

A framework for annotating CSV-like data: online appendix

Marcelo Arenas
PUC Chile
marenas@ing.puc.cl

Cristian Riveros
PUC Chile
cristian.riveros@uc.cl

Francisco Maturana
PUC Chile
fjmaturana@uc.cl

Domagoj Vrgoč
PUC Chile
dvrgoc@ing.puc.cl

ABSTRACT

In this paper, we propose a simple and expressive framework for adding metadata to CSV documents and their noisy variants. The framework is based on annotating parts of the document that can be later used to read, query, or exchange the data. The core of our framework is a language based on extended regular expressions that are used for selecting data. These expressions are then combined using a set of rules in order to annotate the data. We study the computational complexity of implementing our framework and present an efficient evaluation algorithm that runs in time proportional to its output and linear in its input. As a proof of concept, we test an implementation of our framework against a large number of real world datasets and show that it can be efficiently used in practice.

1. INTRODUCTION

Comma-separated values (CSV) are a simple format used to store and exchange data. In their essence, CSV documents are nothing more than data divided by separators that naturally structure the file content into cells, columns, lines, etc. An example of one such document is given in Figure 1. Although traditionally used to send data between organizations, these days CSV files also form a large percentage of data published on the Web under the “open data” initiative whose aim is to make the government data widely available. For example, the Office for National Statistics (ONS), the largest independent body that produces official statistics in the United Kingdom, makes its data available in CSV format [17], as do the Land Registry UK [13] and World Bank [30].

John	,	47	,	15/05/1968	,	3700
Arthur	,	31	,	31/07/1983	,	3005
Eliot	,	30	,	21/08/1985	,	3600

Figure 1: An example of a CSV file.

CSV files are very intuitive and thus easily understood when presented to a human user. However, because of this simplicity CSV data lacks expressive power and at this point there is no standardised way for describing the content of a CSV file, like specifying the structure of the data, the data types used in different columns, or the way it should be displayed. In order to address this issue, the World Wide Web Consortium (W3C), has formed a CSV on the Web working group [28], whose main goal is to develop sound infrastructure for CSV data on the Web. Building this infrastructure is based on allowing CSV data to be enriched by versatile and easily usable metadata. The basic idea of metadata is to allow users to annotate their data with comments, additional meanings, or explanations that can be managed by users or programs processing the data. For example, specifying the meaning of some code that is often repeated in a file is an instance of metadata, as is providing the

default value for an error code or a missing piece of information.

When trying to develop a metadata language for CSV one option is to concentrate on well-structured documents that have the same number of columns in each row, no unexpected line breaks, and always use the same delimiter (comma in most cases). In fact, the current recommendation by the W3C [27] treats CSV files as tables, thus allowing for a concise metadata syntax. However, CSV data can often be noisy, for instance when it is generated as a result of publishing a legacy database in CSV format, which can result in a skewed table with missing values, rows of different length, or non-standard data types. Similar issues occur when using CSV to store data coming from different sources, or when merging two CSV tables. In this case, what is the essence kept in every CSV file? Notice that the main feature of CSV data, which is preserved even in noisy files, is the fact that they use predefined landmarks to separate different data entries: rows are separated using the new line symbol, cells within each row are separated by commas, etc. Moreover, this idea is not exploit only by CSV files but by any sort of format that has predefined landmarks separating the stored data. This includes system and query logs [25], citation data [20], geometric vector files [18], chess game data [9] and many other formats. The main objective of this paper is to develop a metadata language for annotating this more general sort of CSV documents, that we call *CSV-like* documents, which do not necessarily conform to the tabular format prescribed by the W3C [27], but use the mechanism of landmarks to lazily structure the data.

In order to design such a language we must overcome several issues. First, we want the language to be applicable in practical use case scenarios, such as the ones proposed in [29]. Second, notice that our language is meant to provide metadata by annotating different elements of CSV-like files such as cells, rows, columns, or even entire documents. The question is then how can this be done in a uniform way? And lastly, we also need to make our language efficiently implementable in order for it to be applicable in practice. To overcome these issues we deploy several ideas from text processing [3] and information extraction [10, 19] that allow us to handle different elements of potentially noisy CSV files in a uniform and efficient way. Our proposal is centred around the notion of a *span*, first introduced in [19]. Similarly to IBM’s SystemT [19], we view documents as strings. A span is then simply an interval inside this string (i.e. it is defined by the starting point and the ending point of some substring). It is now easy to see that different elements of a CSV-like document, such as cells, rows, or columns, are nothing more than a collection of spans. In order to extract spans from a file we extend the work of [3, 10] to define a class of regular expressions enriched with variables called *span regular expressions*. The idea is to use regular expressions to position ourselves inside a document and then store spans into variables. To attach metadata to documents we allow combining such expressions using a

datalog-like rule language into *annotation programs* that define the annotation attached to spans selected by the program. As we show, this results in an expressive language that can cover all the requirements put forward by the CSV on the Web working group [29] that specify meta data for the file, plus all their validation requirements. Since our language was designed for adding meta data, note that we cannot cover the transformation requirements from [29].

Although the base language we propose serves as a nice theoretical umbrella for the problem of extracting and annotating data, this framework does not internalize the landmark structure of CSV-like documents. This implies that annotation programs are too complicated for working with CSV-like documents, and their high expressive power also implies hard evaluation algorithms, thus hampering the language's usefulness in practice. To overcome these limitations, we further simplify the data extraction process by proposing the so called *navigation expressions* which allow simple navigation through documents by using item separators (e.g. commas, new line symbols) as landmarks. For instance, with the expression `next (,)` we specify that we want to move from the current position to the next comma in the file. Similarly, if we want to capture this part of the document into a variable we say `x:next (,)`, thus storing the resulting span into the variable `x`. The main contribution of this paper is then the language of *navigation programs* which is suitably designed to simplify the extraction process of annotation programs by using navigation expressions instead of span regular expressions. Navigation programs offer an intuitive syntax that is easy to use over CSV-like documents and, at the same time, still allows us to model all use cases put forward by the CSV on the Web working group [29].

In order to determine if our proposal has the potential to be used in practice we study the computational complexity of evaluating expressions and programs over documents. In general, all of these problems are NP-complete. However, if we restrict the structure of navigation expressions and require our programs to be *tree-shaped*, we end up with an expressive subfragment that can be evaluated in polynomial time. The main technical contribution of this paper is then an efficient algorithm for evaluating navigation expressions, thus allowing them to extract spans from CSV-like documents. The algorithm is based on an indexing technique that allows us to use string matching algorithms [3] in order to process our expressions in time that is both linear in the size of the input document and the size of its output (the number of captured spans). The algorithm we develop is also a constant-delay algorithm [21], which means it takes polynomial time to preprocess the input and constant time between producing consecutive outputs.

Finally, as a proof of concept, we also show how an implementation of our framework works in practice by designing a set of experiments that is based on the requirements from the W3C CSV on the Web working group [29] and running it against a wide range of real world CSV datasets. We also test the implementation in the context of annotating query logs and for this we use the data from public query endpoints of the British Museum [23] and DBpedia [8], two large providers of Semantic Web data. Based on this evaluation we conclude that our proposal results in a lightweight framework for annotating CSV-like data that can be efficiently implemented to run on an average laptop or a desktop machine, but is powerful enough to handle files one is likely to encounter in practice.

Related work. We categorise related work as follows.

CSV metadata languages: First, we would like to clarify the connection of our approach with the tentative proposal for annotating CSV files put forward by the W3C's CSV on the Web working group [28]. The main difference with our work is that the working group's proposal assumes the data to conform to a strict format

where all rows have the same number of columns, the user comments appear only in the first row, etc. As discussed previously, one of the main objectives of the language we propose is the ability to handle noisy data that does not necessarily conform to such strict specifications. Furthermore, the metadata proposed by the working group comes from a fixed vocabulary and does not allow arbitrary user defined annotations, as our approach does. Overall, our objective was to design a language that has a high level view of adding the metadata and can also be used to annotate more general files than CSV, such as e.g. query logs, scientific data, etc.

Second, as far as we are aware, apart from [28], the only other body of work addressing the issue of dealing with metadata in CSV files is the paper by Martens et. al [15]. However, note that their proposal addresses only the problem of specifying schemas for CSV data and does not deal with general metadata concepts that our approach covers, such as specifying arbitrary annotations. Furthermore, the approach of [15] assumes that the data is always tabular and is not designed to deal with noisy CSV files. Lastly, the biggest difference between the two approaches is the fact that [15] focuses on providing a good conceptual framework for specifying CSV schemas, but does not deal with implementation issues. On the other hand, the main contribution of our work is a linear time algorithm for extracting data and annotating documents.

Spanners, AQL and SystemT: Annotation query language (AQL), which forms the basis for IBM's SystemT [19], together with its theoretical counterpart named spanners [10], had the biggest influence of the work we present here. In particular, we adopt their idea of viewing a document as a single string, and of using regular expressions with variables to extract data. The main difference between the two approaches is in the intended application domain and in the expressive power. Regarding the application domain, while SystemT is designed to deal with arbitrary text documents, the languages developed in this paper are optimized to work on files that have natural landmarks separating the data. This is also seen in the way that regular expressions are used. More precisely, while SystemT and AQL use regular expressions for extraction, we view them as a way of navigating between landmarks separating the data, and then extracting the portion of the file between two landmarks. In terms of expressive power, it is straightforward to see that the presented languages are incomparable (for instance, we allow full recursive rules which are not available in AQL, while overlap consolidation operators of SystemT are not expressible in our framework). However, since there are enough fragments considered both in [10] and our work to warrant a paper on its own, we leave such comparisons for future work. Finally, as SystemT is a commercial system, due to legality reasons we can not do a direct runtime comparison with our implementation, and furthermore, such a comparison would not be entirely fair as the two systems have different application domains and different expressive power.

