Declarative Cleaning of Inconsistencies in Information Extraction

RONALD FAGIN, BENNY KIMELFELD, and FREDERICK REISS, IBM Research – Almaden
STIJN VANSUMMEREN, Université Libre de Bruxelles (ULB)

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 unambiguously extract the sought information. For example, during extraction, an IE program could annotate a substring as both an address and a person name. When this happens, the extracted information is said to be *inconsistent*, and some way of removing inconsistencies is crucial to compute the final output. 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 through 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.

CCS Concepts: • **Information systems** → **Data management systems**; **Inconsistent data**; **Information extraction**; • **Theory of computation** → **Formal languages and automata theory**; **Data modeling**; **Database query languages (principles)**; **Logic and databases**; *Models of computation*

ACM Reference Format:
Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2016. Declarative cleaning of inconsistencies in information extraction. ACM Trans. Database Syst. 41, 1, Article 6 (February 2016), 44 pages.
DOI: http://dx.doi.org/10.1145/2877202

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

An abridged version of this article has been published in Proceedings of the *33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'14)* [Fagin et al. 2014].
Benny Kimelfeld is currently affiliated with Technion – Israel Institute of Technology.
Authors' addresses: R. Fagin and F. Reiss, IBM Research Almaden, 650 Harry Road, San Jose, CA 95120-6099; email: {fagin, frreiss}@us.ibm.com; B. Kimelfeld, Technion – Israel Institute of Technology, Haifa, Israel; email: bennyk@gmail.com; S. Vansummeren, Université Libre de Bruxelles, 50, Av. F. Roosevelt, CP 165/15, B-1050 Brussels; Belgium; email: stijn.vansummeren@ulb.ac.be.

DOI: http://dx.doi.org/10.1145/2877202

military applications [Grishman and Sundheim 1996], this task is nowadays pervasive in a plethora of computational challenges (especially those associated with Big Data), including social media analysis [Benson et al. 2011], machine data analysis [Fu et al. 2009], healthcare analysis [Xu et al. 2010], customer relationship management [Ajmera et al. 2013], and indexing for semantic search [Zhu et al. 2007]. Moreover, the importance of IE can only be expected to grow in the future since the analysis of data has opened up to a broader range of users because of the widespread adoption of cloud computing and the availability of scalable analytics platforms.

Broadly speaking, there are two main schools of thought on the realization of IE: *probabilistic* (machine learning based) and *rule-based*. In the probabilistic approach, manually labeled unstructured examples are used to train probabilistic models of extraction. Work in this approach includes rule engineering [Freitag 1998; Riloff 1993; Soderland et al. 1995], rule learning [DeJong 1982; Liu et al. 2010], probabilistic graph models [Lafferty et al. 2001; Leek 1997; McCallum et al. 2000], and other statistical models such as Markov Logic Networks [Niu et al. 2011; Poon and Domingos 2007] and probabilistic databases [Dylla et al. 2013]. In the rule-based approach, human experts define logical rules for performing the extraction. Recent work in this approach includes the engineering of general frameworks for the development and scalable execution of IE programs, such as UIMA [Ferrucci and Lally 2004], the General Architecture for Text Engineering (GATE) [Cunningham 2002], Xlog [Shen et al. 2007], and SystemT [Chiticariu et al. 2010].

The distinction between probabilistic and rule-based IE is somewhat artificial. "Probabilistic" IE systems often have a substantial rule-based component [Chiticariu et al. 2013]. For example, the WHISK system [Soderland 1999] uses supervised machine learning to induce regular expression rules from labeled training examples. Other systems use rules to extract basic features and then use additional rules to construct a graphical model from instances of these features. Similarly, "rule-based" systems often incorporate probabilistic components. For example, SystemT ships with utilities for building dictionaries and regular expressions with unsupervised and supervised machine learning [Chiticariu et al. 2011], and recent work has explored using active learning to improve the precision of AQL rule sets [Liu et al. 2010; Roy et al. 2013] (we shall discuss AQL shortly). In general, though, recent research has tended to emphasize the probabilistic approach; while industrial systems, driven by a need for explainable results and ease of maintenance, have leaned more towards using rules [Chiticariu et al. 2013].

In this article, we focus on the rule-based approach to IE. Two systems that are particularly relevant in this respect are GATE and SystemT. GATE, an open-source project by the University of Sheffield, is an instantiation of the *cascaded finite-state transducers* [Appelt and Onyshkevych 1998]. 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 through applying grammar rules and user-defined Java procedures. SystemT is the IE engine behind IBM InfoSphere BigInsights, which is available for free download as IBM BigInsights Quick Start Edition.[1] SystemT exposes an SQL-like declarative language named *AQL* (Annotation Query Language), along with a query plan optimizer [Reiss et al. 2008] and development tooling [Liu et al. 2010]. 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.

[1]http://www.ibm.com/software/data/infosphere/hadoop/trials.html as of August 2015.

It is a common practice among IE rule developers in these systems to write rules that match overlapping spans of the target text. For example, a rule recognizing person names may annotate multiple spans in the input *"Martin Luther King Jr,"* namely *Martin Luther*, *Luther King, Martin Luther King Jr.,* and so on. Moreover, this text could be part of the larger input *"805 Martin Luther King Jr Way, Berkeley, CA 94709,"* which could be annotated as an address by another rule. The practice of writing rules that match overlapping spans exists for a number of reasons, including the following.

- *Separation of concerns:* Large rule sets are often written by teams of developers. To work in parallel, these developers need to divide large extraction tasks into subtasks and to target each subtask with a separate set of rules. Rules from different subtasks frequently match overlapping regions of the text, either because the rules reference the same intermediate results or because different subtasks produce conflicting interpretations for a given span of text. For example, rules for identifying accounting tables and executive salaries in earnings reports would have many overlapping sub-concepts.
- *Performance and ease of maintenance:* Shorter, less complicated rules are generally easier to maintain, and rule engines deliver much higher throughput on less complicated rules. It is often better practice to use multiple simple rules rather than a single complex one. For example, an extractor for person names would include separate rules for different combinations of features like "word that is highly likely to be a first name" or "word that might be a first name or a last name" rather than attempting to fold all of these combinations into a single rule. The simple rules in such a rule set generally produce overlapping matches, since it is difficult to guarantee disjoint results without compromising simplicity.
- *Incomplete information:* In order to identify high-level entities such as expressions of sentiment, it is common to build rules on top of a set of lower-level features. Generic rules that extract these low-level features routinely ascribe multiple roles to a single region of the target text, because these rules do not incorporate information about what role is most relevant in the context of the end-to-end extraction task. For example, a single token could simultaneously be a noun, part of a noun phrase, the object of a preposition, and the name of a product.
- *Recall and precision:* A common strategy to improve the recall of an extractor is to include rules that capture broad categories of patterns. For example, an extractor for company earnings announcements may look for phrases in the form "X announced earnings," where X can be any word. Such a rule helps to protect against cases where the extractor does not correctly identify X as a company. For improving precision, on the other hand, an extractor needs high-precision rules, such as a rule that matches phrases in the form "Y announced earnings on D," where Y is definitely a company name and D is definitely a date. A rule set that includes both high-recall and high-precision rules will have conflicts between the results of these rules.

Depending on the particular application domain, it is nevertheless often undesirable to have overlapping spans annotated by multiple rules in the final output. In our Martin Luther King example above, it is reasonable to assume, for example, that no span should be annotated both as an address and a person. If there is such a span, then we say that the extracted information is *inconsistent*. The pervasiveness of potentially overlapping rules means that logic for resolving conflicts among spans is a key part of any large IE rule set. All of the major rule-based systems for IE provide facilities for conflict resolution. Those systems provide resolution either explicitly, as in CPSL, GATE/JAPE, and SystemT, or implicitly, as in WHISK. 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"

[Appelt and Onyshkevych 1998]. GATE/JAPE, which implements CPSL, generalizes this "best match" 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 of JAPE the annotation procedure (transducer) scans the document left to right; 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 at a specific location, all the annotations that begin there are retained; but after that, scanning continues after the longest annotation. In AQL, the IE extraction language of SystemT, 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 problem is that each system provides a collection of ad hoc resolvers that apply very specific policies (e.g., remove contained, remove contains, left-to-right, etc.). Ideally, we would like to generalize this feature of IE systems and allow developers to define their own policies. Nevertheless, it is not clear how to do so in a way that avoids detailed and imperative coding. As evidence, the policies of GATE/JAPE are described in the manual by means of a procedure that iterates over the spans. In this article, we propose to do IE conflict resolution in a declarative manner by specifying two concepts: conflicts and priorities. We do so by adopting the framework of Staworko et al. [2012] that was proposed in the context of inconsistent databases.

