Cleaning Inconsistencies in Information Extraction via Prioritized Repairs

Ronald Fagin
IBM Research – Almaden
San Jose, CA, USA
fagin@us.ibm.com

Benny Kimelfeld*
LogicBlox, Inc.
Berkeley, CA, USA
bennyk@gmail.com

Frederick Reiss
IBM Research – Almaden
San Jose, CA, USA
frreiss@us.ibm.com

Stijn Vansummeren
Université Libre de Bruxelles (ULB)
Bruxelles, Belgium
stijn.vansummeren@ulb.ac.be

ABSTRACT

The population of a predefined relational schema from textual content, commonly known as Information Extraction (IE), is a pervasive task in contemporary computational challenges associated with Big Data. Since the textual content varies widely in nature and structure (from machine logs to informal natural language), it is notoriously difficult to write IE programs that extract the sought information without any inconsistencies (e.g., a substring should not be annotated as both an address and a person name). Dealing with inconsistencies is hence of crucial importance in IE systems. Industrial-strength IE systems like GATE and IBM SystemT therefore provide a built-in collection of *cleaning* operations to remove inconsistencies from extracted relations. These operations, however, are collected in an ad-hoc fashion through use cases. Ideally, we would like to allow IE developers to declare their own policies. But existing cleaning operations are defined in an algorithmic way and, hence, it is not clear how to extend the built-in operations without requiring low-level coding of internal or external functions.

We embark on the establishment of a framework for declarative cleaning of inconsistencies in IE, though principles of database theory. Specifically, building upon the formalism of *document spanners* for IE, we adopt the concept of *prioritized repairs*, which has been recently proposed as an extension of the traditional database repairs to incorporate priorities among conflicting facts. We show that our framework captures the popular cleaning policies, as well as the POSIX semantics for extraction through regular expressions. We explore the problem of determining whether a cleaning declaration is unambiguous (i.e., always results in a single repair), and whether it increases the expressive power of the extraction language. We give both positive and negative results, some of which are general, and some of which apply to policies used in practice.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*Data models*; H.2.4 [**Database Management**]: Systems—*Textual databases, Relational databases, Rule-based databases*; I.5.4 [**Pattern Recognition**]: Applications—*Text processing*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Algebraic language theory, Classes defined by grammars or automata, Operations on languages*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata, Relations between models*

*Work done while at IBM Almaden – Research.

General Terms

Theory

Keywords

Information extraction, document spanners, regular expressions, extraction inconsistency, database repairs, prioritized repairs

1. INTRODUCTION

Information Extraction (IE) conventionally refers to the task of automatically extracting structured information from text. While early work in the area focused largely on military applications [22], this task is nowadays pervasive in a plethora of computational challenges (especially those associated with Big Data), including social media analysis [4], machine data analysis [21], healthcare analysis [40], customer relationship management [1], and indexing for semantic search [41]. Moreover, contemporary analytics platforms like Hadoop make the analysis of data more accessible to a broad range of users. The techniques used for implementing IE tasks include rule engineering [20,35,37], rule learning [11,28], probabilistic graph models [25, 27, 30], and other statistical models such as Markov Logic Networks [31, 33] and probabilistic databases [12]. There are also general frameworks designated to the development and scalable execution of IE programs, such as UIMA [18], the General Architecture for Text Engineering (GATE) [9], Xlog [36] and SystemT [7].

GATE and SystemT are highly relevant to this paper. GATE, an open-source project by the University of Sheffield, is an instantiation of the *cascaded finite-state transducers* [2]. The core IE engine of GATE, called JAPE, processes a document via a sequence of *phases* (*cascades*), each annotating spans (intervals within the document) with types by processing previous annotations by applying grammar rules and user defined Java procedures. SystemT exposes an SQL-like declarative language named *AQL* (Annotation Query Language), along with a query plan optimizer [34] and development tooling [28]. Conceptually, AQL supports a collection of “direct” extractors of relations from text (e.g., tokenizer, dictionary lookup, regex matcher, part-of-speech tagger, and other morphological analyzers), along with an algebra for relational manipulation.

Databases often involve inconsistency, due to human errors, integration of heterogeneous resources, imprecision in ETL flows, and so on. The database research community has proposed principled ways to capture, manage and resolve data inconsistency [3,5,6,15–17,29,38]. Here, the identification and capturing of inconsistencies is usually done through the use of specialized constraints and dependencies [3,15,16,29,38]. A prominent formalism for specifying inconsistencies in this respect are the *denial constraints* [3,38] that can specify, for example that no two persons can have the same driver's license number, or that no single person can have multiple residential addresses. To manage and resolve data inconsistencies, the database research community has proposed a wide variety of methods, ranging from consistent query answering [3], to formalisms for the declarative specification of conflict resolution by means of prioritized repairs [38], to data cleaning and data fusion tools that introduce domain-specific operators for removing data inconsistencies [5,6,17].

In IE tasks, inconsistencies occur at a low level. For example, an extractor may annotate multiple person mentions inside *"Martin Luther King Jr"*, namely: *Martin Luther*, *Luther King, Martin Luther King Jr.,* etc. Moreover, all of these annotated spans may be contained in the larger span *"1805 Martin Luther King Jr Way, Berkeley, CA 94709,"* which is annotated as an address, and which should not overlap with person mentions.

To handle inconsistency, nearly all rule-based IE systems integrate a cleaning mechanism as a core element the rule language. For example, the CPSL standard, which underpins many commonly-used systems, stipulates that "at each [token position]... one of the matching rules is selected as a 'best match' and is applied" [2]. JAPE, a popular implementation of CPSL, generalizes this "best match" concept to a collection of "controls" that represent different cleaning policies. Every JAPE rule must include a specification of which control applies to matches of the rule. As an example, in the Appelt control the annotation procedure (transducer) scans the document left to right; when at a specific location, it applies only the longest annotation, and continues scanning right after that annotation. In the Brill control scanning is also left to right, and when at a specific location, all the annotations that begin there are retained; but after that, scanning still continues after the longest annotation. In AQL, this cleaning mechanism comes in the form of "consolidation." Specifically, the AQL declaration of a view can include a command to filter out tuples by applying a consolidation policy to one of the columns. There is a built-in collection of such policies, like LeftToRight, which is similar to Appelt, and ContainedWithin that retains only the spans that are not strictly contained in other spans. The Appelt control of JAPE, as well as the ContainedWithin consolidator of AQL, can involve explicitly specified priorities that we ignore here for simplicity; we discuss those in Section 5.2.

The goal of this work is to establish principles for declarative cleaning of inconsistencies in IE programs. We build upon our recent work [14], where we proposed the framework of *document spanners* (or just *spanners* for short) that captures the relational philosophy of AQL. Intuitively, a spanner extracts from a document $\mathbf{d}$ (which is a string over a finite alphabet) a relation over the spans of $\mathbf{d}$. An example of a spanner representation is a *regex formula*: a regular expression with embedded capture variables that are viewed as relational attributes. A *regular* spanner is one that can be expressed in the closure of the regex formulas under relational algebra. We extend the spanners into *extraction programs*, which are non-recursive Datalog programs that can use spanners in the right hand side of the rules. We then include denial constraints (again phrased using spanners) to specify integrity constraints within the program. In the presence of denial constraints, a *repair* of a schema instance is a subset of facts that satisfies all the constraints, and is not strictly contained in any other such subset [3].

Denial constraints do not provide any means to discriminate among repairs, and are therefore insufficient to capture common cleaning policies in IE like the ones aforementioned (Appelt and Brill, etc.); those strategies imply not only which sets of facts are in conflict, but also which fact is in preference to remain or be dropped. To accommodate preferences, we adopt the *prioritized repairs* of Staworko et al. [38], which extend the concept of repairs with priorities among facts that, eventually, translate into priorities among repairs. More precisely, Staworko et al. assume that in addition to denial constraints, the inconsistent database is associated with a binary relation $\succ$, called *priority*, over the facts (where $f_1 \succ f_2$ means that f_1 has priority over f_2). We say that a repair J *can be improved* if we can add to J a new database fact f and retain consistency of J by removing only facts that are inferior to f (according to the priority). The idea is to restrict the set of repairs to those that are *Pareto-optimal* (or just *optimal* hereafter), where an optimal repair is one that cannot be improved.[1]

Staworko et al. [38] made the assumption that the priority relation $\succ$ is given in an explicit manner, and did not provide any syntax for declaring priority at the schema level. Here, we need such a syntax, and we propose what we call a *priority generating dependency*, or just *pgd* for short. A pgd has the logical form $\psi(\mathbf{x}) \rightarrow (\varphi_1(\mathbf{x}) \succ \varphi_2(\mathbf{x}))$, where $\mathbf{x}$ is a sequence of variables, all universally quantified (and all assigned spans in our framework), $\psi(\mathbf{x})$ represents a spanner with variables in $\mathbf{x}$, and the $\varphi_i(\mathbf{x})$ are atomic formulas over the relational schema. A *cleaning update* in an extraction program is specified by a collection of denial constraints and pgds, and it instructs the program to branch into the optimal repairs (which are viewed as possible worlds). In Section 5 we show that common strategies for cleaning inconsistencies in IE, like all those used in JAPE and AQL, can be phrased in our framework, where all involved spanners are regular. We further show that the POSIX semantics for regex formulas is expressible in our framework. The POSIX semantics can be viewed as an extreme cleaning policy for a regex formula, dictating that the evaluation on a string always results in a single match [24].

One difference between the ordinary concept of repairs and the prioritized repairs is that the latter give rise to interesting cases where a *single* optimal repair exists. This is a significant difference, since data management systems are usually not designed to support multiple possible worlds. Here, we refer to this property as *unambiguity*. An extraction program is *unambiguous* if, for all input documents, the result consists of exactly one possible world. This property holds in all of the IE cleaning policies mentioned thus far. The next problem we study is that of deciding whether a given extraction program is unambiguous. We prove that, for the class of programs that use regular spanners, this property is undecidable. To prove that, we give an intermediate result of independent interest: it is undecidable to determine whether a two-way two-head deterministic finite automaton [23] has any immortal finite configuration.

