinism can be introduced to IE languages and study how it affects the complexity.

[2]Note that in this thesis we do not consider the content operator of Arenas et al. (2016), nor the string selection of Fagin et al. (2015), since these do not directly extract information, but rather compare two pieces of existing data.

Finally, we show an application of our framework to the task of annotating documents. This task consists in associating certain regions of a document to specific annotations, which typically function as metadata. To this end, we specify a fragment of our IE language that covers all the common use cases. In addition to this, we provide an algorithm that efficiently evaluates this fragment along with experimental results that show the performance of this algorithm on real-world data.

Organisation. We define documents, spans and mappings in Chapter 2. The unifying framework is introduced in Chapter 3, where we define different notions of extraction expressions and automata, and show how are they are connected. Expressiveness of IE languages is studied in Chapter 4, and the complexity of their evaluation in Chapter 5. We then tackle static analysis in Chapter 6 and present an application of our framework to the task of annotating documents in Chapter 7 . Finally, we conclude in Chapter 8 by outlining our key contributions and discussing future challenges.

2. PRELIMINARIES

Documents and spans. Let Σ be a finite alphabet. A document d, from which we will extract information, is a string over Σ. We define the length of d, denoted by $|d|$, as the length of this string. As done in previous approaches (Arenas et al., 2016; Fagin et al., 2015), and implemented in practical information extraction tools (Chiticariu et al., 2010), we use the notion of a *span* to capture the part of a document d that we wish to extract. Formally, a span p of a document d is a pair (i, j) such that $1 \leq i \leq j \leq |d| + 1$, where $|d|$ is the length of the string d. Intuitively, p represents a continuous region of the document d, whose content is the infix of d between positions i and $j - 1$. The set of all spans associated with a document d, denoted span(d), is then defined as the set $\{(i, j) \mid i, j \in \{1, \ldots, |d| + 1\} \text{ and } i \leq j\}$. Every span $p = (i, j)$ of d has an associated content, which is denoted by $d(p)$ or $d(i, j)$, and is defined as the substring of d from position i to position $j - 1$. Notice that if $i = j$, then $d(p) = d(i, j) = \varepsilon$. Given two spans $s_1 = (i_1, j_1)$ and $s_2 = (i_2, j_2)$, if $j_1 = i_2$ then their concatenation is equal to (i_1, j_2) and it is denoted $s_1 \cdot s_2$.

As an example, consider the following document d_0, where the positions are enumerated and "␣" denotes the white space character:

```
I n f o r m a t i o n ␣ e x t r a c t i o n
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
```

Here the length of d_0 is 22 and the span $p_0 = (1, 23)$ corresponds to the entire document d_0. On the other hand, the span $p_1 = (1, 12)$ corresponds to the first word of our document and its content $d(p_1) = d(1, 12)$ equals the string "`Information`". Similarly, for the span $p_2 = (13, 23)$ we have that $d(p_2) =$ `extraction`, i.e., it spans the second word of our document.

Mappings. In the introduction we argued that the traditional approaches to information extraction that store spans into relations might be somewhat limited when we are processing documents which might have optional parts, or contain incomplete information. Therefore to overcome these issues, we define the process of extracting information from a document d as if we were defining a partial function from a set of variables to the spans of d. The

use of partial functions for managing optional information has been considered before, for example, in the context of the Semantic Web (Pérez et al., 2009). Formally, let $\mathcal{V}$ be a set of variables disjoint from Σ. For a document d, a *mapping*, is a partial function from the set of variables $\mathcal{V}$ to span(d). The *domain* of a mapping μ (denoted dom(μ)) is the set of variables for which μ is defined. For instance, if we consider the document d_0 above, then the mapping μ_0 which assigns the span p_1 to the variable x and leaves all other variables undefined, extracts the first word from d_0.

Two mappings μ_1 and μ_2 are said to be *compatible* (denoted $\mu_1 \sim \mu_2$) if $\mu_1(x) = \mu_2(x)$ for every x in $\text{dom}(\mu_1) \cap \text{dom}(\mu_2)$. If $\mu_1 \sim \mu_2$, then $\mu_1 \cup \mu_2$ denotes the mapping that results from extending μ_1 with the values from μ_2 on all the variables in $\text{dom}(\mu_2) \setminus \text{dom}(\mu_1)$. The *empty mapping*, denoted by $\mu_\emptyset$, is the mapping such that $\text{dom}(\mu_\emptyset) = \emptyset$. Similarly, $[x \to s]$ denotes the mapping that only defines the value of variable x and assigns it to be the span s. The *join* of two set of mappings M_1 and M_2 is defined as follows:

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

Finally, we say that a mapping μ is *hierarchical* if for every $x, y \in \text{dom}(\mu)$, either: $\mu(x)$ is contained in $\mu(y)$, $\mu(y)$ is contained in $\mu(x)$, or $\mu(x)$ and $\mu(y)$ are disjoint. By extension, a set of mappings is said to be hierarchical if it only contains hierarchical mappings.

3. UNIFYING FRAMEWORK

In this chapter , we introduce a theoretical framework for information extraction that encompasses previous approaches proposed in the literature and allows us to manipulate partial and/or incomplete information often arising in practical applications. The idea behind our framework is quite simple, namely, we use mappings in place of relations to allow extracting the maximal amount of information from the document we are processing. For this, we borrow the syntax of *variable regex* and *variable automata* (Fagin et al., 2015), and redefine (and simplify) their semantics to output mappings. We start by redefining regular expressions with capture variables and show how our proposal subsumes extraction languages from Fagin et al. (2015) and Arenas et al. (2016). We then define variable stack and variable set automata for extracting mappings of spans and discuss their connection with extraction expressions under the new semantics.

3.1. Extracting information using RGX

Previous approaches to information extraction (Arenas et al., 2016; Fagin et al., 2015; Shen et al., 2007; Soderland, 1999) use some form of regular expressions with capture variables in order to obtain the desired spans. Intuitively, in such expressions we use ordinary regular languages to move through our document, thus jumping to the start of a span that we want to capture. The variables are then used to store the desired span, with further subexpressions controlling the shape of the span we wish to capture. Following Fagin et al. (2015) and Arenas et al. (2016), we define a class of regular expressions with variables, called *variable regex*.

Let Σ be a finite alphabet and $\mathcal{V}$ a set of variables disjoint with Σ. A *variable regex* (RGX) is defined by the following grammar:

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

where $a \in \Sigma$ is a letter in the alphabet and $x \in \mathcal{V}$ is a variable. For a RGX γ we define $\text{var}(\gamma)$ as the set of all variables occurring in γ.

Just as in the previously introduced information extraction languages, RGX use regular expressions to navigate the document, while a subexpression of the form $x\{\gamma\}$ stores a span starting at the current position and matching γ into the variable x. For example, if we wanted to extract the name of each seller from the document in Table 1.1, we could use the following RGX:

$$\Sigma^* \cdot \texttt{Seller:}\sqcup \cdot x\{(\Sigma - \{\texttt{,}\})^*\}\cdot\texttt{,}\cdot\Sigma^*$$

where Σ stands for the disjunction of all the letters of the alphabet, and where the concatenation symbols between alphabet letters is omitted for convenience. Here the subexpression $\Sigma^* \cdot \texttt{Seller:}\sqcup$ navigates to the position in our document where the name of some seller starts. The variable x then stores a string not containing a comma until it reaches the first comma—that is, the full name of our seller. The rest of the expression then simply matches the rest of the document.

Note that, syntactically, our expressions are the same as the ones introduced by Fagin et al. (2015) and as we show in Chapter 3.2, the expressions of Arenas et al. (2016) can be seen as a restriction of variable regex. The only explicit difference from Fagin et al. (2015) is that we do not allow the empty language $\emptyset$ explicitly in order to make some of the constructions more elegant. Adding this variant would not affect any of the results, though.

In contrast to previous approaches, our semantics views RGX formulas as expressions defining mappings and not only relations. To illustrate how this works, consider again the document in Table 1.1, but now suppose that we want to extract the names of the sellers and, when available, also the amount of tax they paid (recall from the Introduction that not all rows have this information). For this, consider the following RGX:

$$\Sigma^* \cdot \texttt{Seller:}\sqcup \cdot x\{R'\} \cdot \texttt{,} \cdot R' \cdot (\texttt{,} \cdot \sqcup \cdot y\{(\Sigma - \{\text{↲}\})^*\} \vee \varepsilon) \cdot \Sigma^*,$$

where $R' = (\Sigma - \{\texttt{,}, \text{↲}\})^*$. Note that this expression extracts the information about the amount of tax paid into the variable y only when this data is present in the document (otherwise, it matches ε). This defines two types of mappings: the first kind will contain

only the names of sellers (stored into the variable x), while the second kind will contain both the name and the amount of tax paid (stored into y), when the latter information is present in our file.

Table 3.1. The semantics $[\![\gamma]\!]_d$ of a RGX γ over a document d. Here R^2 is a shorthand for $R \cdot R$, similarly R^3 for $R \cdot R \cdot R$, etc.

$$
\begin{aligned}
[\![\gamma]\!]_d &= \{\mu \mid ((1, |d| + 1), \mu) \in [\gamma]_d\} \\
[\varepsilon]_d &= \{(s, \mu_\emptyset) \mid s \in \text{span}(d) \text{ and } s = (i, i)\} \\
[a]_d &= \{(s, \mu_\emptyset) \mid s \in \text{span}(d) \text{ and } d(s) = a\} \\
[x\{R\}]_d &= \{(s, \mu) \mid \exists (s, \mu') \in [R]_d : x \notin \text{dom}(\mu') \text{ and } \mu = \mu' \cup [x \to s]\} \\
[R_1 \cdot R_2]_d &= \{(s, \mu) \mid \exists (s_1, \mu_1) \in [R_1]_d, \exists (s_2, \mu_2) \in [R_2]_d : \\
&\qquad s = s_1 \cdot s_2,\ \text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset,\ \text{and } \mu = \mu_1 \cup \mu_2\} \\
[R_1 \vee R_2]_d &= [R_1]_d \cup [R_2]_d \\
[R^*]_d &= [\varepsilon]_d \cup [R]_d \cup [R^2]_d \cup [R^3]_d \cup \cdots
\end{aligned}
$$

The full semantics of RGX expressions is defined in Table 3.1. As explained above, we view our expression γ as a way of defining a partial mapping $\mu : \text{var}(\gamma) \rightharpoonup \text{span}(d)$. These semantics have two layers. The first layer (denoted $[\gamma]_d$) defines a set of pairs that contain a span of d where γ matches, and the corresponding mapping obtained from that match. For instance, the alphabet letter a must match a part of the document equal to a and it defines no mapping. Correspondingly, a subexpression of the form $x\{R\}$ assigns to x the span captured by R (while preserving the previous variable assignments). Similarly, in the case of concatenation $R_1 \cdot R_2$ we join the mappings defined on the left with the ones defined on the right, while imposing the restriction that the same variable is not assigned in both parts (as this would lead to inconsistencies). The second layer of our semantics (denoted $[\![\gamma]\!]_d$) then simply gives us the mappings that γ defines when matching the entire document. Note

that in the case of an ordinary regular expression we output the empty mapping singleton (representing `TRUE`) when the expression matches the entire document and the empty set (representing `FALSE`) when not, thus making RGX a natural generalisation of ordinary regular expressions with the ability to extract spans.

It is worthwhile mentioning that the denotational semantics introduced here is much simpler than the semantics of variable regex defined by Fagin et al. (2015). In Table 3.1, we give the semantics of our framework directly in terms of spans and mappings. On the other hand, the semantics of variable regex by Fagin et al. (2015) are given through the so-called *parse trees*: syntactical structures that represent the evaluation of an expression over a document. We believe that one contribution of our work is the simplification of the semantics which could help us better understand variable regex and other IE languages.

3.2. Connection with previous approaches

Having the general definition of RGX formulas which define mappings, we can now show how this framework subsumes some previously proposed classes of regular expressions used in information extraction. Most notably, we compare with regex formulas introduced by Fagin et al. (2015) and span regular expressions from Arenas et al. (2016). We start with regex formulas.

Although the expressions from Fagin et al. (2015) use the same syntax as our RGX formulas, the setting they consider dictates that extraction expressions always define relations. This automatically excludes expressions such as $R_1 \vee R_2$ from Chapter 3.1 which allows mappings with different domains. What Fagin et al. (2015) proposes instead is that each mapping defined by an expression assigns precisely the same variables every time (and also all of them); that is, we want our expressions to act as functions. As shown in Fagin et al. (2015), there is a very easy syntactic criteria for this, resulting in functional RGX formulas.

A RGX γ is called *functional with respect to the set of variables X* if one of the following syntactic restrictions holds:

- $\gamma = \varepsilon$, $\gamma = w$ for some $w \in \Sigma$ and $X = \emptyset$.

- $\gamma = \varphi_1 \vee \varphi_2$, where φ_1, φ_2 are functional with respect to X (implying that $\text{var}(\varphi_1) = \text{var}(\varphi_2)$).
- $\gamma = \varphi_1 \cdot \varphi_2$, where φ_1 is functional with respect to $X' \subseteq X$ and φ_2 is functional with respect to $X \setminus X'$ (implying that $\text{var}(\varphi_1) \cap \text{var}(\varphi_2) = \emptyset$).
- $\gamma = (\varphi)^*$, where $\text{var}(\varphi) = \emptyset$ and $X = \emptyset$.
- $\gamma = x\{\gamma'\}$ where $x \in X$ and γ' is functional with respect to $X \setminus \{x\}$.

A RGX γ is called *functional* if it is functional with respect to $\text{var}(\gamma)$.

This condition ensures that each variable mentioned in γ will appear exactly once in every word that can be derived from γ, when we treat γ as a classical regular expression with variables as part of the alphabet. We refer to the class of *functional* RGX as funcRGX. Note that this corresponds to the original definition of regex formulas given by Fagin et al. (2015), even when we consider the new semantics. Thus, we have:

THEOREM 3.1. *Regex formulas of Fagin et al. (2015) are equivalent to the class* funcRGX *defined above.* □

Next, we show how RGX formulas subsume span regular expressions of Arenas et al. (2016). To do this, note that span regular expressions can be seen as RGX formulas defined above, but where the subexpressions of the form $x\{\gamma\}$ allow only for $\gamma = \Sigma^*$. That is, when we have no control over the shape of the span we are capturing, and where we cannot nest variables. We call such formulas *span* RGX *formulas* and denote them by spanRGX. For simplicity, we omit Σ^* after variables when showing these formulas.

To compare spanRGX with span regular expressions, we also need to take note of the semantics proposed in Arenas et al. (2016). One problem with that semantics is that when a variable is not matched by the expression, the resulting mapping is assigned an arbitrary span, which does not seem the correct approach since it can mislead us in some cases. For instance, in our example with names and tax information above we will always assign spans to both variables, thus not knowing if the tax information is really associated with the correct name. Of course, this type of behaviour can easily be simulated by "joining"

the results obtained by spanRGX with the set of all total mappings or, in other words, constructing a new set of mappings where unassigned variables are mapped to all possible spans. Another, more subtle problem, is that the formalism of Arenas et al. (2016) allows expressions of the form $x\{\Sigma^*\} \cdot x\{\Sigma^*\}$ (forcing x to be assigned the empty span at the same position multiple times), while this RGX is not satisfiable. We call span regular expressions which prohibit such behaviour *proper*. We now obtain the following:

THEOREM 3.2. *Let d be a document, γ be a* RGX*, M be the set of all total functions from* $\mathrm{var}(\gamma)$ *to span*(d)*, and let* $[\![\gamma]\!]'_d = M \bowtie [\![\gamma]\!]_d$*. Under these semantics,* spanRGX *and proper span regular expressions of Arenas et al. (2016) are equivalent.* □

We can therefore conclude that the proposed framework indeed generalises the two previously proposed classes of information extraction expressions.

3.3. Automata for information extraction

In this section, we define two automata models for capturing spans. We start with the automata model equivalent to RGX and then extend it in order to allow defining complex sets of mappings. Just as with RGX, the definitions of our automata come from Fagin et al. (2015), however, we need to redefine the semantics to support mappings.

Variable-stack automaton. This class of automata operates in a way analogous to RGX; that is, it behaves as an usual finite state automaton, except that it can also open and close variables. To mimic the way this happens in RGX, variable-stack automata use a stack in order to track which variables are open and when to close them.

Formally, a *variable-stack automaton* ($\mathrm{VA_{stk}}$) is a tuple (Q, q_0, q_f, δ), where: Q is a finite set of *states*; $q_0 \in Q$ is the *initial state*; $q_f \in Q$ is the *final state*; and δ is a *transition relation* consisting of triples of the forms (q, w, q'), (q, ε, q'), $(q, x\vdash, q')$ or $(q, \dashv, q')$, where $q, q' \in Q$, $w \in \Sigma$, $x \in \mathcal{V}$, $\vdash$ is a special *open* symbol, and $\dashv$ is a special *close* symbol. For a

VA_{stk} automaton A we define the set $\text{var}(A)$ as the set of all variables x such that $x\vdash$ appears in some transition of A. Figure 3.1a shows an example of a variable-stack automaton.

A *configuration* of a VA_{stk} automaton A is a tuple (q, V, Y, i), where $q \in Q$ is the *current state*; $V \in \text{var}(A)^*$ is the stack of *active variables*; $Y \subseteq \text{var}(A)$ is the set of *available variables*; and $i \in [1, |d| + 1]$ is the *current position*. A *run* ρ of A over document $d = a_1 a_2 \cdots a_n$ is a sequence of configurations $c_0, c_1, \ldots, c_m$ where $c_0 = (q_0, \emptyset, \text{var}(A), 1)$ and for every $j \in [0, m-1]$, one of the following holds for $c_j = (q_j, V_j, Y_j, i_j)$ and $c_{j+1} = (q_{j+1}, V_{j+1}, Y_{j+1}, i_{j+1})$:

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

(ii) $i_{j+1} = i_j$ and for some $x \in \text{var}(A)$, either
  (a) $x \in Y_j$, $V_{j+1} = V_j \cdot x$, $Y_{j+1} = Y_j \setminus \{x\}$, and $(q_j, x\vdash, q_{j+1}) \in \delta$ (variable push), or
  (b) $V_j = V_{j+1} \cdot x$, $Y_{j+1} = Y_j$ and $(q_j, \dashv, q_{j+1}) \in \delta$ (variable pop).

The set of runs of A over a document d is denoted $\text{Runs}(A, d)$. A run $\rho = c_0, \ldots, c_m$ is *accepting* if $c_m = (q_f, V_m, Y_m, |d| + 1)$. The set of accepting runs of A over d is denoted $\text{ARuns}(A, d)$. Let $\rho \in \text{ARuns}(A, d)$, then for each variable $x \in \text{var}(A) \setminus (Y_m \cup V_m)$ there are configurations $c_b = (q_b, V_b, Y_b, i_b)$ and $c_e = (q_e, V_e, Y_e, i_e)$ such that V_b is the first one in the run where x occurs and V_e (with $e \neq m$) is the last one in the run where x occurs; the span (i_b, i_e) is denoted by $\rho(x)$. The mapping μ^ρ is such that $\mu^\rho(x)$ is $\rho(x)$ if $x \in \text{var}(A) \setminus (Y_m \cup V_m)$, and undefined otherwise. Finally, the semantics of A over D, denoted by $[\![A]\!]_d$, are defined as the set $\{\mu^\rho \mid \rho \in \text{ARuns}(A, d)\}$.

Note here that the only difference between our definition and Fagin et al. (2015) is how we define accepting runs and the mappings μ^ρ. In particular, we do not impose that all the variables in $\text{var}(A)$ should be used in the run, and we also allow some of them to remain on the stack. Furthermore, we leave our mappings undefined for any unused variable.

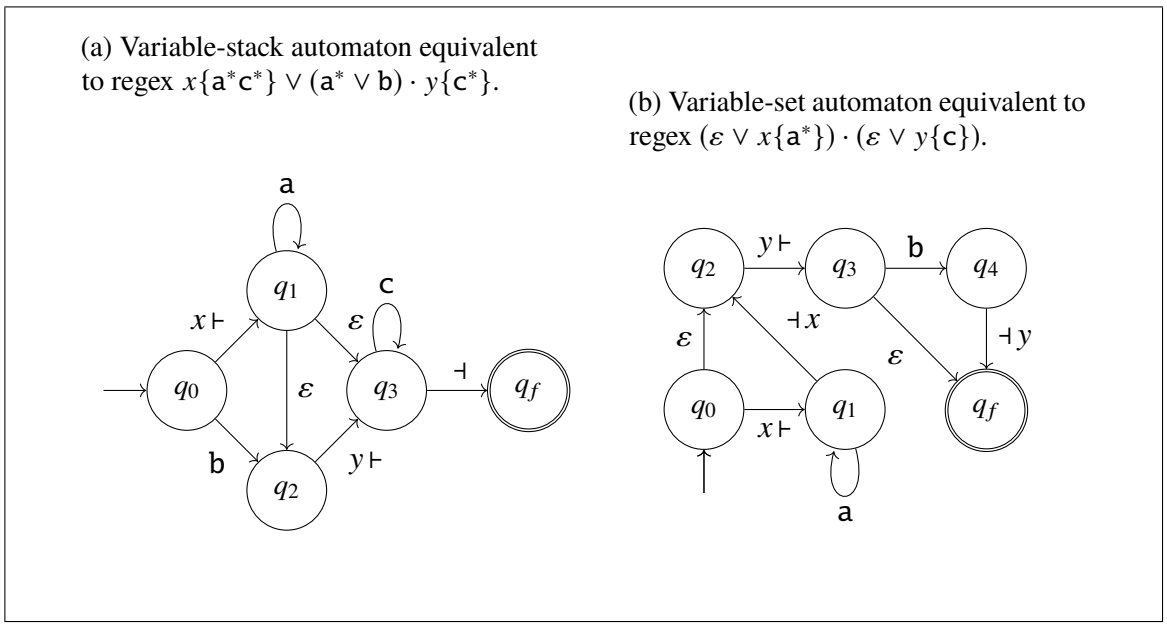


Figure 3.1. Examples of variable automata.

Variable-set automaton. Following Fagin et al. (2015), we introduce a more general class of automata which allow defining mappings that are not necessarily hierarchical as in the case of VA_{stk} automata and RGX. We call these automata *variable-set automata* (VA). The definition of variable-set automata is almost identical to the one of VA_{stk} automata, but we now have transitions of the form $(q, \dashv x, q')$ instead of $(q, \dashv, q')$, that allow us to explicitly state which variable is closed. Likewise, instead of a stack, they operate using a *set*, thus allowing us to add and remove variables in any order. Figure 3.1b shows an example of a variable-set automaton.

The only difference between VA and VA_{stk} automata is in the condition 2.(b) of a run, where we directly stipulate which variable should be removed from the set V_j (this used to be a stack in VA_{stk}). Acceptance is defined analogously as before. To avoid repeating the same definition we refer the reader to Fagin et al. (2015) for details, taking note of the new semantics. We also discuss how the results of Fagin et al. (2015) can be preserved under these new semantics in Chapter 3.4.