Overview of Our Approach

Similarly to inconsistencies in IE, databases also often contain inconsistent data, 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 in relational databases [Arenas et al. 1999; Bertossi et al. 2013; Bleiholder and Naumann 2008; Fan 2008; Fan et al. 2011a, 2011b; Ma et al. 2014; Staworko et al. 2012]. Here the identification and capturing of inconsistencies is usually done through the use of specialized constraints and dependencies [Arenas et al. 1999; Fan 2008; Fan et al. 2011a; Ma et al. 2014; Staworko et al. 2012]. A prominent formalism for specifying inconsistencies in this respect are the *denial constraints* [Arenas et al. 1999; Staworko et al. 2012] 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 [Arenas et al. 1999] to formalisms for the declarative specification of conflict resolution by means of prioritized repairs [Staworko et al. 2012] to data cleaning and data fusion tools that introduce domain-specific operators for removing data inconsistencies [Bertossi et al. 2013; Bleiholder and Naumann 2008; Fan et al. 2011b].

The goal of this article is to establish principles for declarative cleaning of inconsistencies in IE programs. We build upon our recent work Fagin et al. [2015], 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 nonrecursive Datalog programs that can use spanners in the premises 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 consistent subinstance (i.e., a subset of facts that satisfies all the constraints) that is not strictly contained in any other such subset [Arenas et al. 1999].

Denial constraints do not provide 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 cleaning policies imply not only which sets of facts are in conflict but also which facts should remain and which facts should be dropped. To accommodate preferences, we adopt the *prioritized repairs* of Staworko et al. [2012], 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 consistent subinstance 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). Observe that if a consistent subinstance J cannot be improved, then J is necessarily a repair (i.e., it is not strictly contained in any consistent subinstance). 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.[2]

Staworko et al. [2012] 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. For example, a pgd can state that if one person span contains another (e.g., the span of *Martin Luther King Jr* contains that of *Martin Luther*), then we prefer the longer one, or if an address span (e.g., *805 Martin Luther King Jr Way, Berkeley, CA 94709*) overlaps with a person span (e.g., *Martin Luther King Jr*), then we prefer the address span. 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 that dictates that the evaluation on a string always results in a single match [Institute of Electrical and Electronic Engineers and The Open Group 2013].

One difference between the ordinary 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

[2]Staworko et al. [2012] 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 article hold true even if we adopt global optimality instead of Pareto optimality.

possible world. This property holds in all of the IE cleaning policies mentioned thus far. So 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 when the input consists of arbitrary denial constraints and pgds in our framework. 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 [Holzer et al. 2008] has an immortal finite configuration.

Staworko et al. [2012] showed that when two conditions both hold, unambiguity is guaranteed. The first condition (which they assumed throughout their article) 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 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 might be by using a potential function, e.g., as in Fagin et al. [2011]). Such conditions are beyond the scope of this article 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 noncleaning (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 [Fagin et al. 2015]), 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 that of 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 that an extraction program that uses only cleaning updates among these 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

Fig. 1. Document $\mathbf{d}$ in the running example.

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* [Fagin et al. 2015].

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 $\mathbf{d}$. 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, and 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, $[1, 1\rangle \neq [2, 2\rangle$). In IE, such a distinction may be crucial if one wishes to extract point positions such as separation between sentences. 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 article, we consider a restricted alphabet for simplicity. Specifically, we fix 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. 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 $\mathbf{d}$ as entries of tuples. For readability, under each span $[i, j\rangle$ we have written the string $\mathbf{d}_{[i,j\rangle}$; but this string is not a part of the model. Observe that the spans in the top 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 set $V \subseteq \mathsf{SVars}$ of variables and a document $\mathbf{d} \in \Sigma^*$, an *assignment* to V (w.r.t. $\mathbf{d}$) is a mapping $\mu : V \to \mathsf{Spans}(\mathbf{d})$ that assigns a span of $\mathbf{d}$ to each variable in V. We similarly define an assignment to a sequence of variables (i.e., we view the sequence as a set). An assignment to V w.r.t. $\mathbf{d}$ is called a $(V, \mathbf{d})$-*tuple* for short. 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. When V and $\mathbf{d}$ are clear from the context, we will refer to $(V, \mathbf{d})$-tuples simply as

$[\![\rho_{\mathsf{stt}}]\!](\mathbf{d})$

	x
μ_1	$[21, 28\rangle$ (Georgia)
μ_2	$[30, 40\rangle$ (Washington)
μ_3	$[60, 68\rangle$ (Virginia)

$[\![\rho_{\mathsf{loc}}]\!](\mathbf{d})$

	x_1	x_2	y
μ_5	$[13, 19\rangle$ (Plains)	$[21, 28\rangle$ (Georgia)	$[13, 28\rangle$ (Plains, Georgia)
μ_4	$[21, 28\rangle$ (Georgia)	$[30, 40\rangle$ (Washington)	$[21, 40\rangle$ (Georgia, Washington)
μ_6	$[46, 58\rangle$ (Westmoreland)	$[60, 68\rangle$ (Virginia)	$[46, 68\rangle$ (Westmoreland, Virginia)

Fig. 2. Results of spanners in the running example.

tuples and to $(V, \mathbf{d})$-relations simply as relations or as span relations. Thus, a spanner associates with each document a relation whose entries are spans of that document.

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. The μ_i's are tuples. For example, μ_5 is the tuple (x_1, x_2, y), where $x_1 = [13, 19\rangle$, $x_2 = [21, 28\rangle$, and $y = [13, 28\rangle$.

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.

A spanner P is *Boolean* if $\mathsf{SVars}(P)$ is empty. In this case, given a document d, the result $\mathsf{SVars}(P)$ is either the singleton set containing the empty tuple (we think of this singleton set as representing "True") or the empty set (which we think of as representing "False"). If $P(\mathbf{d})$ is the singleton set containing the empty tuple, then we say that P *accepts* the document d. The *language accepted by* P is the set of all the documents d that are accepted by P.

2.3. Spanner Representation Systems

By a *spanner representation system* we refer collectively to any manner of specifying spanners through finite objects. In Fagin et al. [2015] we have defined several

representation systems: (1) a representation system based on regular expressions, (2) a representation system based on relational algebra, and (3) a representation system based on special types of automata. In this section, we recall the basic definitions of these systems. The interested reader is referred to Fagin et al. [2015] for a full study of their properties.

2.3.1. Regular Expression Formulas. 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:

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

The added alternative is $x\{\gamma\}$, where $x \in \mathsf{SVars}$, which intuitively means that x is assigned the span corresponding to γ. We denote by $\mathsf{SVars}(\gamma)$ the set of variables that occur in γ. We use γ^+ as abbreviation of $\gamma \cdot \gamma^*$.

The intuitive semantics of variables regexes is as follows (the formal semantics may be found below). A variable regex can be matched against a document in multiple ways, that is, there can be multiple parse trees showing that a document matches a variable regex. Each such 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 `aa` assigns x to two spans.

The formal semantics of variable regexes is as follows. It is based on the notion of a *parse tree*, which intuitively records how a variable regex matches a given document. To define parse trees, first, define a *tree* over an alphabet Λ of labels recursively as follows: if $t_1, \ldots, t_n$ are trees (where $n \geqslant 0$) and $\lambda \in \Lambda$, then $\lambda(t_1 \cdots t_n)$ is a tree. (Note that trees are hence unranked, that is, the number of children per node is not necessarily constant.)

Then let Λ be the alphabet $\Sigma \cup \mathsf{SVars} \cup \{\epsilon, \vee_1, \vee_2, \cdot, {}^*\}$. Let γ be a variable regex, and let $\mathbf{d}$ be a document. We use the following inductive definition. A tree t over the alphabet Λ is a *γ-parse for* $\mathbf{d}$ if one of the following holds:

- $\gamma = \epsilon$, $\mathbf{d} = \epsilon$, and $t = \epsilon()$.
- $\gamma = \sigma \in \Sigma$, $\mathbf{d} = \sigma$, and $t = \sigma()$.
- $\gamma = \gamma_1 \vee \gamma_2$ and $t = \vee_1(t')$ for a γ_1-parse t' for $\mathbf{d}$.
- $\gamma = \gamma_1 \vee \gamma_2$ and $t = \vee_2(t')$ for a γ_2-parse t' for $\mathbf{d}$.
- $\gamma = \gamma_1 \cdot \gamma_2$ and $t = \cdot(t_1 t_2)$ where t_i is a γ_i-parse for $\mathbf{d}_i$ $(i = 1, 2)$ for some documents $\mathbf{d}_1$ and $\mathbf{d}_2$ such that $\mathbf{d} = \mathbf{d}_1\mathbf{d}_2$.
- $\gamma = \delta^*$ and there are documents $\mathbf{d}_1, \ldots, \mathbf{d}_n$ such that $\mathbf{d} = \mathbf{d}_1 \cdots \mathbf{d}_n$, $t = {}^*(t_1 \cdots t_n)$, and each t_i is a δ-parse for $\mathbf{d}_i$ $(i = 1, \ldots, n)$.
- $\gamma = x\{\delta\}$ and $t = x(t_\delta)$ where t_δ is δ-parse for $\mathbf{d}$.