Staworko et al. [38] show that under two conditions, unambiguity is guaranteed. The first condition (which they assume throughout their paper) is that the priority relation induces an acyclic graph. The second condition is *totality*: every two facts that are jointly involved in a conflict are also comparable in the priority relation. In Section 4 we improve that result by relaxing totality into what we call the *minimum property*: every conflict has a least prioritized member. In our setting, an important example where the minimum

[1] Staworko et al. [38] also study *global optimality* as an alternative to Pareto optimality. As we discuss in Section 3, it turns out that all the results in this paper hold true even if we adopt global optimality instead of the Pareto one.

property (and, in fact, totality) is guaranteed is the special case of a cleaning update consisting of binary conflicts, such that the pgds are specified *precisely* for those pairs in conflict. In fact, we define a special syntax to capture this case: a *denial pgd* is an expression of the form $\psi(\mathbf{x}) \rightarrow (\varphi_1(\mathbf{x}) \rhd \varphi_2(\mathbf{x}))$, which has the same semantics as a pgd, but it also specifies that $\varphi_1(\mathbf{x})$ and $\varphi_2(\mathbf{x})$ are in conflict (and the former is of higher priority). We then look again at extraction programs that use regular spanners. We prove that it is decidable to determine whether the minimum property is guaranteed by a given cleaning update. However, it turns out that it is undecidable to determine whether a pgd guarantees acyclicity of the priority relation. The conclusion is that other (stronger) conditions need to be imposed if we want to automatically verify unambiguity of an extraction program (an example would be by some means of a potential function, e.g., [13]). Such conditions are beyond the scope of this paper, and are left as important directions for future work.

An extreme example of an unambiguous extraction program is a program that does not use cleaning updates (hence, never branches). Given that one is interested in the content of a single relation of the program (which we assume as part of our definition of an extraction program), an unambiguous program can be viewed simply as a representation of a spanner. The next question we explore is whether cleaning updates increase the expressive power (when used in unambiguous programs). We define the property of *disposability* of a cleaning update that, intuitively, means that we can replace the cleaning update with a collection of non-cleaning (ordinary) rules. As usual, this definition is parameterized by the representation system we use for the involved spanners. In the case of unambiguous programs using regular spanners, if all the cleaning updates are disposable, then the spanner defined by the program is also regular. However, we show that there exists an unambiguous program that uses regular spanners and a single cleaning update, such that the resulting spanner is not regular. Moreover, in the case of *core spanners* (which extend the regular spanners with the string-equality selection [14]), each of JAPE's cleaners, as well as the POSIX one, strictly increase the expressive power.

In Section 5 we explore special cases of cleaning updates in extraction programs with regular spanners. The first case is that of acyclic and transitive denial pgds. An example of such a denial pgd is the ContainedWithin consolidation policy mentioned earlier in this section. The second case is the denial pgds that declare the different controls of JAPE. The third case is the POSIX policy for regex formulas, where we show how it is simulated by a sequence of denial pgds. It follows from our results so far that an extraction program that uses only cleaning updates among the three cases is unambiguous. We prove that all of these cleaning updates are disposable. We find these results interesting, since we can now draw the following conclusions. First, the extraction programs that use regular spanners, as well as cleaning updates among the three cases, have the same expressive power as the regular spanners. Second, for every regex formula γ there exists a regex formula γ', such that when evaluated over a document, γ' (without any cleaning applied) gives the same result as γ under the POSIX semantics. We are not aware of any result in the literature showing that the POSIX semantics can be "compiled away" in this sense.

2. DOCUMENT SPANNERS

In this section, we give some preliminary definitions and notation, and recall the formalism of *document spanners* [14].

2.1 Strings and Spans

We fix a finite alphabet Σ of *symbols*. We denote by Σ^* the set of all finite strings over Σ, and by Σ^+ the set of all finite strings of length at least one over Σ. For clarity of context, we will often refer to a string in Σ^* as a *document*. A *language over* Σ is a subset of Σ^*. Let $\mathbf{d} = \sigma_1 \cdots \sigma_n \in \Sigma^*$ be a document. The length n of $\mathbf{d}$ is denoted by $|\mathbf{d}|$. A *span* identifies a substring of $\mathbf{d}$ by specifying its bounding indices. Formally, a span of $\mathbf{d}$ has the form $[i, j\rangle$, where $1 \leqslant i \leqslant j \leqslant n + 1$. If $[i, j\rangle$ is a span of $\mathbf{d}$, then $\mathbf{d}_{[i,j\rangle}$ denotes the substring $\sigma_i \cdots \sigma_{j-1}$. Note that $\mathbf{d}_{[i,i\rangle}$ is the empty string, and that $\mathbf{d}_{[1,n+1\rangle}$ is $\mathbf{d}$. We note that the more standard notation would be $[i, j)$, but we use $[i, j\rangle$ to distinguish spans from intervals. For example, $[1, 1)$ and $[2, 2)$ are both the empty interval, hence equal, but in the case of spans we have $[i, j\rangle = [i', j'\rangle$ if and only if $i = i'$ and $j = j'$ (and in particular, $[1, 1\rangle \neq [2, 2\rangle$). We denote by $\mathsf{Spans}(\mathbf{d})$ the set of all the spans of $\mathbf{d}$. Two spans $[i, j\rangle$ and $[i', j'\rangle$ of $\mathbf{d}$ *overlap* if $i \leqslant i' < j$ or $i' \leqslant i < j'$, and are *disjoint* otherwise. Finally, $[i, j\rangle$ *contains* $[i', j'\rangle$ if $i \leqslant i' \leqslant j' \leqslant j$.

EXAMPLE 2.1. In all of the examples throughout the paper, we consider the example alphabet Σ which consists of the lowercase and capital letters from the English alphabet (i.e., a,...,z and A,...,Z), the comma symbol (" , "), and the underscore symbol ("_") that stands for whitespace. (We use a restricted alphabet for simplicity.) Figure 1 depicts an example document $\mathbf{d}$ in Σ^*. For ease of later reference, it also depicts the index of each character in $\mathbf{d}$. Figure 2 shows two tables containing spans of $\mathbf{d}$. Observe that the spans in the left table are those that correspond to words in $\mathbf{d}$ that are names of US states (Georgia, Washington and Virginia). For example, the span $[21, 28\rangle$ corresponds to Georgia. We will further discuss the meaning of these tables later. □

2.2 Document Spanners

We fix an infinite set SVars of *span variables*; spans may be assigned to these span variables. The sets Σ^* and SVars are disjoint. For a finite set $V \subseteq \mathsf{SVars}$ of variables and a document $\mathbf{d} \in \Sigma^*$, a $(V, \mathbf{d})$*-tuple* is a mapping $\mu \colon V \rightarrow \mathsf{Spans}(\mathbf{d})$ that assigns a span of $\mathbf{d}$ to each variable in V. A $(V, \mathbf{d})$*-relation* is a set of $(V, \mathbf{d})$-tuples. A *document spanner* (or just *spanner* for short) is a function P that is associated with a finite set V of variables, denoted $\mathsf{SVars}(P)$, and that maps every document $\mathbf{d}$ to a $(V, \mathbf{d})$-relation.

EXAMPLE 2.2. Throughout our running example (which started in Example 2.1) we will define several spanners. Two of those are denoted as $[\![\rho_{\mathsf{stt}}]\!]$ and $[\![\rho_{\mathsf{loc}}]\!]$, where $\mathsf{SVars}([\![\rho_{\mathsf{stt}}]\!]) = \{x\}$ and $\mathsf{SVars}([\![\rho_{\mathsf{loc}}]\!]) = \{x_1, x_2, y\}$. Later we will explain the meaning of the brackets, and specify what exactly each spanner extracts from a given document. For now, the span relations (tables) in Figure 2 show the results of applying the two spanners to the document $\mathbf{d}$ of Figure 1. □

Let P be a spanner with $\mathsf{SVars}(P) = V$. Let $\mathbf{d} \in \Sigma^*$ be a document, and let $\mu \in P(\mathbf{d})$ be a $(V, \mathbf{d})$-tuple. We say that μ is *hierarchical* if for all variables $x, y \in \mathsf{SVars}(P)$ one of the following holds: *(1)* the span $\mu(x)$ contains $\mu(y)$, *(2)* the span $\mu(y)$ contains $\mu(x)$, *or (3)* the spans $\mu(x)$ and $\mu(y)$ are disjoint. As an example, the reader can verify that all the tuples in Figure 2 are hierarchical. We say that P is *hierarchical* if μ is hierarchical for all $\mathbf{d} \in \Sigma^*$ and $\mu \in P(\mathbf{d})$. Observe that for two variables x and y of a hierarchical spanner, it may be the case that, over the same document, one tuple maps x to a subspan of y, another tuple maps y to a subspan of x, and a third tuple maps x and y to disjoint spans.

2.3 Spanner Representation Systems

By a *spanner representation system* we refer collectively to any manner of specifying spanners through finite objects. In previous

```
C a r t e r _ f r o m _ P l a i n s , _ G e o r g i a , _ W a s h i n g t o n _ f r o m _ W e s t m o r e l a n d , _ V i r g i n i a
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 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67
```

Figure 1: Document $\mathbf{d}$ in the running example

work [14] we defined several representation systems (by means of regular expressions, special types of automata, and relational algebra). Here, we recall the definition of the *regex formula* system, as well as its closure under relational algebra.

A *regular expression with capture variables*, or just *variable regex* for short, is an expression in the following syntax that extends that of regular expressions:

$$\gamma := \emptyset \mid \epsilon \mid \sigma \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^* \mid x\{\gamma\} \tag{1}$$

The added alternative is $x\{\gamma\}$, where $x \in \mathsf{SVars}$. We denote by $\mathsf{SVars}(\gamma)$ the set of variables that occur in γ. We use γ^+ as abbreviations of $\gamma \cdot \gamma^*$.

A variable regex can be matched against a document in multiple ways, or more formally, there can be multiple parse trees showing that a document matches a variable regex. Each such a parse tree naturally associates variables with spans. It is possible, however, that in a parse tree a variable is not associated with any span, or is associated with multiple spans. If every variable is associated with precisely one span, then the parse tree is said to be *functional*. A variable regex is called a *regex formula* if it has only functional parse trees on every input document. An example of a variable regex that is *not* a regex formula is $(x\{\mathtt{a}\})^*$, because a match against $\mathtt{aa}$ assigns x to two spans. We refer to Fagin et al. [14] for the full formal definition of regex formulas. By RGX we denote the class of regex formulas. A regex formula γ is naturally viewed as representing a spanner, and by $[\![\gamma]\!]$ we denote the spanner that is represented by γ. Following are examples of spanners represented as regex formulas.

EXAMPLE 2.3. In the regex formulas of our running examples we will use the following conventions.

- `[a-z]` denotes the disjunction $\mathtt{a} \vee \cdots \vee \mathtt{z}$;
- `[A-Z]` denotes the disjunction $\mathtt{A} \vee \cdots \vee \mathtt{Z}$;
- and `[a-zA-Z]` denotes the disjunction `[a-z]` $\vee$ `[A-Z]`.

We now define several variable regexes that we will use throughout the paper.

The following regex formula extracts complete words (tokens) from text. (Note that this is a simplistic extraction for the sake of presentation.)

$$\gamma_{\mathsf{tkn}} := \big(\epsilon \vee (\Sigma^* \cdot \_)\big) \cdot x\{[\mathtt{a\text{-}zA\text{-}Z}]^+\} \cdot \Big(\big((, \vee \_) \cdot \Sigma^*\big) \vee \epsilon\Big)$$

When applied to the document $\mathbf{d}$ of Figure 1, the resulting spans include $[1,7\rangle$, $[8,12\rangle$, $[13,19\rangle$ and so on.