Similarly to functional RGX, it also makes sense to consider automata which force their runs to be functional. A run ρ of an automaton A is *functional* if $\mathrm{dom}(\mu^\rho) = \mathrm{var}(A)$ (in other words, μ^ρ is a total function). A *functionally restricted* variable automaton (fVA) is such that it only considers functional runs as valid runs. Note that these restrictions can be defined for $\mathrm{VA}_{\mathrm{stk}}$ automata in the same manner. Lastly, we say that a VA is *hierarchical* if every mapping it produces is hierarchical.

3.4. Comparing expressions to automata

One of the main problems studied by Fagin et al. (2015) was to determine the precise relationship between the automata models introduced in the previous section (restricted such that they always output relations) with the class of functional RGX formulas. As our framework is a generalisation of Fagin et al. (2015) that allows mappings instead of simple relations, here we show how main results on fVA and funcRGX can be generalised to our setting. We start by showing that the class of RGX formulas is also captured by $\mathrm{VA}_{\mathrm{stk}}$ automata in our new setting.

Theorem 3.3 (Fagin et al. 2015). *Every* $\mathrm{VA}_{\mathrm{stk}}$ *automaton has an equivalent* RGX *formula and vice versa. That is* $\mathrm{VA}_{\mathrm{stk}} \equiv \mathrm{RGX}$.

Proof. Just as in the proof for the relational case (Fagin et al., 2015), the main step is to show that $\mathrm{VA}_{\mathrm{stk}}$ automata can be simplified by decomposing them into an (exponential) union of disjoint paths known as $\mathrm{PU}_{\mathrm{stk}}$ (path union $\mathrm{VA}_{\mathrm{stk}}$). In $\mathrm{PU}_{\mathrm{stk}}$ automata each path is essentially a functional RGX formula, thus making the transformation straightforward. The only difference to the proof of Fagin et al. (2015) is that when transforming $\mathrm{VA}_{\mathrm{stk}}$ automaton into a union of paths, we need to consider all paths of length at most $2 \cdot k + 1$ in order to accommodate partial mappings. The notion of a consistent path also changes, since we are allowed to open a variable, but never close it. As a corollary we get that every

RGX is equivalent to (a potentially exponential) union of functional RGX formulas (with this union being empty when the RGX is not satisfiable).

Given a RGX, we can obtain an equivalent VA_{stk} by using an adapted version of *Thompson's construction algorithm* (Hopcroft & Ullman, 1979). This is the classical algorithm for building an automaton from a regular expression, and it can be easily adapted to consider variable operations. It is also worth noting that this algorithm runs in polynomial time with respect to the size of the expression. □

Similarly as in the functional case, it is also straightforward to prove that the mappings defined by VA_{stk} and RGX are hierarchical. Furthermore, just as Fagin et al. (2015) did, one can show that the class of VA automata which produce only hierarchical mappings is equivalent to RGX in the general case.

Theorem 3.4 (Fagin et al. 2015). *Every* VA *automaton that is hierarchical has an equivalent* RGX *formula and vice versa.* □

Both VA and VA_{stk} automata, as well as RGX, provide a simple way of extracting information. To permit a more complex way of defining extracted relations, Fagin et al. (2015) allow combining them using basic algebraic operations of union, projection and join. While defining a union or projection of two automata or RGX is straightforward, in the case of join we now use joins of mappings instead of the natural join (as used by Fagin et al. 2015). Formally, for two VA automatons A_1 and A_2, we define the "join automaton" $A_1 \bowtie A_2$ using the following semantics: for a document d, we have $[\![A_1 \bowtie A_2]\!]_d = [\![A_1]\!]_d \bowtie [\![A_2]\!]_d$. We denote the class of extraction expressions obtained by closing VA under union, projection and join with $\text{VA}^{\{\cup,\pi,\bowtie\}}$, and similarly for VA_{stk} and RGX.

To establish a relationship between algebras based on VA_{stk} and VA automata, Fagin et al. (2015) show that VA is closed under union, projection and join. We can show that the same holds true when dealing with mappings, but now the proofs change quite a bit. That is, while closure under projection is much easier to prove in our setting, closure under join now requires an exponential blowup, since to join mappings, we need to keep track of

variables opened by each mapping in our automaton. Similarly, Fagin et al. (2015) show that each VA automaton can be expressed using the expressions in the algebra $\mathrm{VA}_{\mathrm{stk}}^{\{\cup,\pi,\bowtie\}}$; as this proof holds verbatim in the case of mappings we obtain the following.

THEOREM 3.5 (FAGIN ET AL. 2015). $\mathrm{VA}^{\{\cup,\pi,\bowtie\}} \equiv \mathrm{VA} \equiv \mathrm{VA}_{\mathrm{stk}}^{\{\cup,\pi,\bowtie\}}$. □

As we showed here, the main results from Fagin et al. (2015) can be lifted to hold in the more general setting of mappings, thus suggesting that the added generality does not impact the intuition behind the extraction process.

4. EXPRESSIVENESS OF IE LANGUAGES

In chapter 3 we defined a unifying framework subsuming previous approaches to information extraction. As shown there, RGX formulas of Fagin et al. (2015) strictly subsume spanRGX of Arenas et al. (2016). However, the approach of the latter also allows combining spanRGX formulas into rules. Therefore, in this chapter we study the exact connection between the two frameworks. We start by defining rules based on spanRGX, and then isolate a simple subclass of such rules that is equivalent to RGX. We also study how rules can be simplified by removing cycles, and compare RGX to more general classes of rules.

4.1. Rules for information extraction

Regex formulas allow the user to specify the shape of the span captured by some variable using an expression of the form $x\{\gamma\}$. For example, if we wanted to say that the string corresponding to the span captured by the variable x belongs to the regular language R, we would write $x\{R\}$. As we have seen in the previous chapter , this immediately makes RGX formulas more expressive than spanRGX formulas, since the latter cannot constrain the shape of the span captured by a variable. So, how can one specify that a span captured by a variable inside a spanRGX formula has a specific shape?

A natural approach to solve this issue allows joining spanRGX formulas using a rule-like syntax (Arenas et al., 2016). For instance, to specify that the content of the spans captured by the variable x belongs to a language in the intersection of two regular expressions R_1 and R_2, we can now write $x.R_1 \wedge x.R_2$. Of course, since regular languages are closed under intersection, it is also possible to express this property using RGX, however, the resulting expression will generally be less intuitive than the simple rule above. One can see that the two approaches to information extraction have slightly different design philosophies: on the one hand, we have the concise syntax of RGX, and on the other hand we have a more visual

approach which combines spanRGX formulas into rules. In this chapter we explore how these two approaches compare in terms of expressive power.

To do this, we first need to define a rule-based language for extracting information from documents that is based on spanRGX formulas. In our language, we will allow two types of formulas: R and $x.R$, where R is a spanRGX formula and x a variable. The former is meant to be evaluated over the entire document, while the latter applies to the span captured by the variable x. The semantics of the extraction formula R over a document d is defined as before, and for $x.R$ as follows:

$$[\![x.R]\!]_d \;=\; \{\mu \mid \exists s.\, (s, \mu) \in [x\{R\}]_d\}.$$

We can now define rules for extracting information from a document as conjunctions of extraction formulas. Formally, an *extraction rule* is an expression of the form:

$$\varphi = \varphi_0 \wedge x_1.\varphi_1 \wedge \cdots \wedge x_m.\varphi_m \tag{$\dagger$}$$

where $m \geq 0$, all φ_i are spanRGX formulas, and x_i are variables[1]. Extraction rules typically have an implication symbol and a *head predicate*, which we will omit because it does not affect the analysis performed in this thesis.

While Arenas et al. (2016) has a simple definition of the semantics of extraction rules, lifting this definition to the domain of (partial) mappings requires us to account for the nondeterminism of our expressions. What we mean by this is perhaps best captured by a rule which is of the form $(x \vee y) \wedge x.(ab^*) \wedge y.(ba^*)$, where we first choose which variable is going to be mapped to the entire document, and then we need to satisfy its respective constraint. For instance, if x is matched to the document, we want it to conform to the regular expression ab^*; however, in this case we do not really care about the content of y, so we should leave our mapping undefined on this variable.

Formally, we will define when a rule of the form ($\dagger$) is satisfied by a tuple of mappings $\overline{\mu} = (\mu_0, \mu_1, \ldots, \mu_m)$. To avoid the problem mentioned above, we need the concept of

[1]For simplicity we assume that there is only one formula applying to the entire document; namely φ_0. It is straightforward to extend the definitions below to include multiple formulas of this form.

instantiated variables in our tuple of mappings. For a rule $\varphi = \varphi_0 \wedge x_1.\varphi_1 \wedge \cdots \wedge x_m.\varphi_m$ and a tuple of mappings $\overline{\mu} = (\mu_0, \mu_1, \ldots, \mu_m)$ we define the set of instantiated variables, denoted by $\text{ivar}(\varphi, \overline{\mu})$ as the *minimal set* such that:

- $dom(\mu_0) \subseteq \text{ivar}(\varphi, \overline{\mu})$; and
- If $x_i \in \text{ivar}(\varphi, \overline{\mu})$, then $dom(\mu_i) \subseteq \text{ivar}(\varphi, \overline{\mu})$.

Intuitively, we want to capture only the variables which are used in nondeterministic choices made by the rule. For instance, in the case of the rule $(x \vee y) \wedge x.(\mathtt{ab}^*) \wedge y.(\mathtt{ba}^*)$, if we decide that x should be matched to our document, then we will not assign a value to the variable y and vice versa. We now define that a tuple of mappings $\overline{\mu} = (\mu_0, \mu_1, \ldots, \mu_m)$ satisfies φ over a document d, denoted by $\overline{\mu} \models_d \varphi$, if the following holds:

- $\mu_0 \in [\![\varphi_0]\!]_d$;
- If $x_i \in \text{ivar}(\varphi, \overline{\mu})$ then $\mu_i \in [\![x_i.\varphi_i]\!]_d$;
- If $x_i \notin \text{ivar}(\varphi, \overline{\mu})$ then $\mu_i = \mu_\emptyset$; and
- $\mu_i \sim \mu_j$ for all i, j.

Here the last condition will allow us to "join" all the mappings capturing each subformula φ_i into one. The problem with nondeterminism is handled by the two rules in the middle, since we force all instantiated variables to take a value, and the non-instantiated ones to be undefined. Finally, the definition starts from our main subformula (φ_0) which refers to the entire document and serves as a sort of a root for our mappings.

We can now define the semantics of an extraction rule φ over a document d as follows:

$$[\![\varphi]\!]_d \;=\; \{\mu \mid \exists\, \overline{\mu} \text{ such that } \overline{\mu} \models_d \varphi \text{ and } \mu = \bigcup_i \mu_i\},$$

where $\bigcup_i \mu_i$ signifies the mapping defined as the union of all μ_i.

Extraction rules allow us to define complex conditions about the spans we wish to extract. For instance, if we wanted to extract all spans whose content is a word belonging to (ordinary) regular expressions R_1 and R_2 at the same time, we could use the rule $\Sigma^* \cdot x \cdot \Sigma^* \wedge x.R_1 \wedge x.R_2$. More importantly, using extraction rules, we can now define valuations which cannot be defined using RGX, since they can define mappings which are not hierarchical. For instance, the rule $x \wedge x.\mathtt{a}y\mathtt{aa} \wedge x.\mathtt{aa}z\mathtt{a}$ is one such rule, since it makes

y and z overlap. In some sense, the ability of rules to use conjunctions of variables makes them more powerful than RGX formulas. On the other hand, the ability of RGX formulas to use disjunction of variables poses the same type of problems for spanRGX. Here one separating example is the RGX $\Sigma^* \cdot (x\{y\{\Sigma^*\} \cdot \mathsf{a}\} \vee y\{x\{\Sigma^*\} \cdot \mathsf{a}\}) \cdot \Sigma^*$. Therefore, we have the following:

Proposition 4.1. *Extraction rules and* RGX *formulas are incomparable in terms of the expressive power.*

Proof. First we will show that there is an extraction rule that is not expressible by any RGX. As shown by Fagin et al. (2015), funcRGX are hierarchical. It is clear that this result also extends to non-functional RGX. With this mind, one can realize that the extraction rule $x \wedge x.\Sigma^* \cdot y \cdot \Sigma^* \wedge x.\Sigma^* \cdot z \cdot \Sigma^*$ is not hierarchical, since y and z might be assigned spans that overlap in a non-hierarchical way. It is, therefore, not expressible by RGX.

Now we prove that there is a variable regex that is not expressible by any extraction rule. Consider the following variable regex: $\gamma = (\mathsf{a} \cdot x\{\mathsf{b}\}) \vee (\mathsf{b} \cdot x\{\mathsf{a}\})$. There are only two ways in which a document and mapping can satisfy it: (1) $d_1 = \mathsf{ab}$ and $\mu_1(x) = (2, 3)$; or (2) $d_2 = \mathsf{ba}$ and $\mu_2(x) = (2, 3)$. Suppose that there is an extraction rule φ that is satisfied only by these two document-mapping pairs. By the structure of extraction rules, we know that there is an extraction expression $x.\varphi_x$ such that φ_x is equivalent to the expression $\mathsf{a} \vee \mathsf{b}$; if not, we can construct a document d_3 that satisfies φ and is different from d_1 and d_2. By the same argument, we know that φ_0, the root extraction expression of φ, must be equivalent to $\mathsf{a}x \vee \mathsf{b}x$. Notice, however, that now the document $d_3 = \mathsf{aa}$ and the mapping μ_3 such that $\mu_3(x) = (2, 3)$, satisfy φ. We have reached a contradiction, and therefore conclude that such φ cannot exist. □

Of course, a natural question now is which fragments of the two languages are equivalent.

4.2. Unions of rules

In order to express all RGX formulas, we will show how unions of rules can be used to simulate the disjunction of variables. However, for this we also need to prune the class of rules we allow, since already a single rule can express properties beyond the reach of RGX. As we have seen, allowing conjunctions of the same variable is problematic for RGX. Another thing that makes rules different from RGX, is their ability to enforce cyclic behaviour through expressions of the form $x.y \wedge y.\mathsf{a}x$.

A natural way to circumvent both of these shortcomings is to force the rules to have a tree-like structure. In fact, this class of extraction rules was already considered by Arenas et al. (2016), as it allows faster evaluation than general rules. In order to define the class of *tree-like rules*, we need to explain how each rule can be viewed as a graph.

To each extraction rule $\varphi = \varphi_0 \wedge x_1.\varphi_1 \wedge \cdots \wedge x_m.\varphi_m$ we associate a graph G_φ defined as follows. The set of nodes of G_φ contains all the variables $x_1, \ldots, x_m$ plus one special node labelled doc corresponding to the formula φ_0. There exists an edge (x, y) between two variables in G_φ if and only if there is an *extraction formula* $x.R$ in φ such that y occurs in R. Furthermore, if the variable x occurs in the formula φ_0, we add an edge (doc, x) to G_φ. Then we say that φ is *tree-like* if (1) every variable x appears at most once on the left-hand side of an expression of the form $x.R$; and (2) G_φ is a tree rooted at doc. We can now show that the class of tree-like rules is indeed subsumed by RGX.

Lemma 4.1. *Every* tree-like *rule can be transformed into an equivalent* RGX.

Proof. We can transform tree-like extraction rules into RGX by recursively nesting extraction expressions into their associated variables. The procedure is as follows. Let $\varphi = \varphi_{x_0} \wedge x_1.\varphi_{x_1} \wedge \cdots \wedge x_m.\varphi_{x_m}$ be a tree-like extraction rule, and let G_φ be its graph. Without loss of generality, we assume that every variable $x \in \mathrm{var}(\varphi)$ appears on the left side of an extraction expression (if not, we can add $x.\Sigma^*$ expressions without altering the rule's semantics). For all $i \in [0, m]$ we define the RGX γ_{x_i} as φ_{x_i} where each mention of

each variable $y \in \text{var}(\varphi_{x_i})$ is replaced with $y\{\gamma_y\}$. As an example, consider the tree-like rule $\varphi = (\mathsf{a} \cdot x \cdot \mathsf{b} \cdot y) \wedge x.(\mathsf{abc} \cdot z) \wedge y.(\Sigma^*) \wedge z.(\mathsf{d})$. The resulting RGX in this case would be $\gamma = \mathsf{a} \cdot x\{\mathsf{abc} \cdot z\{\mathsf{d}\}\} \cdot \mathsf{b} \cdot y\{\Sigma^*\}$.

It can be proved with an straightforward induction that the expression γ_{x_0} will be a well-formed RGX and will be equivalent to θ. It is also clear that this procedure terminates since G_φ is a tree. Note, however, that the resulting RGX might be of exponential size with respect to the input extraction rule, since multiple appearances of the same variable can cause the expression to grow rapidly when the replacements are made. □

As we know, RGX alone are more expressive than any single rule, but by allowing unions of tree-like rules, we get a class of extraction rules capturing RGX formulas. Formally, a *union of tree-like rules* is a set of tree-like rules A. The semantics over a document d is defined as:

$$[\![A]\!]_d \;=\; \{\mu \mid \mu \text{ is a variable assignment over } d \text{ such that } \mu \in [\![\varphi]\!]_d, \text{ for some } \varphi \in A\}.$$

With this in hand, we can now better understand the connection between rules and RGX.

Theorem 4.1. *Unions of tree-like rules are equivalent to* RGX *formulas.*

The proof of this theorem relies on the fact that RGX formulas are equivalent to unions of functional RGX which do not use disjunctions of (expressions containing) variables. The latter can then be easily expressed using a tree-like rule, giving us one direction of the theorem. The other direction follows from Lemma 4.1 and the fact that RGX formulas are closed under disjunction. Analysing the proof also shows us that the transformation is exponential in both directions.

Proof. We begin by presenting vstk-graph, vstk-path, and vstk-path union, originally defined by Fagin et al. (2015) (the vset variants are defined analogously). A *vstk-graph* is a tuple $G = (Q, q_0, q_f, \delta)$ defined as a vstk-automaton, except that each transition in δ is of one of the following forms: $(q, \gamma, x\vdash, q')$, $(q, \gamma, \dashv q')$, and (q, γ, q_f), where $q, q' \in Q \setminus \{q_f\}$,

$x \in \mathcal{V}$, and γ is a regular expression over Σ. Configurations are defined in the same way as in the case of vstk-automata. A run ρ of G on a document d is a sequence of configurations $c_0, \ldots, c_m$ where for all $j \in [1, m-1]$ the configurations $c_j = (q_j, V_j, Y_j, i_j)$ and $c_{j+1} = (q_{j+1}, V_{j+1}, Y_{j+1}, i_{j+1})$ are such that $i_j \leq i_{j+1}$ and, depending on the transition used, one of the following holds:

(i) $(q_j, \gamma, x\vdash, q_{j+1}) \in \delta$, the substring $d(i_j, i_{j+1})$ is in $\mathcal{L}(\gamma)$, $x \in Y_j$, $V_{j+1} = V_j \cdot x$, and $Y_{j+1} = Y_j \setminus \{x\}$;

(ii) $(q_j, \gamma, \dashv q_{j+1}) \in \delta$, the substring $d(i_j, i_{j+1})$ is in $\mathcal{L}(\gamma)$, $Y_j = Y_{j+1}$, and $V_j = V_{j+1} \cdot x$; or

(iii) $(q_j, \gamma, q_{j+1}) \in \delta$ (this means $q_{j+1} = q_f$), $d(i_j, i_{j+1})$ is in $\mathcal{L}(\gamma)$, $Y_j = Y_{j+1}$, and $V_j = V_{j+1}$.

Accepting runs, var(G), and the semantics of vstk-graph, are defined the same way as in the case of vstk-automata.

A *vstk-path* P is a vstk-graph that consists of a single path. That is, P has exactly m states $q_1, \ldots, q_m = q_f$ and exactly m transitions such that there is a transition from q_1 to q_2, from q_2 to q_3, and so on. A *vstk-path union* is a vstk-graph that consists of a set of vstk-path such that: (1) each vstk-path is sequential, and (2) every pair of vstk-paths have the same initial state, the same final state, and share no other states.

We also define *path* RGX, which disallow the disjunction of variables. Formally, a path RGX is an expression that can be derived from the following grammar using E as the *start symbol*.

$$E ::= x\{E\},\ x \in \mathcal{V} \mid (E \cdot E) \mid R$$

$$R ::= w,\ w \in (\Sigma \cup \{\varepsilon\}) \mid (R \cdot R) \mid (R \vee R) \mid (R)^*$$

It is easy to see that path RGX are equivalent to vstk-path automata. With this in mind, we will show that every RGX can be transformed into an equivalent set of tree-like rules.

From the proof of Theorem 3.3, it can be deduced that RGX are equivalent to *path union vstk automata* (PU_{stk}), which are vstk automata that consist of a union of disjoint paths. It is apparent that each path in one of the PU_{stk} automata will be equivalent to a path

RGX. This implies that every RGX can be transformed into an equivalent union of path RGX through PU_{stk} automata (notice, however, that this union might be exponential in size with respect to the starting expression).

Given this, it only suffices to show that each path RGX is equivalent to a tree-like rule. Let γ be a path RGX. Given a variable regex α, we denote as $\text{flat}(\alpha)$ the spanRGX that results when replacing every top-level subexpression of the form $x\{\beta\}$ with x. It is easy to notice from the structure of path RGX that each variable in γ can appear at most once. Therefore, we can decompose γ into an extraction rule by using the following procedure: add the extraction expression $\text{flat}(\gamma)$ to the result and, for every subexpression of the form $x\{\gamma_x\}$ in γ, add the extraction expression $x.\,\text{flat}(\gamma_x)$ to the result. It is apparent that the resulting rule is tree-like, and it is straightforward to prove that it is equivalent to γ.

The proof that every set of tree-like rules can be transformed into an equivalent RGX follows from Lemma 4.1 and the fact that RGXs are closed under union (by using the disjunction operator). Notice that this direction will also produce expressions of exponential size. □

4.3. Eliminating cycles from rules

In the previous section we isolated a class of rules whose unions are equivalent to RGX formulas. To do this, we prohibited rules from using conjunctions of the same variable multiple times, and forced them to have a tree-like structure. And while prohibiting arbitrary conjunction was shown to be crucial, we have still not answered if the capability of rules to define cycles is really an obstacle, or if they can be removed. Consider for instance a rule of the form $x \wedge x.y \wedge y.x$, whose underlying graph is cyclic. This rule is clearly equivalent to $x \wedge x.y$, that requires no cycles in order to express the same property. In this chapter we consider the question whether cycles can be eliminated from rules and somewhat surprisingly show that, while generally not possible, in the case where all expressions defining a rule are functional spanRGX formulas this is indeed true.

Let us formally define the classes of rules we will consider in this chapter . First, as discussed above, the capability of an extraction rule to use conjunctions of the same variable multiple times already takes them outside of the reach of RGX. Therefore the most general class of rules we will consider disallows that type of behaviour. We call such rules *simple rules*. Formally, an extraction rule $\varphi = \varphi_0 \wedge x_1.\varphi_1 \wedge \cdots \wedge x_m.\varphi_m$ is *simple*, if all x_i are pairwise distinct. Note that simple rules can still be "cyclic" in nature, namely, the graph G_φ associated with the rule φ can have cycles. We call a rule φ *dag-like* if the graph G_φ contains no cycles. Furthermore, a rule φ is called *functional* with respect to the set of variables X, if all of the spanRGX formulas $\varphi_0, \ldots, \varphi_m$ are functional spanRGX formulas, all nodes in G_φ are reachable from the root, and $\bigcup_{i\in[0,m]} \varphi_i = X$.