(We note that, in order to facilitate the definition of the POSIX disambiguation in Section 6, the above is a slightly different but equivalent notion of parse tree than the one introduced in Fagin et al. [2015].)

To illustrate, consider $\gamma = x\{(0 \vee 01)\} \cdot y\{(1 \vee \epsilon)\}$. Figure 3 shows two γ-parses for $\mathbf{d} = 01$.

A γ-parse t for $\mathbf{d}$ is *functional on* a set V of span variables if all variables in V occur exactly once in t. If t is functional on V, then t naturally defines a $(V, \mathbf{d})$-tuple as follows. If v is a node of t, then the subtree that is rooted at v parses a span s_v of $\mathbf{d}$. Hence, if t is functional on V, then it defines the $(V, \mathbf{d})$ tuple μ^t such that $\mu^t(x) = s_v$, where v is the unique node of t that is labeled by x.

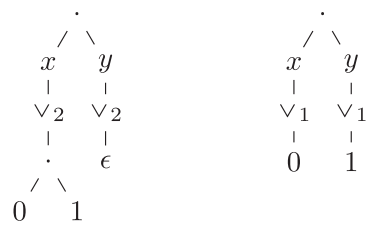

Fig. 3. Two γ-parses for $\mathbf{d} = 01$, with $\gamma = x\{(0 \vee 01)\} \cdot y\{(1 \vee \epsilon)\}$.

In order for a variable regex γ to define a spanner, we require that all of its γ-parses define $(V, \mathbf{d})$ tuples, that is, be functional on $\mathsf{SVars}(\gamma)$, a property we call *functional*. A variable regex γ is *functional* if for every document $\mathbf{d} \in \Sigma^*$ and γ-parse t for $\mathbf{d}$, it holds that t is functional on $\mathsf{SVars}(\gamma)$.

Note that a variable regex can be functional even though it contains multiple occurrences of a variable. An example is the variable regex γ given by $x\{\mathtt{a}\} \vee x\{\mathtt{b}\}$, which has two occurrences of the variable x, although each γ-parse has only one occurrence of x. Although this definition is nonconstructive, functionality of a regex can be tested in polynomial time [Fagin et al. 2015]. The variable regexes that represent spanners are those that are functional, and we call those *regex formulas*.

Let γ be a regex formula . The spanner $[\![\gamma]\!]$ that is represented by γ is the one where $\mathsf{SVars}([\![\gamma]\!])$ is the set $\mathsf{SVars}(\gamma)$, and where $[\![\gamma]\!](\mathbf{s})$ is the span relation $\{\mu^t \mid t \text{ is a } \gamma\text{-parse for } \mathbf{s}\}$.

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

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

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

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

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.

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 $[1, 7\rangle$, $[1, 3\rangle$, $[13, 19\rangle$, $[13, 23\rangle$, 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 x_1 and x_2, and y is the span that starts where x_1 starts and ends

where x_2 ends:

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

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 $([9, 19\rangle, [21, 23\rangle, [9, 23\rangle)$.

2.3.2. Regular Spanners. We denote by RGX the class of variable regex expressions and 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 ($\bowtie$). For example, $\gamma_{\mathsf{tkn}} \bowtie \gamma_{\mathsf{stt}}$, where γ_{tkn} and γ_{stt} refer to the regex formulas defined in Example 2.3, is a REG expression. It is important to note that the standard typing rules for union and projection apply: Union can only be applied to expressions γ_1 and γ_2 if $\mathsf{SVars}(\gamma_1) = \mathsf{SVars}(\gamma_2)$, and $\pi_{\mathbf{x}}$ is only applicable to expression γ if $\mathbf{x} \subseteq \mathsf{SVars}(\gamma)$. Semantically, the spanner $[\![\rho]\!]$ defined by a REG expression ρ is obtained by applying the union, projection, and natural join operators occurring in it to the relations returned by the regex formulas :

$$\begin{aligned}
[\![\pi_{\mathbf{x}}(\gamma_1)]\!](\mathbf{d}) &= \pi_{\mathbf{x}}[\![\gamma_1]\!](\mathbf{d}),\\
[\![\gamma_1 \cup \gamma_2]\!](\mathbf{d}) &= [\![\gamma_1]\!](\mathbf{d}) \cup [\![\gamma_2]\!](\mathbf{d}), \text{ and}\\
[\![\gamma_1 \bowtie \gamma_2]\!](\mathbf{d}) &= [\![\gamma_1]\!](\mathbf{d}) \bowtie [\![\gamma_2]\!](\mathbf{d}).
\end{aligned}$$

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.

Let ρ be an expression in REG and let $\mathbf{x} = x_1, \ldots, x_n$ be a sequence of n distinct variables containing all the variables in $\mathsf{SVars}(\rho)$ (and possibly additional variables). Let $\mathbf{y} = y_1, \ldots, y_n$ 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.

We say that a spanner P is *total on document* $\mathbf{d}$ if $P(\mathbf{d})$ consists of all the $\mathbf{d}$-tuples over $\mathsf{SVars}(P)$. (Note that over a finite set of variables, there are only finitely many $\mathbf{d}$-tuples.) Let $Y \subseteq \mathsf{SVars}$ be a finite set of variables. The *universal spanner over* Y, denoted Υ_Y, is the unique spanner P such that $\mathsf{SVars}(P) = Y$ and P is total on every $\mathbf{d} \in \Sigma^*$. The intersection of two spanners P_1 and P_2 with $\mathsf{SVars}(P_1) = \mathsf{SVars}(P_2)$ is the spanner Q with $\mathsf{SVars}(Q) = \mathsf{SVars}(P_1)$ such that $Q(\mathbf{d}) = P_1(\mathbf{d}) \cap P_2(\mathbf{d})$. The difference is defined similarly. The *complement* of a spanner P is the spanner Q over $\mathsf{SVars}(P)$ such that $Q(\mathbf{d}) = (\Upsilon_Y)(\mathbf{d}) \setminus P(\mathbf{d})$ for all $\mathbf{d} \in \Sigma^*$.

THEOREM 2.5 [FAGIN ET AL. 2015]. *The class of spanners definable in* REG *contains* Υ_Y *for finite subsets* Y *of* SVars*, and is closed under intersection* ($\cap$) *and difference* ($\setminus$)*. Therefore, it is also closed under complement.*

Example 2.6. In some of the proofs that follow, we will use the fact that it is possible to define in REG various relationships among spans.

We denote by precede the regex formula $\Sigma^* \cdot x\{\Sigma^*\} \cdot \Sigma^* \cdot y\{\Sigma^*\} \cdot \Sigma^*$. Hence, precede states that x terminates before y begins. We denote by disjoint the regex formula precede$[x, y] \vee$ precede$[y, x]$. We denote by samespan the regex formula $\Sigma^* \cdot x\{y\{\Sigma^*\}\} \cdot \Sigma^*$. Hence, samespan states that x and y are the same span. We denote by prefix the regex formula $\Sigma^* \cdot y\{x\{\Sigma^*\} \cdot \Sigma^+\} \cdot \Sigma^*$, which states that the span of x is a strict prefix of the span of y. Finally, we denote by contains the regex formula $\Sigma^* \cdot x\{\Sigma^* \cdot y\{\Sigma^*\} \cdot \Sigma^*\} \cdot \Sigma^*$, which states that y is contained in x.

Note that, due to Theorem 2.5, any Boolean combination of these spanners (obtained through $\cup$, $\cap$, $\setminus$, and complement), remains also in REG. For instance, overlap, which states that x and y overlap, can be obtained by taking the complement of disjoint. Therefore, overlap$_{\neq}$, which states that x and y overlap but are distinct can be obtained by the intersection of overlap and the complement of samespan.

For future reference, we also recall the following result.

THEOREM 2.7 [FAGIN ET AL. 2015]. *A spanner is definable in* RGX *if and only if it is regular and hierarchical.*

2.3.3. Variable-Set Automata. A *variable-set automaton* (or *vset-automaton*) is defined to be a tuple (Q, q_0, q_f, δ), where Q is a finite set of *states*, $q_0 \in Q$ is an *initial state*, $q_f \in Q$ is an *accepting state*, and δ is a finite transition relation consisting of triples, each having one of the forms (q, σ, q'), (q, ϵ, q'), $(q, x\vdash, q')$, or $(q, \dashv x, q')$, where $q, q' \in Q$, $\sigma \in \Sigma$, and $x \in$ SVars. We denote by SVars(A) the set of variables that occur in the transitions of A.