Other information extraction tools: There is an extensive body of academic writing and a number of tools for extracting information from text (for a good survey see [14]). Here we will briefly discuss some of the works that influenced our design decisions when defining the languages presented here. First, there is a variety of tools for matching regular expressions, or their extensions with variables [3], to a given string, such as AWK [5] or grep [26]. The whole framework we propose properly subsumes all of these tools (as it allows the use of datalog-like rules), however, the classes of extraction expressions we use when defining our programs can be seen as variants of regular expressions with back referencing [3]. As our semantics is based on the notion of span, we do end with a class of expressions whose expressive power is slightly weaker than that of regular expressions with back referencing. For instance, the lan-

guage of regular expression with back referencing $(x&zx)^*$, defining all words of the form $w_1w_1 \dots w_kw_k$, for some k , is not expressible by the classes of expressions we propose, but once we allow the use of rules we can easily define this language. On the other hand, sacrificing the expressive power allows us to define the semantics of our expressions in a much simpler way than it is done for regular expressions with back referencing [3]. Next, we would like to note that combining atomic extraction processes using rule-based languages is also a common practice in the area of databases and was already studied for less expressive classes of regular expressions in [22, 7, 11]. Similarly as in comparison with SystemT, our approach differs from these because it uses regular expressions for navigating the document, and because it is aimed to be used with a very particular type of files and not on arbitrary text.

Relational tools: Since one of the most natural use cases for our framework is adding metadata to CSV files, we would like to discuss the connection of our work with annotation systems for relational databases. After all, CSV tables can be viewed as relational data, so why not use the existing tools? One reason is that annotation systems for relational data are often row-based [12, 6], and thus do not support all the functionalities required from CSV metadata [29]. Another, perhaps more important reason, is that CSV was designed as a lightweight format that is widely available and independent of internal data storage mechanisms, while annotation tools for relational data [16, 6] rely heavily on the relational model and its indexing capabilities. Lastly, the approach we propose can naturally handle data that is not necessarily tabular (such as CSV files where rows have a different number of columns), which can cause problems for relational systems as table based representation of such data is not straightforward.

Organisation. In Section 2 we give a high level overview of our framework. In Section 3 we formalize the notions that will be used throughout the paper. Next, in Section 4 we analyse the complexity of evaluating our framework. In Section 5 we discuss how the framework can be tuned to cover CSV use cases from [29]. A practical algorithm for navigation expressions is presented in Section 6 and experimental evaluation of the system based on this algorithm is presented in Section 7. We conclude in Section 8. Due to the lack of space proofs of theorems, programs used in experiments and the source code of our implementation are available in our online appendix [1].

2. ANNOTATIONS IN ACTION

To develop an annotation language for CSV-like documents, we propose a framework where documents in general are modelled as strings. In this way, the process of extracting and annotating elements from a document can be simply viewed as a sequence of string operations. In what follows, we introduce the main notions used in our framework, which are then formalised in Section 3.

As a running example, assume that we are given the following document d_0 in the RIS format [20]:

```

TY - JOUR
AU - Simon
AU - Apt

```

This document stores information about a publication, indicating that the type (*TY*) of this publication is journal (*JOUR*), and that the authors (*AU*) of this publication are *Simon* and *Apt*. It is important to notice that every row in the RIS format starts with two letters followed by two spaces, a dash and a space. As mentioned before, document d_0 is viewed as a string in our framework:

$$d_0 = \vdash TY _ _ _ JOUR _ _ _ AU _ _ _ Simon _ _ _ AU _ _ _ Apt _ _ _ _$$

In this case, the string d_0 is defined over an alphabet consisting of the letters a, \dots, z, A, \dots, Z , the symbol dash ($-$), and the special symbols $_$, \leftarrow , \vdash and \dashv , which are used to indicate a space, the end of a line, the start of a document and the end of a document.

In our framework, the process of annotating some elements of a document d is reduced to the process of annotating some continuous regions of the string representation of d , which are called *spans* [10] in our setting. More precisely, a span p of d is a pair of the form (i, j) such that $1 \leq i \leq j \leq |d| + 1$, where $|d|$ is the length of the string d . Thus, p represents a continuous region of the document d , whose content is the infix of d between positions i and $j - 1$. For instance, the following figure includes d_0 and the index of each position in this string:

$$\vdash TY _ _ _ JOUR _ _ _ AU _ _ _ _ Simon _ _ _ _ AU _ _ _ _ Apt _ _ _ _ \dashv$$

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34

In this case, we have that the span $(2, 7)$ has as content the infix of d_0 between positions 2 and 6, that is, the string “*TY -*”, while $(4, 4)$ is assumed to have as content the empty string ε (in general, the content of a span (i, i) is assumed to be ε). Moreover, we have that the span $(1, 35)$ corresponds to the entire document, as the infix of d_0 between positions 1 and 34 is d_0 itself.

The first key ingredient of our framework is the definition of expressions for extracting spans from a document. More precisely, we introduce in this paper the notion of a *span regular expression* (SRE), an extension of classical regular expressions that are designed for defining and extracting spans rather than strings. In a nutshell, SREs use the union (i.e. $+$), the concatenation (i.e. $/$) and the Kleene star of regular expressions to check patterns inside a document. For extracting spans, SREs are enhanced with variables (e.g. x, y, z) which store spans. For instance, suppose that in our running example we want to extract and annotate “*all spans in the first column of a RIS document d* ”, which have the information about the tags for the publications stored in d . This means that we want all spans that: (i) start after a symbol \leftarrow or \vdash , and end before a symbol $_$; and (ii) do not contain a symbol $_$ inside. We can define all spans that satisfy condition (i) with the SRE:

$$R_0 = (\leftarrow + \vdash) / \Sigma^* / _$$

where Σ is the set of all symbols in the document d_0 . Intuitively, R_0 is defining all spans of the form “ $\leftarrow w _$ ” or “ $\vdash w _$ ”, where w is a substring of zero or more symbols in Σ . Since we are interested in capturing spans between two separators, we can refine R_0 and replace Σ^* with a variable x as follows:

$$R_1 = (\leftarrow + \vdash) / x / _$$

Thus, R_1 defines all spans that satisfy condition (i), and it stores in x the span between the symbols \leftarrow or \vdash and the symbol $_$.

To define condition (ii) in the previous example, we need to introduce another ingredient to our framework. More precisely, to evaluate a SRE over a specific span (which can be an entire document or just a fragment of it), we use the so-called *extraction expressions*. For example, define SRE R_2 as follows:

$$R_2 = \Delta^* / R_1 / \Delta^*$$

where $\Delta = \Sigma \cup \{\vdash, \dashv\}$. Then $\text{doc}.R_2$, where doc is a reserved word, is an extraction expression that indicates that R_2 has to be evaluated over the span corresponding to an entire document. For instance, if $\text{doc}.R_2$ is evaluated over d_0 , then we are indicating that R_2 has to be evaluated over the span $(1, 35)$ whose content is d_0 . In particular, this expression indicates that d_0 has to match R_2 , that is, d_0 should be of the form “ $u \leftarrow w _ v$ ” or “ $u \vdash w _ v$ ” where

u and v are substrings of zero or more symbols in Δ , and it also indicates that the span whose content is w should be stored in the variable x (recall that the variable x occurs in R_1).

To indicate that a SRE R has to be evaluated over a span corresponding to a fragment of a document, we use an extraction expression of the form $z.R$ where z is a variable, which specifies that R has to be evaluated over the span stored in z . For example, assume that the extraction expression $\text{doc}.R_2$ has been evaluated over a RIS document d . Then to check that the span stored in the variable x satisfies some condition, we can use an extraction expression of the form $x.R$. More specifically, given that each span stored in x satisfies condition (i) in our running example, we can use another extraction expression to filter x with the condition (ii). In particular, we use the extraction expression $x.F$, with $F = (\Sigma - \{_ \})^*$, to say that x contains zero or more symbols excluding $_$.

The second key ingredient of our framework is a language for annotating spans, which uses rules to combine the results of several extraction expressions and annotations. For instance, in our running example we can use the following rule to extract from a RIS document d every span that corresponds to a row in this document:

$$\text{doc}.\Delta^*/(_ \vdash _)/x/(_ \vdash _)/\Delta^* \wedge x.G \rightarrow \text{Row}(x), \quad (*)$$

where $G = (\Sigma - \{_ \})^*$. That is, a row in d is a string w that appears right after a symbol $_ \vdash$ or $_ \dashv$, does not mention any symbol $_ \vdash$ and appears right before a symbol $_ \vdash$ or $_ \dashv$. Besides, we can reuse Row in other rules to define more specific annotations. For example, we know that if a row starts with the keyword AU , then it contains the name of an author of a publication. Thus, the following rule is used to extract the names of the authors in a RIS document:

$$\text{Row}(x) \wedge x.AU_ _ _ /y \rightarrow \text{Author}(y).$$

The last key ingredient of our framework is a “landmark-oriented” language for extracting spans, called the *navigation language*. Although SREs are a nice theoretical umbrella for extracting spans, they are designed for general span extraction over any kind of document, thus not exploiting the natural structure of CSV-like documents. Namely, as CSV-like documents use separators to organize the data into fragments and subfragments, one would naturally use these separators as anchors to navigate the document and extract the desired data. To this purpose, we base the navigation language on two axis $\text{any}(S)$ and $\text{next}(S)$, which naturally suggest to navigate the document until *any* separator or to the *next* separator in the set S , respectively. For instance, in order to extract every span that corresponds to a row in a document, we can use the following navigation rule:

$$\text{doc}.\text{any}(_ \vdash _)/x : \text{next}(_ \vdash _ \dashv) \rightarrow \text{Row}(x) \quad (**)$$

When this rule is evaluated over a RIS document d , it says that one has to move until “any” place in d that has a symbol $_ \vdash$ or $_ \dashv$, and then move to the “next” symbol $_ \vdash$ or $_ \dashv$, while capturing the span into the variable x . Notice that this rule extracts and annotates the same spans as the rule (*), but it is much easier to define and understand.

As a final example, we consider again the task of extracting the names of the authors in a RIS document, which was carried out before by combining two rules. In what follows we provide a simple navigation rule that takes advantage of the separators used in RIS documents to carry out this task:

$$\text{doc}.\text{any}(AU_ _ _)/x : \text{next}(_ \vdash _ \dashv) \rightarrow \text{Author}(x).$$

In this case, the rule indicates that we have to move until “any” place in a RIS document that has the separator $AU_ _ _$, and then

move to the “next” symbol $_ \vdash$ or $_ \dashv$, while capturing the span into the variable x .