$[\![\rho_{\mathsf{stt}}]\!](\mathbf{d})$	
	x
μ_1	$[21,28\rangle$
μ_2	$[30,40\rangle$
μ_3	$[60,68\rangle$

$[\![\rho_{\mathsf{loc}}]\!](\mathbf{d})$			
	x_1	x_2	y
μ_5	$[13,19\rangle$	$[21,28\rangle$	$[13,28\rangle$
μ_4	$[21,28\rangle$	$[30,40\rangle$	$[21,40\rangle$
μ_6	$[46,58\rangle$	$[60,68\rangle$	$[46,68\rangle$

Figure 2: Results of spanners in the running example

The following variable regex extracts spans that begin with a capital letter.

$$\gamma_{\mathsf{1Cap}} := \Sigma^* \cdot x\{[\mathtt{A\text{-}Z}] \cdot \Sigma^*\} \cdot \Sigma^*$$

When applied to the document $\mathbf{d}$ of Figure 1, the resulting spans include $[1,7\rangle$, $[1,3\rangle$, $[13,19\rangle$, $[13,20\rangle$, and so on.

The following regex formula extracts all the spans that span names of US states. For simplicity, we include just the three in Figure 1. For readability, we omit the concatenation symbol $\cdot$ between two alphabet symbols.

$$\gamma_{\mathsf{stt}} := \Sigma^* \cdot x\{\mathtt{Georgia} \vee \mathtt{Virginia} \vee \mathtt{Washington}\} \cdot \Sigma^*$$

When applied to the document $\mathbf{d}$ of Figure 1, the resulting spans are $[21,28\rangle$, $[30,40\rangle$, and $[60,68\rangle$.

The following regex formula extracts all the triples (x_1, x_2, y) of spans such that the string "`,_`" separates between x_1 and x_2, and y is the span that starts where x_1 starts and ends where x_2 ends.

$$\gamma_{,\_} := \Sigma^* \cdot y\{x_1\{\Sigma^*\} \cdot \mathtt{,\_} \cdot x_2\{\Sigma^*\}\} \cdot \Sigma^*$$

Let $\mathbf{d}$ be the document of Figure 1, and let V be the set $\{x_1, x_2, y\}$ of variables. The $(V, \mathbf{d})$-tuples that are obtained by applying $\gamma_{,\_}$ to $\mathbf{d}$ map (x_1, x_2, y) to triples like $([13,19\rangle, [21,28\rangle, [13,28\rangle)$, and in addition, triples that do not necessarily consist of full tokens, such as the triple $([9,19\rangle, [21,23\rangle, [9,23\rangle)$. $\square$

We denote by REG the class of expressions in the closure of RGX under union ($\cup$), projection ($\pi_{\mathbf{x}}$ where $\mathbf{x}$ is a sequence of variables) and natural join ($\bowtie$). Note that natural join is based on span equality, not on string equality, since our relations contain spans. A spanner is *regular* if it is definable in REG. Fagin et al. [14] proved that the class of regular spanners is closed under the difference operator. As usual, by $[\![\rho]\!]$ we denote the spanner that is represented by the REG expression ρ.

Let ρ be an expression in REG and let $\mathbf{x} = x_1, \dots, x_n$ be a sequence of n distinct variables containing all the variables in $\mathsf{SVars}(\rho)$ (and possibly additional variables). Let $\mathbf{y} = y_1, \dots, y_n$ be a sequence of distinct variables of the same length as $\mathbf{x}$. We denote by $\rho[\mathbf{y}/\mathbf{x}]$ the expression ρ' that is obtained from ρ by replacing every occurrence of x_i with y_i. If $\mathbf{x}$ is clear from the context, then we may write just $\rho[\mathbf{y}]$.

EXAMPLE 2.4. Using γ_{tkn}, γ_{stt} and $\gamma_{,\_}$ from Example 2.3, we define several regular spanners.

- The spanner ρ_{stt} extracts all the tokens that are names of US states: $\rho_{\mathsf{stt}} := \gamma_{\mathsf{tkn}} \bowtie \gamma_{\mathsf{stt}}$.
- The spanner ρ_{1Cap} extracts all the tokens beginning with a capital letter: $\rho_{\mathsf{1Cap}} := \gamma_{\mathsf{tkn}} \bowtie \gamma_{\mathsf{1Cap}}$.
- The spanner ρ_{loc} extracts spans of strings like "*city*, *state*": $\rho_{\mathsf{loc}} := \rho_{\mathsf{1Cap}}[x_1/x] \bowtie \rho_{\mathsf{stt}}[x_2/x] \bowtie \gamma_{,\_}$.

The results of applying the spanners $[\![\rho_{\mathsf{stt}}]\!]$ and $[\![\rho_{\mathsf{loc}}]\!]$ to the document $\mathbf{d}$ of Figure 1 are in Figure 2. $\square$

Fagin et al. [14] proved the following.

THEOREM 2.5. [14] *A spanner is definable in* RGX *if and only if it is regular and hierarchical.*

Both RGX and REG are spanner representation systems that we shall study. In addition, in our proofs we shall later discuss other spanner representation systems, including ones based on automata and ones that extend the system of regular spanners.

3. EXTRACTION PROGRAMS

In the previous section we introduced spanners, along with the representation systems RGX and REG. Here we will use spanners as building blocks for specifying what we call *extraction programs* that involve direct extraction of relations, specification of inconsistencies, and resolution of inconsistencies. We begin with an adaptation of the standard notions of *signature* and *instances* to text extraction.

3.1 Signatures and Instances

A *signature* is a finite sequence $\mathbf{S} = \langle R_1, \dots, R_m \rangle$ of distinct *relation symbols*, where each R_i has an arity $a_i > 0$. In this work, the *data* is a document $\mathbf{d}$, and entries in the instances of a signature are spans of $\mathbf{d}$. Formally, for a signature $\mathbf{S} = \langle R_1, \dots, R_m \rangle$ and a document $\mathbf{d} \in \Sigma^*$, a $\mathbf{d}$-*instance* (*over* $\mathbf{S}$) is a sequence $\langle r_1, \dots, r_m \rangle$, where each r_i is a relation of arity a_i over $\mathsf{Spans}(\mathbf{d})$; that is, r_i is a subset of $\mathsf{Spans}(\mathbf{d})^{a_i}$. A $\mathbf{d}$-*fact* (*over* $\mathbf{S}$) is an expression of the form $R(s_1, \dots, s_a)$, where R is a relation symbol of $\mathbf{S}$ with arity a, and each s_i is a span of $\mathbf{d}$. If f is a $\mathbf{d}$-fact $R(s_1, \dots, s_a)$ and I is a $\mathbf{d}$-instance, both over the signature $\mathbf{S}$, then we say that f is *a fact of I* if $(s_1, \dots, s_a)$ is a tuple in the relation of I that corresponds to R. For convenience of notation, we identify a $\mathbf{d}$-instance with the set of its facts.

EXAMPLE 3.1. The signature $\mathbf{S}$ we will use for our running example consists of three relation symbols:

- The unary relation symbol Loc that stands for *location*;
- The unary relation symbol Per that stands for *person*;
- The binary relation symbol PerLoc that associates persons with locations.

We continue with our running example. Figure 3 shows a $\mathbf{d}$-instance over $\mathbf{S}$, where $\mathbf{d}$ is the document of Figure 1. This instance has 12 facts, and for later reference we denote them by $f_1, \dots, f_{12}$. Note that there are quite a few mistakes in the table (e.g., the annotation of `Virginia` as a person by fact f_9); in the next section we will show how these are dealt with in the framework of this paper. □

3.2 Conflicts and Priorities

The database research community has established the concept of *repairs* as a mechanism for handling inconsistencies in a declarative fashion [3]. Conventionally, *denial constraints* are specified to declare sets of facts that cannot co-exist in a consistent instance. A *minimal repair* of an inconsistent instance is a consistent subinstance that is not properly contained in any other consistent subinstance. Each minimal repair is then viewed as a *possible world* (and the notion of *consistent query answers* can then be applied). Here, we will adapt the concept of denial constraints to our setting. In Section 5 we will illustrate the generality of these constraints w.r.t. conflict resolutions that take place in real life. However, in the world of IE the minimal repairs are not necessarily all equal. In fact, in every example we are aware of, the developer has a clear preference as to which facts to exclude when a denial constraint fires. Therefore, instead of the traditional repairs, we will use the notion of *prioritized repairs* of Staworko et al. [38] that extends repairing by incorporating priorities.

Loc	
f_1	$[13, 28\rangle$
f_2	$[21, 40\rangle$
f_3	$[46, 68\rangle$

Per	
f_4	$[1, 7\rangle$
f_5	$[13, 19\rangle$
f_6	$[21, 28\rangle$
f_7	$[30, 40\rangle$
f_8	$[46, 58\rangle$
f_9	$[60, 68\rangle$

PerLoc		
f_{10}	$[1, 7\rangle$	$[13, 28\rangle$
f_{11}	$[1, 7\rangle$	$[46, 68\rangle$
f_{12}	$[30, 40\rangle$	$[46, 68\rangle$

Figure 3: A $\mathbf{d}$-instance I over the signature of the running example

Let $\mathbf{S}$ be a signature, let $\mathbf{d}$ be a document, and let I be a $\mathbf{d}$-instance over $\mathbf{S}$. A *conflict hypergraph* for I is a hypergraph H over the facts of I; that is, $H = (V, E)$ where V is the set of I's facts and E is a collection of hyperedges (subsets of V). Intuitively, the hyperedges represent sets of facts that together are in conflict. A *priority relation* for I is a binary relation $\succ$ over the facts of I. If f and f' are facts of I, then $f \succ f'$ means intuitively that f is preferred to f'. A *repair* of I is a subinstance of I that does not contain any hyperedge of H. To accommodate priorities in cleaning, we use the notion of *Pareto optimality* [38]: a repair J is an *improvement* of a repair J' if there is a fact $f \in J \backslash J'$ such that $f \succ f'$ for all $f' \in J' \backslash J$; an *optimal repair* is a repair that has no improvement.