To answer the question whether cycles can be eliminated from rules, let us consider the most general case: simple rules over full RGX. It is straightforward to see that in a rule of the form $(x \vee y) \wedge x.(y \vee \Sigma^*) \wedge y.(x \vee \Sigma^*)$, the cycle formed by x and y cannot be broken and the rule cannot be rewritten as a single dag-like rule. The main obstacle here is the fact that in each part of the rule we make a nondeterministic choice which can then affect the value of all the variables.

However, there is one important class of expressions, which would prohibit our rules to define properties such as the one above; that is, functional span regular expressions. The notion of functionality, first introduced by Fagin et al. (2015), was shown to be crucial for having a well-behaved fragment of RGX formulas, and it also has a profound effect on complexity of formula evaluation, since it brings it down from NP-hard to tractable (see chapter 6). As defined previously, functional formulas force that the variables used on the two sides of a disjunction are always the same, and furthermore, require concatenated expressions to share no variables (thus also forcing the star operator to be allowed only over variable-free formulas). Somewhat surprisingly, we can show that in the case of functional rules cycles can always be removed and, in fact, converting a simple rule that is functional into a dag-like rule takes only polynomial time.

THEOREM 4.2. *For every simple rule that is functional there is an equivalent (functional) dag-like rule. Moreover, we can obtain the equivalent rule in polynomial time.*

PROOF. Consider an arbitrary simple rule that is *functional*. We start by analyzing what sort of a value can a mapping assign to the variables which form a cycle. For this, take any rule φ and assume that there is a simple cycle $x_1, \ldots, x_n$ appearing in G_φ and a mapping μ satisfying φ. Then the following must hold:

(i) *All variables in the cycle must be assigned the same value.* This follows from the fact that in a simple rule each edge (x, y) in G_φ implies that $\mu(x)$ contains $\mu(y)$ (see Figure 4.1a).

(ii) *Every variable reachable from a cycle, but not inside it, must be assigned the empty content.* This follows from the observation above, plus the fact that edges (x, y) and (x, z) in G_φ imply that x and y appear in the same spanRGX. By the structure of spanRGX, if $x \neq y$ then $\mu(y)$ and $\mu(z)$ must be disjoint (see Figure 4.1b).

(iii) *If the cycle has a chord, then all the variables inside it must be assigned the empty content.* Here a chord means that we have a path from some x_i to some x_j inside G_φ which consists of nodes not belonging to our cycle, or there is a direct edge between them which is not part of the cycle. In the case there is an intermediate node, we know that it must be assigned ε, therefore x_j and all other nodes in the cycle must be ε as well. If the edge is direct, then by the definition of a chord, x_j is not a successor of x_i in the cycle, so just as in the previous case, the content of the successor of x_i and the content of x_j must be disjoint and equal, which is only possible if they are ε (see Figure 4.1c).

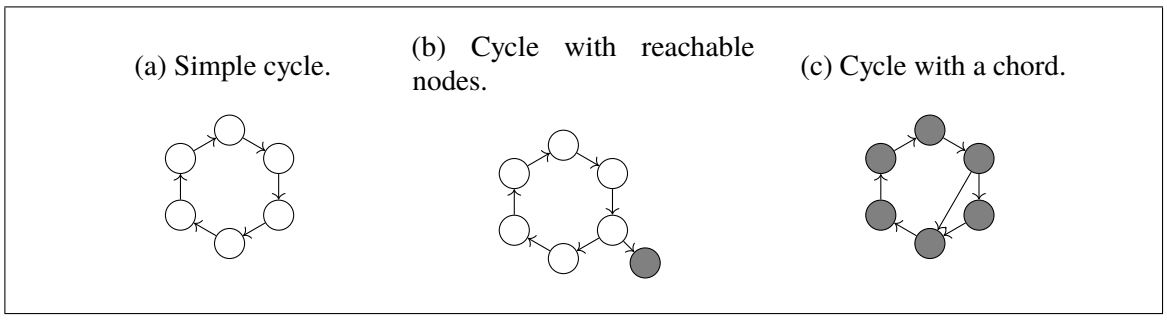


Figure 4.1. Different cycle arrangements in rule graphs. Shaded nodes correspond to variables with empty content.

The procedure for eliminating cycles from simple rules is based on the following colouring scheme for a graph G_φ associated with the rule φ. Let φ be an extraction rule with variables $x_1, \ldots x_n$. We will colour a node x_i *black* if:

- $x_i.\varphi_i$ appears in φ and φ_i is such that, when treating it as a regular expression, every word that can be derived from it must contain a symbol from Σ.

We then paint the graph by assigning the colour *red* to all black nodes, and all nodes which can reach a black node. All other nodes are coloured *green*. It is clear that this procedure can be carried out in polynomial time, since reachability takes only polynomial time. Note that in a black node coming from a conjunct of the form $x_i.\varphi_i$, the content of each variable appearing in φ_i must be *strictly* contained in the content of the variable x_i. This is because φ_i is functional and, since we painted its node black, it must have symbols from Σ which are not part of the content of the variables used in φ. Also note that each cycle has to be coloured using the same color.

If we now have a simple cycle $x_1, \ldots, x_n$ we can eliminate it by considering its color:

- *If the cycle is coloured red, then the rule is not satisfiable, so we can replace it by an arbitrary unsatisfiable dag-like rule.* We have two cases here. First, if a cycle contains a black node, then the content of its successor must be strictly contained inside its own content, which can not happen by the analysis above. Second, if a node x in the cycle can reach some black node not inside the cycle,

then its content must be different from ε, which contradicts point (2) of the above analysis.

- *If the cycle is green we can simplify it using an auxiliary variable.* Let $u_1, \ldots, u_m$ be the variables that are not part of the cycle and for which there is an edge (u_i, x_i). Let $y_1, \ldots, y_l$ be the variables that are not part of the cycle and are reachable from some x_i (they must have empty content, as proved before). We then add an auxiliary variable w and an edge from it to x_1. Each expression associated with some u_i is changed so that it uses w instead of x_i, and all expression associated with some y_i are changed to $y_i.\varepsilon$. Next, for $i < n$, an expression $x_i.\varphi_i$ is changed to $x_i.\varphi_i'$, where φ_i' maintains the possible orderings of variables in φ_i. This is done by removing all other letters or starred subexpressions, and is explained in detail later in this proof. For x_n, we replace the occurrences of x_1 by Σ^*. This yields an equivalent *simple* rule without the mentioned cycle.

As an example of how the rewriting above works, consider the rule $x.y \wedge y.z \wedge z.ux$. This rule can be rewritten to $w.x \wedge x.y \wedge y.z \wedge z.u \cdot \Sigma^* \wedge u.\varepsilon$.

Of course, here we explained only how a single cycle can be removed, but how do we transform a rule with multiple cycles in its graph? For this we start by identifying the strongly connected components of our graph G_φ. Each component can then be either: (a) a single node, (b) a simple cycle, or (c) a simple cycle with additional edges. In the latter two cases, if any component is coloured red, we know that the rule is unsatisfiable, so we can replace it by an arbitrary unsatisfiable dag-like rule. In the case they are coloured green, we can deploy the procedure above to remove the cycles, taking care that in the case (b) our variables can take an arbitrary, but always equal value, while in the case (c) they must be equal to the empty content. In both cases, all the variables reachable from the component are made ε.

Now we precisely describe the procedure for eliminating cycles in rules. Let $\varphi = \varphi_{x_0} \wedge x_1.\varphi_{x_1} \wedge \cdots \wedge x_m.\varphi_{x_m}$ be a simple rule such that each φ_{x_i} $(0 \leq i \leq m)$ is functional, and let G_φ be its graph. We will assume that for every variable $x \in \mathrm{var}(\varphi)$ there is an extraction expression $x.\varphi_x$ in φ. The resulting dag-like rule will be denoted as α.

First, we will color the nodes in G_φ. For this, we define a function ν : spanRGX $\rightarrow$ spanRGX that will indicate when a variable cannot have empty content. Here, $\emptyset$ has the usual definition in the regular expression context, with the following properties: $\emptyset \cdot \alpha = \emptyset$, $\emptyset \vee \alpha = \alpha$, $\emptyset^* = \varepsilon$, where α is any expression.

- $\nu(w) = \emptyset$, where $w \in \Sigma$.
- $\nu(x) = x$, where $x \in \mathcal{V}$.
- $\nu(\varphi_1 \cdot \varphi_2) = \nu(\varphi_1) \cdot \nu(\varphi_2)$.
- $\nu(\varphi_1 \vee \varphi_2) = \nu(\varphi_1) \vee \nu(\varphi_2)$.
- $\nu(\varphi^*) = \varepsilon$.

Thus, a node x_i is painted *black* if $\nu(\varphi_i) = \emptyset$. After this, nodes are painted *red* by "flooding" the graph doing a depth-first search starting at each black node and traversing edges in reverse.

Second, we run the *Strongly Connected Components Algorithm* (Tarjan, 1972). This algorithm computes the strongly connected components (SCCs) in the graph and outputs them in some topological order with respect to the dag formed by the SCCs of G_φ. We denote the ordered SCCs as $S_1, \ldots, S_l$, where each S_i $(1 \leq i \leq l)$ is a set of nodes.

Finally, we process the SCCs in order. Each SCC S_i will be of one of the following types: (1) S_i is a single node; (2) S_i is a simple cycle; or (3) S_i contains a cycle and has additional edges (that is, anything that does not fall under types (1) or (2)). According to the type, the following is done:

- Type (1): let $S_i = \{y\}$. We copy the extraction expression $y.\varphi_y$ to α.
- Type (2): let $S_i = \{y_1, \ldots, y_k\}$, such that (y_k, y_1) and (y_j, y_{j+1}) are edges in G_φ, for $j \in [1, k-1]$. If S_i has a red node, then the rule is unsatisfiable and we may stop and replace α with any unsatisfiable dag-like rule. Otherwise, we add a new auxiliary variable u_i and replace every appearance of variables of S_i in α with u_i. Add the following extraction expressions to α:
  - $u_i.y_1$;
  - $y_j.\nu(\varphi_{y_j})$, for $j \in [1, k-1]$;
  - and $y_k.\psi$, where ψ is $\nu(\varphi_{y_k})$ with all appearances of y_1 replaced with (Σ^*).

After this, mark every SCC reachable from S_i as a type (3) SCC.

- Type (3): let $S_i = \{y_1, \ldots, y_k\}$. If S_i has a red node, then the rule is unsatisfiable and we may stop and replace α with any unsatisfiable dag-like rule. Add an auxiliary variable u_i and add the following rules to α:
  - $u_i.y_1 \cdots y_k$;
  - $y_j.\psi$, for $j \in [1, k]$ where ψ is $\nu(\varphi_{y_j})$ with all appearances of variables $y_1, \ldots, y_k$ replaced with ε.

  After this, mark every SCC reachable from S_i as a type (3) SCC.

The resulting rule α will be dag-like and equivalent to φ. If we take into account the observations presented at the beginning of this proof, then it is straightforward to verify that the transformations outlined above will remove the cycles in G_φ while preserving equivalence. □

We now know that cycles can be eliminated from functional rules, but is there any way to simplify rules that are not functional? As it turns out, although functional and non-functional rules are not equivalent, every non-functional simple rule can in fact be expressed as a union of functional rules.

Proposition 4.2. *Every non-functional simple rule is equivalent to a union of functional simple rules.*

To see this, observe first that every non-functional spanRGX formula can be expressed as a (potentially exponentially big) union of functional spanRGX formulas (Theorem 4.1). Therefore, each non-functional rule can be rewritten into a rule whose formulas are unions of functional spanRGX. Then each such rule can be transformed into a union of functional rules by taking all the possible disjunctions (of which there will generally be exponentially many).

Proof. Let $\varphi = \varphi_0 \wedge x_1.\varphi_1 \wedge \cdots \wedge x_m.\varphi_m$ be a rule such that each φ_i is a spanRGX, where $i \in [0, m]$. We can transform each φ_i into an equivalent disjunction $\psi_{i,1} \vee \cdots \vee \psi_{i,l_i}$ where each $\psi_{i,j}$ is a functional spanRGX (by using the $\mathrm{PU}_{\mathrm{stk}}$ construction from Theorem 3.3).

Specifically, we transform φ_i into a VA_{stk} A and then into a PU_{stk} A'. It is clear that each path in A' can be directly transformed into a functional spanRGX (since paths do not have disjunctions of variables). Therefore, each $\psi_{i,j}$ will be a functional spanRGX. Notice, however, that this transformation might produce exponentially many $\psi_{i,j}$ with respect to the size of φ_i.

As an example of this step, the spanRGX $(x \vee y) \cdot (z \vee w)$ is equivalent to the disjunction $(x \cdot z \vee x \cdot w \vee y \cdot z \vee y \cdot w)$. Note that each of the disjuncts is independently functional.

Rule φ will be equivalent to the set of rules that consist of all possible conjunctions that can be made by taking one disjunct $\psi_{i,j}$ from every extraction expression ($i \in [0, m]$). Formally, φ will be equivalent to $\{\psi_{0,k_0} \wedge x_1.\psi_{1,k_1} \cdots \wedge x_m.\psi_{m,k_m} \mid (k_0, \ldots, k_m) \in [1, l_0] \times \cdots \times [1, l_m]\}$. Note that this will produce another exponential blow-up in size. The resulting set will therefore be double-exponential in size with respect to φ.

For example, consider the rule $\varphi = (x \vee y) \wedge x.(\mathsf{a} \vee \mathsf{b}) \wedge y.(\mathsf{c})$. Then, φ is equivalent to the following set of rules:

$$\{x \wedge x.\mathsf{a} \wedge y.\mathsf{c},\ x \wedge x.\mathsf{b} \wedge y.\mathsf{c},\ y \wedge x.\mathsf{a} \wedge y.\mathsf{c},\ y \wedge x.\mathsf{b} \wedge y.\mathsf{c}\}$$

Finally, we briefly explain why the algorithm is correct. The correctness of the transformation from spanRGX to PU_{stk} carries from the original proof without modification. Given the definition of the semantics for rules, it is fairly easy to observe that taking every possible combination of the disjuncts in each extraction expression will produce an equivalent set of rules. □

4.4. Regex vs. rules

One final thing we would like to determine is the connection of the more general classes of simple and dag-like rules to RGX. We will start by considering functional rules and comparing them to functional RGX formulas. As shown in Theorem 4.1, unions of tree-like rules are equivalent to RGX. It is straightforward to see that this is still true when we

consider functional rules and RGX. Therefore to connect functional rules which are not necessarily tree-like to RGX, we will show how and when can they be viewed as tree-like.

First, observe that a functional RGX formula is always satisfiable; namely, there is always a document on which there is an assignment satisfying this formula. Similarly, every functional tree-like rule is also satisfiable. On the other hand, the functional simple rule $x \wedge x.y \wedge y.\mathsf{a}x$ is clearly not satisfiable, since it forces x and y to be equal and different at the same time. Therefore, to link functional rules with RGX, we need to consider only the satisfiable ones. As we know that each functional simple rule has an equivalent dag-like rule, we start there.

THEOREM 4.3. *Every functional dag-like rule that is satisfiable is equivalent to a union of functional tree-like rules.*

The idea of the proof here is similar to the cycle elimination procedure of Theorem 4.2, but this time considering undirected cycles. One can show that eliminating undirected cycles results in a double exponential number of tree-like rules. In case that the rule was not satisfiable, our algorithm will simply abort.

PROOF. Let $\varphi = \varphi_{x_0} \wedge x_1.\varphi_{x_1} \wedge \cdots \wedge x_n.\varphi_{x_n}$ be a satisfiable dag-like rule such that each φ_{x_i} is a functional spanRGX ($i \in [0, n]$), and let G_φ be its graph. Without loss of generality, we assume that for every variable $x \in \mathrm{var}(\varphi)$ there is an expression $x.\varphi_x$.

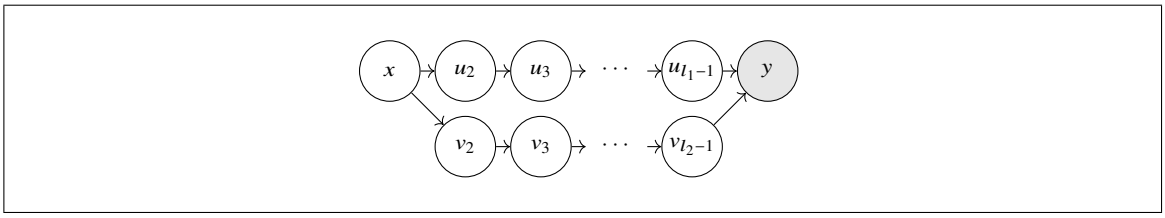


Figure 4.2. Undirected cycle in the graph of a dag-like rule.

First, let us consider how an undirected cycle in a rule can be satisfied. Let x and y be nodes such that there are at least two distinct paths $u_1, \ldots, u_{l_1}$ and $v_1, \ldots, v_{l_2}$ where

$u_1 = v_1 = x$ and $u_{l_1} = v_{l_2} = y$ (see Figure 4.2). Let d and μ be a document and a mapping that satisfy φ. Since all expressions are functional, we know the following: $\mu(x)$ contains $\mu(u_2)$ and $\mu(v_2)$; $\mu(u_2)$ and $\mu(v_2)$ contain $\mu(y)$; $\mu(u_2)$ and $\mu(v_2)$ are disjoint. From these facts we can deduce that $\mu(y)$ must be an empty span. Therefore, if the rule is satisfiable, y must be painted *green* by the coloring procedure. Furthermore, every variable reachable from y must be assigned the empty content, which means that φ may be rewritten as in the proof of Theorem 4.2 to simplify G_φ for all the nodes reachable from y.

The procedure is as follows. Given φ, we first paint all nodes using the same procedure from Theorem 4.2. After this, we transform every spanRGX φ_{x_i} into a disjunction of spanRGX $\varphi_{x_i,1}, \ldots, \varphi_{x_i,m_i}$ by the procedure from the proof of Theorem 4.1.

After this, we generate a new set of rules, where each individual rule consists of a possible combination of extraction expressions made by taking exactly one disjunct φ_{x_i,j_i} for each variable x_i. Formally, we generate the following set of rules:

$$R = \{\varphi_{0,k_0} \wedge x_1.\varphi_{1,k_1} \wedge \cdots \wedge x_n.\varphi_{n,k_n} \mid (k_0, \ldots, k_n) \in [1, m_0] \times \cdots \times [1, m_n]\}.$$

Given a rule $\alpha = \alpha_{x_0} \wedge x_1.\alpha_{x_1} \wedge \cdots \wedge x_n.\alpha_{x_n}$ in R, we can now easily transform it into a tree-like rule. Consider, as before, any pair of nodes x and y such that there are exactly two distinct paths $u_1, \ldots, u_{l_1}$ and $v_1, \ldots, v_{l_2}$ where $u_1 = v_1 = x$ and $u_{l_1} = v_{l_2} = y$ (the proof can be generalized to more paths). Consider, without loss of generality, that u_2 appears to the left of v_2 in φ_x. Then, for α to be satisfiable, everything between u_2 and v_2 in φ must be forced to have empty content. Likewise, everything to the right of u_3 in φ_{u_2} and everything to the left of v_3 in φ_{v_2} must be forced to have empty content. In general, everything to the right of variable u_{i+1} in φ_{u_i} must be forced to have empty content ($1 \leq i < l_1$), and everything to the left of v_{j+1} in φ_{v_j} must be forced to have empty content ($1 \leq j < l_2$). This can be done in polynomial time because it is equivalent to checking if a regular expression accepts the word ε and checking if certain variables were painted green. As we do this, we rewrite the spanRGX, removing everything but the variables from the parts that are forced to have empty content. If at any point we find an expression that cannot be made empty,

we remove α from R. Finally, we remove every occurrence of variable y in $\varphi_{v_{l_2-1}}$, thus removing the edge from v_{l_2-1} to y in G_α and eliminating the undirected cycle.

As an example, consider the following dag-like rule:

$$(x \cdot \Sigma^* \cdot y) \wedge x.(\mathsf{a} \cdot z \cdot \mathsf{b}^*) \wedge y.(\mathsf{b}^* \cdot z \cdot \mathsf{a}) \wedge z.(\Sigma^*)$$

This rule is satisfiable only by the document $d = aa$ and the mapping μ such that $\mu(x) = (1, 2)$, $\mu(y) = (2, 3)$, and $\mu(z) = (2, 2)$. By applying the procedure we described, we obtain the following rule:

$$(x \cdot y) \wedge x.(\mathsf{a} \cdot z) \wedge y.(\mathsf{d}) \wedge z.(\varepsilon)$$

It is simple to observe that this rule is equivalent and tree-like.

Notice that the final expression will be of double-exponential size with respect to the initial dag-like rule: it will experience one exponential blow-up when the spanRGX are transformed into disjunctions of path spanRGX, and another exponential blow-up when we generate a rule for each possible combination of disjuncts.

Given the definitions of the semantics of extraction rules and spanRGX, it can be proven without difficulty that the final set of tree-like rules will be equivalent to the initial dag-like rule. □

With this at hand, we can now also describe the relationship between functional simple rules and RGX.

Theorem 4.4. *The class of* funcRGX *formulas is equivalent to unions of satisfiable simple rules that are functional with respect to the same set of variables.*

Proof. As shown by Theorem 4.2, simple rules are equivalent to dag-like rules in the functional case. Given a functional dag-like rule $\varphi = \varphi_{x_0} \wedge x_1.\varphi_{x_1} \wedge \cdots \wedge x_m.\varphi_{x_m}$, one can obtain an equivalent funcRGX by taking φ_{x_0} and repeatedly replacing each variable x with the corresponding expression $x\{\varphi_x\}$. Since φ is dag-like, this procedure is guaranteed to terminate. It is not difficult to see that the resulting RGX will be functional, since all the

spanRGX in φ are functional. Finally, we take the disjunction of the RGX formulas that resulted from each of the dag-like rules in the union. Since they are all functional with respect to the same set of variables, the final RGX will also be functional.

In the opposite direction, we can use Theorem 4.1, that tells us that RGX formulas can be transformed into unions of tree-like rules. In this case, it is straightforward to prove that if the starting RGX is functional, then the resulting union of tree-like rules will also be functional. □

By using some of the theorems proven in this chapter , we can get a more general results that considers all of RGX.

THEOREM 4.5. *The class of* RGX *formulas is equivalent to unions of simple rules.*

PROOF. Unsatisfiable RGX are trivially equivalent to unsatisfiable simple rules. Therefore, for the rest of this proof, we will only consider satisfiable expressions.