Let $\mathbf{s} = \sigma_1 \cdots \sigma_n$ be a string. A *configuration* of a vset-automaton $A = (Q, q_0, q_f, \delta)$, when running on $\mathbf{s}$, is a tuple $c = (q, V, Y, i)$, where $q \in Q$ is the *current state*, $V \subseteq$ SVars(A) is the set of *active variables*, $Y \subseteq$ SVars(A) is the set of *available variables*, and i is an index in $\{1, \ldots, n+1\}$. A *run* $\mathbf{c}$ of A on $\mathbf{s}$ is a sequence $c_0, \ldots, c_m$ of configurations, where $c_0 = (q_0, \varnothing,$ SVars$(A), 1)$, and for $j = 0, \ldots, m-1$ one of the following holds for $c_j = (q_j, V_j, Y_j, i_j)$ and $c_{j+1} = (q_{j+1}, V_{j+1}, Y_{j+1}, i_{j+1})$:

(1) $V_{j+1} = V_j$, $Y_{j+1} = Y_j$, and either (a) $i_{j+1} = i_j + 1$ and $(q_j, s_{i_j}, q_{j+1}) \in \delta$ (ordinary transition) or (b) $i_{j+1} = i_j$ and $(q_j, \epsilon, q_{j+1}) \in \delta$ (epsilon transition).
(2) $i_{j+1} = i_j$ and for some $x \in$ SVars(A), either (a) $x \in Y_j$, $V_{j+1} = V_j \cup \{x\}$, $Y_{j+1} = Y_j \setminus \{x\}$, and we have $(q_j, x\vdash, q_{j+1}) \in \delta$ (variable insert), or (b) $x \in V_j$, $V_{j+1} = V_j \setminus \{x\}$, $Y_{j+1} = Y_j$ and $(q_j, \dashv x, q_{j+1}) \in \delta$ (variable remove).

Note that, in a run, each configuration (q, V, Y, i) is such that V and Y are disjoint. The run $\mathbf{c} = c_0, \ldots, c_m$ is *accepting* if $c_m = (q_f, \varnothing, \varnothing, n+1)$. We let ARuns$(A, \mathbf{s})$ denote the set of all accepting runs of A on $\mathbf{s}$. If $\mathbf{c} \in$ ARuns$(A, \mathbf{s})$, then for each $x \in$ SVars(A) the run $\mathbf{c}$ has a unique configuration $c_b = (q_b, V_b, Y_b, i_b)$ where x occurs in the current version of V (i.e., V_b) for the first time; later than that $\mathbf{c}$ has a unique configuration $c_e = (q_e, V_e, Y_e, i_e)$, where x occurs in the current version of V (i.e., V_e) for the last time; the span $[i_b, i_e\rangle$ is denoted by $\mathbf{c}(x)$. By $\mu^{\mathbf{c}}$ we denote the $\mathbf{s}$-tuple that maps each variable $x \in$ SVars(A) to the span $\mathbf{c}(x)$. The spanner $[\![A]\!]$ that is represented by A is the one where SVars$([\![A]\!])$ is the set SVars(A) and where $[\![A]\!](\mathbf{d})$ is the (SVars(A), $\mathbf{d}$)-relation $\{\mu^{\mathbf{c}} \mid \mathbf{c} \in$ ARuns$(A, \mathbf{s})\}$. We denote by VA$_{\text{set}}$ the set of all variable-set automata.

THEOREM 2.8 [FAGIN ET AL. 2015]. *A spanner is definable in* VA$_{\text{set}}$ *if, and only if, it is definable in* REG. *Moreover, there is a Turing machine that, given an expression ρ in* REG, *constructs a vset-automaton A such that $[\![A]\!] = [\![\rho]\!]$.*

We say that an expression $\rho \in \mathsf{REG}$ is *nonempty* if there exists some $\mathbf{d} \in \Sigma^*$ such that $[\![\rho]\!](\mathbf{d}) \neq \varnothing$. Observe that a Boolean vset-automaton (i.e., a vset-automaton with $\mathsf{SVars}(A) = \varnothing$) is simply an NFA (nondeterministic finite automaton). Hence, by projecting a given expression in REG on the empty set of variables, and then transforming it into a vset-automaton (which is an NFA in that case), we can test the nonemptiness of the expression through a test for the nonemptiness of an NFA. We then get the following.

THEOREM 2.9. *Nonemptiness is decidable for* REG *and for vset-automata.*

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 $\mathbf{S} = \langle R_1, \ldots, R_m \rangle$ of distinct *relation symbols*, where each R_i has an arity $a_i > 0$. In this work, the input to an extraction program is a document $\mathbf{d}$, and entries in the instances of a signature are spans of $\mathbf{d}$. Formally, for a signature $\mathbf{S} = \langle R_1, \ldots, R_m \rangle$ and a document $\mathbf{d} \in \Sigma^*$, a $\mathbf{d}$-*instance* (*over* $\mathbf{S}$) is a sequence $\langle r_1, \ldots, r_m \rangle$, 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 $R(s_1, \ldots, s_a)$, 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 $R(s_1, \ldots, s_a)$ and I is a $\mathbf{d}$-instance, both over the signature $\mathbf{S}$, then we say that f is *a fact of* I if $(s_1, \ldots, s_a)$ 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 4 shows a $\mathbf{d}$-instance over $\mathbf{S}$, where $\mathbf{d}$ is the document of Figure 1. Later, we shall define *extraction programs*. The instance in Figure 4 consists of relations generated by various steps of the extraction program given in Example 3.7. This instance has 12 facts, and for later reference we denote them by $f_1, \ldots, f_{12}$. 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 article.

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 [Arenas et al. 1999]. Conventionally, *denial constraints* are specified to declare sets of facts that cannot coexist in a consistent instance. A *repair* of an inconsistent instance is a consistent subinstance that is not properly contained in any other consistent subinstance. Each 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

Per

f_4	$[1,7\rangle$ (Carter)
f_5	$[13,19\rangle$ (Plains)
f_6	$[21,28\rangle$ (Georgia)
f_7	$[30,40\rangle$ (Washington)
f_8	$[46,58\rangle$ (Westmoreland)
f_9	$[60,68\rangle$ (Virginia)

Loc

f_1	$[13,28\rangle$ (Plains, Georgia)
f_2	$[21,40\rangle$ (Georgia, Washington)
f_3	$[46,68\rangle$ (Westmoreland, Virginia)

PerLoc

f_{10}	$[1,7\rangle$ (Carter)	$[13,28\rangle$ (Plains, Georgia)
f_{11}	$[1,7\rangle$ (Carter)	$[46,68\rangle$ (Westmoreland, Virginia)
f_{12}	$[30,40\rangle$ (Washington)	$[46,68\rangle$ (Westmoreland, Virginia)

Fig. 4. A **d**-instance I over the signature of the running example.

place in real life. However, in the world of IE the 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. [2012] that extends repairing by incorporating priorities.

Let **S** be a signature, let **d** be a document, and let I be a **d**-instance over **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 *consistent subinstance* of I is a subinstance of I that does not contain any hyperedge of H. A *repair* of I is consistent subinstance that is not strictly contained in any other consistent subinstance. To accommodate priorities in cleaning, we use the notion of *Pareto optimality* [Staworko et al. 2012]: a consistent subinstance J is an *improvement* of a consistent subinstance J' if there is a fact $f \in J \setminus J'$ such that $f \succ f'$ for all $f' \in J' \setminus J$; an *optimal repair* is a consistent subinstance that has no improvement. Observe that an optimal repair is necessarily a repair in the ordinary sense.

Example 3.2. Recall the instance I of our running example (Figure 4). Figure 5 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) define priorities, where $f_i \to f_j$ denotes that $f_i \succ f_j$. Later, we shall explain the preferences

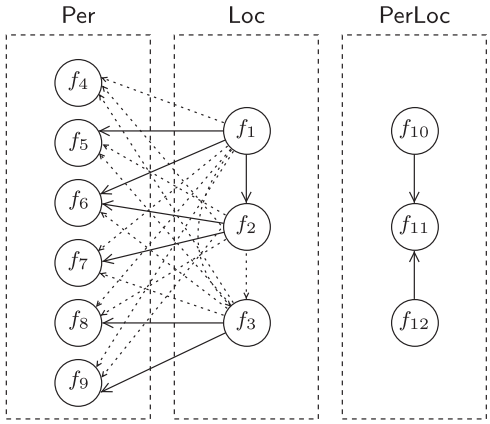

Fig. 5. A conflict graph with priorities in the running example.

Per	
f_4	$[1,7\rangle$ (Carter)
f_7	$[30,40\rangle$ (Washington)

Loc	
f_1	$[13,28\rangle$ (Plains,_Georgia)
f_3	$[46,68\rangle$ (Westmoreland,_Virginia)

PerLoc		
f_{10}	$[1,7\rangle$ (Carter)	$[13,28\rangle$ (Plains,_Georgia)
f_{12}	$[30,40\rangle$ (Washington)	$[46,68\rangle$ (Westmoreland,_Virginia)

Fig. 6. A $\mathbf{d}$-instance J_3 over the signature of the running example.