3. ANNOTATING CSV-LIKE DATA

In this section, we formalise the different notions of extraction expressions and annotation programs that were discussed in the previous section, and which are studied in this paper.

From now on, assume that Σ is a finite alphabet that is used in documents and does not contain the reserved symbols $_ \vdash$ and $_ \dashv$. Then a document d is just a string in the language $_ \vdash \Sigma^* _ \dashv$, that is, d is a string of the form $_ \vdash w _ \dashv$ with $w \in \Sigma^*$. Notice that we can treat a document just as a string, so, for example, we define $|d|$ as the length of the string d . However, we will continue using the term document to emphasize the fact it is a string with starting symbol $_ \vdash$ and ending symbol $_ \dashv$.

As discussed in the previous section, a fundamental notion associated to a document d is the concept of a span, which corresponds to a continuous region in d . Formally, we define $\text{span}(d)$ as the set $\{(i, j) \mid i, j \in \{1, \dots, |d| + 1\} \text{ and } i \leq j\}$, where each element $(i, j) \in \text{span}(d)$ is called a span of d . Every span $p = (i, j)$ of d has an associated content, which is denoted by $d(p)$ or $d(i, j)$, and it is defined as the infix of d from position i to position $j - 1$ (notice that if $i = j$, then $d(p) = d(i, j) = \varepsilon$).

3.1 Extraction expressions

As pointed out in Section 2, an extraction expression defines a way to retrieve spans from a document. The most general form of such statements considered in this article will be defined by using the notion of span regular expression. To define such notion, we first need to assume that \mathcal{V} is a set of variables that is disjoint with Σ . Then a *span regular expression* (SRE) is defined by the following grammar:

$$R ::= w, w \in (\Sigma \cup \{_ \vdash, _ \dashv\})^* \mid x, x \in \mathcal{V} \mid \\ (R/R) \mid (R + R) \mid (R)^* \mid \langle R \rangle$$

We have already shown in Section 2 some examples of SREs, and how they are used to define a set of spans from a document. Besides, we have also briefly described in Section 2 how variables are used in SREs to store spans. To define the semantics of SREs, we consider variables in SREs as free variables and, thus, we need to consider a variable assignment when evaluating a SRE over a document d . More precisely, a variable assignment σ over d is a function from \mathcal{V} to $\text{span}(d)$, that is, a function that assigns a span to every variable in \mathcal{V} . Then the semantics of SRE R , with respect to a document d and a variable assignment σ over d , denoted by $\llbracket R \rrbracket_{d, \sigma}$, is defined recursively as follows:

$$\begin{aligned} \llbracket w \rrbracket_{d, \sigma} &= \{p \mid p \in \text{span}(d) \text{ and } d(p) = w\} \\ \llbracket x \rrbracket_{d, \sigma} &= \{\sigma(x)\} \\ \llbracket R_1/R_2 \rrbracket_{d, \sigma} &= \{(i, j) \mid (i, j) \in \text{span}(d) \text{ and } \exists k : \\ &\quad (i, k) \in \llbracket R_1 \rrbracket_{d, \sigma} \text{ and } (k, j) \in \llbracket R_2 \rrbracket_{d, \sigma}\} \\ \llbracket R_1 + R_2 \rrbracket_{d, \sigma} &= \llbracket R_1 \rrbracket_{d, \sigma} \cup \llbracket R_2 \rrbracket_{d, \sigma} \\ \llbracket R^* \rrbracket_{d, \sigma} &= \llbracket \varepsilon \rrbracket_{d, \sigma} \cup \llbracket R \rrbracket_{d, \sigma} \cup \llbracket R^2 \rrbracket_{d, \sigma} \cup \llbracket R^3 \rrbracket_{d, \sigma} \cup \dots \\ \llbracket \langle R \rangle \rrbracket_{d, \sigma} &= \{p \in \text{span}(d) \mid \exists p' \in \llbracket R \rrbracket_{d, \sigma} : d(p) = d(p')\}, \end{aligned}$$

where R^2 is a shorthand for R/R , similarly R^3 for $R/R/R$, etc.

EXAMPLE 3.1. Consider the following document d storing information about temperatures:

$_ \vdash 15.2, _ \dashv 20.3 _ \vdash 14.3, _ \dashv 14.3 _ \vdash 14.2, _ \dashv 18.9 _ \dashv$
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34

Notice that below the string d we have included the index of each position in it. Now consider $R_3 = (15.2 + 14.3)$ that uses the union operator $+$ to define the set of spans whose content is either 15.2 or 14.3 . Then given an arbitrary variable assignment σ , we have that $\llbracket R_3 \rrbracket_{d,\sigma} = \{(2, 6), (13, 17), (19, 23)\}$ since $d(2, 6) = 15.2$, $d(13, 17) = 14.3$, $d(19, 23) = 14.3$, and for every other $p \in \text{span}(d)$, it holds that $d(p) \neq 15.2$ and $d(p) \neq 14.3$. Notice that R_3 does not include any variable, so σ does not play any role when evaluating R_3 over d .

Several variables can be included in a SRE, such as in the case of $R_4 = \swarrow/x/\swarrow/y/\swarrow$. To evaluate R_4 over d , we need to consider the values assigned to both x and y . For instance, assume that σ_2 is a variable assignment such that $\sigma_2(x) = (13, 17)$ and $\sigma_2(y) = (19, 23)$. It is then easy to check that $(12, 24) \in \llbracket R_4 \rrbracket_{d,\sigma_2}$.

If we replace R_4 by a SRE $R_5 = \swarrow/x/\swarrow/x/\swarrow$, then it may look like we are defining the set of spans $p \in \text{span}(d)$ such that the content associated to p corresponds to a row having the same temperature value in both columns. But in this case we have that $\llbracket R_5 \rrbracket_{d,\sigma}$ is empty for every variable assignment σ . The reason for this is that two spans (i, j) and (k, ℓ) are considered to be equal if and only if $i = k$ and $j = \ell$. To overcome this limitation, span regular expressions contain the operator $\langle \cdot \rangle$, which allows to compare the contents associated to two spans. Thus, if we consider the SRE $R_6 = \swarrow/x/\swarrow/\langle x \rangle/\swarrow$ instead of R_5 , then we will be effectively defining the set of spans corresponding to rows with the same temperature value in both columns. In fact, if σ_3 is a variable assignment such that $\sigma_3(x) = (13, 17)$, then we have that $(12, 24) \in \llbracket R_6 \rrbracket_{d,\sigma_3}$ as $(13, 17) \in \llbracket x \rrbracket_{d,\sigma_3}$, and $(19, 23) \in \llbracket \langle x \rangle \rrbracket_{d,\sigma_3}$ (given that $d(19, 23) = d(\sigma_3(x)) = d(13, 17) = 14.3$). \square

Notice that the semantics of SREs is defined with respect to a valuation σ . However, we want to use SREs in order to extract spans from a document and store them into some variables, thus defining valuations. This is achieved by using the notion of extraction expressions. More precisely, an *extraction expression* is an expression of the form either $\text{doc}.R$ or $x.R$, where R is a SRE, doc is a reserved word and x is a variable. Moreover, the evaluation of these expressions over a document d is defined as follows:

$$\begin{aligned} \llbracket \text{doc}.R \rrbracket_d &= \{ \sigma \mid \sigma \text{ is a variable assignment over } d \\ &\quad \text{such that } (1, |d| + 1) \in \llbracket R \rrbracket_{d,\sigma} \} \\ \llbracket x.R \rrbracket_d &= \{ \sigma \mid \sigma \text{ is a variable assignment over } d \\ &\quad \text{such that } \sigma(x) \in \llbracket R \rrbracket_{d,\sigma} \} \end{aligned}$$

Intuitively, an expression of the form $\text{doc}.R$ is used to extract valuations from a span $(1, |d| + 1)$ corresponding to an entire document d , therefore the use of the keyword doc , while an expression of the form $x.R$ is used to extract valuations from the spans assigned to the variable x . Note that the expression of the form $x \vdash R \dashv$ is equivalent to $\text{doc}.R$, however, as the latter is used more often we keep it in the language.

EXAMPLE 3.2. Let d be the document in Example 3.1, and R be SRE $\Delta^*/\swarrow/x/\swarrow/\langle x \rangle/\swarrow/\Delta^*$, where $\Delta = \Sigma \cup \{\vdash, \dashv\}$. This SRE can be used to extract spans from d whose content is a temperature that is repeated in a row. In fact, we have that $\llbracket \text{doc}.R \rrbracket_d$ is the set of all variable assignments σ such that $\sigma(x) = (13, 17)$. \square

3.2 General annotation programs

The goal of this section is to introduce a general form of programs for annotating documents. To this end, assume from now on that Γ is a finite alphabet of annotations that is disjoint with Σ and \mathcal{V} . Then an annotation program consists of a set of rules that indicate how to assign annotations from Γ to the spans in a document.

These rules are constructed by combining extraction expressions with formulas of the form $A(x)$, where $A \in \Gamma$ and $x \in \mathcal{V}$. More precisely, a *rule* is an expression of the form:

$$\varphi_1 \wedge \cdots \wedge \varphi_m \wedge A_1(x_1) \wedge \cdots \wedge A_n(x_n) \rightarrow A(x), \quad (\dagger)$$

where (a) $m \geq 0$ and $n \geq 0$; (b) φ_i ($1 \leq i \leq m$) is an extraction expression; (c) $A_j \in \Gamma$ and $x_j \in \mathcal{V}$ ($1 \leq j \leq n$); and (d) $A \in \Gamma$, $x \in \mathcal{V}$ and x occurs in the body of the rule. Moreover, an *annotation program* Π is a finite set of rules.