COMMENT 3.2. Another notion of optimality in [38] is *global optimality*, where a repair J is an *improvement* of a repair J' if for every fact $f' \in J' \backslash J$ there is a fact $f \in J \backslash J'$ such that $f \succ f'$. As before, an *optimal repair* is a repair that has no improvement. When *(1)* all conflicts are binary and *(2)* the priority relation includes all the pairs in conflict, then it follows from results in [38] that if we also make the natural assumption of acyclicity of the priority relation, we have that global optimality and Pareto optimality coincide. Assumptions *(1)* and *(2)* hold in all our results that involve optimal repairs, except for Theorem 4.3 where the same proof works for both notions of optimality. □

EXAMPLE 3.3. Recall the instance I of our running example (Figure 3). Figure 4 shows both a conflict hypergraph (which is a graph in this case) and a priority relation over I. Specifically, the figure has two types of edges. Dotted edges (with small arrows)

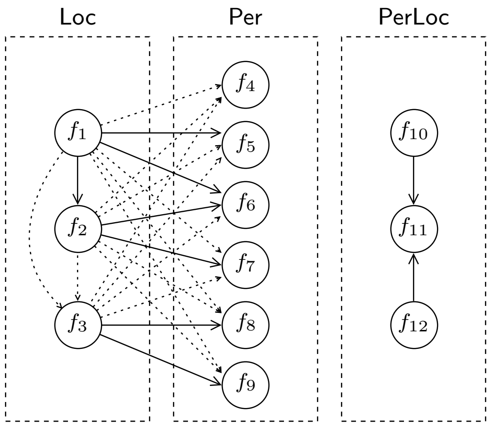

Figure 4: A conflict graph with priorities in the running example

Loc	
f_1	$[13, 28\rangle$
f_3	$[46, 68\rangle$

Per	
f_4	$[1, 7\rangle$
f_7	$[30, 40\rangle$

PerLoc		
f_{10}	$[1, 7\rangle$	$[13, 28\rangle$
f_{12}	$[30, 40\rangle$	$[46, 68\rangle$

Figure 5: A $\mathbf{d}$-instance J_3 over the signature of the running example

define priorities, where $f_i \rightarrow f_j$ denotes that $f_i \succ f_j$. Later, we shall explain the preferences (such as $f_1 \succ f_4$). Solid edges (with bigger arrows) define both conflicts and priorities: $f_i \rightarrow f_j$ denotes that $\{f_i, f_j\}$ is an edge of the conflict hypergraph, and that $f_i \succ f_j$.

Consider the following sets of facts.

- $J_1 = \{f_2, f_3, f_4, f_5, f_{11}\}$;
- $J_2 = \{f_1, f_3, f_4, f_7, f_{11}\} = (J_1 \cup \{f_1, f_7\}) \setminus \{f_2, f_5\}$;
- $J_3 = \{f_1, f_3, f_4, f_7, f_{10}, f_{12}\} = (J_2 \cup \{f_{10}, f_{12}\}) \setminus \{f_{11}\}$.

Observe that each J_i is a repair of I. The repair J_2 is an improvement of J_1, since both $f_1 \succ f_2$ and $f_1 \succ f_5$ hold. The repair J_3 is an improvement of J_2, since $f_{10} \succ f_{11}$ (and $f_{12} \succ f_{11}$). Note that J_3 is not an improvement of J_1, since no fact in J_3 is preferred to both f_2 and f_{11}. Thus, "is an improvement of" is not transitive. The reader can verify that J_3 is an optimal repair. In fact, it can easily be verified (and it also follows from Theorem 4.3 below) that J_3 is the *unique* optimal solution. The instance J_3 is depicted in Figure 5. $\square$

3.3 Denial Constraints and Priority Generating Dependencies

We now discuss the syntactic declaration of conflicts and priorities.

To specify a conflict hypergraph at the signature level (i.e., to say how to define the conflict hypergraph for every instance), we use the formalism of denial constraints. Let $\mathbf{S}$ be a signature, and let $\mathcal{R}$ be a spanner representation system. A *denial constraint* in $\mathcal{R}$ (*over* $\mathbf{S}$), or just $\mathcal{R}$*-dc* (or simply *dc*) for short, has the form $\forall \mathbf{x}[P \rightarrow \neg\Psi(\mathbf{x})]$, where $\mathbf{x}$ is a sequence of variables in SVars, P is a spanner specified in $\mathcal{R}$ with all variables in $\mathbf{x}$, and Ψ is a conjunction of atomic formulas $\varphi(\mathbf{x})$ over $\mathbf{S}$. (Note that an *atomic formula* φ is an expression of the form $R(x_1, \dots, x_a)$, where R is an a-ary relation symbol in $\mathbf{S}$.) We usually omit the universal quantifier, and specify a dc simply by $P \rightarrow \neg\Psi(\mathbf{x})$.[2]

EXAMPLE 3.4. We now define dcs in our running example. We denote by $\mathsf{precede}$ the regex formula $\Sigma^* \cdot x\{\Sigma^*\} \cdot \Sigma^* \cdot y\{\Sigma^*\} \cdot \Sigma^*$. Hence, $\mathsf{precede}$ states that x terminates before y begins. We denote by $\mathsf{disjoint}$ the regex formula $\mathsf{precede}[x, y] \vee \mathsf{precede}[y, x]$. We denote by $\mathsf{overlap}$ an expression in REG that represents the complement of $\mathsf{disjoint}$. Note that $\mathsf{overlap}$ exists, since regular spanners are closed under complement [14]. Finally, we denote by $\mathsf{overlap}_{\neq}$ an expression in REG that restricts the pairs in $\mathsf{overlap}$ to those (x, y) satisfying $x \neq y$ (i.e., x and y are not the same span). It is easy to verify that $\mathsf{overlap}_{\neq}$ indeed exists.

The following dc, denoted d_{loc}, states that the spans of locations are disjoint.

$$d_{\mathsf{loc}} := \mathsf{overlap}_{\neq}[x, y] \rightarrow \neg\big(\mathsf{Loc}(x) \wedge \mathsf{Loc}(y)\big)$$

[2]We note that instead of being written as $P \rightarrow \neg\Psi(\mathbf{x})$, denial constraints are typically written (as in [38]) in the equivalent form $\neg(P \wedge \Psi(\mathbf{x}))$. In the literature, the premise P is typically taken to be a conjunction of atomic formulas, whereas for us P represents a spanner.

Similarly, the following dc, denoted d_{lp}, states that spans of locations are disjoint from spans of persons.

$$d_{\mathsf{lp}} := \mathsf{overlap}[x, y] \rightarrow \neg\big(\mathsf{Loc}(x) \wedge \mathsf{Per}(y)\big) \quad \square$$

To specify a priority relation $\succ$, we propose what we call here a *priority generating dependency*, or just *pgd* for short. Let $\mathbf{S}$ be a signature, and let $\mathcal{R}$ be a spanner representation system. A pgd in $\mathcal{R}$ (for $\mathbf{S}$) has the form $\forall \mathbf{x}[P \rightarrow (\varphi(\mathbf{x}) \succ \varphi'(\mathbf{x}))]$, where $\mathbf{x}$ is a sequence of variables in SVars, P is a spanner specified in $\mathcal{R}$ with all variables in $\mathbf{x}$, and φ and φ' are atomic formulas over $\mathbf{S}$. Again, we usually omit the universal quantifier and write just $P \rightarrow (\varphi(\mathbf{x}) \succ \varphi'(\mathbf{x}))$.

EXAMPLE 3.5. The following pgd, denoted p_{loc}, states that for spans in the unary relation Loc, spans that start earlier are preferred, and moreover, when two spans begin together, the longer one is preferred.

$$p_{\mathsf{loc}} := \rho[x, y] \rightarrow \big(\mathsf{Loc}(x) \succ \mathsf{Loc}(y)\big)$$

Here, $\rho[x, y]$ is the following expression in REG.

$$\pi_{x,y}\Big(\big(\Sigma^* \cdot x\{z\{\epsilon\} \cdot \Sigma^*\} \cdot \Sigma^*\big) \bowtie \big(\Sigma^* \cdot z\{\epsilon\} \cdot \Sigma^+ \cdot y\{\Sigma^*\} \cdot \Sigma^*\}\big)\Big) \vee \big(\Sigma^* \cdot x\{y\{\Sigma^*\}\Sigma^+\} \cdot \Sigma^*\big)$$

Intuitively, the first disjunct says that x begins before y, because x begins with the empty span z, and y begins strictly after z begins. The second disjunct says that x and y begin together, but x ends strictly after y ends.

The following pgd, denoted p_{lp}, states that all the facts of Loc are preferred to all the facts of Per (e.g., because the extraction made for Loc is deemed more precise). We use the Boolean spanner true that is true on every document.

$$p_{\mathsf{lp}} := \mathsf{true} \rightarrow \big(\mathsf{Loc}(x) \succ \mathsf{Per}(y)\big) \quad \square$$

As we will discuss in Section 5, common resolution strategies translate into a dc and a pgd, such that the dc is binary, and the pgd defines priorities precisely on the facts that are in conflict. To refer to such a case conveniently, we write $P \rightarrow (\varphi(\mathbf{x}) \rhd \varphi'(\mathbf{x}))$ to jointly represent the dc $P \rightarrow \neg(\varphi(\mathbf{x}) \wedge \varphi'(\mathbf{x}))$ and the pgd $P \rightarrow (\varphi(\mathbf{x}) \succ \varphi'(\mathbf{x}))$. We call such a constraint a *denial pgd*.

EXAMPLE 3.6. We denote by $\mathsf{contains}_{\neq}[x, y]$ a regex formula that produces all pairs (x, y) of spans where x strictly contains y. Let $\mathsf{encloses}[z, x, y]$ denote a specification in REG that produces all the triples (z, x, y), such that z begins where x begins and ends where y ends. For presentation sake, we avoid the precise specification of these formulas.

The following denial pgd, denoted dp_{enc}, states that in the relation PerLoc, two facts are in conflict if the span that covers the two elements of the one strictly contains that span of the other; in that case, the smaller span is prioritized (since a smaller span indicates closer relationship between the person and the location).

$$\mathsf{encloses}[z, x, y] \bowtie \mathsf{encloses}[z', x', y'] \bowtie \mathsf{contains}_{\neq}[z', z] \rightarrow \mathsf{PerLoc}[x, y] \rhd \mathsf{PerLoc}[x', y'] \quad \square$$

EXAMPLE 3.7. Consider again the $\mathbf{d}$-instance I of Figure 3. The reader can verify that dcs d_{loc} and d_{lp} from Example 3.4, the pgds p_{loc} and p_{lp} in Example 3.5 and the denial pgd dp_{enc} of Example 3.6, together define the conflicts and priorities discussed in Example 3.3 (Figure 4). $\square$

3.4 Extraction Programs

Let $\mathcal{R}$ be a spanner representation system. An *extraction program in* $\mathcal{R}$, or just $\mathcal{R}$*-program*, for short, is a triple $\langle \mathbf{S}, U, \varphi \rangle$, where $\mathbf{S}$ is a signature, U is a finite sequence $u_1, \dots, u_m$ of *updates*, and φ is an atomic formula over $\mathbf{S}$ (representing the result of the program). There are two types of updates u_i:

1. **CQ updates**. These updates are conjunctive queries of the form $R(y_1, \dots, y_a) :- \alpha_1 \wedge \cdots \wedge \alpha_k$, where R is a relation symbol of $\mathbf{S}$ of arity a, and each α_i is either an atomic formula over $\mathbf{S}$ or a spanner in $\mathcal{R}$. The α_i are called *atoms*. We make the requirement that each y_i occurs in at least one atom.