(such as $f_1 \succ f_4$). Solid edges (with bigger arrows) define both conflicts and priorities: $f_i \to f_j$ 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 consistent. The set J_2 is an improvement of J_1, since both $f_1 \succ f_2$ and $f_1 \succ f_5$ hold. The set 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 6.

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 of its 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 $R(x_1, \ldots, x_a)$, where R is an a-ary relation symbol in $\mathbf{S}$.) We usually omit the universal quantifier and specify a dc simply by $P \rightarrow \neg\Psi(\mathbf{x})$.[3] Semantically, $P \rightarrow \neg\Psi(\mathbf{x})$ is interpreted in the usual first-order-logic sense while viewing P as a predicate that contains all of the tuples in its output; that is, $P \rightarrow \neg\Psi(\mathbf{x})$ is satisfied in a document $\mathbf{d}$ if for every assignment μ to $\mathbf{x}$ (w.r.t. $\mathbf{d}$), if $P(\mathbf{d})$ contains the restriction of μ to $\mathsf{SVars}(P)$, then at least one of the conjuncts of Ψ must be false under μ.

Example 3.3. We now define dcs in our running example. Recall from Example 2.6 the definitions of the regex formulas precede, disjoint, overlap, and $\mathsf{overlap}_{\neq}$. The following dc, denoted d_{loc}, states that the spans of locations are disjoint:

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

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

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

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 of its variables in $\mathbf{x}$, and φ and φ' are atomic formulas over $\mathbf{S}$. Again, we usually omit the universal quantifier and write just $P \rightarrow (\varphi(\mathbf{x}) \succ \varphi'(\mathbf{x}))$. Again, the semantics of $P \rightarrow (\varphi(\mathbf{x}) \succ \varphi'(\mathbf{x}))$ is the obvious one: $f \succ f'$ holds between two ground facts f and f' if there exists an assignment μ to $\mathbf{x}$ such that $P(\mathbf{d})$ contains the restriction of μ to $\mathsf{SVars}(P)$, and f and f' are obtained from $\varphi(\mathbf{x})$ and $\varphi'(\mathbf{x})$, respectively, by replacing every variable x with the span $\mu(x)$.

Example 3.4. 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

[3]We note that instead of being written as $P \rightarrow \neg\Psi(\mathbf{x})$, denial constraints are typically written (as in Staworko et al. [2012]) 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.

the Boolean spanner true that is true on every document.

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

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 3.5. 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's 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).

$$dp_{\mathsf{enc}} := \mathsf{encloses}[z, x, y] \bowtie \mathsf{encloses}[z', x', y'] \bowtie \mathsf{contains}_{\neq}[z', z]$$
$$\rightarrow \mathsf{PerLoc}[x, y] \rhd \mathsf{PerLoc}[x', y'].$$

Example 3.6. Consider again the **d**-instance I of Figure 4. The reader can verify that dcs d_{loc} and d_{lp} from Example 3.3, the pgds p_{loc} and p_{lp} in Example 3.4, and the denial pgd dp_{enc} of Example 3.5 together define the conflicts and priorities discussed in Example 3.2 (Figure 5).

3.4. Extraction Programs

We now formally define *extraction programs*. Such a program takes as input a document, but, unlike a spanner, its output is a *set* of span relations (rather than a single one) since the program may involve cleaning steps that may result in multiple possible repairs.

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 $u_1, \ldots, u_m$ 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 $R(y_1, \ldots, y_a) :- \alpha_1 \wedge \cdots \wedge \alpha_k$, 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 $\textsc{Clean}(\delta_1, \ldots, \delta_d)$, 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 3.7. 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.[4] The signature is, as usual, that of Example 3.1. The sequence U

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

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) $\textsc{Clean}(d_{\mathsf{loc}}, d_{\mathsf{lp}}, p_{\mathsf{loc}}, p_{\mathsf{lp}})$ (see Examples 3.3 and 3.4)
(4) $\mathsf{PerLoc}(x, y) :- \mathsf{Per}(x) \wedge \mathsf{Loc}(y) \wedge \mathsf{precede}[x, y]$ (see Example 3.3)
(5) $\textsc{Clean}(d_{\mathsf{enc}})$ (see Example 3.5)
(6) $\textsc{Return}\ \mathsf{PerLoc}(x, y)$.

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 $U = \langle u_1, \ldots, u_m \rangle$, 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 $i = 1, \ldots, m$, we denote by $\mathbf{I}_i$ the result of executing the updates $u_1, \ldots, u_i$ 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 $R(x_1, \ldots, x_a) :- \alpha_1 \wedge \cdots \wedge \alpha_k$, 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 $\textsc{Clean}(\delta_1, \ldots, \delta_d)$, 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.

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 $U = \langle u_1, \ldots, u_m \rangle$ and $\varphi = R(x_1, \ldots, x_a)$. Let $V = \{x_1, \ldots, x_a\}$ 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 $R(\mu(x_1), \ldots, \mu(x_a))$ 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 3.8. Consider again the REG-program $\mathcal{E}$ of Example 3.7. 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 4, and the other two relations are empty.
(2) In $\mathbf{I}_2$, the relations Loc and Per are as shown in Figure 4, and PerLoc is empty.
(3) In $\mathbf{I}_3$, the relations Loc and Per are as shown in Figure 6, 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 4.
(5) $\mathbf{I}_5$ is the instance shown in Figure 6.

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 $([30, 40\rangle, [46, 68\rangle)$.

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 a spanner.

Let $\mathcal{R}$ be a spanner representation system. An $\mathcal{R}$-program is said to be *noncleaning* if it does not contain cleaning updates (hence, it consists of only CQ updates). Clearly, if an extraction program is noncleaning, then it is unambiguous. The following proposition states that in the case where $\mathcal{R}$ is REG or RGX, the noncleaning $\mathcal{R}$-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 noncleaning* REG*-program,* (2) *P is representable by a noncleaning* 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 is given in Section 4.1.2.

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. [2012] 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.

In all of the examples we have given so far, each conflict involved only two elements, so each hyperedge of the conflict hypergraph was simply an edge of a graph. An example of a conflict involving three elements is due to the following denial constraint: there cannot be three distinct spans A, B, and C such that A and B are contained in C, where A and B represent state names (including abbreviations of state names) and C represents an address. To illustrate this conflict, consider the following address (State Capitol of Arizona):

```
1700 West Washington, Phoenix, AZ 85007.
```

If the entire string is marked as an address, then at most one of `Washington` and `AZ` can be annotated as states, or otherwise we have a conflict. In this case, if we mark the first state as inferior to the second, then that annotation is a minimum element.

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.*

PROOF. Staworko et al. [2012] proved that there exists an optimal repair as long as $\succ$ is acyclic. (The result also requires the assumption that no hyperedge of the conflict hypergraph is empty. This additional assumption holds for us, because it is implied by the minimum property.) The idea of that proof is to define an ordering $\gg$ on the consistent subinstances, where $J \gg J'$ for consistent subinstances J, J' if J is an improvement of J', as defined in Section 3.2. It is then shown in Staworko et al. [2012] that acyclicity of $\succ$ implies acyclicity of $\gg$. A maximal consistent subinstance under $\gg$ is clearly an optimal repair.

Now we need to prove uniqueness. Suppose that (1) and (2) hold but that there are two distinct optimal repairs I_1 and I_2; we shall derive a contradiction. Since $I_1 \neq I_2$, the symmetric difference $D = (I_1 \setminus I_2) \cup (I_2 \setminus I_1)$ is nonempty. Since I is a $\mathbf{d}$-instance, and $\mathbf{d}$ is finite, so are I and D. Since D is finite and nonempty, and since $\succ$ is acyclic, it follows that there is a maximal element a of D (that is, there is no $b \in D$ such that $b \succ a$). Since $a \in D$, we can assume without loss of generality that $a \in I_1$ and $a \notin I_2$. Since $a \notin I_2$, and I_2 is an optimal repair, it cannot be the case that $I_2 \cup \{a\}$ is consistent; therefore, $I_2 \cup \{a\}$ contains a hyperedge h of H. Observe that such a hyperedge h necessarily contains a (since I_2 does not contain h), and every member of h other than a is in I_2. Let $h_1, \ldots, h_r$ be all such hyperedges.

We now show that for each i with $1 \leqslant i \leqslant r$ the element a cannot be a minimum member of h_i. For assume that a is a minimum member of h_i; we shall derive a contradiction. Let $b_1, \ldots, b_k$ be the members of h_i other than a. Since a is a minimum, we have that $b_j \succ a$ for $1 \leqslant j \leqslant k$. It cannot happen that every b_j is in I_1, since then we would have $h_i \subseteq I_1$, because $a \in I_1$. But then I_1 would contain a hyperedge, which is a contradiction, since I_1 is consistent. So select j such that $b_j \notin I_1$. Then $b_j \in D$, since $b_j \in I_2 \setminus I_1$. Since $b_j \succ a$, this contradicts our assumption that a is a maximal element of D. This contradiction shows that, indeed, for each i with $1 \leqslant i \leqslant r$ the element a is not a minimum member of h_i.