A ground annotation over a document d is an expression of the form $A(p)$, where $A \in \Gamma$ and $p \in \text{span}(d)$. Assume that Λ is a set of ground annotations over d . Then Λ is said to satisfy the rule (\dagger) if for every variable assignment σ over d such that $\sigma \in \llbracket \varphi_i \rrbracket_d$ for every $i \in \{1, \dots, m\}$, it holds that:

$$\text{if } A_i(\sigma(x_i)) \in \Lambda \text{ for all } i \in \{1, \dots, n\}, \text{ then also } A(\sigma(x)) \in \Lambda.$$

Moreover, Λ is said to satisfy an annotation program Π if Λ satisfies each rule in Π . The result of evaluating an annotation program over a document is a set of ground annotations. More precisely, given a document d and an annotation program Π , the evaluation of Π over d , denoted by $\text{EVAL}(\Pi, d)$, is defined as follows:

$$\begin{aligned} \{ A(p) \mid \text{for every set } \Lambda \text{ of ground annotations over } d \\ \text{such that } \Lambda \text{ satisfies } \Pi, \text{ it holds that } A(p) \in \Lambda \} \end{aligned}$$

That is, $\text{EVAL}(\Pi, d)$ is the set of all ground annotations $A(p)$ that hold in every possible model of Π according to d . It is important to notice that this semantics is based on the usual semantics for (recursive) Datalog programs [2, 7]. In fact, $\text{EVAL}(\Pi, d)$ is the result of taking the intersection of all possible sets Λ of ground annotations over d satisfying Π , which, given that the rules in Π do not include any negative atoms, is equivalent to computing the minimum such set Λ under set inclusion. This corresponds to the usual least fixed point semantics for Datalog programs without negation.

EXAMPLE 3.3. Assume that we want to indicate that if a document d starts with the string *temp*, then d stores information about temperatures. To do this, we use the following program to annotate d with the tag *TempDoc* if it satisfies the previous condition:

$$x \vdash /temp/\Sigma^*/\dashv \rightarrow \text{TempDoc}(x).$$

Notice that $\sigma \in \llbracket x \vdash /temp/\Sigma^*/\dashv \rrbracket_d$ if, and only if, the content associated to the span $\sigma(x)$ is a string of the form $\vdash tempw \dashv$ with $w \in \Sigma^*$, that is, if and only if $\sigma(x)$ corresponds to the document d , which starts with the string *temp*. Second, assume that in every document storing information about temperatures, we need to annotate every value with the tag *TempVal*. If $D = \{0, 1, \dots, 9\}$ and $N = (\Sigma \setminus D)$, then this is done by using an annotation program that combines the rule defining *TempDoc* with the following rule:

$$\begin{aligned} \text{TempDoc}(x) \wedge x \vdash / \Sigma^* / N / y / N / \Sigma^* / \dashv \wedge \\ y.R \rightarrow \text{TempVal}(y), \end{aligned}$$

In the body of the rule, the annotation $\text{TempDoc}(x)$ indicates that x stores a span whose content is a document storing information about temperatures, while the expression $x \vdash / \Sigma^* / N / y / N / \Sigma^* / \dashv$ is used to retrieve from x and store in y a span that could represent a temperature value. Notice that we do not want to extract a part of a temperature value, for example we do not want to extract 12 if the stored temperature is 12.3; thus, we check that the symbols right before and after y are not digits (they belong to N). Finally, assuming that $P = \{1, \dots, 9\}$, we have that R is the SRE $(0 + PD^*).D^*$. Hence, the expression $y.R$ is used to check that the content of y is a decimal number. \square

3.3 Annotation programs for CSV-like data

Annotation programs are designed to deal with general documents, relying on span regular expressions to extract spans from them. In this sense, annotation programs do not take advantage of the structure of CSV-like documents where separators play a key role. In fact, in many practical scenarios users only need a simple language that is oriented towards navigating CSV-like documents using these separators as landmarks. In these scenarios span regular expressions are, in general, too expressive, so we propose in this section a simple language that separates the navigational features needed in CSV-like documents from the use of regular 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:

$$\begin{aligned} \psi & ::= \psi/\psi \mid \text{any}(S) \mid \text{next}(S) \mid \\ & \quad x : \text{next}(S), x \in \mathcal{V} \mid \langle x \rangle : \text{next}(S), x \in \mathcal{V} \\ S & ::= w, w \in \Delta^+ \mid S + S \end{aligned}$$

where S is assumed to be prefix free, that is, every expression S is of the form $w_1 + \dots + w_n$, 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 $\text{any}(S)$ is used to move forward in a document reading any sequence of symbols ending with a word in S , while the axis $\text{next}(S)$ is used to move forward to the next occurrence of a word in S . Moreover, $x : \text{next}(S)$ and $\langle x \rangle : \text{next}(S)$ perform the same form of navigation as $\text{next}(S)$, 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 $\text{any}(S)$, $\text{next}(S)$, $x : \text{next}(S)$ and $\langle x \rangle : \text{next}(S)$ are useful to restrict the navigation between two or more separators, which is a very common operation on CSV-like 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 variable assignment σ over d , the base case $\llbracket S \rrbracket_{d,\sigma}$ and the recursive case $\llbracket \psi_1/\psi_2 \rrbracket_{d,\sigma}$ are defined as for the case of span regular expressions. Moreover, $\llbracket \text{any}(S) \rrbracket_{d,\sigma}$ is defined as $\llbracket (\Sigma \cup \{\vdash, \dashv\})^*/S \rrbracket_{d,\sigma}$. Finally, the evaluation of $\text{next}(S)$, $x : \text{next}(S)$ and $\langle x \rangle : \text{next}(S)$ are defined as follows assuming that $S = w_1 + \dots + w_n$ and r_S is the SRE $\Delta^*w_1\Delta^* + \dots + \Delta^*w_n\Delta^*$ with $\Delta = \Sigma \cup \{\vdash, \dashv\}$:

$$\begin{aligned} \llbracket \text{next}(S) \rrbracket_{d,\sigma} &= \{(i, j) \in \text{span}(d) \mid \\ & \quad \exists k \geq i : (k, j) \in \llbracket S \rrbracket_{d,\sigma} \text{ and } (i, j-1) \notin \llbracket r_S \rrbracket_{d,\sigma}\} \\ \llbracket x : \text{next}(S) \rrbracket_{d,\sigma} &= \{(i, j) \in \text{span}(d) \mid \exists k \geq i : \sigma(x) = (i, k), \\ & \quad (k, j) \in \llbracket S \rrbracket_{d,\sigma} \text{ and } (i, j-1) \notin \llbracket r_S \rrbracket_{d,\sigma}\} \\ \llbracket \langle x \rangle : \text{next}(S) \rrbracket_{d,\sigma} &= \{(i, j) \in \text{span}(d) \mid \exists k \geq i : (i, k) \in \\ & \quad \llbracket \langle x \rangle \rrbracket_{d,\sigma}, (k, j) \in \llbracket S \rrbracket_{d,\sigma} \text{ and } (i, j-1) \notin \llbracket r_S \rrbracket_{d,\sigma}\} \end{aligned}$$

Notice that a span p belongs to $\llbracket r_S \rrbracket_{d,\sigma}$ 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 $(i, j) \in \llbracket \text{next}(S) \rrbracket_{d,\sigma}$, 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 in Section 3.1, that is, if ψ is an NE, then $\text{doc}.\psi$ and $x.\psi$ are considered to be extraction expressions if $x \in \mathcal{V}$. However, the semantics of these formulae are defined in a slightly different way:

$$\begin{aligned} \llbracket \text{doc}.\psi \rrbracket_d &= \{\sigma \mid \sigma \text{ is a variable assignment over } d \text{ such that} \\ & \quad (1, k) \in \llbracket \psi \rrbracket_{d,\sigma} \text{ for some } k, 1 \leq k \leq |d| + 1\} \\ \llbracket x.\psi \rrbracket_d &= \{\sigma \mid \sigma \text{ is a variable assignment over } d \text{ such that} \\ & \quad \sigma(x) = (i, j) \text{ and } (i, k) \in \llbracket \psi \rrbracket_{d,\sigma} \text{ for some } k, i \leq k \leq j\} \end{aligned}$$

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, $\sigma \in \llbracket \psi \rrbracket_d$ 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 $k+1, \dots, |d|$.

We are ready to define our simplified notion of annotation program. A *navigation program* Ω is a finite set of rules of the form (\dagger) where (a) $m \geq 0$ and $n \geq 0$; (b) φ_i ($1 \leq i \leq m$) is an expression of the form either $\text{doc}.\psi$ or $x.\psi$ or $x.R$, where $x \in \mathcal{V}$, ψ is an NE and R is a SRE without variables; (c) $A_j \in \Gamma$ and $x_j \in \mathcal{V}$ ($1 \leq j \leq n$); and (d) $A \in \Gamma$, $x \in \mathcal{V}$ and x occurs in the body of the rule. The evaluation of Ω over a document d , denoted by $\text{Eval}(\Omega, d)$, is defined as for the case of annotation programs.

EXAMPLE 3.4. Assume that d is a document where values are separated by commas. Moreover, suppose that the elements of the first column of d should form a primary key, so that two different rows of d should have distinct values in their first columns. We can annotate with the tag `Error` any violation to this constraint:

$$\begin{aligned} \text{doc.any}(\vdash + \dashv)/x : \text{next}(,)/\text{any}(\dashv)/\langle x \rangle : \text{next}(,) \\ \rightarrow \text{Error}(x) \end{aligned}$$

Thus, we can check if the first column of d is a primary key by verifying whether any span has been annotated with the tag `Error`. \square

Recall that the navigation program $(**)$ defined in Section 2 is equivalent to annotation program $(*)$ from the same section. In fact, it can be proved that each navigation program can be transformed into an equivalent annotation program, thus showing that navigation programs are a simplification of annotation programs.

4. COMPLEXITY OF ANNOTATING CSV

The goal of this section is to study the complexity of evaluating an annotation program over a CSV-like document, for which we also have to study the evaluation problem for extraction expressions. This study allows us to identify the features of our framework that need more resources, and the fragments that can be evaluated efficiently.