We know, from Proposition 4.2, that simple rules are equivalent to unions of functional simple rules. Also, Theorem 4.2 implies that unions of functional simple rules are equivalent to unions of functional dag-like rules. From Theorem 4.3 we can deduce that unions of functional dag-like rules are equivalent to unions of functional tree-like rules. Finally, Theorem 4.1 indicates that unions of functional tree-like rules are equivalent to RGX, completing the proof. □

5. EVALUATION OF IE LANGUAGES

In this chapter , we study the computational complexity of evaluating an extraction expression γ over a document d, namely, the complexity of enumerating all mappings $\mu \in [\![\gamma]\!]_d$. Given that we are dealing with an enumeration problem, our objective is to obtain an *incremental polynomial time* algorithm (Johnson et al., 1988), i.e., an algorithm that enumerates all the mappings in $[\![\gamma]\!]_d$ by taking time polynomial in the size of γ and d between outputting two consecutive results. For this analysis, we model our problem as a decision problem and relate it to the enumeration problem. Formally, let $\perp$ be a new symbol. An *extended mapping* μ over d is a partial function from $\mathcal{V}$ to $\text{span}(d) \cup \{\perp\}$. Intuitively, in our decision problem $\mu(x) = \perp$ will represent that the variable x will not be mapped to any span. Furthermore, we usually treat μ as a normal mapping by assuming that x is not in $\text{dom}(\mu)$ for all variables x that are mapped to $\perp$. Given two extended mappings μ and μ', we say that $\mu \subseteq \mu'$ if, and only if, $\mu(x) = \mu'(x)$ for every $x \in \text{dom}(\mu)$. Also, given a mapping μ, we will denote as $\mu_\perp$ the extended mapping such that, for all $x \in \text{var}(\gamma)$, it holds that $\mu_\perp(x) = \mu(x)$ if $x \in \text{dom}(\mu)$ and $\perp$ otherwise. Now, for any language $\mathcal{L}$ for information extraction we define the main decision problem for evaluating expressions from $\mathcal{L}$:

Problem:	EVAL[$\mathcal{L}$]
Input:	An expression $\gamma \in \mathcal{L}$, a document d, and an extended mapping μ.
Question:	Does there exist μ' such that $\mu \subseteq \mu'$ and $\mu' \in [\![\gamma]\!]_d$?

In other words, in EVAL[$\mathcal{L}$] we want to check whether μ can be extended to a mapping μ' that satisfies γ in d. Note that in our analysis we will consider the combined complexity of EVAL[$\mathcal{L}$].

We claim that EVAL[$\mathcal{L}$] correctly models the problem of enumerating all mappings in $[\![\gamma]\!]_d$. Indeed, if we can find a polynomial time algorithm for deciding EVAL[$\mathcal{L}$], one

can have an incremental polynomial time algorithm for computing $[\![\gamma]\!]_d$ as presented in Algorithm 5.1.

```
Function Enumerate(γ, d, μ, V)
    if Eval[L](γ, d, μ) is false then return
    if V = ∅ then yield μ and return
    let x be some element from V
    for s ∈ span(d) ∪ {⊥} do
        Enumerate(γ, d, μ[x → s], V \ {x})
```

Figure 5.1. Algorithm for enumerating all spans in $[\![\gamma]\!]_d$.

The procedure starts with the empty mapping $\mu = \mu_\emptyset$ and the set V of variables yet to be assigned equal to var(γ). We will recursively build a mapping μ such that $\mu \in [\![\gamma]\!]_d$. First, we check if the current mapping μ can be extended to satisfy the input expression (line 5.1). If not, then the procedure returns; this ensures that we will only proceed if this will lead us to a satisfactory mapping. Next, if all variables have been assigned a span or the $\perp$ symbol (that is, $V = \emptyset$ in line 5.1) then we output μ, since the previous check guarantees that μ is an satisfactory mapping. Finally, the recursive step simply picks some unassigned variable (line 5.1) and iterates over all $s \in \text{span}(d)$ (or the symbol $\perp$) recursively calling the procedure with $\mu[x \to s]$ (which is a new extended mapping equivalent to μ, except that x is assigned to s) and the remaining unassigned variables ($V \setminus \{x\}$).

Now we will show that, as long as Eval[$\mathcal{L}$] can be decided efficiently, this procedure efficiently enumerates all spans in $[\![\gamma]\!]_d$.

Proposition 5.1. *If* Eval[$\mathcal{L}$] *is in* PTIME*, then enumerating all mappings in* $[\![\gamma]\!]_d$ *can be done in incremental polynomial time.*

Proof. It is easy to observe that $\mu \in [\![\gamma]\!]_d$ if and only if Eval[$\mathcal{L}$]($\gamma, d, \mu_\perp$). It is also easy to observe that for every mapping μ, it holds that $\mu_\emptyset \subseteq \mu$. From these two observations,

and since line 5.1 iterates over all possible spans, it is straightforward to prove by induction that the algorithm will eventually output all (and only) $\mu \in [\![\gamma]\!]_d$.

Now, we prove that this is an *incremental polynomial time* algorithm. As noted above, the algorithm will only recurse if there exists a mapping μ' such that $\mu' \in [\![\gamma]\!]_d$ and $\mu \subseteq \mu'$. Since the size of span(d) is in $O(|d|^2)$ and the algorithm can only recurse up to a depth of $|\mathcal{V}|$, the function Eval[$\mathcal{L}$] will be called $O(|\mathcal{V}||d|^2)$ times before an output is reached (or the algorithm terminates). Given that Eval[$\mathcal{L}$] can be decided in polynomial time, then time to produce the next output will be polynomial. □

Two decision problems associated with Eval[$\mathcal{L}$], that have a long standing history in data management (Abiteboul, Hull, & Vianu, 1995), are the *model checking problem* and the *non-emptiness problem*. Formally, the model checking problem (non-emptiness problem), denoted by ModelCheck[$\mathcal{L}$] (NonEmp[$\mathcal{L}$] resp.), asks whether $\mu \in [\![\gamma]\!]_d$ ($[\![\gamma]\!]_d \neq \emptyset$ resp.) for any $\gamma \in \mathcal{L}$, document d and mapping μ. One can easily see that both problems are actually restricted instances of Eval[$\mathcal{L}$], namely:

$$\begin{aligned} \text{ModelCheck}[\mathcal{L}](\gamma, d, \mu) &= \text{Eval}[\mathcal{L}](\gamma, d, \mu_\perp) \\ \text{NonEmp}[\mathcal{L}](\gamma, d) &= \text{Eval}[\mathcal{L}](\gamma, d, \mu_\emptyset) \end{aligned}$$

This implies that if we can find an efficient algorithm for Eval[$\mathcal{L}$] then the same holds for ModelCheck[$\mathcal{L}$] and NonEmp[$\mathcal{L}$]. In contrast, if we show that ModelCheck[$\mathcal{L}$] or NonEmp[$\mathcal{L}$] are NP-hard, then we will have that Eval[$\mathcal{L}$] is NP-hard and, consequently, the enumeration of $[\![\gamma]\!]_d$ is inefficient. We use these relations to simplify results that follow.

Now that we identified the appropriate decision problem, we start by understanding the complexity of Eval[$\mathcal{L}$] in the most general case. First of all, one can see that checking Eval[$\mathcal{L}$] is in NP for all languages and computational models considered in this thesis. Indeed, given a mapping μ' such that $\mu \subseteq \mu'$ one can check in PTIME if $\mu' \in [\![\gamma]\!]_d$ by using finite automata evaluation techniques (Hopcroft & Ullman, 1979). As the following result shows, this is the best that one can do if RGX or variable-set automata contain the language of spanRGX.

THEOREM 5.1. NONEMP[spanRGX] *is* NP*-comp.*

We would like to remark that this result was proved by Arenas et al. (2016) and here we just rephrased it in light of the unifying framework presented in this thesis.

PROOF. To prove that NONEMP[spanRGX] is NP-hard, we provide a reduction from 1-IN-3-SAT. The input of 1-IN-3-SAT is a propositional formula $\alpha = C_1 \wedge \cdots \wedge C_n$, where each C_i $(1 \leq i \leq n)$ is a disjunction of exactly three propositional variables (negative literals are not allowed). Then the problem is to verify whether there exists a satisfying assignment for α that makes exactly one variable per clause true. 1-IN-3-SAT is known to be NP-complete (Garey & Johnson, 1979).

For the reduction, we construct a spanRGX γ_α such that $[\![\gamma_\alpha]\!]_d$ is not empty if and only if there exists a satisfying assignment for α that makes exactly one variable per clause true, with $d = \varepsilon$. In this reduction, we assume that for every clause C_i in α $(1 \leq i \leq n)$, it holds that $C_i = (p_{i,1} \vee p_{i,2} \vee p_{i,3})$, where each $p_{i,j}$ is a propositional variable. Notice that distinct clauses can have propositional variables in common, which means that $p_{i,j}$ can be equal to $p_{k,\ell}$ for $i \neq k$.

To define γ_α we consider two sets of variables: $\{x_{i,j} \mid 1 \leq i \leq n \text{ and } 1 \leq j \leq 3\}$ and $\{y_{i,j,k,\ell} \mid 1 \leq i < k \leq n \text{ and } 1 \leq j, \ell \leq 3\}$. With these variables we encode the truth values assigned to the propositional variables in α; in particular, a span is assigned to the variable $x_{i,j}$ if and only if the propositional variable $p_{i,j}$ is assigned value true. Moreover, γ_α is used to indicate that exactly one of $p_{i,1}$, $p_{i,2}$ and $p_{i,3}$ is assigned value true, which is essentially represented by a spanRGX of the form $(x_{i,1} \vee x_{i,2} \vee x_{i,3})$, indicating that exactly one of $x_{i,1}$, $x_{i,2}$ and $x_{i,3}$ has to be assigned a span. Besides, γ_α is used to indicate that if $p_{i,j}$ is assigned value true, then we are forced to assign value false not only to $p_{i,k}$ with $k \neq j$ but also to some propositional variables in other clauses. This idea is formalized by means of the notion of conflict between propositional variables. More precisely, we say that $p_{i,j}$ is in conflict with $p_{k,\ell}$ if $i < k$ and one of the following conditions holds:

- there exists $m \in \{1, 2, 3\}$ such that $p_{i,j} = p_{k,m}$ and $m \neq \ell$;
- there exists $m \in \{1, 2, 3\}$ such that $p_{i,m} = p_{k,\ell}$ and $m \neq j$.

Thus, if $p_{i,j}$ is assigned value true and $p_{i,j}$ is in conflict with $p_{k,\ell}$, then we know that $p_{k,\ell}$ has to be assigned value false. In γ_α, the variable $y_{i,j,k,\ell}$ is used to indicate the presence of such a conflict; in particular, a span is assigned to $y_{i,j,k,\ell}$ if and only if the propositional variable $p_{i,j}$ is in conflict with the propositional variable $p_{k,\ell}$. We collect all the conflicts of $p_{i,j}$ in the set conflict($p_{i,j}$):

$$\{y_{i,j,k,\ell} \mid p_{i,j} \text{ is in conflict with } p_{k,\ell}\} \cup \{y_{k,\ell,i,j} \mid p_{k,\ell} \text{ is in conflict with } p_{i,j}\}$$

The variable $y_{i,j,k,\ell}$ is used as follows in γ_α. If some spans have been assigned to $x_{i,j}$ and $y_{i,j,k,\ell}$, then no span is assigned to $x_{k,\ell}$, as the propositional variable $p_{i,j}$ has been assigned value true and $p_{i,j}$ is in conflict with the propositional variable $p_{k,\ell}$. To encode this restriction, define the spanRGX $\gamma_{i,j}$ as the concatenation of the variables in conflict($p_{i,j}$) in no particular order. For example, if

$$\text{conflict}(p_{3,1}) \quad = \quad \{y_{1,2,3,1}, y_{1,3,3,1}, y_{3,1,4,1}, y_{3,1,5,2}\},$$

then

$$\gamma_{3,1} \quad = \quad y_{1,2,3,1} \cdot y_{1,3,3,1} \cdot y_{3,1,4,1} \cdot y_{3,1,5,2}$$

Finally, for every clause C_i ($1 \leq i \leq n$) define spanRGX γ_i as:

$$(x_{i,1} \cdot \gamma_{i,1} \ \vee \ x_{i,2} \cdot \gamma_{i,2} \ \vee \ x_{i,3} \cdot \gamma_{i,3})$$

With this notation, we define spanRGX γ_α as follows:

$$\gamma_\alpha \quad = \quad \gamma_1 \cdots \gamma_n$$

At this point it is important to understand how the variables $y_{i,j,k,\ell}$ are used in the spanRGX γ_α. Assume that $p_{1,1} = p_{2,1}$, so that $p_{1,1}$ is in conflict with $p_{2,2}$. Then if we assigned value true to $p_{1,1}$, we have that $p_{2,1}$ is also assigned value true, so $p_{2,2}$ has to be assigned value false. This restriction is encoded by using the variable $y_{1,1,2,2}$. More precisely, $\gamma_\alpha = \gamma_1 \cdot \gamma_2 \cdots \gamma_n$,

where γ_1 is of the form:

$$(x_{1,1} \cdots y_{1,1,2,2} \cdots \ \vee \ x_{1,2} \cdot \gamma_{1,2} \ \vee \ x_{1,3} \cdot \gamma_{1,3}),$$

given that $y_{1,1,2,2} \in \text{conflict}(p_{1,1})$, and γ_2 is of the form:

$$(x_{2,1} \cdot \gamma_{2,1} \ \vee \ x_{2,2} \cdots y_{1,1,2,2} \cdots \ \vee \ x_{2,3} \cdot \gamma_{2,3}),$$

given that $y_{1,1,2,2} \in \text{conflict}(p_{2,2})$. Thus, if $x_{1,1}$ is assigned a span, representing the assignment of value true to the propositional variable $p_{1,1}$, then also $y_{1,1,2,2}$ is assigned a span (both spans will have empty content by the definition of γ_α and d). If we now try to assign a span to $x_{2,2}$, then we are forced to assign a span to $y_{1,1,2,2}$ again. This, however, violates the definition of the semantics of RGX, because the mappings for concatenated expressions must have disjoint domains (in other words, they cannot both assign the same variable).

Based on the previous intuition, it is straightforward to prove that $[\![\gamma_\alpha]\!]_d$ is not empty if and only if there exists a satisfying assignment for α that makes exactly one variable per clause true, which was to be shown. As before, we take d to be ε. □

The previous result implies that the evaluation problem for RGX and variable-set automata is NP-complete. However, it does not say anything about the model checking problem. By adapting the proof of the previous result it can be shown that, unfortunately, the model checking problem is also hard for RGX and variable automata.

Theorem 5.2. ModelCheck[spanRGX] *is* NP*-complete.*

Proof. To prove this result, we will modify the proof of Theorem 5.1. In this case, given a propositional formula α we define the spanRGX γ'_α as follows:

$$\gamma'_\alpha = \gamma_\alpha \cdot \psi_\alpha, \text{ where } \psi_\alpha = \Big(\bigvee_{x \in \text{var}(\gamma_\alpha)} x \Big)^*$$

In addition to this, we set d to an empty document and μ to the mapping that assigns the span $(1, 1)$ to every variable in $\text{var}(\gamma_\alpha)$. The newly added part of the spanRGX will allow us to match μ by assigning all the variables that were not assigned in the first part of the formula. This, however, does not make the problem easier because we have to make sure that there exists some mapping μ' that satisfies the first part of the formula. Then, there is a satisfying assignment for α that makes exactly one variable per clause true if and only if $\mu \in [\![\gamma'_\alpha]\!]_d$.

Now we show why this construction is correct. Consider the definition of the semantics of RGX. In this case, since $d = \varepsilon$ all the spans will be $(1, 1)$ so we will omit them. The definition of concatenation tells us, then, that for μ to satisfy γ'_α there must exist mappings μ_1 and μ_2 such that μ_1 satisfies γ_α, μ_2 satisfies ψ_α, $\text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset$, and $\mu = \mu_1 \cup \mu_2$. Since ψ_α is satisfied by any mapping that assigns some subset of $\text{var}(\gamma_\alpha)$, deciding the model checking problem in this case is equivalent to determining if there exists a mapping that satisfies γ_α. This is, in other words, the non-emptiness problem for γ_α, which we showed it was hard. □

With the previous negative results, we examine several syntactical restrictions of RGX and variable-set automata that make the evaluation problem tractable. Note that the previous negative results are considering a more general setting than the one presented by Fagin et al. (2015), where RGX and variable-set automata are restricted to be *functional* which forces them to only generate relations of spans. Interestingly, the functional restriction decreases the complexity of the non-emptiness and evaluation problem for RGX as the following result shows.

Proposition 5.2. Eval[funcRGX] *is in* PTIME.

We will omit the proof, since this proposition is implied by Theorem 5.3 introduced later in this section.

This result proves that the functional restriction for RGX introduced by Fagin et al. (2015) is crucial for getting tractability. The question that remains is what the necessary

restrictions are that make the evaluation of RGX tractable when outputting mappings and how to extend these restrictions to other classes like variable-set automata. One possible approach is to consider variable-set automata that produce only relations. Formally, we say that a variable-set automaton A is *relational* if for all documents d, the set $[\![A]\!]_d$ forms a relation. As the next result shows, this semantic restriction is not enough to force tractability on Eval[VA].

Proposition 5.3. ModelCheck *and* NonEmp *of relational variable-set automata are* NP*-complete.*

Proof. First, we prove that both problems belong to NP. In both cases, we only need to guess a run for the variable automaton (that conforms to the input document and mapping) and verify that it is accepting. The size of the runs that we need to consider is bounded by a polynomial in the size of the document and the automaton. This is because any sufficiently long valid run will have a sequence of consecutive ε-transitions that form a cycle, which means that it is equivalent to a shorter (polynomially bounded) run that does not contain that kind of cycles.

To prove the NP-hardness of the ModelCheck problem we will describe a reduction from the *Hamiltonian path problem*. This problem consists in deciding whether or not a directed graph has a path that visits every vertex exactly once, and it is known to be NP-hard (Garey & Johnson, 1979). Let $G = (V, E)$ be a graph and let $A = (Q, q_0, q_f, \delta)$ be the variable automaton that results from reducing G. We will construct A in such a way that G has a Hamiltonian path if and only if $\mu_\varepsilon \in [\![A]\!]_d$, where $d = \varepsilon$ and μ_ε is such that $\mu_\varepsilon(x) = (1, 1)$ for all $x \in \mathrm{var}(A)$.

The automaton A is built as follows: (1) for every vertex $v \in V$, add states $p_{v,1}, p_{v,2}, \ldots, p_{v,|V|}$ to Q; (2) for every edge (u, v) and every $i \in [1, |V| - 1]$ add the transitions $(p_{u,i}, \dashv x_v, p_{v,i+1})$, $(q_0, \dashv x_v, p_{v,1})$ to δ; (3) add two fresh states for q_0, q_f and, for every $v \in V$ add transitions $(p_{v,|V|}, \varepsilon, q_f)$, $(q_0, x_v \vdash, q_0)$ to δ. Figure 5.2 shows an example of this reduction.

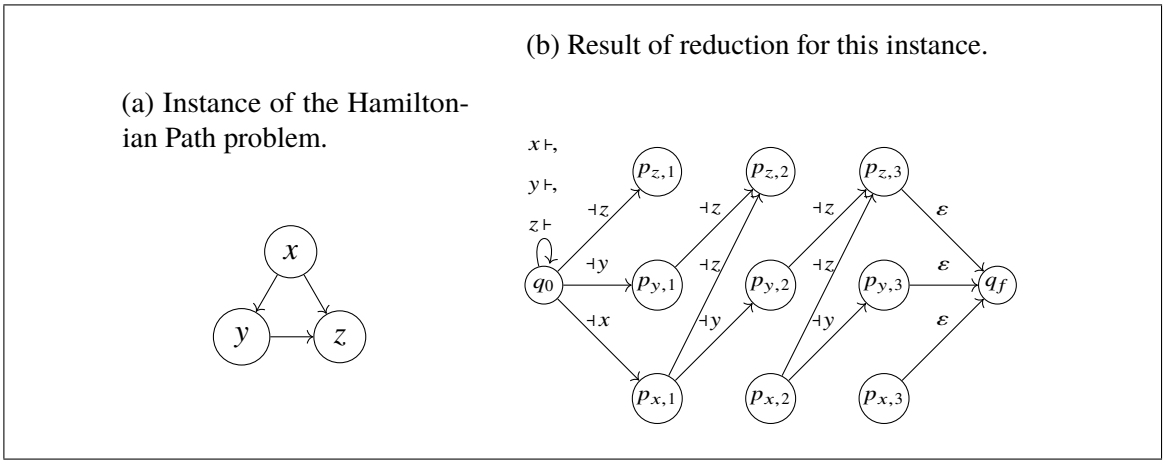


Figure 5.2. Example of reduction from Hamiltonian Path to ModelCheck of relational VA (Theorem 5.3).

Notice that every accepting run of A assigns every variable to the span $(1, 1)$, since to go from q_0 to q_f it must go through $|V|$ closing transitions (which must be different if the run is valid). Thus, A is relational. Because the states and transitions in A correspond to the vertices and edges in G there will be a one-to-one correspondence between runs in A and Hamiltonian paths in G. That is, if there is an accepting run that goes through states $p_{v_1,1}, \ldots, p_{v_{|V|},|V|}$, then there is a Hamiltonian path through the vertices of G. Proving this last statement is straightforward given the way A was built.

To see why the NonEmp is also NP-hard, notice that in the aforementioned construction when graph G does not have a Hamiltonian path there will be no accepting runs. Therefore, it holds that $[\![A]\!]_d$ is not empty if and only if G has a Hamiltonian path. □

By taking a close look at the proof of the previous result, one can note that a necessary property for getting intractability is that, during a run, the automaton can see the same variable on potential transitions many times but not use it if it has closed the same variable in the past. Intuitively, this cannot happen in functional RGX formulas where for every subformula of the form $\varphi_1 \cdot \varphi_2$ it holds that $\mathrm{var}(\varphi_1) \cap \mathrm{var}(\varphi_2) = \emptyset$. Actually, we claim that this is the restriction that implies tractability for evaluating RGX formulas. Formally, we

say that a RGX formula γ is *sequential* if for every subformula of the form $\varphi_1 \cdot \varphi_2$, φ^*, or $x\{\varphi\}$ it holds that $\text{var}(\varphi_1) \cap \text{var}(\varphi_2) = \emptyset$, $\text{var}(\varphi) = \emptyset$, and $x \notin \text{var}(\varphi)$, respectively.

We can also extend these ideas of sequentiality from RGX formulas to variable-set automata as follows. A path π of a variable-set automaton $A = (Q, q_0, q_f, \delta)$ is a finite sequence of transitions $\pi : (q_1, s_2, q_2), (q_2, s_3, q_3) \ldots, (q_{m-1}, s_m, q_m)$ such that $(q_i, s_{i+1}, q_{i+1}) \in \delta$ for all $i \in [1, m-1]$. We say that a path π of A is sequential if for every variable $x \in \mathcal{V}$ it holds that: (1) there is at most one $i \in [1, m]$ such that $s_i = x\vdash$; (2) there is at most one $j \in [1, m]$ such that $s_j = \dashv x$; and (3) if such a j exists, then i exists and $i < j$. We say that variable-set automaton A is sequential if every path in A is sequential. Finally, we denote the class of sequential RGX and sequential variable-set automata by seqRGX and seqVA, respectively.

The first natural question about sequentiality is whether this property can be checked efficiently. As the next proposition shows, this is indeed the case.

Proposition 5.4. *Deciding if a variable-set automaton is sequential can be done in* NLOGSPACE.

Proof. We describe an algorithm for checking if a variable automaton is sequential that is in coNLOGSPACE, which is known to be equal to NLOGSPACE (Immerman, 1988).