Let c_i be a minimum member of h_i, for $1 \leqslant i \leqslant r$ (by assumption, such elements c_i exist). We just showed that for $1 \leqslant i \leqslant r$, we have that $c_i \neq a$. Let I_2' be the result of adding a to I_2 and removing $c_1, \ldots, c_r$. We now show that I_2' does not contain any hyperedges. This is because I_2 does not contain any hyperedges, and so each hyperedge contained in I_2' must have a as a member. But the only candidates for such a hyperedge would be $h_1, \ldots h_r$. However, none of these are contained in I_2', because $c_i \notin I_2'$, for $1 \leqslant i \leqslant r$.

Therefore, I_2' is an improvement of I_2, which contradicts our assumption that I_2 is optimal. □

Theorem 4.3 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.*

PROOF. Let u be a given cleaning update. We will devise a procedure that constructs an NFA A (or, equivalently, a Boolean vset-automaton), such that A rejects every string if and only if u satisfies the minimum property. This is sufficient to prove the theorem, since it is well known that it is decidable if a given NFA rejects every string. We assume that the set of variables that occur in the dcs of u is disjoint from the set of variables that occur in the pgds of u. We can make this assumption, since variables can be renamed.

Let δ be a dc in u, and suppose that δ is given by $\tau_\delta[\mathbf{x}] \rightarrow \neg\Psi_\delta(\mathbf{x})$. We denote by Φ_δ the set of all pairs $\langle\varphi(\mathbf{x}), \varphi'(\mathbf{x})\rangle$, such that $\varphi(\mathbf{x})$ and $\varphi'(\mathbf{x})$ are (not necessarily distinct) atomic formulas in Ψ_δ. Let $c = \langle\varphi(\mathbf{x}), \varphi'(\mathbf{x})\rangle$ be a pair in Φ_δ. A pgd p of the form $\tau_p[\mathbf{y}] \rightarrow (\eta(\mathbf{y}) \succ \eta'(\mathbf{y}))$ is said to *match* c if φ and η have the same relation symbol and φ' and η' have the same relation symbol. We denote by Π_c the set of all the pgds p in u, such that p matches c.

If f and f' are two facts, then we will use the notation $f \succeq f'$ as a replacement of $f \succ f'$ or $f = f'$.

Let $c = \langle\varphi(\mathbf{x}), \varphi'(\mathbf{x})\rangle$ be a pair in Φ_δ and let p be a pgd $\tau_p[\mathbf{y}] \rightarrow (\eta(\mathbf{y}) \succ \eta'(\mathbf{y}))$ in Π_c. We will construct a regular spanner $\tau_{c,p}[\mathbf{x}]$, such that $\tau_{c,p}$ contains all the assignments μ for $\mathbf{x}$ such that the facts f_1 and f_2 obtained by applying μ to φ and φ', respectively, satisfy $f_1 \succeq f_2$ according to p. We assume the following.

- $\varphi(\mathbf{x}) = R(x_1, \ldots, x_m)$
- $\eta(\mathbf{y}) = R(y_1, \ldots, y_m)$
- $\varphi'(\mathbf{x}) = R'(x'_1, \ldots, x'_{m'})$
- $\eta'(\mathbf{y}) = R'(y'_1, \ldots, y'_{m'})$.

Note that each of the above tuples of variables may include repetitions (e.g., $x_i = x_j$ for $i \neq j$). Moreover, recall that $\mathbf{x}$ and $\mathbf{y}$ are disjoint.

We denote by $\rho_c^=[\mathbf{x}]$ the regular spanner that behaves as follows:

- If R and R' are the same relation symbol, then $\rho_c^=[\mathbf{x}]$ produces all the assignments μ to $\mathbf{x}$ such that $x_i = x'_i$ for all $i = 1, \ldots, m$.
- If R and R' are different relation symbols, then $\rho_c^=[\mathbf{x}]$ is empty.

It is easy to verify that, indeed, $\rho_c^=[\mathbf{x}]$ can be constructed as an expression in REG.

Let $\gamma[\mathbf{x}, \mathbf{y}]$ be a REG expression representing the spanner that produces all the assignments to $\mathbf{x}$ and $\mathbf{y}$, such that $x_i = y_i$ for $i = 1, \ldots, m$. Similarly, let $\gamma'[\mathbf{x}', \mathbf{y}']$ be a REG expression representing the spanner that produces all the assignments to $\mathbf{x}'$ and $\mathbf{y}'$, such that $x'_i = y'_i$ for $i = 1, \ldots, m'$. Then we define $\tau_{c,p}[\mathbf{x}]$ as follows:

$$\tau_{c,p}[\mathbf{x}] \stackrel{\text{def}}{=} \rho_c^=[\mathbf{x}] \cup \pi_{\mathbf{x}}\big(\tau_\delta[\mathbf{x}] \bowtie \tau_p[\mathbf{y}] \bowtie \gamma[\mathbf{x}, \mathbf{y}] \bowtie \gamma'[\mathbf{x}', \mathbf{y}']\big).$$

In words, the left disjunct $\rho_c^=[\mathbf{x}]$ produces all the assignments μ to $\mathbf{x}$ such that that the facts f_1 and f_2 obtained by applying μ to φ and φ', respectively, are equal. Hence, $f_1 \succeq f_2$ vacuously holds. The right disjunct, in contrast, produces all the assignments such that the facts f_1 and f_2 obtained by applying μ to φ and φ' satisfy $f_1 \succ f_2$ according to p.

The spanner τ_c is the disjunction of the $\tau_{c,p}$ across all the pgds that match c.

$$\tau_c[\mathbf{x}] \stackrel{\text{def}}{=} \bigcup_{p \in \Pi_c} \tau_{c,p}[\mathbf{x}].$$

Hence, τ_c produces all the assignments μ for $\mathbf{x}$ such that the facts f_1 and f_2 obtained by applying μ to φ and φ', respectively, satisfy $f_1 \succeq f_2$ according to some pgd in u.

Let $a' = \varphi'(\mathbf{x})$ be an atom of the dc δ. We denote by $\Phi_{\delta,a'}$ the subset of Φ_δ that consists of pairs $\langle a, a' \rangle$ for some atom a of δ. We now define the following expression $\tau_{\delta,a'}$ in REG:

$$\tau_{\delta,a'}[\mathbf{x}] \stackrel{\text{def}}{=} \bowtie_{c \in \Phi_{\delta,a'}} \tau_c[\mathbf{x}].$$

In words, $\tau_{\delta,a'}$ produces all the assignments μ for $\mathbf{x}$ such that the fact f' obtained by applying μ to the atom a' satisfies $f \succeq f'$ for every fact f obtained by applying μ to a different atom a of δ.

Next, $\tau_{\delta,*}$ is the union of the $\tau_{\delta,a'}$ across all atoms a' of δ:

$$\tau_{\delta,*}[\mathbf{x}] \stackrel{\text{def}}{=} \bigcup_{a' \in \delta} \tau_{\delta,a'}[\mathbf{x}].$$

In words, $\tau_{\delta,*}$ produces all the assignments μ for $\mathbf{x}$ such that for at least one atom a' of δ, the fact f' obtained by applying μ to the atom a' satisfies $f \succeq f'$ for every fact f obtained by applying μ to a different atom a of δ.

Next, υ_δ is the Boolean spanner that is true if and only if there is an assignment to the left-hand-side of δ that does not satisfy $\tau_{\delta,*}$.

$$\upsilon_\delta \stackrel{\text{def}}{=} \pi_\varnothing\big(\tau_\delta[\mathbf{x}] \setminus \tau_{\delta,*}[\mathbf{x}]\big)$$

Hence, υ_δ is true if and only if the minimum property is violated on a hyperedge defined by δ. Note that υ_δ is regular, since regular spanners are closed under complement (Theorem 2.5). Finally, υ is the disjunction of the υ_δ:

$$\upsilon \stackrel{\text{def}}{=} \bigcup_{\delta \in u} \upsilon_\delta.$$

It is easy to see that υ is true on any document $\mathbf{d}$ if and only if the minimum property is violated when u is applied to some $\mathbf{d}$-instance I. Since υ is Boolean and regular, we get from Theorem 2.8 that we can translate it to an NFA A. And since A is empty if and only if u has the minimum property, we get that A is as claimed. □

For this and later decidability results, we gain insight into the complexity by seeing which known decidable problem P, with what input size, we reduce the problem in our decidability result to (in the case of Theorem 4.5 the known decidable problem is the emptiness problem for NFA's). Establishing finer complexity classes for this and later decidability results is the subject of future work.