We start by analysing the computational complexity of evaluating an extraction expression defined by either a span regular expression or a navigation expression. Given a document d and an extraction expression of the form $\text{doc}.\psi$, where $\{x_1, \dots, x_k\}$ is the set of variables occurring in ψ , we are interested in computing the set $\{(\sigma(x_1), \dots, \sigma(x_k)) \mid \sigma \in \llbracket \text{doc}.\psi \rrbracket_d\}$, that is, we are interested in enumerating all the variable assignments in $\llbracket \text{doc}.\psi \rrbracket_d$. To provide a lower bound for the complexity of this problem, and also to simplify the analysis, we consider the decision problem of verifying whether $\llbracket \text{doc}.\psi \rrbracket_d$ is not empty. Formally, if \mathcal{L} is either the class \mathcal{SRE} of span regular expressions or the class \mathcal{NE} of navigation expressions, then we consider the following problem:

Problem:	NONEMP(\mathcal{L})
Input:	A document d and an expression $\psi \in \mathcal{L}$.
Question:	Is $\llbracket \text{doc}.\psi \rrbracket_d$ not empty?

Unfortunately, we can show that the problem is intractable.

THEOREM 4.1. *NONEMP($\mathcal{SR}\mathcal{E}$) and NONEMP($\mathcal{N}\mathcal{E}$) are both NP-complete.*

The proof of Theorem 4.1 reveals a first source of intractability in NONEMP(\mathcal{L}): the overuse of variables and the content operator $\langle \cdot \rangle$ in extraction expressions. In practice, a restriction on the use of the content operator is reasonable, as checking whether two spans have the same content is not used very often (see [29]). The following theorem shows that under this assumption, the complexity decreases for the case of navigation expressions.

THEOREM 4.2. *If the content operator is not allowed, then NONEMP($\mathcal{SR}\mathcal{E}$) is NP-complete and NONEMP($\mathcal{N}\mathcal{E}$) is in PTIME.*

The membership of NONEMP($\mathcal{N}\mathcal{E}$) in PTIME opens a window to find an efficient algorithm for our initial enumeration problem for navigation expressions without the content operator. In fact, we provide such an algorithm in Section 6.

Now we move our attention to the evaluation of annotation programs. Similarly to the case of extraction expressions, we are interested in enumerating all the A -annotations in EVAL(Π, d) for an annotation program Π , a document d , and an annotation label $A \in \Gamma$. In terms of complexity analysis, we define the natural decision problem associated to this computation problem, assuming that \mathcal{P} is either the class \mathcal{AP} of annotation programs or the class $\mathcal{NV}\mathcal{P}$ of navigation programs:

Problem:	MEMB(\mathcal{P})
Input:	A document d , a program $\Pi \in \mathcal{P}$, and a ground annotation $A(p)$.
Question:	Is $A(p) \in \text{EVAL}(\Pi, d)$?

We know from Theorem 4.1 that MEMB(\mathcal{AP}) and MEMB($\mathcal{NV}\mathcal{P}$) are both intractable problems (i.e. NP-hard). The following result shows that these problems are indeed NP-complete.

THEOREM 4.3. *MEMB(\mathcal{AP}) and MEMB($\mathcal{NV}\mathcal{P}$) are both NP-complete.*

A consequence of Theorems 4.2 and 4.3 is that evaluating annotation programs without the content operator is still NP-complete. We do not know if the same happens for navigation programs without the content operator. However, we do know that evaluating navigation programs without the content operator and with a tree-like structure on each rule can be done efficiently. Formally, let $\theta = \ell_1.\psi_1 \wedge \dots \wedge \ell_m.\psi_m$ be a conjunction of extraction expressions, where each ψ_i is a navigation expression and $\ell_i \in \mathcal{V} \cup \{\text{doc}\}$ ($1 \leq i \leq m$), and assume that G_θ is a graph obtained from θ as follows. The set of nodes of G_θ is the set of variables occurring in θ , and there exists an edge (x, y) in G_θ if and only if $x = \ell_i$ and y occurs in ψ_i , for some $i \in \{1, \dots, m\}$. Then we say that θ is *tree-like* if (1) $\ell_i \neq \ell_j$ for every $i, j \in \{1, \dots, m\}$ such that $i \neq j$; and (2) G_θ is a rooted-forest (i.e. a set of rooted-trees). For example, the following conjunction of extraction expressions is tree-like:

$\text{doc}.\text{any}(\leftarrow)/x:\text{next}(\leftarrow) \wedge x.y:\text{next}(,)/z:\text{next}(,) \wedge z.0^*$

We say that a navigation program Ω is tree-like if for every rule in Ω of the form (\dagger) (see Section 3.2), it holds that $\varphi_1 \wedge \dots \wedge \varphi_m$ is tree-like.

THEOREM 4.4. *If we restrict to tree-like navigation programs not using the content operator, then MEMB($\mathcal{NV}\mathcal{P}$) is in PTIME.*

The tree-like restriction just defined is a reasonable assumption over navigation programs. In fact, all the navigation programs in this paper are tree-like, and all the use cases in [29] that can be annotated by annotation programs can also be annotated by tree-like navigation programs.

5. USABILITY OF THE FRAMEWORK

With a language for annotating CSV-like data and the complexity analysis of evaluating them at hand, we briefly discuss how usable is this framework for the task of specifying metadata of CSV-like files. For this, we consider the use cases put forward by the CSV on the Web working group [29] as a yardstick to check whether our framework is expressive enough to be used in practice. Note that the requirements in [29] also include other use cases that go beyond metadata specification (e.g. transforming CSV data into another format). Since our language was designed for adding metadata, we do not consider these use cases in our analysis and we just concentrate in the ones related to metadata over CSV.

The use cases presented in [29] are very diverse and are mostly designed for tabular CSV data like the file presented in Figure 1. By simple inspection, one can easily check that navigational programs can fulfill most of the requirements in [29], but not all of them. In particular, navigational programs cannot define *Foreign Key References* and *Association of Code Values With External Definitions* (see [29] for the full definition). To satisfy the missing requirements, one can extend our framework with two natural features: (1) managing multiple documents and (2) annotating with n -ary relations. For the former extension, one can easily add the use of multiple documents to our framework by allowing extraction expressions of the form $d.\psi$ where d is the name of a document. Note that this extension does not change the complexity analysis in Section 4; in particular, all the positive and negative results still hold for expressions and programs with multiple documents. For the latter extension, we can enrich our Datalog-like rules with n -ary predicates, and then define their semantics in the same ways as in Section 3. Instead of giving the formal definitions of both extensions, we illustrate them with an example.

EXAMPLE 5.1. *The requirement Foreign Key References asks to cross-reference data between different CSV files [29]. For example, suppose that we have two CSV documents, called d_1 and d_2 , and we want to say that a row in d_2 is related with a row in d_1 if they have the same value in the first cell. The following rule in the extended framework allows to define this requirement:*

$$d_1.\text{any}(\leftarrow)/x:\text{next}(\leftarrow) \wedge d_2.\text{any}(\leftarrow)/y:\text{next}(\leftarrow) \wedge x.z:\text{next}(,) \wedge y.\langle z \rangle:\text{next}(,) \rightarrow \text{FKKey}(x, y)$$

Although requirement (1) is harmless for the framework (we did not include it in the formal presentation for the sake of readability), adding n -ary relations will have a considerable impact on the complexity of the evaluation problem. In fact, it is well-known that evaluating Datalog programs with n -ary (intentional) predicates is EXPTIME-complete [2, 7] and, thus, all results in Section 4 will not hold in the extended framework.

It should be mentioned that navigational programs for annotating CSV-like data satisfy all the use-cases in [29] if multiple documents and n -ary relations are allowed. However, as mentioned above, extending the framework with n -ary predicates will have a significant impact in the evaluation. For this reason, we are convinced that navigational programs are a natural “sweet spot” between the ex-

pressiveness needed to deal with metadata over CSV-like files and the efficiency needed to deal with such files in practice.

6. EFFICIENT EVALUATION OF NAVIGATION EXPRESSIONS

The use of our framework requires an efficient algorithm for enumerating all the variable assignments satisfying a navigation expression. The complexity results provided in Section 4 show that such a procedure does not exist if the content operator is allowed (unless $\text{PTIME} = \text{NP}$). On the other hand, we provide in this section such a procedure for navigation expressions without the content operator. This algorithm 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 [21], namely, enumeration algorithms that take polynomial time in pre-processing the input (i.e. ψ and d), and constant time between two consecutive outputs (i.e. variable assignments).

6.1 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 $\varphi_0/\text{any}(S_1)/\varphi_1/\dots/\text{any}(S_k)/\varphi_k$, where $k \geq 0$, φ_0 is a sequence of `next`-operators and $\varphi_1, \dots, \varphi_k$ 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 variable assignment σ over d , it holds that $\llbracket \varphi \rrbracket_{d,\sigma} = \llbracket \psi \rrbracket_{d,\sigma}$. Then we consider the following rewriting rules to meet our goal:

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

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 6.1. Consider the following navigation expression:

$$\varphi = \text{any}(\text{;})/\text{next}(\text{;})/\text{any}(\leftarrow)/x:\text{next}(\leftarrow)/\text{any}(\text{;})$$

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:

$$\varphi_1 = \text{next}(\text{;})/\text{next}(\text{;})/\text{any}(\leftarrow)/x:\text{next}(\leftarrow)/\text{any}(\text{;})$$

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:

$$\varphi_2 = \text{next}(\text{;})/\text{next}(\text{;})/\text{any}(\leftarrow)/x:\text{next}(\leftarrow)/\text{next}(\text{;})$$

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 $O(|\varphi|)$.

Data: A document d and a prefix-free set of words S

Result: An array $A[1..|d| + 1]$ of $\text{span}(d)$

Function `separators_match`(d, S)

```

    aho_corasick.init( $d, S$ )
     $m := 1$ 
    while ( $s_1, s_2$ ) := aho_corasick.next() do
        for  $i = m$  to  $s_1$  do
             $A[i] := (s_1, s_2)$ 
             $m := s_1 + 1$ 
        return  $A$ 

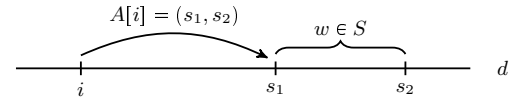
```

Figure 2: Finding all matches for a set of separators.