The algorithm non-deterministically traverses the automaton searching for a non-sequential path. To do so, it remembers the current variable's status, which can be either *available*, *open* or *closed*. If it finds a transition which is incompatible with the current status (e.g. opening an already open variable), it accepts, indicating that the variable automaton is not sequential. More formally, let $A = (Q, q_0, q_f, \delta)$ be a variable automaton, let q_{curr} denote the current state and let s_{curr} denote the current variable's status. For every variable $x \in \mathcal{V}$ the algorithm proceeds as follows:

- Set q_{curr} to q_0 and s_{curr} to *available*;
- while $q_{curr} \neq q_f$:
  - non-deterministically pick a transition $(q_{curr}, a, q_{next}) \in \delta$;

  - if a is incompatible with the status s_{curr}, then accept; otherwise, update q_{curr} to q_{next} and s_{curr} according to a;
- reject.

It is simple to realize that the algorithm is correct, since if there is a non-sequential path, then there is a sequence of non-deterministic decisions that will lead the algorithm to accept. On the other hand, it is also apparent that the algorithm uses only logarithmic space, because it only has to store the current variable, current state, next state and variable status; in other words, a constant amount of information that is at most logarithmic in size with respect to the input. □

Sequentiality is a mild restriction over extraction expressions since it still allows many RGX formulas that are useful in practice. For example, all extraction expressions discussed in Chapter 3 are sequential. Furthermore, as we now show, no expressive power is lost when restricting to sequential RGX or automata.

Proposition 5.5. *For every* RGX *(*VA *automaton), there exists a sequential* RGX *(sequential* VA*, respectively) that defines the same extraction function.*

Proof. By using the PU_{stk} construction (mentioned in Theorem 3.3), we know that we can transform any RGX (or VA) into an equivalent *path union vstk-automaton*. Let A be the automaton that results from such construction. It is easy to see from the way A is constructed that the individual path automata that compose A will always be sequential. Each of these path automata can be transformed into an equivalent RGX that will clearly be sequential as well. Therefore, A is equivalent to the disjunction of a finite set of sequential RGX, which is evidently also a sequential RGX. □

We believe that sequentiality is a natural syntactical restriction of how to use variables in extraction expressions: in simple terms users should not reuse variables by concatenation since this can easily make the formula unsatisfiable. Furthermore, the more important advantage for users is that RGX and variable-set automata that are sequential can be evaluated efficiently.

THEOREM 5.3. EVAL[seqRGX] *and* EVAL[seqVA] *is in* PTIME.

PROOF. We first show that EVAL[seqRGX] can be reduced to EVAL[seqVA] in polynomial time. By Theorem 3.3, we know that RGX can be efficiently transformed into VA_{stk}. Therefore, it only remains to show that the resulting variable automaton will be sequential if the starting RGX is sequential. This can be proven with a straightforward induction over the structure of RGX, showing that, for each operator, the construction algorithm preserves the sequentiality of the output automaton.

Now, we prove that the EVAL[seqRGX] problem is in PTIME. The main idea behind this proof, and many of the following, will be to embed in document d the variable operations corresponding to mapping μ. This will allow us to then treat variable operation transitions as normal transitions. This is an advantage because then we can use classical algorithms for finite automata to decide problems.

Let $\text{Op}(A) = \{x\vdash, \dashv x \mid x \in \mathcal{V}(A)\}$, that is, the set of all variable operations for variables in A. Let ρ be a run for document d and mapping μ on a variable automaton A. We refer to the *label* of ρ, denoted $L(\rho)$, as the string $\lambda \in (\Sigma \cup \text{Op}(A))$ that is the concatenation of the labels of the transitions in ρ, in the order they are used.

Given a label λ, we may easily generate the document-mapping pair (d, μ) from the run of λ in logarithmic-space. We simply scan λ from left to right, outputting symbols of Σ to d, then we do a second scan, counting symbols to determine the spans that compose μ. It is simple to see that if we change the order of consecutive variable operation in λ, then the generated (d, μ) will be the same.

As an example, consider the document $d = \mathsf{abc}$ and the mapping μ such that $\mu(x) = (1, 3)$ and $\mu(y) = (3, 3)$. Some labels that correspond to these are $\lambda_1 = x\vdash, \mathsf{a}, \mathsf{b}, y\vdash, \dashv x, \dashv y, \mathsf{c}$ or $\lambda_2 = x\vdash, \mathsf{a}, \mathsf{b}, \dashv x, y\vdash, \dashv y, \mathsf{c}$.

Similarly, for every pair (d, μ) and finite set of variables, there is a finite set of possible labels of runs that correspond to d and μ. By the previous paragraph, it is an easy observation that the labels in this set will differ only on the ordering of consecutive variable operations, and variables that are opened but never closed.

Since the ordering of variable operations will be a problem in most proofs, we will frequently use the technique of *coalescing* consecutive variable operations. What this means, is that we will consider a set of consecutive variable operations as a single symbol. We will usually accompany this by introducing new transitions to the automata that recognize these coalesced symbols.

Let $A = (Q, q_0, q_f, \delta)$ be the sequential automaton, d the document and μ the mapping. First, let λ be some label for (d, μ). Let $\tau = T_1, \ldots, T_\ell$ be a partition of $\mathrm{dom}(\mu)$ such that two variable operations o_1 and o_2 belong to the same T_i if and only if $o_1 \cdot w \cdot o_2$ is a substring of λ and w is ε or consists solely of variable operations. We treat the sets in τ as new symbols of the alphabet. We will coalesce all sequences of consecutive variable operations in λ replacing them with their respective T_i, and call the result d'.

Let $A' = (Q, q_0, q_f, \delta')$ be as follows. For each transition $(p, a, q) \in \delta$: (1) if $a \in \Sigma \cup \{\varepsilon\}$, then $(p, a, q) \in \delta'$; (2) if a is a variable operation for x and $x \notin \mathrm{dom}(\mu)$, then $(p, \varepsilon, q) \in \delta'$; otherwise, ignore the transition. Finally, for every set T_i ($i \in [1, \ell]$), transition $(p, T_i, q) \in \delta'$ if there exists a path from p to q in A satisfying the following conditions: (1) every transition in the path is either an ε-transition or corresponds to a variable operation in T_i; and (2) for every variable operation in T_i, there is exactly one transition in the path that corresponds to it. Notice that A' has no variable operations, and therefore, behaves exactly like a non-deterministic finite automaton. Therefore, the problem has been reduced to that of deciding whether the non-deterministic finite automaton A' accepts the word d', which is known to be in PTIME (Hopcroft & Ullman, 1979).

Except for the last step, it is clear that this reduction runs in polynomial time. Therefore, in order to complete this part of the proof, we only need to provide an algorithm that given states p, q and $i \in [1, n+1]$ decides whether $(p, T_i, q) \in \delta'$. We will describe an algorithm that finds a path in A that in NLOGSPACE, which is contained in PTIME (Papadimitriou, 1993). Taking into account that A is sequential, we know that the paths will not repeat operations nor execute them in a wrong order, therefore, we only need to count the number of variable operations.

The algorithm starts from state p and sets a counter c to 0. Then, at each step it guesses the next transition, and checks that it is either an ε-transition or corresponds to a variable transition in T_i. If it is the latter, then it increments c by one. If the algorithm reaches q, it accepts only if $c = |T_i|$. From the description of the algorithm it is straightforward to prove that it is correct and uses logarithmic-space.

Now we prove the correctness of the algorithm. Namely, we will prove that there exists an extension μ' of μ if and only if A' accepts d'. We will consider the three cases that can happen to a variable x with respect to μ: (1) $x \notin \text{dom}(\mu)$, (2) $\mu(x) = \perp$, and (3) $\mu(x) = (i, j)$ for $i, j \in [1, n + 1]$. In case (1), we have that x may or may not be in $\text{dom}(\mu')$. This agrees with the fact that variable operations for x are replaced with ε in A'. Furthermore, because A is sequential, we know that there are no valid runs in A' that would be invalid in A. In case (2), μ' cannot assign x, which agrees with A' because variable operations for x were removed. Finally, in case (3), we know that μ' will be compatible with μ on x because each of the T_i symbols we introduced can be matched by A' if and only if there exists a path in A that performs the variable operations in T_i in some order. Given these observations it is very apparent that there is a one-to-one correspondence between accepting runs in A and A', which finishes the proof of correctness. □

It is important to recall that this result implies, by Proposition 5.1, that the evaluation of sequential RGX formulas can be done in incremental polynomial time. Moreover, ModelCheck and NonEmp of this class can also be decided in PTIME. This provides us with a good upper-bound for evaluating formulas and shows that it might be possible to find an efficient algorithm that works in practice. With this, we refer to a *constant delay algorithm* (Johnson et al., 1988) like the one presented by Arenas et al. (2016) for the so-called navigational formulas—a class strictly subsumed by sequential RGX. Thus, sequential RGX seem like a good direction for extending the constant delay algorithm for navigational formulas given by Arenas et al. (2016).

Now that we have captured an efficient fragment of RGX, we will analyze what happens with the complexity of the evaluation problem for extraction rules. First, we show that

evaluating rules is in general a hard problem. In fact, non-emptiness is already NP-hard, even when restricted to functional RGX and dag-like rules.

THEOREM 5.4. NONEMP *of functional dag-like rules is* NP*-complete.*

PROOF. First, we show that the problem is in NP. Consider a functional dag-like rule φ, and a document d. To decide the problem we can guess a mapping μ, which is of polynomial size, and check that $\mu \in [\![\varphi]\!]_d$. This can be done in polynomial time as follows. From Theorem 5.3, we know that MODELCHECK of sequential (and thus functional) RGX is in PTIME. With this in mind, we can check that μ respects the semantics of rules detailed in Chapter 4 (with regards to instantiated variables, among other conditions) and for each relevant extraction expression $x.\varphi_x$, check that μ restricted to $Var(\varphi_x)$ satisfies φ_x, when d is restricted to $\mu(x)$ (which, as we mentioned, is in PTIME).

To show that the problem is NP-hard, we will describe a polynomial time reduction from the 1-IN-3-SAT problem. The input for 1-IN-3-SAT consists of a propositional formula $\alpha = C_1 \wedge \cdots \wedge C_n$ where each clause C_i $(1 \leq i \leq n)$ is a disjunction of three positive literals: $p_{i,1}$, $p_{i,2}$, and $p_{i,3}$. The problem is to determine if there is a truth assignment that makes *exactly* one literal true in each clause. This problem is known to be NP-complete (Garey & Johnson, 1979).

Given the propositional formula α, the reduction will output a functional dag-like rule φ, and a document d such that $[\![\varphi]\!]_d$ is non-empty if and only if α is satisfiable. Let V be the set of variables used in α. The expression φ will use the variables in V plus fresh variables c_i for $i \in [1, n]$, and two extra variables: T and F. Here, variables c_i will represent the clauses in α, and T, F will signal those variables that are *true* and *false*, respectively. The rule φ consists of the following extraction expressions:

- $T \cdot c_1 \cdot F$;
- $c_i.(p_{i,1} \cdot c_{i+1} \cdot p_{i,2} \cdot p_{i,3}) \vee (p_{i,2} \cdot c_{i+1} \cdot p_{i,1} \cdot p_{i,3}) \vee (p_{i,3} \cdot c_{i+1} \cdot p_{i,1} \cdot p_{i,2})$ for $i \in [1, n-1]$; and

- $c_n.(p_{i,1} \cdot T \cdot \# \cdot F \cdot p_{i,2} \cdot p_{i,3}) \vee (p_{i,2} \cdot T \cdot \# \cdot F \cdot p_{i,1} \cdot p_{i,3}) \vee (p_{i,3} \cdot T \cdot \# \cdot F \cdot p_{i,1} \cdot p_{i,2})$, where # is a symbol in the alphabet.

Note that every spanRGX is functional, and that the rule is dag-like.

The intuition behind the reduction is that every variable placed to the left of the # symbol should be assigned a true value, and every variable placed to the right of the symbol should be assigned a false value. Notice that φ can only be satisfied by the document $d = \#$ and a mapping μ such that $\mu(T) = (1, 1)$ and $\mu(F) = (2, 2)$. If μ satisfies φ, we can make the following observations: (1) for every $x \in V$, either $\mu(x) = (1, 1)$ or $\mu(x) = (2, 2)$; and (2) for every $i \in [1, n]$, there is exactly one $j \in \{1, 2, 3\}$ such that $\mu(p_{i,j}) = (1, 1)$. With these observations in mind, it is easy to see that every satisfying mapping of φ will correspond to a satisfying truth assignment of α and vice versa, thus proving the reduction correct. □

The difficulty in this case arises from the fact that dag-like rules allow referencing the same variable from different extraction expressions. A natural way to circumvent this is to use tree-like rules. Indeed, the fact that, in a tree-like rule, different branches are independent, causes the evaluation problem to become tractable. In fact, the functionality constraint is not really needed here, as the result holds even for sequential rules.

THEOREM 5.5. EVAL *of sequential tree-like rules is in* PTIME.

PROOF. In order to prove that EVAL of sequential tree-like rules is in PTIME, we will describe an algorithm that first does some polynomial-time preprocessing of the input, and then runs in *alternating logarithmic space* (ALOGSPACE), which is known to be equivalent to PTIME (Papadimitriou, 1993).

Let $\varphi = \varphi_{x_0} \wedge x_1.\varphi_{x_1} \wedge \cdots \wedge x_m.\varphi_{x_m}$ be a sequential tree-like rule with graph G_φ, let d be a document, and let μ be a mapping. We assume, without loss of generality, that for every variable x in φ there is an extraction expression $x.\varphi_x$ in φ.

We may immediately reject in two cases: (1) μ is not hierarchical; and (2) there are variables x and y such that $\mu(x) = \mu(y)$, the content of $\mu(x)$ is not empty, and there is

no directed path in G_φ that connects x and y. These two cases can easily be checked in polynomial-time, and will help us simplify the proceeding analysis.

For the purpose of this proof, we say that two variables x and y are *indistinguishable* if $\mu(x) = \mu(y) = (i, i)$ for some $i \in [1, n + 1]$ and they are siblings in G_φ; that is, there exists a variable z such that (z, x) and (z, y) are edges in G_φ. The problem with these variables is that we cannot deduce from μ and φ the order in which they must be encountered when processing the document. Therefore, we will coalesce each set of indistinguishable variables into a single variable. This means removing these variables from the global set of variables and replacing them with a single new variable that represents the set. We refer to these new variables as *coalesced variables*, and we refer to mapping μ updated to reflect this change as μ'.

By coalescing indistinguishable variable, however, we will be destroying the subtrees rooted at them. Therefore, we must check that μ agrees with these subtrees. Let U be a maximal set of pairwise indistinguishable variables. For each $x \in U$ we perform the following "emptiness" check. Transform φ_x into a variable automaton A_x and check that: (1) there is a path from the initial state to the final state of A_x that uses only ε-transitions and variable operations; (2) this path opens and closes every variable y such that (x, y) is in G_φ; (3) for every variable y used in this path, either $\mu(x) = \mu(y)$ or $y \notin \mathrm{dom}(\mu)$; and (4) recursively perform the "emptiness" check on y and φ_y. This may be done in polynomial time by using similar techniques to those shown on the proof of Theorem 5.3.

For this proof, we will use again the idea of *labels* (defined in the proof of Theorem 5.3). Notice that if we fix an order $\preceq_{\mathrm{Op}}$ over the variable operations and limit to those variables in $\mathrm{dom}(\mu)$, then there is a unique label for (d, μ) in which consecutive variable operations are ordered according to $\preceq_{\mathrm{Op}}$. We denote this label $L(d, \mu, \preceq_{\mathrm{Op}})$, and we may compute it easily in polynomial time.

In addition to the above, we say that a label λ is *balanced* if all of its opening and closing variable operations are correctly balanced (like parentheses). It is clear that given a valid (d, μ), μ is hierarchical if and only if (d, μ) have at least one balanced label.

Now, notice that if we take into account μ', G_φ and indistinguishable variables, then there is a unique order in which variable operations could be seen by the rule if the document is processed sequentially. We will use this order as the order $\preceq_{\mathrm{Op}}$, which we can compute as follows. Let $V = \{x \in \mathrm{dom}(\mu') \mid \mu(x) \neq \perp\}$ and consider the induced subgraph $T = G_\varphi[V]$. A node x in T precedes its sibling y if $\mu'(x) = (i, j)$, $\mu'(y) = (k, l)$, and $\min(i, j) < \max(k, l)$. Since we coalesced indistinguishable variables, we know that there is a unique way to put siblings in this order. Finally, the order can be obtained by doing an ordered depth-first search on T: when we enter a node x we add $x\vdash$ to the output, when we finish processing the subtree rooted at x we add $\dashv x$ to the output. With this in mind, we define the document $d' = L(d, \mu', \preceq_{\mathrm{Op}})$.

Next, we transform each sequential spanRGX φ_{x_i} into a non-deterministic finite automaton $A_{x_i} = (Q, q_0, q_f, \delta)$. For each coalesced variable X that represents the set of indistinguishable variables U, we add a new state q_X and transitions $(p, X\vdash, q_X)$ and $(q_X, \dashv X, q)$ if there is a path from p to q that uses only ε-transitions and variable transitions such that every variable in set U is opened and closed in this path. This can be done in polynomial-time because all expressions are sequential (the same way it was done on the proof of Theorem 5.3).

Now, we run the alternating logarithmic space algorithm. We will have two pointers: i_{curr} and i_{end}. They will denote the part of the document that we are considering at any given time, and will start as 1 and $|d'| + 1$ respectively. The algorithm works by traversing the automata guessing transitions. Every time we choose a transition in A_x that opens variable y, we find the position i_{close} in d' where y is closed (or guess it if $y \notin \mathrm{dom}(\mu')$) and check two conditions in parallel (by use of alternation): (1) A_y recursively accepts (d', μ') on the interval (i_{curr}, i_{close}); and (2) A_x accepts (d', μ') on the interval (i_{close}, i_{end}), continuing from the current state. More specifically, the algorithm is the following:

(i) Set i_{curr} to 1, i_{end} to $|d'| + 1$, and x_{curr} to x_0.

(ii) Let $A_{x_{curr}}$ be (Q, q_0, q_f, δ).

(iii) Set q_{curr} to q_0.

(iv) While $q_{curr} \neq q_f$ and $i_{curr} \leq i_{end}$:

(a) Non-deterministically pick a transition $(q_{curr}, a, q_{next}) \in \delta$.

(b) If $a = \varepsilon$, set q_{curr} to q_{next} and continue.

(c) Else if $a = x\vdash$ for some variable x (that is not coalesced), do as follows. If $x \in \text{dom}(\mu')$, then check that $a = a_{i_{curr}}$, then find the position i_{close} such that $a_{i_{close}} = \dashv x$. Else if $x \notin \text{dom}(\mu')$, guess $i_{close} \geq i_{curr}$ and set q_{next} to the state reached by following the $\dashv x$-transition from the current q_{next}. Do the following two things in parallel:

- Set i_{curr} to i_{close}, q_{curr} to q_{next}, and continue.
- Set i_{end} to i_{close}, x_{curr} to x, increment i_{curr}, and go to step 2.

(d) Else if a is $a = a_{i_{curr}}$, then set q_{curr} to q_{next} and increment i_{curr}.

(e) Otherwise, reject.

(v) If $i_{curr} = i_{end}$, accept.

Now we will sketch a proof of correctness. By the definition of the semantics of rules, it is clear that there is a correspondence between mappings and a set of runs for the automata that compose the rule. It is easy to see that the algorithm described above will find accepting runs for each of the automatons that correspond to variables instanced by the rule. These runs will correspond to a mapping ν which is an extension of μ' and that can be easily be transformed into an extension of μ by separating the coalesced variables. To see why the algorithm will accept whenever such a ν exists, consider the following. It can be proven without much difficulty that, given the nested structure of tree-like rules and the plainness of sequential spanRGX, the way in which we ordered the variable operations in d' is the only way in which they might be actually seen. The only case in which this is not true, is in the case of indistinguishable variables, which we handled as a separate case. Therefore, the algorithm will accept whenever there exists an extension to μ that satisfies φ. □

This implies that we should focus on sequential tree-like rules if we wish to have efficient algorithms for rules. Luckily, these do not come at a high price in terms of expressiveness, since Theorem 4.2, Proposition 4.2, and Theorem 4.3 imply that every satisfiable simple rule is equivalent to a union of sequential tree-like rules.

The previous results show how far we can go when syntactically restricting the class of RGX formulas, variable-set automata, or extraction rules in order to get tractability. The next step is to parametrize the size of the query not only in terms of the length, but also in terms of meaningful parameters that are usually small in practice. In this direction, a natural parameter is the number of variables of a formula or automata since one would expect that this number will not be huge. Indeed, if we restrict the number of variables of a RGX formula or variable-set automata we can show that the problem becomes tractable. We express this by using the notion of *fixed parameter tractability* and the complexity class FPT (Flum & Grohe, 2006).

Theorem 5.6. Eval[RGX] *and* Eval[VA] *parametrized by the number of variables is* FPT.

Proof. We know from Theorem 3.3 that RGX can be transformed into equivalent variable automata in polynomial time (and without altering the parameter). Therefore, the rest of the proof will focus only on VA.

Let A be a variable automaton, d a document, μ a mapping and k the number of variables in A, that is, $k = |\operatorname{var}(A)|$. We can decide this instance of the Eval[VA] problem using the same reduction from the proof of Theorem 5.3, but with two modifications.

First, we change the algorithm that decides if $(p, T_i, q) \in \delta'$, for some given states $p, q \in Q$ and $i \in [1, n+1]$. The original algorithm will not work in this case because A might not be sequential. Thus, now we iterate over all possible total orders over the set T_i (there are $|T_i|!$ such orders) and let $(t_1, \ldots, t_{|T_i|})$ be a sequence with the elements of T_i according to that order. We give $(t_1, \ldots, t_{|T_i|})$ as an additional input to the algorithm and proceed in a similar way than before, but we keep an additional counter e with the current position in the new sequence (we set e to 1 at the start). Whenever the algorithm chooses a transition with a variable operation, it compares it with t_e: if it is the same, it increments e; otherwise, it rejects. At the target state q we accept if and only if $e = |T_i| + 1$, which means

we saw all the variable operations of $|T_i|$ exactly once. Notice that this gives an algorithm that runs in time at most $k! \cdot p(n)$, where p is a polynomial.

Second, we slightly change the way we handle a variable x when $x \notin \text{dom}(\mu)$. Instead of replacing the variable operation transitions of x with ε-transitions, we preserve them as they are. In this part of the algorithm, we will iterate over all valid sequences of variable operations in $\{x\vdash, \dashv x \mid x \in (\text{var}(A) \setminus \text{dom}(\mu))\}$. We say that a sequence of variable operations is *valid* if, for every variable x: (1) the operations $x\vdash$ and $\dashv x$ appear at most once; (2) if $\dashv x$ is in the sequence, then $x\vdash$ is in the sequence at an earlier position. For example, $[x\vdash, y\vdash, \dashv x, \dashv y]$ and $[x\vdash, z\vdash, \dashv x, y\vdash]$ would be two valid sequences of operations for variables x, y, z. Given a sequence of operations, the modified automaton, and the modified document, the problem then reduces to checking if the final state of the variable automaton is reachable from its initial state, subject to the constraint that the chosen transitions must match the sequence of operations and the document.