2. **Cleaning updates**. A cleaning update is an update of the form $\text{CLEAN}(\delta_1, \dots, \delta_d)$, where each δ_i is a dc or a pgd (for convenience, we will also allow denial pgds).

In the program of the following example, we specify an extraction program $\langle \mathbf{S}, U, \varphi \rangle$ using only U along with a special RETURN statement that specifies φ. We then assume that $\mathbf{S}$ consists of precisely the relation symbols that occur in the program.

EXAMPLE 3.8. We now define the REG-program $\mathcal{E}$ of our running example. Intuitively, the goal of the program is to extract pairs (x, y), where x is a person and y is a location associated with x.[3] The signature is, as usual, that of Example 3.1. The sequence U of updates is the following. Note that we are using the notation we established in the previous examples.

1. $\mathsf{Loc}(x) :- \rho_{\mathsf{loc}}[x]$ (see example 2.4)
2. $\mathsf{Per}(y) :- \rho_{\mathsf{1Cap}}[y]$ (see example 2.4)
3. $\text{CLEAN}(d_{\mathsf{loc}}, d_{\mathsf{lp}}, p_{\mathsf{loc}}, p_{\mathsf{lp}})$ (see Examples 3.4 and 3.5)
4. $\mathsf{PerLoc}(x, y) :- \mathsf{Per}(x) \wedge \mathsf{Loc}(y) \wedge \mathsf{precede}[x, y]$ (see Example 3.4)
5. $\text{CLEAN}(dp_{\mathsf{enc}})$ (see Example 3.6)
6. RETURN $\mathsf{PerLoc}(x, y)$

Note that lines 1, 2 and 4 are CQ updates, whereas lines 3 and 5 are cleaning updates. □

We now define the semantics of evaluating an extraction program over a document. Let $\mathcal{E} = \langle \mathbf{S}, U, \varphi \rangle$ be an $\mathcal{R}$-program with $U = \langle u_1, \dots, u_m \rangle$, and let $\mathbf{d} \in \Sigma^*$ be a document. Let $\mathbf{I}_0$ be the singleton $\{I_\varnothing\}$, where $I_\varnothing$ is the empty instance over $\mathbf{S}$. For $i = 1, \dots, m$, we denote by $\mathbf{I}_i$ the result of executing the updates $u_1, \dots, u_i$ as we describe below. Since the cleaning operation can result in multiple instances (optimal repairs), each $\mathbf{I}_i$ is a *set* of $\mathbf{d}$-instances, rather than a single one. For $i > 0$ we define the following.

1. If u_i is the CQ update $R(x_1, \dots, x_a) :- \alpha_1 \wedge \cdots \wedge \alpha_k$, then $\mathbf{I}_i$ is obtained from $\mathbf{I}_{i-1}$ by adding to each $I \in \mathbf{I}_{i-1}$ all the facts (over R) that are obtained by evaluating the CQ over I.

2. If u_i is the cleaning $\text{CLEAN}(\delta_1, \dots, \delta_d)$, then $\mathbf{I}_i$ is obtained from $\mathbf{I}_{i-1}$ by replacing each $I \in \mathbf{I}_{i-1}$ with all the optimal repairs of I, as defined by the conflict hypergraph and priorities implied by all the δ_j.

[3]In real life, such a program would of course be much more involved; here it is extremely simplistic, for the sake of presentation.

Recall that a spanner is a function that maps a document into a $(V, \mathbf{d})$-relation (see Section 2). An extraction program acts similarly, except that a document is mapped into a set of $(V, \mathbf{d})$-relations (since it branches into multiple repairs). Later, we are going to investigate cases where the extraction program produces precisely one $(V, \mathbf{d})$-relation, and then we will view the extraction program simply as a spanner. Next, we define the output of an extraction program $\mathcal{E} = \langle \mathbf{S}, U, \varphi \rangle$, where $U = \langle u_1, \dots, u_m \rangle$ and $\varphi = R(x_1, \dots, x_a)$. Let $V = \{x_1, \dots, x_a\}$ be the set of variables in φ. From a $\mathbf{d}$-instance I over $\mathbf{S}$ (that does not involve any variables) we naturally obtain a $(V, \mathbf{d})$-relation (that involves the variables in V): the $(V, \mathbf{d})$-relation consisting of all the assignments $\mu : V \to \mathsf{Spans}(\mathbf{d})$ such that $R(\mu(x_1), \dots, \mu(x_a))$ is a fact in I. We denote this relation by $I[\varphi]$. The *result* $\mathcal{E}(\mathbf{d})$ of evaluating the program $\mathcal{E}$ over the document $\mathbf{d}$ is the set of all the $(V, \mathbf{d})$-relations $I[\varphi]$ with $I \in \mathbf{I}_m$, where $\mathbf{I}_m$ is the result (as defined earlier) of the last update u_m.

EXAMPLE 3.9. Consider again the REG-program $\mathcal{E}$ of Example 3.8. We will now follow the steps of evaluating the program $\mathcal{E}$ on the document $\mathbf{d}$ of our running example (Figure 1). It turns out that, in this example, each $\mathbf{I}_i$ is a singleton, since every cleaning operation results in a unique optimal repair. Hence, we will treat the $\mathbf{I}_i$ as instances.

1. In $\mathbf{I}_1$, the relation Loc is as shown in Figure 3, and the other two relations are empty.
2. In $\mathbf{I}_2$, the relations Loc and Per are as shown in Figure 3, and PerLoc is empty.
3. In $\mathbf{I}_3$, the relations Loc and Per are as shown in Figure 5, and PerLoc is empty. The cleaning process is described throughout Sections 3.2 and 3.3.
4. In $\mathbf{I}_4$, the relations Loc and Per are as in $\mathbf{I}_3$, and PerLoc is as shown in Figure 3.
5. $\mathbf{I}_5$ is the instance shown in Figure 5.

The result $\mathcal{E}(\mathbf{d})$ is the (singleton containing the) $(\{x, y\}, \mathbf{d})$-relation that has two mappings: the first maps (x, y) to $([1, 7\rangle, [13, 28\rangle)$, and the second to $([30, 40\rangle, [46, 48\rangle)$. □

4. PROPERTIES OF REGULAR PROGRAMS

In this section, we discuss some fundamental properties of extraction programs, and focus on the class of REG-programs, which we refer to as *regular programs*.

4.1 Unambiguity

Recall that a spanner maps a document $\mathbf{d}$ into a $(V, \mathbf{d})$-relation, for a set V of variables, while an extraction program maps $\mathbf{d}$ into a *set* of $(V, \mathbf{d})$-relations. The first property we discuss for extraction programs is that of *unambiguity*, which is the property of having a single possible world when the program is evaluated over any given document. Formally, we say that extraction program $\mathcal{E}$ is *unambiguous* if $\mathcal{E}(\mathbf{d})$ is a singleton $(V, \mathbf{d})$-relation for every document $\mathbf{d}$. We may view an unambiguous extraction program $\mathcal{E}$ simply as a specification of spanner.

Let $\mathcal{R}$ be a spanner representation system. An $\mathcal{R}$-program is said to be *non-cleaning* if it does not contain cleaning updates (hence, it consists of only CQ updates). Clearly, if an extraction program is non-cleaning, then it is unambiguous. The following proposition states that in the case where $\mathcal{R}$ is REG or RGX, the non-cleaning $\mathcal{R}$-programs do not have any expressive power beyond the regular spanners. The proof is straightforward from the definitions.

PROPOSITION 4.1. *Let P be a spanner. The following are equivalent:* (1) *P is representable by a non-cleaning* REG*-program,* (2) *P is representable by a non-cleaning* RGX*-program,* and (3) *P is regular.*

The following theorem states that, unfortunately, in the presence of cleaning updates unambiguity cannot be verified for regular extraction programs.

THEOREM 4.2. *Whether a* REG*-program is unambiguous is co-recursively enumerable but not recursively enumerable. In particular, it is undecidable.*

The proof of Theorem 4.2 is an adaptation of the proof of Theorem 4.6 that we later present and discuss in detail.

Let I be a $\mathbf{d}$-instance over a signature $\mathbf{S}$. Let H and $\succ$ be a conflict hypergraph and a priority relation over I, respectively. Staworko et al. [38] give the following sufficient condition for the existence of a single optimal solution (under the assumption that there are no empty hyperedges). Suppose that (1) $\succ$ is acyclic, and that (2) $\succ$ is total on every hyperedge of H. Then there is exactly one optimal repair. We have obtained an improvement of this result. We say that $(\succ, H)$ satisfies the *minimum property* if every hyperedge h of H contains a minimum element, that is, an element a such that $b \succ a$ for every member of h other than a. Intuitively, for every conflict, the minimum element is a natural candidate to remove to break the conflict. It is clear that, in the presence of acyclicity (and the absence of empty hyperedges), totality on every hyperedge is strictly more restrictive than our minimum property. We have the following.

THEOREM 4.3. *Let I be a $\mathbf{d}$-instance over a schema $\mathbf{S}$. Let H and $\succ$ be a conflict hypergraph and a priority relation over I, respectively. Suppose that (1) $\succ$ is acyclic, and that (2) the pair $(\succ, H)$ satisfies the minimum property. Then there is exactly one optimal repair.*

The proof of Theorem 4.3 goes by way of contradiction; specifically, it shows how, in the presence of (1) and (2), when given two distinct optimal repairs we can construct an improvement of one of the two (contradicting its optimality). The theorem suggests the following condition for when a cleaning update does not introduce ambiguity. Let u be a cleaning update. We say that u is *acyclic* if, for every document $\mathbf{d}$ and $\mathbf{d}$-instance I over $\mathbf{S}$, the priority relation implied by the pgds of u is acyclic. We say that u is *minimum generating* if, for every document $\mathbf{d}$ and $\mathbf{d}$-instance I over $\mathbf{S}$, for the priority relation $\succ$ implied by the pgds of u and the conflict hypergraph H implied by u, we have that $(\succ, H)$ satisfies the minimum property. As an example, if u consists of only denial pgds, then u is minimum generating (since the hyperedges are all of size 2). From Theorem 4.3 we conclude the following.

COROLLARY 4.4. *Let $\mathcal{E}$ be an $\mathcal{R}$-program for some spanner representation system $\mathcal{R}$. If every cleaning update of $\mathcal{E}$ is acyclic and minimum generating, then $\mathcal{E}$ is unambiguous.*

The good news is that testing whether a regular cleaning update is minimum generating is a decidable problem.