6.2 An efficient algorithm for evaluating NEs

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 variable assignments σ such that $\sigma \in \llbracket \text{doc}.\psi \rrbracket_d$. 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 `any`(S) or `next`(S), the set S is assumed to be prefix-free). The procedure then runs the Aho-Corasick algorithm [4] 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 $i \in \{1, \dots, |d|\}$. 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 2. To analyse the algorithm, observe that we repeatedly run the Aho-Corasick string matching procedure. The iterator m starts at the beginning 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 $O(|S| + |d|)$.

The next part of the algorithm, presented in Figure 3, deals with computing the possible valuations for a *context*, that is, a subformula of the expression that is of the form `any`(S)/ v_1 :`next`(S_1)/ \dots / v_n :`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 $\varphi_0/E_1/\dots/E_n$ 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 .

Data: A document d , a prefix-free set of words S , and a sequence $(v_1, S_1), \dots, (v_n, S_n)$ where each S_i is a prefix-free set of words and $v_i \in \mathcal{V} \cup \perp$.

Result: An array $C[1..|d| + 1]$ of triples (r_1, r_2, σ) , where $(r_1, r_2) \in \text{span}(d)$ and $\sigma : \mathcal{V} \rightarrow \text{span}(d)$ is a partial function.

```

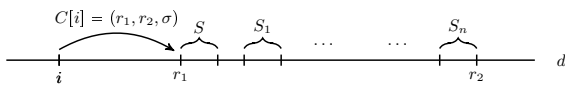
Function context_match( $d, S, (v_1, S_1), \dots, (v_n, S_n)$ )
   $A := \text{separators\_match}(d, S)$ 
  for  $i = 1$  to  $n$  do
     $B[i] := \text{separators\_match}(d, S_i)$ 
   $m := 1$ 
  while  $A[m] \neq \text{null}$  do
     $(s_1, s_2) := A[m]$ 
     $r_1 := s_1$ 
     $\sigma := \emptyset$ 
    for  $i = 1$  to  $n$  do
      if  $B[i][s_2] = \text{null}$  then break
       $(t_1, t_2) := B[i][s_2]$ 
      if  $v_i \neq \perp$  then  $\sigma(v_i) := (s_2, t_1)$ 
       $(s_1, s_2) := (t_1, t_2)$ 
    if  $i = n + 1$  then
       $r_2 := s_2$ 
      for  $i = m$  to  $r_1$  do
         $C[i] := (r_1, r_2, \sigma)$ 
       $m := r_1 + 1$ 
    else if  $A[r_1 + 1] \subseteq A[r_1]$  then
       $A[m] := A[r_1 + 1]$ 
    else break
  return  $C$ 

```

Figure 3: Finding all matches for a context.

To start the computation, the algorithm `context_match` in Figure 3 calls the function `separators_match` from Figure 2 for each of the input sets of words S, S_1, \dots, 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 $i = 1 \dots n$, 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 .

The main loop of the algorithm now proceeds to try and match the context expression $\text{any}(S)/v_1 : \text{next}(S_1)/\dots/v_n : \text{next}(S_n)$ 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 $A[m] \neq \text{null}$), 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 $(k_1, k_2) \subseteq (\ell_1, \ell_2)$, 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.



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

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

Function forward_index(d, C_1, \dots, C_n)

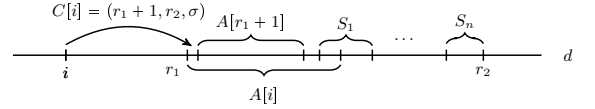
```

 $k := n$ 
for  $i = |d| + 1$  to  $1$  do
  if  $k = 0$  then
     $R[i] := 1$ 
  else if  $C_k[i] = \text{null}$  then
     $R[i] := k + 1$ 
  else
     $(r_1, r_2, \sigma) := C_k[i]$ 
    if  $R[r_2] = k + 1$  then  $k := k - 1$ 
     $R[i] := k + 1$ 
return  $R$ 

```

Figure 4: Computing the forward index.

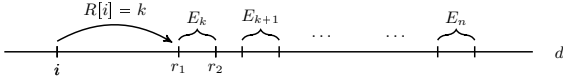
If there is a match for S that starts after the position r_1 , ends after the ending position of the current match for S , and allows to successfully match all the sets S_1 through S_n , then so does the current match of S (after all our expression only asks for the next position that matches each S_i 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, \dots, 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 $i = 1 \dots n$. We can thus conclude that running the procedure `context_match` from Figure 3 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 $\text{any}(S)/v_1 : \text{next}(S_1)/\dots/v_n : \text{next}(S_n)$. 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, \dots, E_n of such subexpressions, and we have computed the corresponding arrays C_1, \dots, C_n using the algorithm `context_match` from Figure 3.

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 4, 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 $E_k/E_{k+1}/\dots/E_n$ 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 4), 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, \dots, 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 5, computes all the possible valuations that make an expression $\psi = E_1/E_2/\dots/E_n$ true, where each E_i is a context of the form `any(S)/v1 : next(S1)/.../vn : next(Sn)`. 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 $E_m/\dots/E_n$. 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 pre-computed 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 $\sigma_1, \dots, \sigma_n$ to produce a variable assignment 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 possible matches for ψ .

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

Data: A document d , an array $R[1..|d| + 1]$ over $\{1, \dots, n + 1\}$, and a sequence C_1, \dots, C_n such that each C_i is an array $C_i[1..|d| + 1]$ of triples (r_1, r_2, σ) , where $(r_1, r_2) \in \text{span}(d)$ and $\sigma : \mathcal{V} \rightarrow \text{span}(d)$ is a partial function.

Result: A set O of partial functions $\sigma : \mathcal{V} \rightarrow \text{span}(d)$.

Function `all_matches`(d, R, C_1, \dots, C_n)

```

Let  $T[0..n + 1]$  be an array of integers
Let  $\sigma_1, \dots, \sigma_n$  be partial functions from  $\mathcal{V}$  to  $\text{span}(d)$ 
 $T[1] := 1$ 
 $m := 1$ 
while  $m \geq 1$  do
  if  $R[T[m]] > m$  then
     $m := m - 1$ 
     $T[m] := T[m] + 1$ 
  else
     $(r_1, r_2, \sigma) := C_m[T[m]]$ 
     $T[m] := r_1$ 
     $\sigma_m := \sigma$ 
     $m := m + 1$ 
     $T[m] := r_2$ 
  if  $m > n$  then
     $O := O \cup \{\sigma_1 \uplus \dots \uplus \sigma_n\}$ 
     $m := m - 1$ 
     $T[m] := T[m] + 1$ 
return  $O$ 

```

Figure 5: Computing all possible matches.

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 $O(|\psi| \cdot |d| + |\text{Output}|)$. Finally, the complete procedure is a constant-delay algorithm [21] given that it takes time $O(|\psi| \cdot |d|)$ in preprocessing the input and constant time between two consecutive outputs.

7. EXPERIMENTAL EVALUATION

To illustrate that the algorithm from Section 6 does not only have good theoretical complexity, in this section we describe how a system based on its implementation performs over real world datasets. In particular, we describe the datasets and the experiments used to test the efficiency of this algorithm, providing enough information so that the reader can understand the steps in our experimental evaluation. However, some details of our evaluation cannot be included due to the lack of space. Complete description of the missing details, together with more information about the datasets used can be found at [1]. In this link, we also provide the complete source code and documentation of our implementation, more detailed results of our experiments and additional data used for testing.

Datasets. We test our implementation on a number of CSV datasets and query logs which are explained below.

CSV files come from the use cases considered by W3C's CSV on the Web working group [29] and use data from: The World Bank (WB) [30], Office for National Statistics UK (ONS) [17] and the City of Palo Alto tree data (PA) [24]. 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

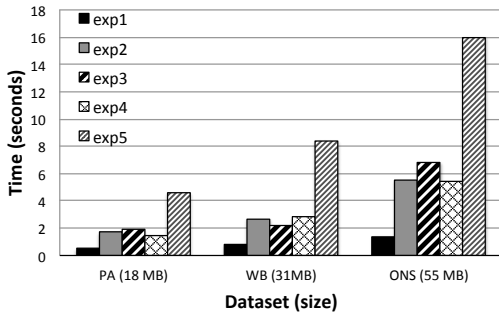


Figure 6: Running time on medium sized CSV files.

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 Linked SPARQL Queries Dataset team [25] and come from public SPARQL endpoints of the British Museum [23] and DBPedia [8]. For the experiments we used the raw log files provided by [25] 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 we used for testing we will present the evaluation result for only a handful of them. However, all files of similar size show same trends in evaluation time, and we provide the complete results for all the files used in [1].

Implementation syntax. In our implementation we use a slightly modified syntax from the one in Section 3. In particular, to facilitate keyboard input we use $\backslash n$ in place of \leftarrow , $?x$ when specifying a variable and $\&x$ for its content. Similarly, \wedge and \S denote the beginning and the end of file, respectively. Additionally for writing annotation programs like $A_1(x_1) \wedge A_2(x_2) \rightarrow A(x)$ we use the syntax $A(?x) :- A_1(?x1), A_2(?x2)$. To illustrate how the experiments are expressed using this syntax consider the following navigation program:

```
C(?x) :- /any(\n)/?x:next(,)/,
        ?x."[A-Z][0-9]{8}"
```

This program annotates all the spans in the first column whose content is a capital letter followed by eight digits (this type of value is used as a code in the ONS dataset).

Experiments for CSV files. These experiments were motivated by the requirements for CSV metadata proposed by the W3C CSV on the Web working group [29]. 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

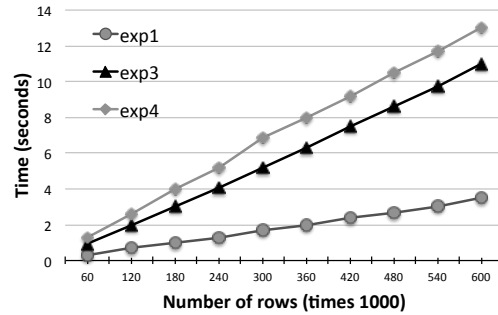


Figure 7: Scaling based on file size.