Formally, the algorithm would be the following. Let $A' = (Q, q_0, q_f, \delta')$ be the modified variable automaton, and let $d' = a_1 a_2 \cdots a_n$ be the modified input document (the label). Throughout the algorithm we will keep: i_{doc}, the current position in the document; i_{seq}, the current position in the sequence of operations; and q_{curr}, the current state. For every valid sequence of operations $s_1, s_2, \ldots, s_m$ we proceed as follows:

(i) Set q_{curr} to q_0, i_{doc} to 1, and i_{seq} to 1.

(ii) While $q_{curr} \neq q_f$:

  (a) Non-deterministically pick a transition $(q_{curr}, a, q_{next}) \in \delta$ such that $a = a_{i_{doc}}$ or $a = s_{i_{seq}}$. If no such transition exists, then reject.

  (b) Set q_{curr} to q_{next}, and if $a = a_{i_{doc}}$, increment i_{doc} by one; otherwise, increment i_{seq} by one.

(iii) if $i_{doc} = n + 1$, then accept; otherwise, reject.

If at any point the counters go "out of bounds", then we also reject. This part of the algorithm will run in time at most $(2k)! \cdot q(n)$, for some polynomial q.

It is straightforward to prove that these modification will not alter the correctness of the algorithm. Also, by combining the different parts of the algorithm, we will get a total

running time of $k! \cdot p(n) + (2k)! \cdot q(n) + r(n)$ where p, q, r are polynomials. This is in $O(f(k)n^c)$ for some constant c and some function f. Therefore, the problem is in FPT. □

6. STATIC ANALYSIS AND COMPLEXITY

In this chapter , we study the computational complexity of static analysis problems for information extraction languages like satisfiability and containment. Determining the exact complexity of these problems is crucial for query optimization (Abiteboul et al., 1995) and data integration (Lenzerini, 2002), and it gives us a better understanding of how difficult it is to manage RGX formulas and VA. Although formal frameworks for defining rule-based information extraction languages were introduced by Fagin et al. (2015) and Arenas et al. (2016), and further studied by Freydenberger and Holldack (2016), there is still no analysis of static properties for regular IE languages (that is, languages without the content operator). Furthermore, as we have seen in Chapter 5, the results for regular languages generally do not extend to RGX or VA. Therefore, it is important to analyse the complexity of static problems for IE languages.

We start with the satisfiability problem for RGX formulas and VA. Formally, given an information extraction language $\mathcal{L}$, the satisfiability problem of $\mathcal{L}$ is defined as follows.

Problem:	SAT[$\mathcal{L}$]
Input:	An expression $\gamma \in \mathcal{L}$.
Question:	Does there exist a document d such that $[\![\gamma]\!]_d$ is non-empty?

SAT[$\mathcal{L}$] is the natural generalization of the satisfiability problem for ordinary regular languages: if γ does not contain variables, then asking if $[\![\gamma]\!]_d \neq \emptyset$ for some document d is the same as asking whether the language of γ is non-empty. It is a folklore result that satisfiability of regular languages given by regular expressions or NFAs has low-complexity (Hopcroft & Ullman, 1979). Unfortunately, in the information extraction context, this problems is intractable even for spanRGX.

THEOREM 6.1. SAT[VA] *and* SAT *of extraction rules are in* NP. *Furthermore,* SAT[spanRGX] *is* NP*-hard.*

To prove a part of the previous result, we have to show that Sat[VA] is in NP. In order to do this, we will first prove a lemma that will limit the size of the documents we must consider.

Lemma 6.1. *Let $A = (Q, q_0, q_f, \delta)$ be a* VA*, and let $\mathcal{V} = \mathcal{V}(A)$. If A is satisfiable, then there exists a document of size at most $(2 \cdot |\mathcal{V}| + 1)|Q|$ that satisfies it.*

Proof. The proof of this lemma follows a similar idea to the idea behind the pumping lemma for regular languages (Hopcroft & Ullman, 1979). Suppose the smallest document $d = a_1 \cdots a_n$ that satisfies A is of size greater than $(2 \cdot |\mathcal{V}| + 1)|Q|$, and let μ be its corresponding mapping. Then, there must exist a substring $a_k \cdots a_l$ in d of size at least $|Q| + 1$ inside which A does not use any variable operations (since A can use at most $2 \cdot |\mathcal{V}|$ variable operations). Denote the state of A after processing a_i as q_i. Since A has $|Q|$ states, there must exist $i, j \in [k, l]$ such that $i < j$, $q_i = q_j$, and $|a_k \cdots a_i a_{j+1} \cdots a_l| \leq |Q|$. Because A does not use any variable operations in this substring, it is clear that if A accepts d and μ, then it will accept $d' = a_1 \cdots a_i a_{j+1} \cdots a_n$ and μ', where μ' is μ with all the positions greater than j adjusted by $(j - i)$. If we repeat this for all substrings of size greater than $|Q|$ with no variable operations, then the final document will have size at most $(2 \cdot |\mathcal{V}| + 1)|Q|$, contradicting our initial supposition. This proves the lemma. □

With this result in mind, we can now prove the previous theorem.

Proof of Theorem 6.1. A direct consequence of Lemma 6.1 is that every satisfiable VA A has an accepting run that is at most polynomial in size with respect to A. Therefore, a NP algorithm for Sat[VA] is to simply guess a run and check that it is an accepting run (which can easily be done in polynomial-time).

Now, we prove that Sat[spanRGX] is NP-hard. Consider the proof of Theorem 5.1. Notice that the expression γ_α is satisfiable if and only if it is satisfied by document $d = \varepsilon$, since γ_α only matches empty documents. Therefore, 1-IN-3-SAT can be reduced to Sat[spanRGX]. Since the former is NP-hard, the latter is also NP-hard. □

These results show that satisfiabilty is generally NP-complete for all IE languages considered in this thesisṪhe next step is to consider syntactic restrictions of RGX or VA, i.e., sequentiality introduced in Chapter 5. Indeed, when considering sequential VA we can restore tractability.

THEOREM 6.2. SAT[seqVA] *is in* NLOGSPACE.

PROOF. Let $A = (Q, q_0, q_f, \delta)$ be a sequential variable automata. Notice that any sequential path from q_0 to q_f corresponds to an accepting run, because sequential paths respect the correct use of variables. Since A is sequential, finding an accepting run for A is as easy as finding a path from q_0 to q_f. This problem is equivalent to the problem of reachability on graphs, which is in NLOGSPACE (Papadimitriou, 1993). □

It is interesting to note that this result is very similar to satisfiability of finite state automata: given a sequential VA the NLOGSPACE algorithm simply checks reachability between initial and final states. This again shows the similarity between finite state automata and VA if the sequential restriction is imposed.

Next, we consider extraction rules combined with the sequential and functional spanRGX. Similarly as before, SAT of extraction rules remains intractable even for the class of functional dag-like rules. However, if we consider sequential tree-like rules we can restore tractability since tree-like rules are always satisfiable.

THEOREM 6.3. SAT *of functional dag-like rules is* NP*-hard. Furthermore, any sequential tree-like rule is always satisfiable.*

PROOF. Consider the proof of Theorem 5.4. Notice that the rule φ in this proof is satisfiable if and only if it is satisfied by the document $d = \#$, since φ matches only one # symbol. Therefore, 1-IN-3-SAT can be reduced to SAT of functional dag-like rules in polynomial time. Since the former problem is NP-hard, the latter must also be NP-hard. □

It is important to make the connection here between regular expressions and sequential RGX: both formalisms are trivially satisfiable. We believe that this gives more evidence that sequential RGX are the natural extension of regular expressions, as they inherit all the good properties of its predecessor.

We continue by considering the classical problems of containment and equivalence of expressions. Since equivalence can be defined in terms of containment, we only formalize the latter.

Problem:	CONTAINMENT[$\mathcal{L}$]
Input:	Expressions γ_1 and γ_2 in $\mathcal{L}$.
Question:	Does it hold that $[\![\gamma_1]\!]_d \subseteq [\![\gamma_2]\!]_d$ for every document d?

It is well known that containment for regular languages is PSPACE-complete (Stockmeyer & Meyer, 1973), even for restricted classes of regular expressions (Martens, Neven, & Schwentick, 2009). Since our expressions are extensions of regular expressions and automata, these results imply that a PSPACE bound is the best we can aim for. Given that the complexity of evaluation and satisfiability for VA increases compared to regular languages, one would expect the complexity of containment to do the same. Fortunately, this is not the case. In fact, containment of all IE languages we consider is PSPACE-complete.

THEOREM 6.4. CONTAINMENT *of extraction rules and* CONTAINMENT[VA] *are* PSPACE*-complete.*

PROOF. As previously stated, it is easy to see that regular expressions are a subset of RGX, and it is known that the containment problem for regular expressions is PSPACE-hard. Therefore, we will only prove that CONTAINMENT[VA] is in PSPACE.

Let $A_1 = (Q_1, q_1^0, q_1^f, \delta_1)$ and $A_2 = (Q_2, q_2^0, q_2^f, \delta_2)$ be two variable automata. We will prove that deciding if $[\![A_1]\!]_d \subseteq [\![A_2]\!]_d$ for every document d is in PSPACE by describing a non-deterministic algorithm that decides its complement. The algorithm will attempt to

prove that there exists a counterexample, that is, a document d and a mapping μ such that $\mu \in [\![A_1]\!]_d$ and $\mu \notin [\![A_2]\!]_d$. At every moment, we will have sets $S_1 \subseteq Q_1$ and $S_2 \subseteq Q_2$ that will hold the possible states in which A_1 and A_2 might be. We will also have sets V and Y which will hold the available and open variables respectively.

Assume, without loss of generality, that $\mathcal{V} = \text{var}(A_1) = \text{var}(A_2)$ and $\mathcal{O} = \text{Op}(A_1) = \text{Op}(A_2)$. We define the ε-closure of a state q, denoted $E(q)$, as the set of states reachable from q by using only ε-transitions (including q). Similarly, we define $S(q, a) = \{q' \mid (q, a, p) \in \delta \text{ and } q' \in E(p)\}$, where $a \in (\Sigma \cup \mathcal{O})$ and δ is the relevant transition relation. Given a set of states R, we define $E(R) = \bigcup_{q \in R} E(q)$ (and $S(R)$ analogously). Lastly, we define $S(R, aw) = S(S(R, a), w)$, where $w \in (\Sigma \cup \mathcal{O})^*$.

The algorithm proceeds as follows:

(i) Set S_1 to $E(q_1^0)$, set S_2 to $E(q_2^0)$, set V to $\mathcal{V}$, and set Y to $\emptyset$.

(ii) If $q_1^f \in S_1$ and $q_2^f \notin S_2$, then accept. Otherwise, guess either an element a from Σ or a set of variable operations $P \subseteq \mathcal{O}$.

(iii) If the algorithm guessed $a \in \Sigma$ then:

  (a) Set S_1 to $S(S_1, a)$ and S_2 to $S(S_2, a)$.

  (b) Go to step 2.

(iv) If the algorithm guessed a set P of variable operations, then:

  (a) Check that P is compatible with V and Y. If they are, the update V and Y accordingly; if not, reject.

  (b) Let $\text{Perm}(P)$ be the set of all strings that are permutations of P.

  (c) Set S_i to $\bigcup_{w \in \text{Perm}(P)} S(S_i, w)$ for $i \in \{1, 2\}$.

  (d) Go to step 2.

It is clear that this algorithm uses only polynomial-space, since we are only guessing strings of polynomial size, and storing information about variables and states.

Now we prove that the algorithm is correct. Notice that if the algorithm accepts, then there exists strings w_1 and w_2 differing only on the ordering of consecutive variable operations, such that $q_1^f \in S(E(q_1^0), w_1)$ and $q_2^f \notin S(E(q_2^0), w_2)$. Moreover, $q_1^f \in S(E(q_1^0), w_1)$ if and only if there exists a document d and mapping μ such that $\mu \in [\![A_1]\!]_d$. Since w_1 and

w_2 generate the same document-mapping pairs, and the algorithm tries all the possible permutations of consecutive variable operations, it is clear that there is no accepting run in A_2 with label w_2. Therefore $\mu \notin [\![A_2]\!]_d$, which concludes the proof. □

Given that all RGX subfragments contain regular expressions, it does not make sense to consider the functional or sequential restrictions of RGX to lower the complexity. Instead, we have to look for subclasses of regular languages where containment can be decided efficiently like, for example, deterministic finite state automata (Hopcroft & Ullman, 1979). It is well-known that containment between deterministic finite state automata can be checked in PTIME (Stockmeyer & Meyer, 1973). Then a natural question is: what is the deterministic version of VA? One possible approach is to consider a deterministic model that, given any document produces a mapping deterministically. Unfortunately, this idea is far too restrictive since it will force the model to output at most one mapping for each document. A more reasonable approach is to consider an automata model that behaves deterministically *both* in the document and the mapping. This can be formalized as follows: a VA (Q, q_0, q_f, δ) is *deterministic* if δ does not have ε-transitions and for every $p \in Q$ and $v \in \Sigma \cup \{x\vdash, \dashv x \mid x \in \mathcal{V}\}$ there exists at most one $q \in Q$ such that $(p, v, q) \in \Delta$. That is, the transition relation of a deterministic VA is functional with respect to both Σ and $\mathcal{V}$. Although the deterministic version of VA seems obvious, as far as we know, this is the first attempt to introduce this notion for IE languages.

The first natural question to ask is whether deterministic VA can still define the same class of mappings as the non-deterministic version. Indeed, one can easily show that every VA can be determinized by following the standard determinization procedure (Hopcroft & Ullman, 1979).

Proposition 6.1. *For every* VA *A, there exists a deterministic* VA A^{det} *such that* $[\![\mathcal{A}]\!]_d = [\![A^{det}]\!]_d$ *for every document* d.

Proof. Let $A = (Q, q_0, q_f, \delta)$ be a variable automaton. We will determinize A by using the classical method of *subset construction* (Hopcroft & Ullman, 1979). Without loss of

generality, we will allow a set of final states instead of a single final state. We reuse the definitions of $E(q)$ and $S(q)$ from the proof of Theorem 6.4.

We define the deterministic variable automaton $A^{det} = (Q', q'_0, F', \delta')$ as follows. Let $Q' = 2^Q$, $q'_0 = E(q_0)$, $F' = \{P \in Q' \mid q_f \in P\}$. The transition $(P, a, P') \in \delta'$ if and only if $P' = \bigcup_{q \in P} S(q, a)$.

Now we will prove that for every document d and mapping μ, $\mu \in [\![A]\!]_d$ if and only if $\mu \in [\![A^{det}]\!]_d$. Let ρ be an accepting run for d and μ on A. Then it is easy to prove by induction that ρ can be mapped to an accepting run ρ' in A^{det}. For the base case, we have that $q_0 \in q'_0$. For the inductive case, consider that ρ uses transition (p, a, p'), and that the last state we appended to ρ' is P: if $a = \varepsilon$ then $p' \in P$ and we do nothing to ρ'; if $a \in (\Sigma \cup Op)$ then there exist $(P, a, P') \in \delta'$ such that $p' \in P'$, so we add P' to ρ'. Since ρ' uses the same transitions as ρ (except for ε-transitions), A^{det} will also accept d and μ.

Now consider the opposite direction: if there is an accepting run ρ' in A^{det}, then there is an accepting run ρ in A. This is also easily proved with induction. In this case the inductive hypothesis is that if there exists a path from P to P' using a certain sequence of symbols and variable operations, then for all $p' \in P'$ there exists $p \in P$ such that there is a path from p to p' using the same sequence of symbols and operations. For the base case we have that $E(q_0) = q'_0$, so it is trivial. For the inductive case, consider that ρ' uses transition (P, a, P'). Consider some state $p' \in P'$. By definition, there is some state $q \in P'$ such that $p' \in E(q)$ and there exists a state $p \in P$ such that $(p, a, q) \in \delta$. By composing the different paths between states, we get the path that proves our hypothesis. By considering the last state in ρ' then, we can build an accepting run ρ. □

As mentioned previously, the motivation of having a deterministic model is to look for subclasses of VA where Containment has lower complexity. We can indeed show that this is the case for deterministic VA, although the drop in complexity is not as dramatic as with regular languages.

Theorem 6.5. Containment *of deterministic* VA *is in* Π_2^p. *Moreover,* Containment *of deterministic sequential* VA *is* coNP*-complete.*

Proof. Let $A_1 = (Q_1, q_1^0, q_1^f, \delta_1$ and $A_2 = (Q_2, q_2^0, q_2^f, \delta_2)$ be deterministic variable automata. Assume, without loss of generality, that $O = \text{Op}(A_1) = \text{Op}(A_2)$ and $\mathcal{V} = \text{var}(A_1) = \text{var}(A_2)$. We will prove the theorem by showing that the complement of this problem is in Σ_2^P. We describe an algorithm that will accept if there exists a document d and mapping μ such that $\mu \in [\![A_1]\!]_d$ and $\mu \notin [\![A_2]\!]_d$. We will use the fact that when we fix some linear order $\preceq_{\text{Op}}$ over the variable operations, then there is a unique label λ to each document-mapping pair (d, μ), denoted $L(d, \mu, \preceq_{\text{Op}})$, d.

First, we guess a document d, a mapping μ, and a linear order $\preceq^1_{\text{Op}}$ over O. Then, for all linear orders $\preceq^2_{\text{Op}}$ over O, we execute the following polynomial-time procedure. We compute the label $\lambda_1 = L(d, \mu, \preceq^1_{\text{Op}})$ and the label $\lambda_2 = L(d, \mu, \preceq^2_{\text{Op}})$. Finally, we check if there is a run in A_i that has λ_i as a label, for $i \in \{1, 2\}$. This is equivalent to checking if a deterministic finite automaton accepts a word, and therefore it can be done in polynomial time. If A_1 accepts λ_1 and A_2 rejects λ_2, then we accept; otherwise, we reject.

It is straightforward to prove that the algorithm is correct. Therefore it only remains to show that the guessed document d is of polynomial size (since that will determine the size and running time of the rest). This can be done by using the same "pumping lemma" argument from the proof of Theorem 6.1. In this case, the substrings without variable operations will be of size at most $|Q_1| \cdot |Q_2|$; if its longer, then there are indices i and j such that the pair of states of A_1 and A_2 will be the same at position i and j, and therefore we can shorten the substring by removing the characters in between. Therefore, we only need to consider documents of size at most $(2|\mathcal{V}| + 1)|Q_1||Q_2|$.

Now we prove that for deterministic sequential variable automata A_1, A_2 the problem is in coNP. As in the previous case, we show that the complement of the problem is in NP. To do this, we guess a document d and a mapping μ and then check that $\mu \in [\![A_1]\!]_d$ and $\mu \notin [\![A_2]\!]_d$. This is the ModelCheck problem, and since A_1 and A_2 are sequential, Theorem 5.3 guarantees that we can check this in polynomial time. The same argument made in the previous case for the size of d applies here.

It only remains to prove that Containment of deterministic sequential variable automata is coNP-hard. For this we will describe a polynomial-time reduction from the *disjunctive*

normal form validity problem. The problem consists in determining whether a propositional formula α in disjunctive normal form is valid, that is, all valuations make α true. We may assume, without loss of generality, that every clause in α has exactly three literals. This problem is known to be coNP-complete, since it can be easily shown to be the complement of the *conjunctive normal form satisfiability* problem (Garey & Johnson, 1979).

Let $\alpha = C_1 \vee \cdots \vee C_m$ be a propositional formula in disjunctive normal form with propositional variables $\{p_1, \ldots, p_n\}$, and let $C_i = l_{i,1} \wedge l_{i,2} \wedge l_{i,3}$ ($i \in [1, m]$), where each $l_{i,j}$ is a literal. We assume, without loss of generality, that for each clause C_i, the variables corresponding to its literals are pairwise distinct.

Now, we will describe the procedure for constructing automata $A_1 = (Q_1, q_1^0, q_1^f, \delta_1)$ and $A_2 = (Q_2, q_2^0, q_2^f, \delta_2)$. The construction will only use variable operation transitions so, in order to simplify the construction, we use transitions of the form (p, x, q) to represent a "gadget" that opens and closes variable x in succession, that is, a fresh state r and the transitions $(p, x\vdash, r)$ and $(r, \dashv x, q)$. For the automata, we are going to use variables $p_1, \ldots, p_n$ to represent positive literals; $\overline{p_1}, \ldots, \overline{p_n}$ to represent negative literals; and $c_1, \ldots, c_m$ to represent clauses. Thus, we have a total of $2n + m$ variables.

The automaton A_1 is will consist of a long chain with two parts. In the first part, states are joined with two parallel transitions p_i and $\overline{p_i}$, for every propositional variable p_i. This forces the automaton to choose a valuation for the propositional variables. The second part consists of a path with all the clause variables c_i. This will make the automaton compatible with A_2. Formally, A_1 is defined as follows:

$$Q_1 = \{r_1, \ldots, r_{n+m+1}\} \qquad q_1^0 = r_1 \qquad q_1^f = r_{n+m+1}$$

$$\delta_1 = \{(r_i, p_i, r_{i+1}), (r_i, \overline{p_i}, r_{i+1}) \mid i \in [1, n]\} \cup \{(r_{n+i}, c_i, r_{n+1+i}) \mid i \in [1, m]\}$$

The automaton A_2 will consist of m independent branches, each one representing a clause. Each branch has three parts, which we refer to as parts (a), (b), and (c). Part (a) starts with the clause variable c_i, and then follows with the variables corresponding to the literals in C_i. This forces any run that assigns c_i to also assign its literals. In part (b), states are joined with two parallel transitions p_j and $\overline{p_j}$, for every propositional variable p_j not

used in C_i. This lets the run decide the truth value of the variables not used in the clause. Finally, part (c) consists of a path with all the clause variables c_k such that $i \neq k$. That is, we let the run assign the rest of the variables corresponding to clauses. Therefore, every accepting run in this automaton corresponds to a variable assignment that makes some clause in α true. Figure 6.1 shows how a branch of this automaton looks for a particular clause.

We will give the formal definition of the automata that correspond to the branches of A_2. Formally, the automaton $A_2^i = (Q, q_0, q_f, \delta)$ corresponding to clause C_i is defined as follows ($i \in [1, m]$).

$$Q = \{s_1, \ldots, s_{n+m+1}\} \qquad q_0 = s_1 \qquad q_f = s_{n+m+1}$$

We define the transition relation δ in three sets, δ^a, δ^b, δ^c, corresponding to the three different parts of the automaton.

$$\begin{aligned}
\delta^a &= \{(s_1, c_i, s_2), (s_2, l_{i,1}, s_3), (s_3, l_{i,2}, s_4), (s_4, l_{i,3}, s_5)\} \\
\delta^b &= \{(s_{j+4}, p'_j, s_{j+5}), (s_{j+4}, \overline{p'_j}, s_{j+5}) \mid p'_1, \ldots, p'_{n-3} \text{ are the variables} \\
&\qquad \text{not in } C_i, \text{ in some order, and } j \in [1, n-3]\} \\
\delta^c &= \{(s_{n+j+1}, c'_j, s_{n+j+2}) \mid c'_1, \ldots, c'_{m-1} \text{ are the clause variables} \\
&\qquad \text{different from } c_i, \text{ renumbered, and } j \in [1, m-1]\}
\end{aligned}$$

Thus, $\delta = \delta^a \cup \delta^b \cup \delta^c$. Finally, A_2 is defined as the automaton that results when we take all the A_2^i automata ($i \in [1, m]$) and fuse their initial states into a single initial state, and fuse all their final states into one.