THEOREM 4.5. *Whether a cleaning update in* REG *is minimum generating is decidable.*

Establishing finer complexity classes for this and later decidability results is the subject of future work. In the proof of Theorem 4.5, we show how to construct, from a regular cleaning update u, a regular spanner P, represented as a *variable-set automaton* [14], such that u is minimum generating if and only if P is empty. We then use results of Fagin et al. [14] that imply a procedure to test whether a given variable-set automaton is empty.

Unfortunately, acyclicity is an undecidable property of regular cleaning updates. Moreover, undecidability remains even if the update consists of a single denial pgd.

THEOREM 4.6. *Whether a denial pgd in* REG *is acyclic is co-recursively enumerable but not recursively enumerable. In particular, it is undecidable.*

In Section 4.1.1 we discuss our proof of Theorem 4.6.

What about the decidability of the two conditions (1) and (2) of Theorem 4.3 together? Those two together are also undecidable, since as we observed, denial pgds are automatically minimum generating, and so Theorem 4.6 implies that deciding whether a denial pgd in REG satisfies conditions (1) and (2) of Theorem 4.3 together is undecidable.

4.1.1 Proof of Undecidability

The proof of Theorem 4.2 is a variation of the proof of Theorem 4.6. To prove the latter, we use results on the emptiness of multi-head automata [23], and prove an intermediate result of independent interest. To present this intermediate result, we need some definitions.

Let k be a natural number. A *nondeterministic two-way k-head finite automaton* (or just *2NFA(k)* for short) is a tuple of the form $\langle \Theta, Q, \delta, q_0, F \rangle$ where Θ is a finite alphabet, Q is a finite set of states, δ is a transition function that maps each element in $Q \times (\Theta \cup \{\vdash, \dashv\})^k$ to a subset of $Q \times \{\mathsf{l}, \mathsf{r}, \mathsf{h}\}^k$, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of accepting states. The symbols $\vdash$ and $\dashv$ are the left and right endmarkers of input strings, respectively, and are not in Θ. If the image of δ consists of only singletons and the empty set, then A is *deterministic* and we may replace "NFA" with "DFA."

The semantics of a 2NFA(k) $A = \langle \Theta, Q, \delta, q_0, F \rangle$ is as follows. Let $\mathbf{s} \in \Theta^*$ be a string, and denote it as $s_1, \ldots, s_n$. The machine A does not read $\mathbf{s}$ directly, but rather the augmentation $\vdash\mathbf{s}\dashv$ (i.e., endpoints are marked), which we denote by $s_0, s_1, \ldots, s_n, s_{n+1}$. The machine has a single current state and k heads, each on some symbol in $\vdash\mathbf{s}\dashv$. In the initial configuration, the state is q_0 and all the heads are on s_0. The transition $(q', t_1, \ldots, t_k) \in \delta(q, a_1, \ldots, a_k)$, where $t_i \in \{\mathsf{l}, \mathsf{r}, \mathsf{h}\}$ and $a_i \in \Theta \cup \{\vdash, \dashv\}$, means that if the current state is q and the ith head is on the symbol a_i, then in a possible next configuration the state is q' and each head i acts according to t_i: moves one left (l), moves one right (r), or holds in place (h). If t_i moves the head outside the input, it is treated as h. A *halting* configuration is one without possible next configurations (because $\delta(q, a_1, \ldots, a_k) = \emptyset$). A halting configuration is *accepting* if its state q is in F; otherwise, the halting configuration is *rejecting*. We say that A *accepts* a sting $\mathbf{s}$ if there is a run (legal sequence of configurations) on $\vdash\mathbf{s}\dashv$ that starts with the initial configuration and ends with an accepting configuration.

Let $\mathbf{C}$ be a class of multi-head automata (e.g., 2DFA(2)). The *finite mortality problem* for $\mathbf{C}$ is the following. Given an automaton $A \in \mathbf{C}$, determine whether A terminates on every input string from every possible configuration; in that case, we say that *A has no immortal configurations*. We can prove the following lemma.[4]

[4]In private communication with the authors, Martin Kutrib independently proved this result.

LEMMA 4.7. *For 2DFA(2), the finite mortality problem is co-recursively enumerable but not recursively enumerable. In particular, it is undecidable.*

The proof of Lemma 4.7 uses a result by Holzer et al. [23], stating that the emptiness problem (i.e., whether no string is accepted) is undecidable for 1DFA(2). Specifically, the proof reduces the emptiness problem for 1DFA(2) to the finite mortality problem for 2DFA(2), where the main idea is to "restart" the run of the given 1DFA(2) whenever it enters an accepting configuration.

Lemma 4.7 is used in the proof of Theorem 4.6 as follows. From an input 2DFA(2) A, we construct a binary regular spanner $\rho[x, y]$. Given a document $\mathbf{d}$, we view the spans in $[\![\rho]\!](\mathbf{d})$ as representing configurations of A over some input string, and $(a, b) \in [\![\rho]\!](\mathbf{d})$ implies that b is the next configuration following a. We then define the denial pgd $\rho[x, y] \rightarrow (R(y) \rhd R(x))$, where R is a unary relation symbol. It then follows that A has no immortal configurations if and only if this pgd is acyclic.

4.2 Disposability

Next, we address the question of whether cleaning updates increase the expressive power of extraction programs.

Let $\mathcal{R}$ be a spanner representation system. A cleaning update u defined in $\mathcal{R}$ is said to be *$\mathcal{R}$-disposable* if the following holds: for every $\mathcal{R}$-program $\mathcal{E}$ that has u as its single cleaning update, there exists a non-cleaning $\mathcal{R}$-program that is equivalent to $\mathcal{E}$. Of course, we have the following.

PROPOSITION 4.8. *Let $\mathcal{R}$ be a spanner representation system and let $\mathcal{E}$ be an $\mathcal{R}$-program. If every cleaning update of $\mathcal{E}$ is $\mathcal{R}$-disposable, then $\mathcal{E}$ is equivalent to a non-cleaning $\mathcal{R}$-program.*

We say that a denial pgd p is *$\mathcal{R}$-disposable* if the cleaning update that consists of only p is $\mathcal{R}$-disposable. The following theorem implies that cleaning updates, and in fact a single acyclic denial pgd, increase the expressive power of regular extraction programs. Recall that a program that uses an acyclic denial pgd as its single cleaning update is unambiguous (Theorem 4.3).

THEOREM 4.9. *There exists an acyclic denial pgd in* REG *that is not* REG*-disposable.*

In the proof of Theorem 4.9, we build a REG-program $\mathcal{E}$ with a single cleaning update u. We then assume, by contradiction, that $\mathcal{E}'$ is a non-cleaning REG-program that is equivalent to $\mathcal{E}$, and then we show how to construct from $\mathcal{E}'$ an NFA that accepts the language $\mathcal{L}$ of all the strings $\mathbf{s}\#\mathbf{t}$ where $\mathbf{s}$ and $\mathbf{t}$ are in $\{0, 1\}^*$, and their lengths are equal and even. But it follows immediately from the literature on regular languages that $\mathcal{L}$ is not regular (hence, no NFA accepts it), and we therefore have a contradiction.

In Section 5, we are going to discuss specific regular cleaning updates that are, in fact, REG-disposable. We will also discuss some general conditions that suffice for REG-disposability.

5. SPECIAL CLEANING STRATEGIES

In this section, we discuss several classes of cleaning strategies that are used in practice. We will show that the strategies in each class are expressible as cleaning updates (in fact, denial pgds) in REG. Moreover, we will prove that all of these cleaning updates are REG-disposable.

5.1 Transitive Denial Pgds

A denial pgd is *transitive* if the relationship "fact f is in conflict with and has priority over fact g" is transitive. More formally, let p be a denial pgd $P \rightarrow (\varphi(\mathbf{x}) \rhd \varphi'(\mathbf{x}))$ over a schema $\mathbf{S}$. Let I be a $\mathbf{d}$-instance over $\mathbf{S}$, and let f and f' be two facts in I. By $p \models f \rhd f'$ we denote the fact that there is a span assignment for $\mathbf{x}$ that is true on P, and that maps $\varphi(\mathbf{x})$ and $\varphi'(\mathbf{x})$ to f and f', respectively. We say that p is *transitive* if for every $\mathbf{d}$-instance I over $\mathbf{S}$ and for every three facts f_1, f_2 and f_3 in I, if $p \models f_1 \rhd f_2$ and $p \models f_2 \rhd f_3$ then $p \models f_1 \rhd f_3$.

An example of a transitive denial pgd is the *maximal container* denial pgd, which has the form $\mathsf{contains}_{\neq}[x, y] \rightarrow (R(\mathbf{x}) \rhd R(\mathbf{y}))$, where R is a relation symbol, and $\mathbf{x}$ and $\mathbf{y}$ are disjoint sequences of variables that contain x and y, respectively. This denial pgd is among the standard collection of "consolidation" strategies provided by SystemT [7], along with the analogous *minimal contained* (that favors shorter strings, and is expressed by the denial pgd dp_{enc} in Example 3.6), which is also transitive. Another example of a transitive denial pgd that captures a popular strategy is that of *rule priority*, and it has the form $P \rightarrow (R(\mathbf{x}) \rhd S(\mathbf{y}))$, where P is a spanner, and R and S are *distinct* relation symbols. Hence, the rule states that if the condition ρ holds (e.g., some attributes overlap), the fact that is from R is preferred to the fact that is from S (perhaps because the source of R is more trusted). Here, transitivity holds in a vacuous manner. Later in this section we will encounter additional transitive denial pgds.

Next, we give results about transitive denial pgds in REG, even in conjunction with acyclicity. The first result states that transitivity is a decidable property.

THEOREM 5.1. *The following are decidable for a given denial pgd p in* REG. (1) *Determine whether p is transitive;* (2) *Determine whether p is both transitive and acyclic.*

Like in the proof of Theorem 4.5, the proof here shows how to reduce each of the two conditions to the emptiness of variable-set automata.

Recall that every cleaning update that consists of a single acyclic denial pgd is unambiguous. Interestingly, if that denial pgd is a transitive denial pgd in REG, then the cleaning update is also REG-disposable.

THEOREM 5.2. *If p is a transitive denial pgd in* REG*, then p is* REG*-disposable.*

To prove Theorem 5.2 we use the observation that, under the condition of the theorem, the single optimal repair is the one that consists of only the maximal elements under $\rhd$. Then we show how to construct a spanner that selects those maximal elements from the relevant relations.

5.2 JAPE Controls

JAPE [10] is an instantiation of the Common Pattern Specification Language (CPSL) [2], a rule based framework for IE. A JAPE program (or "phase") can be viewed as an extraction program where all the relation symbols are unary. This system has several built-in cleaning strategies called "controls." Here, we will define these strategies in our own terminology—denial pgds.

JAPE provides four controls (in addition to the All control stating that no cleaning is to be applied). These translate to the following denial pgds. Here, R is assumed to be a unary relation in an extraction program.