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 Theorem 4.6. To prove Theorem 4.6, we use results on the emptiness of multihead automata [Holzer et al. 2008] 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.

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 $\delta(q, a_1, \ldots, a_k) = \varnothing$). 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 A be a 2NFA(k). If the range of δ consists of only singletons and the empty set, then we say that A is *deterministic* and may replace "NFA" with "DFA" in 2NFA(k). If no transition moves heads to the left, then the A is *one way*, and we may replace "2" with "1" in "2NFA(k)." For example, 1DFA(2) is a one-way deterministic 2-head automaton. We have the following due to Holzer et al. [2008].

THEOREM 4.7 [HOLZER ET AL. 2008]. *The emptiness problem (i.e., whether no string is accepted) is undecidable for 1DFA(2).*

Let $\mathbf{C}$ be a class of multihead 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 now prove the following lemma.

LEMMA 4.8. *For 2DFA(2), the finite mortality problem is undecidable.*

PROOF. We will show how the emptiness problem for 1DFA(2) reduces to the finite mortality problem for 2DFA(2). Let A be a given 1DFA(2), and our goal is to determine whether there exists a string that is accepted by A. We will construct a 2DFA(2) B, such that B has no immortal configurations if and only if A recognizes the empty language.

Without loss of generality, we will assume that in every transition, at least one head of A moves to the right. We can make that assumption, since we can easily change A to satisfy it without affecting its set of accepting strings. In particular, starting from every string and configuration, A always terminates.

Observe that an accepting halting configuration of A is such that the heads are pointing to two symbols a_1 and a_2, respectively, the current state q is an accepting state, and $\delta(q, a_1, a_2)$ is empty. We obtain the 2DFA(2) B from A by applying the following modification to A. Whenever the configuration is an accepting halting configuration, the automaton enters a special state q' where both heads are going left, all the way

to the beginning, until the first symbol, and then the state changes to the initial one; thereby, the initial configuration is obtained.

The reader can now easily verify that the automaton A has an accepting input if and only if B has an immortal configuration. □

We now proceed to complete the proof of Theorem 4.6. We will show a reduction from the problem of finite mortality for 2DFA(2). Throughout this section we fix an input $A = \langle \Theta, Q, \delta, q_0, F \rangle$, which is a 2DFA(2). We will construct a binary regular spanner ρ with two variables x and y, such that the following holds. Given a document $\mathbf{d}$, we view $[\![\rho]\!](\mathbf{d})$ as representing the edges of a directed graph over the spans of $\mathbf{d}$. Then A has no immortal configurations if and only if for all documents $\mathbf{d}$, the graph represented by $[\![\rho]\!](\mathbf{d})$ is acyclic. From ρ we can construct the following denial pgd, which is acyclic if and only if $[\![\rho]\!](d)$ is acyclic on every $\mathbf{d}$:

$$\rho[x, y] \rightarrow R(y) \rhd R(x).$$

Combined with Lemma 4.8, we then get undecidability of the question as to whether a denial pgd in REG is acyclic.

Let $\vdash\mathbf{s}\dashv$ be a given input for A. Suppose that Q is the set $\{q_1, \ldots, q_m\}$. Denote $\vdash\mathbf{s}\dashv$ by $\sigma_1 \cdots \sigma_n$, where $\sigma_1 = \vdash$, $\sigma_n = \dashv$, and $\sigma_i \in \Theta$ for $i = 2, \ldots, n-1$. We will assume that the alphabet Σ, which we use for spanners, contains $\vdash$, $\dashv$, all the symbols of Θ, and all the states of Q. We define the document $\mathbf{d_s}$ as the concatenation $\mathbf{e}_1 \cdots \mathbf{e}_n$, where each $\mathbf{e}_i$ is the string $\sigma_i q_1 \cdots q_m$.

Next, we show how to represent a configuration of A over $\vdash\mathbf{s}\dashv$ using a span over $\mathbf{d_s}$. Recall that a configuration of A consists of the positions of the two heads and the state. Denote the heads of A by h_1 and h_2. Suppose that in the configuration we represent, h_1 and h_2 are positioned on indices i_1 and i_2, respectively. Let j_1 be position of the first symbol of $\mathbf{e}_{i_1}$ in $\mathbf{d_s}$. Suppose that the state in our configuration is q_j. Let j_2 be the position of the symbol q_j of $\mathbf{e}_{i_2}$ in $\mathbf{d_s}$. We have the following:

- $j_1 = (m+1)(i_1 - 1) + 1$
- $j_2 = (m+1)(i_2 - 1) + 1 + j$

We represent our configuration with the span $[j_1, j_2\rangle$ if $j_1 < j_2$, or $[j_2, j_1\rangle$ if $j_2 < j_1$. Observe that this span, indeed, uniquely identifies the configuration. (We can distinguish which position corresponds to the head h_1 and which to the head h_2 because of the additive term j in j_2.) Moreover, every configuration has precisely one such representation by a span over $\mathbf{d_s}$. We need the following lemma. The proof is straightforward yet tedious and hence is omitted.

LEMMA 4.9. *Expressions in* REG *defining the following spanners can be constructed by a Turing machine:*

- *A Boolean spanner P_1 such that $P_1(\mathbf{d})$ is true if and only if $\mathbf{d}$ has the form $\mathbf{d_s}$ for some string $\mathbf{s} \in \Theta^*$.*
- *A unary spanner P_2 such that $P_2(\mathbf{d_s})$ consists of precisely the spans $[j_1, j_2\rangle$ that represent configurations of A.*
- *A binary spanner P_3 such that $P_3(\mathbf{d_s})$ consists of all the pairs x and y, such that both x and y represent configurations of A.*
- *A binary spanner P_4, which is the restriction of P_3 to all the pairs (x, y) where y is the subsequent configuration of x.*

Consider the spanners P_1 and P_4 of Lemma 4.9. Let P be the spanner $P_1 \bowtie P_4$. Given a document $\mathbf{d}$, the binary relation $P(\mathbf{d})$ is empty if $\mathbf{d}$ is not of the form $\mathbf{d_s}$. Otherwise, $P(\mathbf{d})$ represents all the pairs of consecutive configurations. Let G be the graph of spans that is represented by $P(\mathbf{d_s})$ for some $\mathbf{d_s}$. Then G has a cycle if and only if A has

a cyclic sequence of configurations over $\vdash\mathbf{s}\dashv$. However, A has only a finite number of configurations over $\mathbf{s}$, and hence, the existence of a such a cycle is logically equivalent to A having an immortal configuration. We conclude that P represents an acyclic graph for all documents $\mathbf{d}$ if and only A has no immortal configurations, as claimed.

It remains to show that the problem is co-recursively enumerable but not recursively enumerable. It is well known that a problem is decidable if and only if both the problem and its complement are recursively enumerable. Since our problem is undecidable, it suffices to show that its complement is recursively enumerable. A Turing machine that enumerates all the nonacyclic denial pgds in REG can be one that enumerates all denial pgds p, documents $\mathbf{d}$, and $\mathbf{d}$-instances I over the relation symbols of p (in an order that guarantees that every p, $\mathbf{d}$, and I is considered), evaluates p on I, and tests whether there is a cycle; if so, then p is printed.

This concludes the proof of Theorem 4.6. Beyond the theorem itself, we gain two insights from the proof. First, we obtain Lemma 4.8, which is interesting in its own right. Second, for this and later undecidability results, we see which known undecidable problem can be reduced to the problem under consideration.

4.1.2. Proof of Theorem 4.2. The proof of Theorem 4.2 is a variation of the proof of Theorem 4.6. We now need a definition and a lemma.

Let A be a 2NFA(k). We denote by $\mathcal{L}(A)$ the set of strings accepted by A. We define a *full cycle* of A to be a sequence $\mathbf{c} = c_0, \ldots, c_{m-1}$ of configurations, such that:

- c_0 is the initial configuration;
- c_i is a legal subsequent configuration of c_{i-1} for $i = 1, \ldots, m-1$;
- c_0 is a legal subsequent configuration of c_{m-1}.

The *length* of $\mathbf{c}$ is m. Note that a 2DFA(k) A has at most one full cycle for each input string (since it is deterministic).

LEMMA 4.10. *There is a Turing machine that, given a 1DFA(2) A, constructs a 2DFA(2) B such that:*

(1) *If $\mathcal{L}(A) = \varnothing$, then B has no finite immortal configurations.*
(2) *If $\mathcal{L}(A) \neq \varnothing$, then B has a full cycle of odd length on at least one input string.*

PROOF. We will adjust the construction used in the proof of Lemma 4.8. That construction alone already guarantees that if $\mathcal{L}(A) = \varnothing$, then B has no finite immortal configurations. Moreover, if $\mathcal{L}(A) \neq \varnothing$, then the construction guarantees that, on at least one string, there is full cycle $\mathbf{c} = c_0, \ldots, c_{m-1}$. However, m may be even. Without loss of generality, assume that the only initial configuration in $\mathbf{c}$ is c_0. We can then adjust B to guarantee that m is odd, as follows.