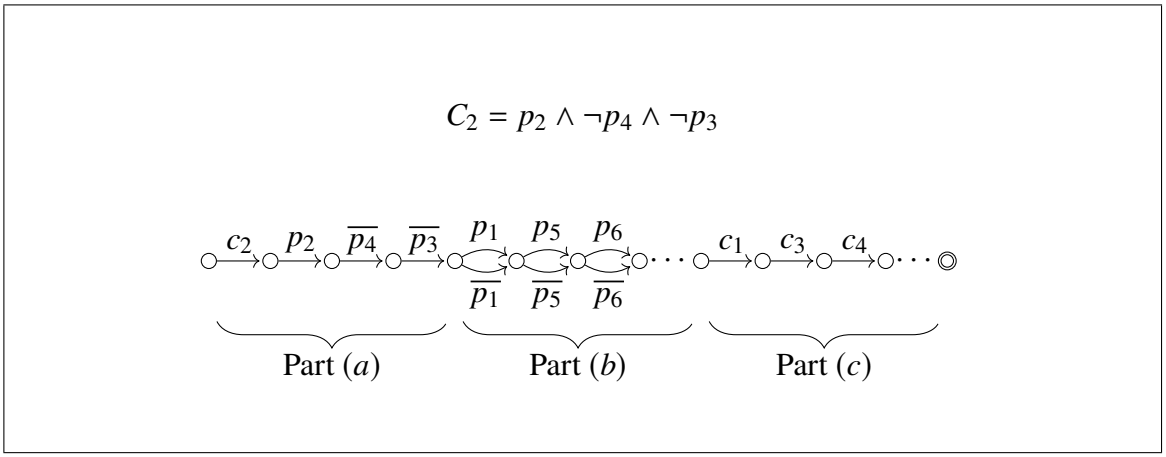


Figure 6.1. Example of automaton A_2^i, where $i = 2$, from the proof of Theorem 6.5.

Now we prove that $[\![A_1]\!]_d \subseteq [\![A_2]\!]_d$ for every document d if and only if α is valid. First, notice that we need only consider $d = \varepsilon$, since this is the only document that may satisfy A_1 and A_2. First, it is easy to see that each mapping μ corresponds to a valuation ν, namely, by considering $\nu(p) = 1$ if $p \in \mathrm{dom}(\mu)$, and $\nu(p) = 0$ if $\overline{p} \in \mathrm{dom}(\mu)$. The automaton A_1 will accept the set of mappings that correspond to all possible valuations over $p_1, \ldots, p_n$. It is also easy to see that the branch i in A_2 will accept mapping μ if and only if μ corresponds to a valuation that satisfies clause C_i. Therefore, if $[\![A_1]\!]_d \subseteq [\![A_2]\!]_d$, then A_2 accepts the mappings corresponding to all possible valuations. This means that for each valuation ν there is a clause in α satisfied by ν, which means that α is valid. □

Since the complexity is still intractable, it would be interesting to isolate a subclass of deterministic VA, where containment can be solved in PTIME. It is not difficult to show that this can be obtained by considering deterministic VA which never produce a mapping extracting two spans that intersect at their limit points.

7. APPLICATION TO DOCUMENT ANNOTATION

The ideas and results presented in this thesis can also be applied in a practical scenario. One such scenario is the annotation of semi-structured documents, such as *comma-separated values* (CSV) files.

In this context, an *annotation* is a relation over spans, which is used to denote that the tuples in this relation have a certain property or are related to a certain concept. For example, one may use an annotation to indicate all spans in a document that correspond to dates in "yyyy-mm-dd" format, or to indicate all pairs of spans in a CSV file that violate a primary-key constraint. To accomplish this objective, one may use the rules explored in this thesis to specify the regions in a document that will be associated with a specific annotation.

In their paper, Arenas et al. (2016) propose the language of *navigation expressions*, which is a subset of spanRGX that fulfills most of the common use cases related to this task. In this chapter , we present this language and explain how the results of this thesis apply to it.

7.1. Navigation expressions

We start by defining a navigation language, which uses span regular expressions in a very restricted form but can express most of the span-directed extraction used in practice. A *navigation expression* (NE) ψ is defined by the following grammar:

$$\psi \;::=\; \psi/\psi \;\mid\; \texttt{any}(S) \mid \texttt{next}(S) \mid x : \texttt{next}(S),\ x \in \mathcal{V} \;\mid\; \langle x \rangle : \texttt{next}(S),\ x \in \mathcal{V}$$

$$S \;::=\; w,\ w \in \Delta^+ \;\mid\; S + S$$

where S is assumed to be prefix free, that is, every expression S is of the form $w_1 + \cdots + w_n$, where (a) $\Delta = \Sigma \cup \{\vdash, \dashv\}$, (b) every $w_i \in \Delta^+$ $(1 \leq i \leq n)$, and (c) w_i is not a prefix of w_j $(1 \leq i, j \leq n$ and $i \neq j)$.

An NE is constructed as a sequence of expressions using either `any` or `next`. The axis $\texttt{any}(S)$ is used to move forward in a document reading any sequence of symbols ending

with a word in S, while the axis $\texttt{next}(S)$ is used to move forward to the next occurrence of a word in S. Moreover, $x : \texttt{next}(S)$ and $\langle x \rangle : \texttt{next}(S)$ perform the same form of navigation as $\texttt{next}(S)$, but in the former case the traversed span is stored in the variable x, while in the latter case it is checked whether the content of the traversed span coincides with the content of the span stored in x. Thus, the expressions $\texttt{any}(S)$, $\texttt{next}(S)$, $x : \texttt{next}(S)$ and $\langle x \rangle : \texttt{next}(S)$ are useful to restrict the navigation between two or more separators, which is a very common operation on CSV documents. Notice that we assume that S is prefix free, as the set of separators used in practice usually satisfies this restriction (e.g comma and semicolon).

We define the semantics of NEs in the same way as for span regular expressions. More precisely, given a document d and a mapping μ over d, the base case $[\![S]\!]$ and the recursive case $[\![\psi_1/\psi_2]\!]$, which denotes concatenation, are defined as for the case of span regular expressions. Moreover, $[\![\texttt{any}(S)]\!]$ is defined as $[\![(\Sigma \cup \{\vdash, \dashv\})^*/S]\!]$. Finally, the evaluation of $\texttt{next}(S)$, $x : \texttt{next}(S)$ and $\langle x \rangle : \texttt{next}(S)$ are defined as follows assuming that $S = w_1 + \cdots + w_n$ and r_S is the spanRGX $\Delta^* w_1 \Delta^* + \cdots + \Delta^* w_n \Delta^*$ with $\Delta = \Sigma \cup \{\vdash, \dashv\}$:

$$[\![\texttt{next}(S)]\!] = \{(i,j) \in \text{span}(d) \mid \exists k \geq i : (k,j) \in [\![S]\!] \text{ and } (i, j-1) \notin [\![r_S]\!]\}$$

$$[\![x : \texttt{next}(S)]\!] = \{(i,j) \in \text{span}(d) \mid \exists k \geq i : \mu(x) = (i,k), (k,j) \in [\![S]\!] \text{ and } (i, j-1) \notin [\![r_S]\!]\}$$

Notice that a span p belongs to $[\![r_S]\!]$ if the content associated to p is a string of the form uw_iv with $u, v \in \Delta^*$ and $1 \leq i \leq n$, that is, if one of the separators in S occurs in the content associated to p. Thus, if $(i,j) \in [\![\texttt{next}(S)]\!]$, then we know that there exists a position k such that $i \leq k < j$, the content of (k,j) is a word in S and no separator in S occurs between positions i and $j-1$. Hence, in this case we know that k is the next position from i where a separator from S occurs.

Extraction expressions based on NEs are defined exactly as for spanRGX, that is, if ψ is an NE, then ψ and $x.\psi$ are considered to be extraction expressions if $x \in \mathcal{V}$. However, the

semantics of these formulas are defined in a slightly different way:

$$\begin{aligned}
[\![\psi]\!]_d \;=\; \{\mu \mid \mu \text{ is a mapping over } d \text{ such that} \\
(1,k) \in [\![\psi]\!] \text{ for some } k, 1 \leq k \leq |d|+1\} \\
[\![x.\psi]\!]_d \;=\; \{\mu \mid \mu \text{ is a mapping over } d \text{ such that} \\
\mu(x) = (i,j) \text{ and } (i,k) \in [\![\psi]\!] \text{ for some } k, i \leq k \leq j\}
\end{aligned}$$

This definition formalises the fact that NEs are intended to be used to navigate forward in a document until we find a separator, and without taking into consideration the symbols after this separator. Thus, $\mu \in [\![\psi]\!]_d$ if there exists a prefix $(1,k)$ of the span $(1,|d|+1)$ representing the entire document d such that $(1,k)$ conforms to the conditions encoded in ψ, thus without taking into account the symbols in the positions $k+1, \ldots, |d|$.

A *navigation rule* is defined the same way as before, but using NE instead of spanRGX. The notion of *tree-like navigation rules* is defined analogously.

7.2. Complexity of evaluating Navigation Expressions

Given that this is a language designed to be used in a practical setting, we would expect it to have good complexity properties. In their paper, Arenas et al. (2016) proved that tree-like navigation rules can be efficiently evaluated. This, however, should not come as a surprise to the reader of this thesis, since tree-like navigation rules combine the two properties that we have shown are essential to have good complexity properties: sequentiality and tree-like structure.

Given that NEs are a restriction of sequential spanRGX, and that we have already shown that sequential tree-like rules can be evaluated efficiently (Theorem 5.5), we obtain the following result.

Theorem 7.1. Eval *of tree-like navigation rules is in* PTIME. □

Even though the above theorem gives us an algorithm for evaluating NEs, that algorithm is far too general, and would probably have an unsatisfactory performance in a real-world system. For this purpose, we designed and implemented an algorithm that is tailored to NEs.

7.3. Efficient evaluation of NEs

The use of our framework requires an efficient algorithm for enumerating all the mappings satisfying a navigation expression. We provide an algorithm that has running time $O(|\psi| \cdot |d| + |\text{Output}|)$, where ψ is a navigation expression, d is a document and $|\text{Output}|$ is the size of the output. Furthermore, this algorithm belongs to the class of constant-delay algorithms (Segoufin, 2014), namely, enumeration algorithms that take polynomial time in preprocessing the input (i.e. ψ and d), and constant time between two consecutive outputs (i.e. mappings).

7.3.1. A normal form for navigation expressions

The initial step for evaluating a navigation expression is to remove unnecessary `any`-operators from the input navigation expression. For this purpose, we introduce a normal form for navigation expressions and show that every formula can be transformed into this normal form. Specifically, we say that an NE φ is a *next-formula* if it is the concatenation of `next`-operators and at least one variable occurs in φ. Then a navigation expression is in *next normal form* (NNF) if it is of the form $\varphi_0/\texttt{any}(S_1)/\varphi_1/\ldots/\texttt{any}(S_k)/\varphi_k$, where $k \geq 0$, φ_0 is a sequence of `next`-operators and $\varphi_1, \ldots, \varphi_k$ are next-formulas. Thus, between any pair of contiguous `any`-operators there must exist at least one variable that captures a span.

The next step is to show that every NE φ can be efficiently converted into an equivalent NE ψ in NNF. Here we say that φ and ψ are equivalent if for every document d and every mapping μ over d, it holds that $[\![\varphi]\!] = [\![\psi]\!]$. Then we consider the following rewriting rules

to meet our goal:

$$\begin{aligned} \psi_1/\texttt{any}(S)/\varphi/\texttt{any}(S')/\psi_2 &\rightarrow \psi_1/\texttt{next}(S)/\varphi/\texttt{any}(S')/\psi_2 \\ \psi_1/\texttt{any}(S)/\varphi &\rightarrow \psi_1/\texttt{next}(S)/\varphi \end{aligned}$$

where φ is a sequence of zero or more `next`-operators without variables, and ψ_1, ψ_2 are arbitrary NEs. The following example illustrates how these two rules can be used to convert a navigation expression into an equivalent one in NNF.

EXAMPLE 7.1. *Consider the following navigation expression:*

$$\varphi = \texttt{any}(;)/\texttt{next}(;)/\texttt{any}(\hookleftarrow)/x:\texttt{next}(\hookleftarrow)/\texttt{any}(;)$$

This NE is not in NNF as it starts with two `any`*-operations without variables in between. This can be solved by applying the first rewriting rule, giving us:*

$$\varphi_1 = \texttt{next}(;)/\texttt{next}(;)/\texttt{any}(\hookleftarrow)/x:\texttt{next}(\hookleftarrow)/\texttt{any}(;)$$

Notice that φ and φ_1 are equivalent NEs. Now φ_1 is not in NNF as it ends with an `any`*-expression. This can be solved by applying the second rewriting rule, resulting in:*

$$\varphi_2 = \texttt{next}(;)/\texttt{next}(;)/\texttt{any}(\hookleftarrow)/x:\texttt{next}(\hookleftarrow)/\texttt{next}(;)$$

Finally, we have that φ_2 is in NNF and φ_2 is equivalent to φ.

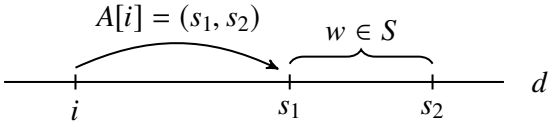
It can be proved that every NE φ can be transformed into an equivalent NE ψ in NNF by using the previous rewriting rules. Moreover, this transformation can be performed in time $O(|\varphi|)$.

7.3.2. An efficient algorithm for evaluating NE

We divide the evaluation of an NE into four steps. The input of this process is an NE ψ in NNF and a document d, and then the output is the set of mappings μ such that $\mu \in [\![\psi]\!]_d$.

We assume that ψ has no repeated variables, as if it had then the evaluation of ψ would be empty (recall that the content operator is not used, and two spans (i, j) and (k, ℓ) are assumed to be equal if $i = k$ and $j = \ell$). Besides, it can be easily checked whether ψ has repeated variables.

The first procedure takes as input a document d and a prefix-free set of words S (recall that in an NE of the form $\texttt{any}(S)$ or $\texttt{next}(S)$, the set S is assumed to be prefix-free). The procedure then runs the Aho-Corasick algorithm (Aho & Corasick, 1975) to produce an array A that is of the length of the input document, and such that $A[i]$ stores the next span in d that matches a word from S for every $i \in \{1, \ldots |d|\}$. This idea is illustrated in the following figure.



In this figure, and others illustrating how the evaluation works, the straight line represents the input document d, while the markings i, s_1, s_2 denote positions inside the document. Recall that the content of a span (i, j) is the infix of d between position i and $j - 1$.

The algorithm itself (called separators_match) is given in Figure 7.1. To analyse the algorithm, observe that we repeatedly run the Aho-Corasick string matching procedure. The iterator m starts at the begining of the document and after we find a match (s_1, s_2) for some string in S we store this span into $A[m]$ through $A[s_1]$. After this the iterator m is set to $s_1 + 1$, as this is the position of the next possible match. It is important to stress that there are no matches for strings in S beginning between positions m and s_1, therefore (s_1, s_2) is the first possible match. Besides, due to the fact that S is prefix-free, this is also the only possible match starting at s_1. As the running time of the Aho-Corasick algorithm is $O(|S| + |d|)$ (since S is prefix-free), and the only overhead we have is assigning spans to the array A, the total time of the algorithm separators_match is still $O(|S| + |d|)$.

```
Data: A document d and a prefix-free set of words S
Result: An array A[1..|d| + 1] of span(d)
Function separators_match(d, S)
    aho_corasick. init(d, S)
    m := 1
    while (s1, s2) := aho_corasick. next() do
        for i = m to s1 do
            A[i] := (s1, s2)
        m := s1 + 1
    return A
```

Figure 7.1. Finding all matches for a set of separators.

The next part of the algorithm, presented in Figure 7.2, deals with computing the possible valuations for a *context*, that is, a subformula of the expression that is of the form $\texttt{any}(S)/v_1 : \texttt{next}(S_1)/\ldots/v_n : \texttt{next}(S_n)$, where v_i is either a variable, or a placeholder $\perp$, specifying that `next` is used without a variable. Note that contexts are the building blocks of any expression in NNF since any NNF-expression is of the form $\varphi_0/E_1/\ldots/E_n$ where φ_0 is a sequence of `next`-operators and each subformula E_i is a context. Similarly to the previous algorithm, here we will again return an array whose i-th position will contain the information about the next possible match that occurs after the position i.

To start the computation, the algorithm context_match in Figure 7.2 calls the function separators_match from Figure 7.1 for each of the input sets of words S, S_1, …, S_n. The information for the set S is stored in an array A, while the information about each S_i is stored in an array $B[i]$. Therefore, A and $B[i]$, for $i = 1 \ldots n$, are all arrays of size $|d|$. This means that we can also refer to B as a matrix whose entry $B[i][j]$ contains the information about the next span matching a word from the set S_i after the position j of the input document d.

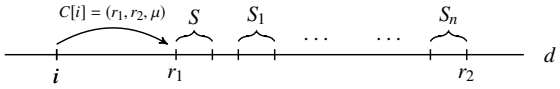
Data: A document d, a prefix-free set of words S, and a sequence $(v_1, S_1), \ldots (v_n, S_n)$ where each S_i is a prefix-free set of words and $v_i \in \mathcal{V} \cup \bot$.
Result: An array $C[1..|d|+1]$ of triples (r_1, r_2, μ), where $(r_1, r_2) \in \text{span}(d)$ and $\mu : \mathcal{V} \to \text{span}(d)$ is a partial function.

```
Function context_match(d, S, (v_1, S_1), ... (v_n, S_n))
    A := separators_match(d, S)
    for i = 1 to n do
        B[i] := separators_match(d, S_i)
    m := 1
    while A[m] ≠ null do
        (s_1, s_2) := A[m]
        r_1 := s_1
        μ := ∅
        for i = 1 to n do
            if B[i][s_2] = null then break
            (t_1, t_2) := B[i][s_2]
            if v_i ≠ ⊥ then μ(v_i) := (s_2, t_1)
            (s_1, s_2) := (t_1, t_2)
        if i = n + 1 then
            r_2 := s_2
            for i = m to r_1 do
                C[i] := (r_1, r_2, μ)
            m := r_1 + 1
        else if A[r_1 + 1] ⊑ A[r_1] then
            A[m] := A[r_1 + 1]
        else break
    return C
```

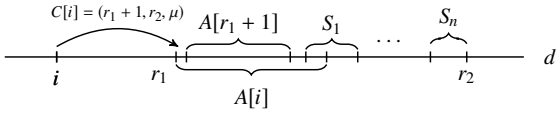
Figure 7.2. Finding all matches for a context.

The main loop of the algorithm now proceeds to try and match the context expression $\texttt{any}(S)/v_1:\texttt{next}(S_1)/\ldots/v_n:\texttt{next}(S_n)$ to the input document d to the right of the position m (beginning with $m = 1$). As long as the algorithm can keep on matching S (the condition

$A[m] \neq \texttt{null}$), it stores the information about the next possible match and tries to match S_1 through S_n (now using the matrix B in the for loop). If the matching was successful we store the information about the used-up portion of the document d into the (context) array C (note that this means that all positions from m to the start of the match for S, namely r_1, will have this information), and then we start matching again from the position $r_1 + 1$. In the case we did not manage to match all of the context expression, the only other possibility of a successful match is when we have that $A[r_1 + 1]$ is contained in $A[r_1]$, where a span (k_1, k_2) is contained in an span (ℓ_1, ℓ_2), denoted by $(k_1, k_2) \sqsubseteq (\ell_1, \ell_2)$, if $\ell_1 \leq k_1$ and $k_2 \leq \ell_2$. To clarify why this is so consider the following illustration of what the algorithm does.



If there is a match for S that starts after the position r_1, ends after the ending position of the current match for S, and allows to successfully match all the sets S_1 through S_n, then so does the current match of S (after all our expression only asks for the next position that matches each S_i and nothing more). Therefore, if the match starting at r_1 fails, so does one starting after r_1 that is longer than it. On the other hand, if the match at r_1 fails, but there is one starting at or after $r_1 + 1$ that is contained in it, there is a possibility for this match to be extended to cover all the sets S_1 to S_n. This possibility is illustrated in the figure below.



Notice that if all of the possibilities fail, we have exhausted our options and the algorithm finishes. To analyse the running time of the algorithm context_match, we first notice that running separately the function separators_match for the sets $S, S_1, \ldots, S_n$ takes total time $O(|S| + \sum_{i=1}^{n} |S_i| + (n + 1) \cdot |d|)$. Moreover, the while loop of the algorithm takes time $O(n \cdot |d|)$, as we have to loop over every set S_i, for $i = 1 \ldots n$. We can thus conclude that running the procedure context_match from Figure 7.2 takes time $O(|S| + \sum_{i=1}^{n} |S_i| + n \cdot |d|)$, which is indeed $O(|\psi| \cdot |d|)$ where $|\psi|$ is the size of the input navigation expression.

With the two procedures presented before, we are able to find all matchings for a context of the form $\texttt{any}(S)/v_1:\texttt{next}(S_1)/\ldots/v_n:\texttt{next}(S_n)$. Recall that every expression in NNF is simply a concatenation of such contexts, plus an easily evaluable initial segment. In fact, the last two steps of the algorithm assume that we have partitioned our input navigation expression ψ into contexts $E_1, \ldots, E_n$ of such subexpressions, and we have computed the corresponding arrays $C_1, \ldots, C_n$ using the algorithm context_match from Figure 7.2.

Data: A document d and a sequence $C_1, \ldots, C_n$ such that each C_i is an array $C_i[1..|d|+1]$ of triples (r_1, r_2, μ), where $(r_1, r_2) \in \text{span}(d)$ and $\mu : \mathcal{V} \to \text{span}(d)$ is a partial function.

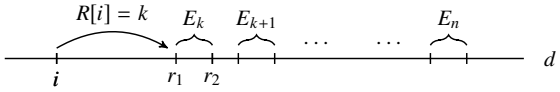
Result: An array $R[1..|d|+1]$ over $\{1, \ldots, n+1\}$.

Function forward_index($d, C_1, \ldots, C_n$)

```
k := n
for i = |d| + 1 to 1 do
    if k = 0 then
        R[i] := 1
    else if C_k[i] = null then
        R[i] := k + 1
    else
        (r1, r2, μ) := C_k[i]
        if R[r2] = k + 1 then k := k − 1
        R[i] := k + 1
return R
```

Figure 7.3. Computing the forward index.

The next step of our procedure is a quick index building algorithm (called forward_index) that will allow us to discard positions not leading to a match in an efficient manner. This step, presented in Figure 7.3, computes an array R such that for each position i in the input file (e.g. CSV-like document), $R[i]$ contains the least number k such that, starting from position i, it is possible to find a match for the subexpression $E_k/E_{k+1}/\cdots/E_n$ of the input expression. This idea can be depicted as follows:



As expected, the algorithm forward_index traverses the input document d backwards and at each position tries to match the context E_k, starting with $k = n$. If a match is possible (the second else clause in Figure 7.3), we take note of that and reduce the index k by one in order to move to the previous subexpression E_{k-1}. Since all of the information is already stored in arrays $C_1, \ldots, C_n$, the cost of this part of the algorithm is simply the cost of traversing the input document d once, or in other words $O(|d|)$.