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 6, 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 index. 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 600000 lines. From this we generated ten different files, each containing first $k \times 60000$ 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, the results do scale as expected from the formal analysis in Section 6. Here we selected experiments 1, 3 and 4 as they are the most representative. The other two experiments behave similarly (see [1] and Table 1).

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 1. As we can see, for files less than 200 MB in size there are no significant problems in terms of the evaluation.

	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

Table 1: Running times (in seconds) on large files.

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.

Our final round of experiments for CSV files was conducted using noisy documents which do not conform to the tabular format as described in [27]. The theoretical analysis of our programs in Section 6 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 [1].

As far as the experimental results on query logs are concerned they show similar performance as the ones on CSV files. In Figure 8 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.

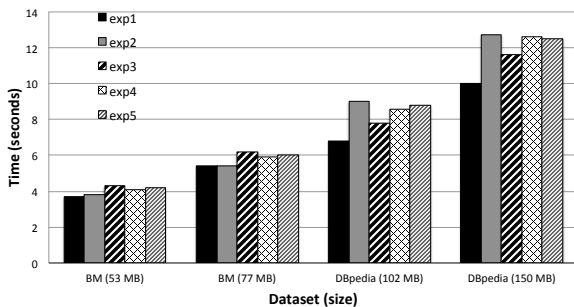


Figure 8: Running times on query logs.

8. CONCLUSIONS

Motivated by the recent initiative of the W3C to standardize metadata for the CSV format [28] we proposed a framework for annotating CSV-like documents based on expressions that extract the data and annotate it using a rule-based language. We study this framework from a theoretical perspective and develop efficient algorithms for evaluating it in practice. As a proof of concept we also test the implementation of our framework on a number of real world CSV files and query logs and show that it can handle document sizes one is likely to encounter in practice.

As future work, we plan to work on extending our framework to allow transforming CSV and CSV-like data to other formats, as this requirement often arises in practical scenarios [29]. Furthermore, as our implementation was meant to serve only as a proof of concept, we would also like to develop a full fledged system based on our framework that would be capable of handling vast amounts of data one is likely to encounter when dealing with corporate records.

9. REFERENCES

- [1] Online Appendix. <http://dvr.goc.ing.puc.cl/CSV>, 2016.
- [2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [3] A. V. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 255–300. 1990.
- [4] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.
- [5] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.
- [6] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *The VLDB Journal*, 14(4):373–396, 2005.
- [7] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.
- [8] DBpedia team. DBpedia. <http://dbpedia.org>, 2015.
- [9] S. J. Edwards. Portable game notation. https://en.wikipedia.org/wiki/Portable_Game_Notation.
- [10] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *Journal of the ACM*, 62(2), 2015.
- [11] T. Furche, G. Gottlob, G. Grasso, O. Gunes, X. Guo, A. Kravchenko, G. Orsi, C. Schallhart, A. J. Sellers, and C. Wang. Diadem: domain-centric, intelligent, automated data extraction methodology. In *WWW*, 2012.
- [12] F. Geerts and J. V. den Bussche. Relational completeness of query languages for annotated databases. *J. Comput. Syst. Sci.*, 77(3):491–504, 2011.
- [13] HM Government Land Registry. Price Paid Data. <https://www.gov.uk/government/collections/price-paid-data>.
- [14] B. Kimelfeld. Database principles in information extraction. In *PODS’14*, pages 156–163, 2014.
- [15] W. Martens, F. Neven, and S. Vansummeren. SCULPT: A Schema Language for Tabular Data on the Web. In *WWW*, 2015.
- [16] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *ACM SIGMOD*, pages 193–204. ACM, 2003.
- [17] Office for National Statistics. ONS Data Explorer. <http://www.ons.gov.uk/ons/data/web/explorer>.
- [18] Open Geospatial Consortium. Well-known text. https://en.wikipedia.org/wiki/Well-known_text.
- [19] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE 2008*, pages 933–942, 2008.
- [20] Research Information systems. RIS (file format). [https://en.wikipedia.org/wiki/RIS_\(file_format\)](https://en.wikipedia.org/wiki/RIS_(file_format)), 2015.
- [21] L. Segoufin. A glimpse on constant delay enumeration. In *STACS 2014*, pages 13–27, 2014.
- [22] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, pages 1033–1044, 2007.
- [23] The British Museum. British Museum public endpoint. <http://bm.rkbexplorer.com>, 2015.
- [24] The City of Palo Alto, California Urban Forest Section. Palo Alto tree data. http://w3c.github.io/csvw/use-cases-and-requirements/Palo_Alto_Trees.csv, 2015.
- [25] The LSQ team. The Linked SPARQL Queries Dataset. <http://aksw.github.io/LSQ/>, 2015.
- [26] Unix.com. Man page for grep. <http://www.unix.com/man-page/linux/1/grep/>.
- [27] W3C. Model for Tabular Data and Metadata on the Web. <http://www.w3.org/TR/tabular-data-model/>.
- [28] W3C. CSV on the Web Working Group. <http://www.w3.org/2013/csvw/>, 2013.
- [29] W3C. CSV on the Web: Use Cases and Requirements, Editor’s Draft. <http://w3c.github.io/csvw/use-cases-and-requirements/>, December 2014.
- [30] World Bank. Data Records. <http://data.worldbank.org/>.

APPENDIX

Proof of Theorem 4.1

In this section, we show that $\text{NONEMP}(\mathcal{N}\mathcal{E})$ is NP-complete. The proof that the same holds for $\text{NONEMP}(\mathcal{SRE})$ is analogous, and can be easily derived from this proof.

For the upper bound of $\text{NONEMP}(\mathcal{N}\mathcal{E})$ we use the usual guess and check algorithm. Assume that d is a document and ψ is a navigation expression. To check whether $\llbracket \text{doc}.\psi \rrbracket_d$ is not empty, for each variable appearing in ψ we guess a span $p \in \text{span}(d)$ that will be assigned to it. After this the problem boils down to checking whether a string belongs to the language defined by a regular expression, which can be solved in polynomial time [3].

For the lower bound we do a reduction from 3-CNF-SAT. Let $\alpha = (a_1 \vee b_1 \vee c_1) \wedge \dots \wedge (a_k \vee b_k \vee c_k)$ be an arbitrary 3-CNF propositional formula, where $a_i, b_i,$ and c_i are literals. Furthermore, assume that y_1, \dots, y_n are all the propositional variables occurring in α , and that $\bar{y}_1, \dots, \bar{y}_n$ are the negations of these propositional variables.

For the reduction, we construct a string w_α , a navigation expression R_α using the variables x_1, \dots, x_k and define the document d to be $\vdash w_\alpha \dashv$. We then show that $\llbracket \text{doc}.\mathcal{R}_\alpha \rrbracket_d$ is not empty if and only if α is satisfiable. The main idea behind w_α and R_α is that w_α will encode the structure of α and R_α will check whether there exists a satisfying assignment for α . Let $\Sigma_\alpha = \{\#, ', \bar{\cdot}, \lrcorner, y_1, \bar{y}_1, \dots, y_n, \bar{y}_n\}$. For the sake of readability, we define the substrings $\text{eq}(y, z)$ and u_α as follows, where y and z are two propositional variables:

$$\begin{aligned} \text{eq}(y, z) &= \begin{cases} \#y, z\#y, \bar{z}\#\bar{y}, z\#\bar{y}, \bar{z} & y \neq z \\ \#y, y\#\bar{y}, \bar{y} & y = z \end{cases} \\ u_\alpha &= \text{eq}(y_1, y_1) \text{eq}(y_1, y_2) \dots \text{eq}(y_1, y_n) \\ &\quad \text{eq}(y_2, y_1) \text{eq}(y_2, y_2) \dots \text{eq}(y_2, y_n) \\ &\quad \dots \\ &\quad \text{eq}(y_n, y_1) \text{eq}(y_n, y_2) \dots \text{eq}(y_n, y_n) \end{aligned}$$

The substring $\text{eq}(y, z)$ is displaying all consistent combinations between y, z and their negations, and u_α is displaying all combinations between propositional variables of α . Notice that $\#y, \bar{y}\#$ or $\#\bar{y}, y\#$ are not substrings of u_α for every propositional variable y . The string w_α is defined as:

$$w_\alpha = \#, a_1, b_1, c_1, \# \dots \#, a_k, b_k, c_k, \# \underbrace{\lrcorner u_\alpha \dots \lrcorner u_\alpha}_{\frac{k \cdot (k-1)}{2} \text{ times}}$$

The document d is now defined as the string $\vdash w_\alpha \dashv$.

We show next the navigation expression R_α . Let x_1, \dots, x_k be k different variables. Similar that for w_α , in order to define R_α we use subformulas to simplify its presentation. For a variable x_i , consider the following expressions:

$$\begin{aligned} \text{cap}(x_i) &= \text{next}(\#)/\text{any}(\cdot)/x_i:\text{next}(\cdot) \\ \text{comp}(x_i) &= \text{next}(\lrcorner)/ \\ &\quad \text{any}(\#)/\langle x_i \rangle:\text{next}(\cdot)/\langle x_{i+1} \rangle:\text{next}(\#)/ \\ &\quad \dots / \\ &\quad \text{next}(\lrcorner)/ \\ &\quad \text{any}(\#)/\langle x_i \rangle:\text{next}(\cdot)/\langle x_k \rangle:\text{next}(\#)/ \end{aligned}$$

Above, the expression $\text{cap}(x_i)$ is used to capture a literal in the substring $\#, a_i, b_i, c_i$, and $\text{comp}(x_i)$ is used to compare the content of x_i with the variables x_{i+1}, \dots, x_k in the substring $\lrcorner u_\alpha \dots \lrcorner u_\alpha$. Then the expression R_α is defined as follows:

$$R_\alpha = \text{cap}(x_1)/\dots/\text{cap}(x_k)/\text{comp}(x_1)/\dots/\text{comp}(x_{k-1})$$

To finish the proof, one can easily check that there is a satisfying assignment for α if and only if $\llbracket \text{doc}.\mathcal{R}_\alpha \rrbracket_d$ is not empty.