- Under the Appelt control, $R(x) \rhd R(y)$ holds if *(1)* x and y overlap and x starts earlier than y, *or (2)* x and y start at the same position but x is longer than y. The same strategy is used is also provided by SystemT [7] (as a "consolidator"). This control also involves *rule priority*, which we ignore for now and discuss later.

- The Brill control is similar to Appelt, with the exclusion of option (2); that is, $R(x) \rhd R(y)$ holds if x and y overlap and x starts earlier than y.

- The First control is similar to Appelt with "longer" replaced with "shorter."

- The Once control states that a single fact should remain in R (unless R is empty), which is the one that starts earliest, where a tie is broken by taking the one that ends earliest. Hence, $R(x) \rhd R(y)$ if and only if x is that remaining fact and $x \neq y$.

EXAMPLE 5.3. Suppose that $\Sigma = \{0, 1\}$, and that R is defined by the following regex formula:

$$\Sigma^* \cdot x\{1^+0^+1^+\} \cdot \Sigma^*$$

Now, consider the following two documents:

$$\mathbf{d}^1 = 100110100101 \quad \mathbf{d}^2 = 000110100101$$

Note that the two documents differ only in their first symbol. The spans in R for $\mathbf{d}^1$ are $[1, 5\rangle$, $[1, 6\rangle$, $[4, 8\rangle$, $[5, 8\rangle$, $[7, 11\rangle$, and $[10, 13\rangle$. The spans in R for $\mathbf{d}^2$ are $[4, 8\rangle$, $[5, 8\rangle$, $[7, 11\rangle$, and $[10, 13\rangle$.

- By applying the Appelt control, the spans that remain in R for $\mathbf{d}^1$ are $[1, 6\rangle$ and $[7, 11\rangle$, and the spans that remains in R for $\mathbf{d}^2$ are $[4, 8\rangle$ and $[10, 13\rangle$.

- By applying the Brill control, the spans that remain in R for $\mathbf{d}^1$ are $[1, 5\rangle$, $[1, 6\rangle$ and $[7, 11\rangle$, and the spans that remains in R for $\mathbf{d}^2$ are $[4, 8\rangle$ and $[10, 13\rangle$.

- By applying the First control, the spans that remain in R for $\mathbf{d}^1$ are $[1, 5\rangle$, $[5, 8\rangle$ and $[10, 13\rangle$, and the spans that remains in R for $\mathbf{d}^2$ are $[4, 8\rangle$ and $[10, 13\rangle$.

- By applying the Once control, the span that remains in R for $\mathbf{d}^1$ is $[1, 5\rangle$, and the span that remains in R for $\mathbf{d}^2$ is $[4, 8\rangle$.

As can be seen in the example, the change in the first symbol of the document affects the extracted spans all over the document. □

It is easy to show that each of the above denial pgds is acyclic, and that it can be expressed in REG. For example, the Applet control is presented in Example 3.5 with R being the relation symbol Loc. We can also show the following.

THEOREM 5.4. *Each of the denial pgds that correspond to the four JAPE controls is* REG*-disposable.*

In the case of Once, the proof of Theorem 5.4 is by by using Theorem 5.2 (since the corresponding denial pgd is transitive in a vacuous sense). The challenging part of the theorem is for Appelt, Brill and First. To handle the three cases, we prove Lemma 5.5 below that is of independent interest. We first need a definition.

Let P be a unary spanner. Define P^* to be the spanner Q with $\mathsf{SVars}(P) = \mathsf{SVars}(Q)$, where for each document $\mathbf{d}$, we have that $Q(\mathbf{d})$ consists of those spans $[a, a'\rangle$ such that the following holds: there are indices $a_1 \leqslant \cdots \leqslant a_n$ where $n \geqslant 1$, $a_1 = a$, $a_n = a'$, and $[a_i, a_{i+1}\rangle$ is in $P(\mathbf{d})$ for all $i = 1, \ldots, n-1$. Observe that for all documents $\mathbf{d}$, every empty span $[a, a\rangle$ is in P^*; this is obtained by letting $a' = a$ and $n = 1$.

LEMMA 5.5. *Let P be a unary spanner. If P is regular, then P^* is regular.*

The proof of Lemma 5.5 is by constructing a variable-set automaton that simulates an unbounded number runs of the variable-set automaton that specifies P.

We now sketch a proof that the denial pgd for the Appelt control is REG-disposable. The proofs for the Brill and First denial pgds are similar. Given the unary spanner P, let Q be a regular spanner that gives the strict prefixes of spans from P. By closure of regular spanners under complement [14], the complement Q' of Q is a regular spanner. Now define M to be a regular spanner that gives spans in $P \cap Q'$. So M gives the right-maximal spans from P (which means the spans that are maximal when we ignore possible extensions to the left). Next, let T be the regular spanner that gives spans that contain a starting point for spans of P. Let us denote the complement of T by N. By closure under complement, N is also a regular spanner. Intuitively, N gives the spans that do not contain any starting point for spans of P. Next, define K to be the spanner $(M \cup N)^*$. Intuitively, K gives us a sequence of right-maximal spans, possibly preceded by or followed by spans that do that do not contain any starting point for a span of P. By closure of regular spanners under union and the Kleene star (Lemma 5.5), we know that K is a regular spanner. Finally, define S to be the spanner that gives those spans from M that are preceded by a span from K. Then S is a regular spanner that gives those spans in the result of Appelt cleaning.

5.2.1 Rule Priority

In its general form, Appelt involves rule priority. In JAPE, the priority of a rule is determined by an explicit numerical priority (e.g., `Priority:20`) and, in the case of ties, the position of the rule in the program definition. In our terminology, we have n unary views $R_1, \ldots, R_n$ (ordered by decreasing priority), and we apply the cleaning of Appelt simultaneously to all the views, while in the end remove from each R_i all the spans that occur in $R_1 \cup \cdots \cup R_{i-1}$. We can extend Theorem 5.4 to include priorities, as follows. We first define R as the union of the R_i (which we can do in an extraction program). Next, we apply Appelt, as previously defined, to R, and then join each R_i with R using $R_i(x) :- R_i(x) \wedge R(x)$. Finally, we apply the (transitive) cleaners $\textsc{Clean}(\mathsf{true} \rightarrow R_i(x) \rhd R_j(x))$ for $1 \leqslant i < j \leqslant n$, one by one, in order of increasing i.

5.3 POSIX Disambiguation

Recall from Section 2.3 that a regex formula γ defines a spanner by considering all possible ways that the input document $\mathbf{d}$ can be matched by γ; that is, it considers all possible parse trees of γ on $\mathbf{d}$. Each such parse tree generates a new $(V, \mathbf{d})$-tuple, where $V = \mathsf{SVars}(\gamma)$, in the resulting span relation. In contrast, regular-expression pattern-matching facilities of common UNIX tools, such as `sed` and `awk`, or programming languages such as Perl, Python, and Java, do not construct all possible parse trees. Instead, they employ a *disambiguation policy* to construct only a single parse tree among the possible ones. As a result, a regex formula in these tools always yields a single $(V, \mathbf{d})$-tuple per matched input document $\mathbf{d}$ instead of multiple such tuples.[5]

In this section, we take a look at the POSIX disambiguation policy [19, 24], which is followed by all POSIX compliant tools such as `sed` and `awk`. Formalizations of this policy have been proposed by Vansummeren [39] and Okui and Suzuki [32], and multiple efficient algorithms for implementing the policy are known [8, 26, 32]. We show in particular that the POSIX disambiguation policy can be expressed in our framework as a REG-program that uses only

[5]While our syntax $x\{\gamma\}$ for variable binding is not directly supported in these tools, it can be mimicked through the use of so-called *parenthesized expressions* and *submatch addressing*.

cleaning updates with transitive denial pgds. Other disambiguation policies, such as the first and greedy match policy followed by Perl, Python, and Java (see, e.g. [39] for a description of this policy) are left for future work.

POSIX essentially disambiguates as follows when matching a document $\mathbf{d}$ against regex formula γ.[6] A formal definition may be found in [32, 39]. If γ is one of $\emptyset$, ϵ, or $\sigma \in \Sigma$ then at most one parse tree exists; disambiguation is hence not necessary. If γ is a disjunction $\gamma_1 \vee \gamma_2$, then POSIX first tries to match $\mathbf{d}$ against γ_1 (recursively, using the POSIX disambiguation policy to construct a unique parse tree for this match). Only if this fails it tries to match against γ_2 (again, recursively). If, on the other hand, γ is a concatenation $\gamma_1 \cdot \gamma_2$ then POSIX first determines the longest prefix $\mathbf{d_1}$ of $\mathbf{d}$ that can be matched by γ_1 such that the corresponding suffix $\mathbf{d_2}$ of $\mathbf{d}$ can be matched by γ_2. Then, $\mathbf{d_1}$ and $\mathbf{d_2}$ are recursively matched under the POSIX disambiguation policy by γ_1 and γ_2, respectively, to construct a unique parse tree for γ. If γ is a Kleene closure δ^* and $\mathbf{d}$ is nonempty, then POSIX views γ equivalent to its expansion $\delta \cdot \delta^*$. In line with the rule for concatenation, it hence first determines the longest prefix $\mathbf{d_1}$ of $\mathbf{d}$ that can be matched by δ such that the corresponding suffix $\mathbf{d_2}$ of $\mathbf{d}$ can be matched by δ^*. Then, a unique parse tree for $\mathbf{d}$ against γ is constructed by matching $\mathbf{d_1}$ recursively against δ and $\mathbf{d_2}$ against δ^*. If, on the other hand, $\mathbf{d}$ is empty, then a special parse tree is constructed for this case. The following example illustrates the policy.

EXAMPLE 5.6. Consider $\gamma = x\{(0 \vee 01)\} \cdot y\{(1 \vee \epsilon)\}$ and $\mathbf{d} = 01$. Under the POSIX disambiguation policy, subexpression $x\{(0 \vee 01)\}$ will match as much of $\mathbf{d}$ as possible while still allowing the rest of the expression, namely $y\{(1 \vee \epsilon)\}$, to match the remainder of $\mathbf{d}$. As such, $x\{(0 \vee 01)\}$ will match $\mathbf{d}$ entirely, and $y\{(1 \vee \epsilon)\}$ will match the empty string. We hence bind x to the span $[1, 3\rangle$ and y to $[3, 3\rangle$.

In contrast, when $\gamma = (x\{0\} \cdot y\{(1 \vee \epsilon)\}) \vee (x\{01\} \cdot y\{(1 \vee \epsilon\})$ and $\mathbf{d} = 01$, under the POSIX disambiguation policy we bind x to the span $[1, 2\rangle$ and y to the span $[2, 3\rangle$. □

By $\mathsf{posix}[\gamma]$ we denote the spanner represented by the regex formula γ under the POSIX disambiguation policy; this is the spanner such that $\mathsf{posix}[\gamma](\mathbf{d})$ is empty if $\mathbf{d}$ cannot be matched by γ, and consists of the unique $(V, \mathbf{d})$-tuple resulting from matching $\mathbf{d}$ against γ under the POSIX disambiguation policy otherwise. We can prove the following.