The final part of the algorithm, presented in Figure 7.4, computes all the possible valuations that make an expression $\psi = E_1/E_2/\ldots/E_n$ true, where each E_i is a context of the form $\texttt{any}(S)/v_1 : \texttt{next}(S_1)/\ldots/v_n : \texttt{next}(S_n)$. As input, all_matches takes all of the information computed by the previous algorithms. In particular it has at its disposal the arrays C_i corresponding to the context E_i (computed by the function context_match), as well as the array R from forward_index. With this information the algorithm proceeds to compute the output in time that is proportional to the number of valuations that allow for a successful match of ψ.

The idea of the evaluation is to simulate the typical recursive approach that tries all possible combinations of matchings for E_1 through E_n and backtracks as necessary. To do this, we use the array T whose i-th position stores the next possible starting point for a match of the context E_i. Intuitively, T acts as a stack where we store the current position of a match for the context E_i in $T[i]$. Whenever $T[1]$ to $T[i]$ contains a match of E_1 to E_i, respectively, we try to match the contexts E_{i+1} to E_n (and compute all the successful matches for them) before we move the starting point of the next match for E_i one position to

the right. The procedure is then repeated until we have exhausted the search space. What makes our approach efficient is the fact that we only explore a branch of the search tree that is guaranteed to lead to a match. Specifically, we reduce the search space by using the index R from forward_index, which tell us if a match from the current position is possible. Next we describe this process in more detail.

The algorithm all_matches uses the iterator m to denote which context E_m is currently processing. It starts with $m = 1$ and with $T[1] = 1$, thus assuming that it will be possible to match the entire expression ψ to the input document d. In each step the algorithm then checks if it can match the part of the expression starting from E_m, namely, the subexpression $E_m/\ldots/E_n$. If this is not possible (the condition $R[T[m]] > m$ is true), then we simply move to the previous subexpression and try to match it from the next position to the right. Note that the use of array R allows us to terminate the evaluation of a branch that will not lead to a successful match at the first possible occasion. If a match is possible, we take note of that (σ_m stores the valuation for E_m using the information precomputed in array C_m) and move to the next context. This step is executed in the else clause of the algorithm all_matches. Finally, if m reaches $n + 1$, then we manage to match the entire expression (the final if clause), so we take the union of $\mu_1, \ldots, \mu_n$ to produce a mapping that makes ψ true (this union is well defined as ψ does not have repeated variables). Then we try to match E_n again, but this time starting one position to the right of the previous match. The algorithm then moves downwards to find the next match for E_{n-1} and so on until it found all the matches for ψ.

Data: A document d, an array $R[1..|d|+1]$ over $\{1,\ldots,n+1\}$, and a sequence $C_1,\ldots,C_n$ such that each C_i is an array $C_i[1..|d|+1]$ of triples (r_1, r_2, μ), where $(r_1, r_2) \in \text{span}(d)$ and $\mu : \mathcal{V} \to \text{span}(d)$ is a partial function.
Result: A set O of partial functions $\mu : \mathcal{V} \to \text{span}(d)$.

Function all_matches($d, R, C_1, \ldots, C_n$)
- **Let** $T[0..n+1]$ be an array of integers
- **Let** $\mu_1, \ldots, \mu_n$ be partial functions from $\mathcal{V}$ to span(d)
- $T[1] := 1$
- $m := 1$
- **while** $m \geq 1$ **do**
  - **if** $R[T[m]] > m$ **then**
    - $m := m - 1$
    - $T[m] := T[m] + 1$
  - **else**
    - $(r_1, r_2, \mu) := C_m[T[m]]$
    - $T[m] := r_1$
    - $\mu_m := \mu$
    - $m := m + 1$
    - $T[m] := r_2$
  - **if** $m > n$ **then**
    - $O := O \cup \{\mu_1 \uplus \cdots \uplus \mu_n\}$
    - $m := m - 1$
    - $T[m] := T[m] + 1$
- **return** O

Figure 7.4. Computing all possible matches.

To analyse the running time of the algorithm, first notice that the total running time needed to precompute arrays C_i and R is bounded by $O(|\psi| \cdot |d|)$. As discussed above, the final part of the algorithm simply outputs all valid matchings, and does so knowing in advance if a branch in the tree of all possible matches will result in a valid match. Therefore the running time of that part of the algorithm is equal to the number of valid matchings for the expression, or size of the output. Summing up, the total running time of the algorithm

is $O(|\psi| \cdot |d| + |\text{Output}|)$. The algorithm is also constant-delay (Segoufin, 2014), as it takes $O(|\psi| \cdot |d|)$ time to preprocess the input, and constant time between two consecutive outputs.

7.4. Experimental evaluation

To illustrate that the algorithm from the preceding section does not only have good theoretical complexity, in this section we describe how a system based on its implementation performs over real world datasets. Here we describe the datasets and the experiments used to test the efficiency of this algorithm, and compare it with the stream editing tool AWK. Due to the lack of space programs used in the experiments have been omitted, but are made available at *Annotating CSV documents: An Online Appendix* (2015), where we also provide the complete source code and documentation of our implementation and more detailed results of the experiments.

Implementation details. Our prototype implements a restricted but functional version of navigation programs that covers all use cases in W3C (2014). To simplify the implementation, we restrict navigational expressions to use at most one content operator, rules to be tree-like, and navigational programs to be non-recursive. It is important to add that to cope with the requirement *ForeignKeyReferences* we allow binary relations in the head of rules for storing results. For the evaluation of these programs, we use the algorithm discussed in this chapter to evaluate navigational expressions. Finally, for the evaluation of non-recursive programs we compute each rule in order, evaluating its navigational expressions separately, and intersecting their results with the precomputed annotations mentioned in the rule.

Datasets. We test our implementation on a number of CSV datasets and query logs. CSV files come from the use cases considered by W3C's CSV on the Web working group (W3C, 2014) and use data from: World Bank (2015) (WB), Office for National Statistics (2015) (ONS) and The City of Palo Alto, California Urban Forest Section (2015) (PA). While the Palo Alto tree data provides us with only one CSV file, for other two datasets we used many files of different sizes. In the end we tested our implementation on a total of thirteen different CSV files with sizes ranging from 6 to 183 MB. Note that most of the files

published by these organisations are of much smaller size, however, we decided to use larger files in order to show how the implementation works in extreme circumstances. As all of the CSV files we obtained were well structured, we also tested how our implementation behaves on noisy data by modifying the existing files by including additional empty rows, changing the expected values in one column in 5% of the rows, and adding an extra column to 5% of the rows. For each CSV document we created its noisy variant and used it for testing. The query log files we use were collected by The LSQ team (2015) and come from public SPARQL endpoints of The British Museum (2015) and DBpedia team (2015). For the experiments we used the raw log files provided by The LSQ team (2015) and tested our implementation on a total of 19 files, their sizes ranging between 2 and 190 MB. Due to a large number of files,and since files of similar size show same trends in evaluation times, we will present the evaluation result for only a handful of them. We provide the complete results for all the files in *Annotating CSV documents: An Online Appendix* (2015).

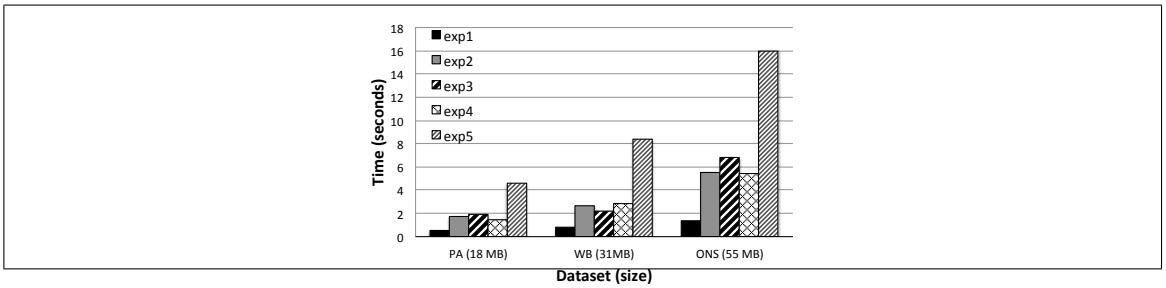


Figure 7.5. Running time on medium sized CSV files.

Experiments for CSV files. These experiments were motivated by the requirements for CSV metadata proposed by the W3C CSV on the Web working group (W3C, 2014). To test our implementation we will use five different experiments that run navigation programs which annotate CSV files, or check if some constraint is violated. Our first experiment (`exp1`) annotates the entire file. This type of annotation is used when we want to specify that the file is in particular language, or that it is to be displayed in a particular way. Next, in `exp2` we annotate a single column within a file, which is used when one wants to specify

that this column contains values that are of a certain type, or that it forms a primary key. In `exp3` we check if a primary key constraint is violated in our file. We continue with `exp4` where conformance of a column to a datatype specification is tested. Finally, in `exp5` we use a more complex navigation program consisting of three rules, each of which annotates a different column and checks that its values are of a certain datatype.

Experiments for query logs. Here we were annotating the data one might naturally want to obtain when managing log files, such as the actual text of the query, the time it was executed, the endpoint used, etc. Over these datasets we also use five experiments. The first experiment `exp1` annotates all the queries which use the OPTIONAL operator of the SPARQL query language. Similarly, in `exp2` we find all the queries which have more than one occurrence of this operator. In `exp3` we annotate all the queries using the FILTER operator that were executed between noon and one o'clock. Next, in `exp4` we find the queries with double OPTIONAL and using a binary output relation store such queries together with the time when they were executed. The final experiment, `exp5`, works similarly, but stores queries with two occurrences of FILTER, together with the details about the endpoint used to execute them.

Results. The testing was done using a Laptop with an Intel Core i7-4510u processor and 8 GB of main memory, running Arch Linux x86_64, kernel 4.2.2. Each experiment was ran three times and the average score was reported (we also note that there were no significant deviations from the average).

The first set of results, presented in Figure 7.5, shows the running times when CSV files of reasonable size are used. Here we test on three files: the Palo Alto trees file is 18 MB in size, while the World Bank file weights 35 MB and the ONS one 55 MB. As expected, the running times scale according to the size of the file, but as all of the times were really fast (less than sixteen seconds), we can conclude that on average sized files our implementation runs well considering that we use raw data with no precomputed indices. One can notice that times are much longer for experiment 5, however, this is to be expected, as the program `exp5` does about three times more work than any of the other programs, as discussed above.

Next we wanted to see how the results scale when similar files of increasing sizes are used. To test this we selected one large CSV file from the World Bank dataset of size 130 MB and containing around 600,000 lines. From this we generated ten different files, each containing first $k \times 60{,}000$ lines of the original file, with k between one and ten. The experiments were then ran against each of these files. As we can see from Figure 7.6, the results do scale as expected from the formal analysis this chapter. Here we selected experiments 1, 3 and 4 as they are the most representative. The other two experiments behave similarly (see *Annotating CSV documents: An Online Appendix* (2015) and Table 7.1).

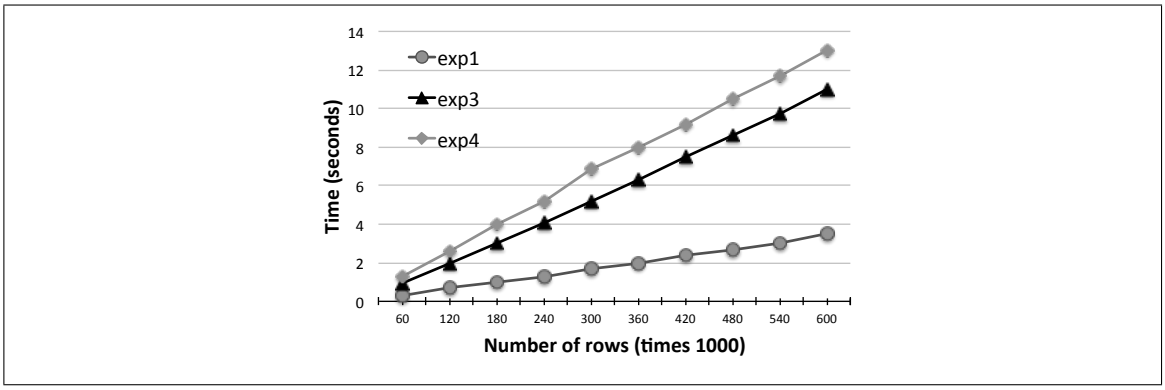


Figure 7.6. Scaling based on file size.

We also considered three large files (one obtained from the World Bank and two from the ONS). Although these files are in no way representative of the average size appearing in practice, we wanted to see if our implementation could still be used in these cases. The file from the World Bank dataset was 160 MB in size, while the two from the ONS weighed 175 MB and 183 MB. The test results are show in Table 7.1. As we can see, for files less than 200 MB in size there are no significant problems in terms of the evaluation. Since in practice one is likely to work with files that fall on the smaller end of the spectrum, we believe that our experiments serve to illustrate that navigation programs can be used efficiently.

Table 7.1. Running times (in seconds) on large files.

	exp1	exp2	exp3	exp4	exp5
WB (160 MB)	4	13	12	13	39
ONS (175 MB)	5	16	17	16	47
ONS (183 MB)	5	17	18	17	50

Our final round of experiments for CSV files was conducted using noisy documents which do not conform to the tabular format as described in W3C (2015). The theoretical analysis of our programs in this chapter showed that, whether a file is noisy or not, this should not have much impact on the performance of the evaluation and our experiments on CSV files with synthetically created noise show this. In particular, since the amount of noise we added did not significantly change the size of the files, the performance for each experiment was essentially the same as on original data (in fact the results on noisy files are generally slightly faster as less data passes the filters and gets stored). The details on noisy files and the precise performance of the experiments over them can be found at *Annotating CSV documents: An Online Appendix* (2015).

As far as the experimental results on query logs are concerned they show similar performance as the ones on CSV files. In Figure 7.7 we present evaluation times of our five programs for four different query log files. From the logs of the British Museum we selected the smallest and the largest file available, and from DBPedia we selected two files on the larger end of the spectrum. The sizes were selected so that they further illustrate the fact that the performance scales as the size of the document increases.

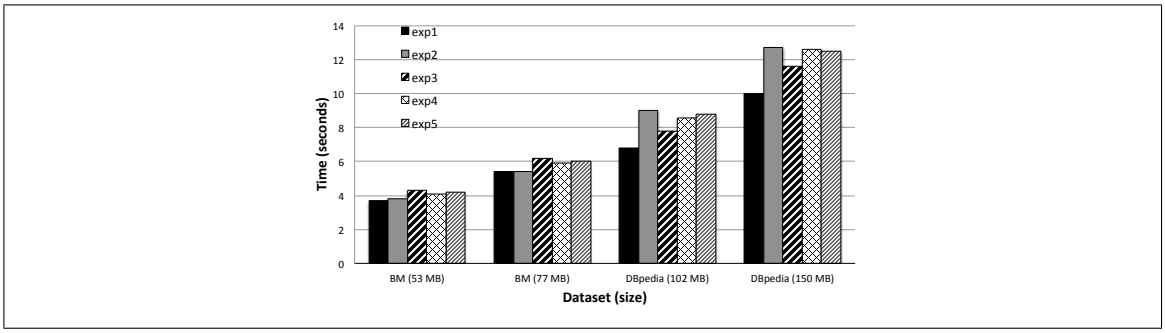


Figure 7.7. Running times on query logs.

Comparison with stream editing tools. We also compare our implementation with the standard stream editing tool AWK (Aho, Kernighan, & Weinberger, 1988). Although AWK is less expressive than the framework we propose, in some cases AWK programs which simulate navigation expressions can be constructed. To carry out the comparison we created AWK programs equivalent to navigation expressions used in `exp1` through `exp4` over query logs. Of course, when allowed to match just a single pattern line by line, AWK performs better than our implementation; however, when asked to produce the same set of annotations as our experiments over the files being treated as a single line, the majority of AWK programs took more than a minute, or ran out of memory, while processing the DBPedia and British Museum query logs. In comparison, our implementation computed the answers in less than 14 seconds (Figure 7.7). Complete programs and running times can be found at *Annotating CSV documents: An Online Appendix* (2015). We can conclude that when annotations spanning multiple lines, or with noisy files missing many new line symbols, AWK techniques might not be the best choice for producing annotations.

8. CONCLUSION

Throughout this work, we have seen many of the different properties of variable regexes, variable automata, and some of their variants. To finalize, we will discuss some of the possible avenues for future research and summarize our key contributions.

8.1. Future work

Implementation. Even though we have shown that many problems related to these IE languages are tractable, some of the algorithms we have provided would be inadequate for real-world implementations. Therefore, there is still a need for practical algorithms with finer complexity guarantees and a concern for possible implementation constraints and issues.

Full Datalog rules. In this work, we studied the expressiveness and complexity of span regex in conjunction with a very constrained form of Datalog. The next step, thus, would be to study how do bounds change when we consider additional features from Datalog, such as recursion. On a similar note, one could study the usability of this approach and how would it relate to an extension of context-free grammars with variables.

Computation models. Variable automata seem like a very natural extension of finite automata. In this regard, there is a vast amount of computation models that could be similarly extended with variable operations (e.g., Pushdown Automata). It would therefore be interesting how would this different models would compare to variable automata in terms of expressiveness and complexity.

8.2. Final remarks

In this thesis we showed how previous frameworks for information extraction can be extended in order to incorporate incomplete or missing information, a feature often needed when processing data from noisy sources, such as the Web. We did so by redefining the

semantics of regex formulas introduced by Fagin et al. (2015), allowing them to output mappings instead of relations. This addition also permitted us to subsume other approaches to IE, thus creating a general framework for studying expressiveness and complexity of information extraction languages.

From our analysis it follows that several variants of expressions proposed by Fagin et al. (2015) and Arenas et al. (2016) are in fact equivalent, and that obtaining an efficient algorithm for enumerating all of their outputs is generally not possible. To overcome the latter, we isolated a class of sequential regex formulas, which extend the functionality constraint proposed by Fagin et al. (2015), and show that these can be efficiently evaluated both in isolation, and when combined into tree-like rules of Arenas et al. (2016). Finally, we showed that the good properties of sequential tree-like rules are also preserved when considering main static tasks, such as satisfiability, non-emptiness, and containment. This suggests that these kind of expressions have the potential to serve as a theoretical base of information extraction languages.

REFERENCES

Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of databases*. Addison-Wesley.

Aho, A. V., & Corasick, M. J. (1975). Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, *18*(6), 333–340.

Aho, A. V., Kernighan, B. W., & Weinberger, P. J. (1988). *The awk programming language*. Addison-Wesley.

Annotating CSV documents: An Online Appendix. (2015). `http://dvrgoc.ing.puc.cl/CSV`.

Arenas, M., Maturana, F., Riveros, C., & Vrgoč, D. (2016, July). A framework for annotating CSV-like data. *Proceedings of the VLDB Endowment*, *9*(11), 876–887.

Califf, M. E., & Mooney, R. J. (1999). Relational learning of pattern-match rules for information extraction. In *Proceedings of the sixteenth national conference on artificial intelligence and eleventh conference on innovative applications of artificial intelligence* (pp. 328–334). Orlando, Florida, USA.

Chiticariu, L., Krishnamurthy, R., Li, Y., Raghavan, S., Reiss, F., & Vaithyanathan, S. (2010). SystemT: An algebraic approach to declarative information extraction. In *Proceedings of the 48th annual meeting of the association for computational linguistics* (pp. 128–137). Uppsala, Sweden.

Chiticariu, L., Li, Y., & Reiss, F. R. (2013, October). Rule-based information extraction is dead! long live rule-based information extraction systems! In *Proceedings of the 2013 conference on empirical methods in natural language processing* (pp. 827–832). Seattle, Washington, USA: Association for Computational Linguistics.

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

DBpedia team. (2015). *DBpedia.* `http://dbpedia.org`.

Fagin, R., Kimelfeld, B., Reiss, F., & Vansummeren, S. (2014). Cleaning inconsistencies in information extraction via prioritized repairs. In *PODS* (pp. 164–175).

Fagin, R., Kimelfeld, B., Reiss, F., & Vansummeren, S. (2015). Document spanners: A formal approach to information extraction. *Journal of the ACM*, *62*(2).

Flum, J., & Grohe, M. (2006). *Parameterized complexity theory*. Springer.

Freydenberger, D. D., & Holldack, M. (2016). Document spanners: From expressive power to decision problems. In *LIPIcs-Leibniz international proceedings in informatics* (Vol. 48).

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-Completeness*. W. H. Freeman.

Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to automata theory, languages and computation*. Addison-Wesley.

Immerman, N. (1988). Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, *17*(5), 935-938.

Johnson, D. S., Yannakakis, M., & Papadimitriou, C. H. (1988). On generating all maximal independent sets. *Information Processing Letters*, *27*(3), 119–123.

Kimelfeld, B. (2014). Database principles in information extraction. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems* (pp. 156–163). Snowbird, Utah, USA.

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

Lenzerini, M. (2002). Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems* (pp. 233–246). Madison, Wisconsin, USA.

Martens, W., Neven, F., & Schwentick, T. (2009). Complexity of decision problems for XML schemas and chain regular expressions. *SIAM Journal on Computing*, *39*(4), 1486–1530.

Office for National Statistics. (2015). *ONS Data Explorer*. `http://www.ons.gov.uk/ons/data/web/explorer`.

Papadimitriou, C. H. (1993). *Computational complexity*. Addison-Wesley.

Pérez, J., Arenas, M., & Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, *34*(3).

Segoufin, L. (2014). A glimpse on constant delay enumeration. In *31st international symposium on theoretical aspects of computer science (stacs 2014), stacs 2014, march 5-8, 2014, lyon, france* (pp. 13–27).

Shen, W., Doan, A., Naughton, J. F., & Ramakrishnan, R. (2007). Declarative information extraction using datalog with embedded extraction predicates. In *VLDB* (pp. 1033–1044).

Soderland, S. (1999). Learning information extraction rules for semi-structured and free text. *Machine Learning*, *34*(1-3), 233–272.

Stockmeyer, L. J., & Meyer, A. R. (1973). Word problems requiring exponential time (preliminary report). In *Proceedings of the fifth annual ACM symposium on theory of computing* (pp. 1–9).

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, *1*(2), 146-160.

The British Museum. (2015). *British Museum public endpoint.* `http://bm.rkbexplorer.com`.

The City of Palo Alto, California Urban Forest Section. (2015). *Palo Alto tree data.* `http://w3c.github.io/csvw/use-cases-and-requirements/Palo_Alto_Trees.csv`.

The LSQ team. (2015). *The Linked SPARQL Queries Dataset.* `http://aksw.github.io/LSQ/`.

W3C. (2014, December). *CSV on the Web: Use Cases and Requirements, Editor's Draft.* `http://w3c.github.io/csvw/use-cases-and-requirements/`.

W3C. (2015). *Model for Tabular Data and Metadata on the Web.* `http://www.w3.org/TR/tabular-data-model/`.

World Bank. (2015). *Data Records.* `http://data.worldbank.org/`.