Proof of Theorem 4.2

If the content operator is not allowed, then the membership of the problem $\text{NONEMP}(\mathcal{N}\mathcal{E})$ in PTIME is a consequence of the existence of a polynomial-time algorithm for the enumeration problem for navigation expressions without the content operator (see Section 6). Moreover, the membership of $\text{NONEMP}(\mathcal{SRE})$ is NP under this restriction is a consequence of the fact that the unrestricted version of this problem is in NP (see Theorem 4.1).

To prove that $\text{NONEMP}(\mathcal{SRE})$ is NP-hard even if the content operator is not allowed, we provide a reduction from 1-IN-3-SAT. The input of 1-IN-3-SAT is a propositional formula $\alpha = C_1 \wedge \dots \wedge C_n$, 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 (see *M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, 1979*).

For the reduction, we construct a string w_α and a navigation expression R_α such that $\llbracket \text{doc}.\mathcal{R}_\alpha \rrbracket_d$ is not empty if and only if there exists a satisfying assignment for α that makes exactly one variable per clause true, with d the string $\vdash w_\alpha \dashv$. 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 R_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, R_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 SRE of the form $(x_{i,1} + x_{i,2} + x_{i,3})$, indicating that exactly one of $x_{i,1}, x_{i,2}$ and $x_{i,3}$ has to be assigned a span. Besides, R_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 R_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 $\text{conflict}(p_{i,j})$:

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

The variable $y_{i,j,k,\ell}$ is used as follows in R_α . If some spans have been assigned to $x_{i,j}$ and $c_{i,j,k,\ell}$, 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 SRE $R_{i,j}$ as the concatenation of the vari-

ables in $\text{conflict}(p_{i,j})$ in no particular order. For example, if

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

then

$$R_{3,1} = y_{1,2,3,1}/y_{1,3,3,1}/y_{3,1,4,1}/y_{3,1,5,2}$$

Moreover, assuming that $\text{conflict}(p_{i,j})$ contains ℓ variables, define $S_{i,j}$ as $0^{3n-\ell-1}$, that is, a string consisting of $(3n-\ell-1)$ symbols 0. Finally, for every clause C_i ($1 \leq i \leq n$) define SRE R_i as:

$$\#/(x_{i,1}/R_{i,1}/S_{i,1} + x_{i,2}/R_{i,2}/S_{i,2} + x_{i,3}/R_{i,3}/S_{i,3})$$

With this notation, we define SRE R_α as follows:

$$R_\alpha = \vdash/R_1/\cdots/R_n/\neg$$

To conclude the reduction, we define the string w_α as u^n (the concatenation of n strings u), where u is defined as the string $\#0^{3n}$. At this point it is important to understand how the variables $y_{i,j,k,\ell}$'s are used in the SRE R_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, $R_\alpha = \vdash/R_1/R_2/\cdots/R_n/\neg$, where R_1 is of the form:

$$\#(x_{1,1}/\cdots/y_{1,1,2,2}/\cdots/S_{1,1} + x_{1,2}/R_{1,2}/S_{1,2} + x_{1,3}/R_{1,3}/S_{1,3}),$$

given that $y_{1,1,2,2} \in \text{conflict}(p_{1,1})$, and R_2 is of the form:

$$\#(x_{2,1}/R_{2,1}/S_{1,1} + x_{2,2}/\cdots/y_{1,1,2,2}/\cdots/S_{2,2} + x_{2,3}/R_{2,3}/S_{2,3}),$$

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, say p (both spans will have content 1 by the definition of R_α and w_α). If we now try to assign a span to $x_{2,2}$, then we are forced to assign a span to $y_{1,1,2,2}$, say q . But then p should be equal to q , which is not possible as two spans (d, r_1, s_1) and (d, r_2, s_2) are equal if $r_1 = r_2$ and $s_1 = s_2$, and in this case p and q are located in different positions of w_α and, thus, of d .

Based on the previous intuition, it is straightforward to prove that $\llbracket d \circ c.R_\alpha \rrbracket_d$ 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 $\vdash w_\alpha \neg$.

Proof of Theorem 4.3

Consider the problem of verifying, given a document d , a variable assignment σ over d and an NE R , whether $\sigma \in \llbracket d \circ c.R \rrbracket_d$. By following the same ideas as in the proof of Theorem 4.1, it can be shown that this problem is in NP. Moreover, the same problem but with NEs replaced by SREs can also be shown to be in NP. Thus, the membership of $\text{MEMB}(\mathcal{AP})$ and $\text{MEMB}(\mathcal{NVP})$ in NP is a consequence of these two facts, together with the fact that the evaluation problem for monadic datalog is NP-complete with respect to combined complexity (see Proposition 3.4 in *Gottlob, Georg and Koch, Christoph, Monadic datalog and the expressive power of languages for web information extraction, Journal of the ACM, 51(1), 2004*).

Proof of Theorem 4.4

The PTIME evaluation of a tree-like navigation program without content operator consists of two steps. The first step is to transform

each rule into an equivalent set of rules such that each rule contains at most one navigation expression. Then the second step consists in evaluating the resulting navigation program. Thus, the following presentation of the algorithm is divided in these two steps.

First step. Consider a generic rule τ of the form:

$$\theta \wedge A_1(x_1) \wedge \cdots \wedge A_n(x_n) \rightarrow A(x)$$

where θ is a tree-like conjunction of navigation expressions and G_θ is the graph obtained from θ . Our goal is to rewrite this rule in a set of rules \mathcal{S} where each rule is of the form:

$$\ell.\psi \wedge B_1(y_1) \wedge \cdots \wedge B_m(y_m) \rightarrow B(x)$$

such that $A(p) \in \text{EVAL}(\{\tau\}, d)$ if, and only if, $A(p) \in \text{EVAL}(\mathcal{S}, d)$ for any document d . For the sake of simplification, in this construction we assume that G_θ consists of one rooted-tree.

Fix a variable y in θ . We denote by G_θ^y the induced subtree of all the descendants of y in G_θ . Furthermore, consider the following formulas of τ with respect to y :

- $y.\varphi_y(z_1, \dots, z_n)$ is the only navigation expression in θ starting with y (whenever it exists),
- α_y is the conjunction of all annotations predicates in θ that mention y , and
- β_y is the conjunction of all navigation expressions and annotations in θ that mention variables in G_θ^y except from y .

It is easy to check that $y.\varphi_y(z_1, \dots, z_n) \wedge \alpha_y \wedge \beta_y$ is the maximum subformula of the body of τ that contains all variables in G_θ^y . Indeed, we can define the rule:

$$\tau_y : y.\varphi_y(z_1, \dots, z_n) \wedge \alpha_y \wedge \beta_y \rightarrow C_y(y)$$

where C_y is a fresh annotation symbol. That is, C_y will contain all the spans that satisfies the subformula induced by the graph G_θ^y .

Next, we show a set of rules \mathcal{S}_y that is equivalent to τ_y for every variable y . We define the set \mathcal{S}_y inductively as follows. Suppose that y is a leaf in G_θ and there exists an expression $y.R$ where R is a regular expression (if not, define $R = \Sigma^*$ where Σ is the whole alphabet). Then define the rule:

$$\tau'_y : y.R \wedge \alpha_y \rightarrow C_y(y)$$

Clearly, τ'_y is equivalent to τ_y (i.e. β_y is empty) and, thus, we define $\mathcal{S}_y = \{\tau'_y\}$. Now, by induction suppose that $\mathcal{S}_{z_1}, \dots, \mathcal{S}_{z_n}$ are already defined and consider the formula:

$$\tau'_y : y.\varphi_y(z_1, \dots, z_n) \wedge \alpha_y \wedge C_{z_1}(z_1) \wedge \cdots \wedge C_{z_n}(z_n) \rightarrow C_y(y)$$

Then one can easily check that the set of rules $\mathcal{S}_y = \{\tau'_y\} \cup \mathcal{S}_{z_1} \cup \cdots \cup \mathcal{S}_{z_n}$ is equivalent to τ_y .

The next step in the construction is to extract the variable x . Notice that, if x is the variable at the root of G_θ , we are done because the set of rules $\mathcal{S}_x \cup \{C_x(x) \rightarrow A(x)\}$ is equivalent to τ . However, if x is not the root of G_θ , we have to add more rules to filter the ancestors of x . Let y_1, \dots, y_k be a path in G_θ from the root y_1 to $y_k = x$ for some $k \geq 1$. Then define \mathcal{S}' as the set that contains $C_{y_1}(y_1) \rightarrow D_{y_1}(y_1)$ and:

$$y_i.\varphi_{y_i}(z_1, \dots, z_n) \wedge D_{y_i}(y_i) \wedge C_{z_1}(z_1) \wedge \cdots \wedge C_{z_n}(z_n) \rightarrow D_{y_{i+1}}(y_{i+1})$$

for every $1 < i \leq k$ (recall that $y_{i+1} \in \{z_1, \dots, z_n\}$ since y_{i+1} is a child of y_i in G_θ). Then one can easily check that the set $\mathcal{S} = \mathcal{S}_{z_1} \cup \mathcal{S}' \cup \{D_x(x) \rightarrow A(x)\}$ is equivalent to τ , each rule in \mathcal{S} has at most one navigation expression, and \mathcal{S} is of polynomial size with respect to τ .

Second step. We discuss now how to check in PTIME that $A(p) \in \text{EVAL}(\Pi, d)$ for a document d , a navigation program Π with one navigation expression per rule, and a ground annotation $A(p)$. This can be done in $\text{ASPACE}(\log(n))$, namely, an alternating log-space turing machine. For this, we can first suppose that each rule is totally order with respect to its corresponding rule. Then the alternating machine must keep one pointer for the current rule r , one pointer for the current variable x , and two pointer for the current span p (i.e. the span that satisfies x in r). Since each rule is totally order with respect to its navigation expression, the machine can universally decide to either check if p satisfies a particular annotation predicate in r and follow the rule that define it, or to check that the next variable in the navigation expression of r can be satisfied. For the second case, one can existentially choose a span that satisfies the next variable and continue. All this process can be done with a fix number of pointers, that is, with logarithmic space. It is important to notice that the base case consists in checking that a span satisfies a specific regular expression, but this can be checked in NLOGSPACE.