LEMMA 5.7. *For every regex formula γ there exists a* REG-*program $\mathcal{E}$ such that $\mathcal{E}(\mathbf{d}) = \{\mathsf{posix}[\gamma](\mathbf{d})\}$, for every document $\mathbf{d}$. Furthermore, all cleaning updates in $\mathcal{E}$ are of the form* CLEAN(p) *where p is a transitive denial pgd.*

The proof proceeds by induction on γ, showing how to encode the POSIX policy by means of transitive denial pgds.

Since every transitive denial pgd is REG-disposable by Theorem 5.2, we immediately obtain from Lemma 5.7 that the spanner $\mathsf{posix}[\gamma]$ is regular, for every regex formula γ. Moreover, since it is easily verified that $\mathsf{posix}[\gamma]$ is hierarchical, it follows by Theorem 2.5 that $\mathsf{posix}[\gamma]$ is itself definable in RGX by a regex formula δ. While $[\![\gamma]\!]$ may produce many tuples for a given input document, $[\![\delta]\!]$ is always guaranteed to produce only one—corresponding to the tuple defined by the γ-parse constructed by the POSIX disambiguation policy.

THEOREM 5.8. *For every regex formula γ, the spanner* $\mathsf{posix}[\gamma]$ *is definable in* RGX.

[6]For simplicity, we restrict ourselves here to the setting where the entire input is required to match γ. Our results naturally extend to the setting where partial matches of $\mathbf{d}$ against γ are sought.

5.4 Core Spanners

Recall that REG is the closure of RGX under union, projection, and natural join. The class Core of *core spanners* [14] is obtained by adding to the list of operators the *string-equality selection*, denoted $\varsigma^=$. Formally, given an expression ρ in REG and two variables $x, y \in \mathsf{SVars}(\rho)$, the spanner defined by $\varsigma^=_{x,y}(\rho)$ selects all those tuples from ρ in which x and y span equal strings (though x and y can be different spans). The following theorem implies that the results given earlier in this section to not extend to the core spanners.

THEOREM 5.9. *If p is the maximal-container denial pgd or one of the JAPE denial pgds, then p is* not Core-*disposable.*

To prove Theorem 5.9, we use the following definition. Let p be a denial pgd of the form $\rho[x, y] \rightarrow R(x) \rhd S(y)$, where R and S are not-necessarily distinct unary relation symbols. We say that *p favors strict containers* if for all spans a and b, if b is contained in a, and b is neither a prefix or a suffix of a, then $R(a) \rhd S(b)$ is implied by p. Note that all the denial pgds in Theorem 5.9 favor strict containers. It also applies to POSIX, which we did not define formally as a denial pgd. Our proof is by showing that if p favors strict containers, then p is *not* Core-disposable. In particular, if a denial pgd that favors strict containers is Core-disposable, then there is a Boolean core spanner that recognizes the language of all the strings $\mathbf{s}\#\mathbf{t}$ where $\mathbf{s}, \mathbf{t} \in \{0, 1\}^+$ and $\mathbf{s}$ is *not* a substring of $\mathbf{t}$. Then, we use the result of Fagin et al. [14] stating that no such core spanner exists. Note that Theorem 5.9 does not mention any *rule priority* denial pgd (Section 5.1), since this rule is underspecified. Nevertheless, we can show that the cleaning update consisting of the denial pgd $\mathsf{true} \rightarrow (R(x) \rhd S(x))$, where R and S are distinct unary relation symbols, is not Core-disposable.

6. CONCLUSIONS

By incorporating the concept of prioritized repairs, we have generalized the framework of spanners into extraction programs that involve cleaning updates. We showed that existing cleaning policies can be represented, in a unified formalism, as denial pgds in REG. We discussed the problem of unambiguity, and showed that it is undecidable for REG-programs, as is the related problem of testing if a given pgd in is acyclic. We also investigated disposability (i.e., whether a cleaning update can be simulated by CQ updates alone). We showed that cleaning updates in REG (and denial pgds as special cases) are not always REG-disposable, even if the program is unambiguous. Hence, cleaning updates increase the expressive power of unambiguous REG-programs as a representation system for spanners. Finally, we looked at special cases of cleaning updates in REG, namely transitive denial pgds, JAPE controls and POSIX, and showed that they all are REG-disposable. Of course, this does not mean that the programs that simulate these cleaners are of manageable sizes; the complexity of simulating disposable cleaners is a direction left here for future investigation. Another direction derived directly from this work is the challenge of devising a sufficient condition for unambiguity of cleaners, featuring low complexity, and robustness to realistic needs. Finally, we believe that it is of importance to explore the impact of *recursion* on our extraction programs (and in particular, associating an ordinary, order-independent Datalog semantics to our programs), either with or without cleaning updates.

Acknowledgments

We are extremely grateful to Phokion G. Kolaitis for suggestions that had significant impact on the paper. We also thank Martin

Kutrib and Frank Neven for insightful discussions on multi-head automata.

7. REFERENCES

[1] J. Ajmera, H.-I. Ahn, M. Nagarajan, A. Verma, D. Contractor, S. Dill, and M. Denesuk. A CRM system for social media: challenges and experiences. In *WWW*, pages 49–58, 2013.

[2] D. E. Appelt and B. Onyshkevych. The common pattern specification language. In *Proceedings of the TIPSTER Text Program: Phase III*, pages 23–30, Baltimore, Maryland, USA, 1998.

[3] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79, 1999.

[4] E. Benson, A. Haghighi, and R. Barzilay. Event discovery in social media feeds. In *ACL*, pages 389–398, 2011.

[5] L. E. Bertossi, S. Kolahi, and L. V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Theory Comput. Syst.*, 52(3):441–482, 2013.

[6] J. Bleiholder and F. Naumann. Data fusion. *ACM Comput. Surv.*, 41(1), 2008.

[7] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, and S. Vaithyanathan. SystemT: An algebraic approach to declarative information extraction. In *ACL*, pages 128–137, 2010.

[8] R. Cox. Regular expression matching: the virtual machine approach. digression: Posix submatching, December 2009. http://swtch.com/ rsc/regexp/regexp2.html.

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

[10] H. Cunningham, D. Maynard, and V. Tablan. JAPE: a Java Annotation Patterns Engine (Second Edition). Research Memorandum CS–00–10, Department of Computer Science, University of Sheffield, November 2000.

[11] G. DeJong. An overview of the frump system. In W. G. Lehnert and M. H. Ringle, editors, *Strategies for natural language processing*, pages 149–176. Lawrence Erlbaum Associates, 1982.

[12] M. Dylla, I. Miliaraki, and M. Theobald. A temporal-probabilistic database model for information extraction. *PVLDB*, 6(14):1810–1821, 2013.

[13] R. Fagin, B. Kimelfeld, Y. Li, S. Raghavan, and S. Vaithyanathan. Rewrite rules for search database systems. In *PODS*, pages 271–282, 2011.

[14] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Spanners: a formal framework for information extraction. In *PODS*, pages 37–48, 2013.

[15] W. Fan. Dependencies revisited for improving data quality. In *PODS*, pages 159–170, 2008.

[16] W. Fan, H. Gao, X. Jia, J. Li, and S. Ma. Dynamic constraints for record matching. *VLDB J.*, 20(4):495–520, 2011.

[17] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD Conference*, pages 469–480, 2011.

[18] D. A. Ferrucci and A. Lally. Uima: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348, 2004.

[19] G. Fowler. An interpreation of the posix regex standard (2003), 2003. http://gsf.cococlyde.org/download/re-interpretation.tgz.

[20] D. Freitag. Toward general-purpose learning for information extraction. In *COLING-ACL*, pages 404–408, 1998.

[21] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*, pages 149–158, 2009.

[22] R. Grishman and B. Sundheim. Message understanding conference-6: A brief history. In *COLING*, pages 466–471, 1996.

[23] M. Holzer, M. Kutrib, and A. Malcher. Multi-head finite automata: Characterizations, concepts and open problems. In *CSP*, volume 1 of *EPTCS*, pages 93–107, 2008.

[24] Institute of Electrical and Electronic Engineers and the Open group. The open group base specifications issue 7, 2013. IEEE Std 1003.1, 2013 Edition.

[25] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, pages 282–289, 2001.

[26] V. Laurikari. Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology, 2001.

[27] T. R. Leek. Information extraction using hidden markov models. Master's thesis, UC San Diego, 1997.

[28] B. Liu, L. Chiticariu, V. Chu, H. V. Jagadish, and F. Reiss. Automatic rule refinement for information extraction. *PVLDB*, 3(1):588–597, 2010.

[29] S. Ma, W. Fan, and L. Bravo. Extending inclusion dependencies with conditions. *Theor. Comput. Sci.*, 515:64–95, 2014.

[30] A. McCallum, D. Freitag, and F. C. N. Pereira. Maximum entropy markov models for information extraction and segmentation. In *ICML*, pages 591–598, 2000.

[31] F. Niu, C. Ré, A. Doan, and J. W. Shavlik. Tuffy: Scaling up statistical inference in Markov Logic Networks using an RDBMS. *PVLDB*, 4(6):373–384, 2011.

[32] S. Okui and T. Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. In M. Domaratzki and K. Salomaa, editors, *CIAA*, volume 6482 of *Lecture Notes in Computer Science*, pages 231–240, 2010.

[33] H. Poon and P. Domingos. Joint inference in information extraction. In *AAAI*, pages 913–918. AAAI Press, 2007.

[34] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE*, pages 933–942, 2008.

[35] E. Riloff. Automatically constructing a dictionary for information extraction tasks. In *AAAI*, pages 811–816, 1993.

[36] 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.

[37] S. Soderland, D. Fisher, J. Aseltine, and W. G. Lehnert. CRYSTAL: Inducing a conceptual dictionary. In *IJCAI*, pages 1314–1321, 1995.

[38] S. Staworko, J. Chomicki, and J. Marcinkowski. Prioritized repairing and consistent query answering in relational databases. *Ann. Math. Artif. Intell.*, 64(2-3):209–246, 2012.

[39] S. Vansummeren. Type inference for unique pattern matching. *ACM Trans. Program. Lang. Syst.*, 28(3):389–428, 2006.

[40] H. Xu, S. P. Stenner, S. Doan, K. B. Johnson, L. R. Waitman, and J. C. Denny. Application of information technology: Medex: a medication information extraction system for clinical narratives. *JAMIA*, 17(1):19–24, 2010.

[41] H. Zhu, S. Raghavan, S. Vaithyanathan, and A. Löser. Navigating the intranet with high precision. In *WWW*, pages 491–500, 2007.