First, we can change B so the initial state q_0 is used only in the initial configuration. Second, we can add to each state q of B, except for the initial state, a new state $\hat{q}$, such that all the transitions into q will first go through $\hat{q}$. Formally, suppose we have the following:

$$\delta(q', \sigma_1, \sigma_2) = \{(q, \mathsf{l}, \mathsf{r})\}.$$

Then after we change B, we will get the following two transitions:

$$\delta(q', \sigma_1, \sigma_2) = \{(\hat{q}, \mathsf{l}, \mathsf{r})\}$$
$$\delta(\hat{q}, \sigma_1, \sigma_2) = \{(q, \mathsf{h}, \mathsf{h})\}.$$

It is easy to verify that, with this change, we are guaranteed that the full cycle $\mathbf{c}$ (which, by assumption, has only one occurrence of the initial state), is of odd length, as required. □

We can now prove Theorem 4.2. We will show how to adjust the proof of Theorem 4.6. By applying Lemma 4.10, we can assume that if the 2DFA(2) B has any finite immortal configuration, then it has a full cycle of an odd length. Consider the following program, which we denote by $\mathcal{E}$:

(1) $R(x) :- \Sigma^* \cdot x\{\Sigma^*\} \cdot \Sigma^*$
(2) $\textsc{Clean}(\rho[x, y] \rightarrow R(y) \triangleright R(x))$
(3) $\textsc{Return}(R(x))$.

If B has no immortal configuration, then the pgd in the program is acyclic, and then by Theorem 4.3 we get that $\mathcal{E}$ is unambiguous. So now suppose that B has an immortal configuration. We will show that $\mathcal{E}$ is not unambiguous. Suppose, by way of contradiction, that $\mathcal{E}$ is unambiguous.

Let $\mathbf{s}$ be a string such that B has a full cycle of odd length when evaluated over $\mathbf{s}$. Consider the document $\mathbf{d_s}$ that is constructed in the proof of Theorem 4.6 for the string $\mathbf{s}$. Suppose that $\mathcal{E}(\mathbf{d_s}) = \{J\}$. We will obtain a contradiction by reasoning about J.

Let $c_0, \ldots, c_{m-1}$ be the full cycle of A when run over $\mathbf{s}$. (Recall that m is odd.) For $i = 0, \ldots, m-1$, let f_i be the fact $R(s_i)$, where s_i represents the configuration c_i.

We will use the following observation. Since B is deterministic, we have $f \triangleright f_i$, for exactly one f, namely the one corresponding to the configuration that legally follows f_i, that is, $f = f_j$ for $j = (i + 1) \mod m$. We will also use the following observation. A cycle with an odd number of nodes, some of which are black and the rest white, must have at least one pair of consecutive nodes with the same color.

From the fact that m is odd it follows that there must be two facts f_i and f_j, where $j = (i + 1) \mod m$, such that either (1) both f_i and f_j are in J or (2) neither f_i nor f_j is in J. Case (1) cannot be true, since f_i and f_j are in conflict (where f_j is the winner). Therefore, Case (2) is true. However, we can then improve J again by adding f_i and removing all those facts that are in conflict with f_i—all of those are dominated by f_i, since the only fact that dominates f_i is f_j, which is not in J. Hence, we get a contradiction.

It remains to show that the problem is not recursively enumerable. We use the same argument as that used in the proof of Theorem 4.6: It suffices to show that the complement of our problem is recursively enumerable. Indeed, a Turing machine that enumerates all the ambiguous extraction programs in REG can be one that enumerates all documents $\mathbf{d}$ and programs $\mathcal{E}$ in REG (in an order that guarantees that every $\mathbf{d}$ and $\mathcal{E}$ is considered), evaluates $\mathcal{E}(\mathbf{d})$, and prints $\mathcal{E}$ if $\mathcal{E}(\mathbf{d})$ is not a singleton.

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 noncleaning $\mathcal{R}$-program that is equivalent to $\mathcal{E}$. Of course, we have the following.

PROPOSITION 4.11. *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 noncleaning $\mathcal{R}$-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 4.12. *There exists an acyclic denial pgd in* REG *that is not* REG*-disposable.*

In the remainder of this section, we prove Theorem 4.12. We will assume that our language contains the symbols 0, 1, and $\#$. We first define the following two languages over Σ:

$$L^2 = \{\mathbf{s}\#\mathbf{t} \mid \mathbf{s}, \mathbf{t} \in \{0,1\}^+ \text{ and both } |\mathbf{s}| \text{ and } |\mathbf{t}| \text{ are even}\}$$
$$L^= = \{\mathbf{s}\#\mathbf{t} \mid \mathbf{s}\#\mathbf{t} \in L^2 \text{ and } |\mathbf{s}| = |\mathbf{t}|\}.$$

We need the following lemma. The proof is by standard arguments on the expressiveness of regular languages and the fact that Boolean regular spanners coincide with the regular languages (Theorem 2.8).

LEMMA 4.13. *The language L^2 is regular, but the language $L^=$ is not. Equivalently, there exists a Boolean regular spanner that accepts L^2, but no Boolean regular spanner accepts $L^=$.*

We now introduce some terminology. Let $[i, j\rangle$ and $[i', j'\rangle$ be two spans of some document $\mathbf{d}$, and let m be a natural number. We say that $[i', j'\rangle$ is a *right m-shift* of $[i, j\rangle$ if $i' = i + m$ and $j' = j + m$. Now consider a document $\mathbf{d} = \mathbf{s}\#\mathbf{t}$ in L^2. We call the span that corresponds to the prefix $\mathbf{s}$ the *left side* of $\mathbf{d}$, and we call the span that corresponds to the suffix $\mathbf{t}$ the *right side* of $\mathbf{d}$. For example, if $\mathbf{d} = 0100\#10$, then the left side is $[1, 5\rangle$ and the right side is $[6, 8\rangle$.

We shall construct the program $\mathcal{E} = \langle \mathbf{S}, U, \varphi \rangle$ as follows:

(1) $R(x) :- \quad \gamma^2 \ \wedge \ \Sigma^* \cdot x\{\Sigma^*\} \cdot \Sigma^*$
(2) CLEAN($\rho[x, y] \rightarrow R(y) \rhd R(x)$)
(3) $S(x) :- \quad R(x) \ \wedge \ x\{(0 \vee 1)^*\} \cdot \# \cdot (0 \vee 1)^*$
(4) RETURN($S(x)$).

In the program, the symbol γ^2 represents a regular expression that accepts the language L^2 (Lemma 4.13). The spanner $\rho[x, y]$ will be described next. Note that $R(x)$ contains every possible span if the input document is in L^2, and is empty otherwise.

Our definition of $\rho[x, y]$ will be such that in the end of the execution, the relation S (which is the output of the program) is either empty or the singleton that consists of left side of $\mathbf{d}$. Moreover, the latter happens precisely when $\mathbf{d}$ is in L^2 and R contains the left side of $\mathbf{d}$ after the cleaning of line 2.

Our expression $\rho[x, y]$ is the union $\rho_1[x, y] \cup \rho_2[x, y]$ where:

- $\rho_1[x, y]$ produces all the spans x and y, such that y is a right 1-shift of x:

$$\rho_1[x, y] \stackrel{\text{def}}{=} \left(\Sigma^* \cdot x\{\Sigma \cdot z\{\Sigma^*\}\} \cdot z'\{\Sigma\} \cdot \Sigma^*\right) \bowtie \left(\Sigma^* \cdot y\{z\{\Sigma^*\} \cdot z'\{\Sigma\}\} \cdot \Sigma^*\right).$$

- $\rho_2[x, y]$ produces exactly one tuple when applied to a document $\mathbf{d}$ in L^2: x is the right side of $\mathbf{d}$ and y is the empty span in the end of the document:

$$\rho_2[x, y] \stackrel{\text{def}}{=} \left(\Sigma^* \cdot \# \cdot x\{\Sigma^*\} \cdot y\{\epsilon\}\right).$$

Hence, the spanner $[\![\rho]\!]$ produces all the pairs (x, y) of spans, such that either y is the right 1-shift of x or x is the right side and y is the empty span at the end of the document. It is an easy observation that the update of $\mathcal{E}$ is an acyclic denial pgd. In particular, we have that $\mathcal{E}$ can be viewed as a spanner. We have the following.

LEMMA 4.14. *A document $\mathbf{d}$ is in $L^=$ if and only if $\mathcal{E}(\mathbf{d})$ is nonempty.*

PROOF. Let $\mathbf{d}$ be a document. As explained above, if $\mathbf{d}$ is not in L^2, then $\mathcal{E}(\mathbf{d})$ is empty. So in the remainder of the proof we will assume that $\mathbf{d}$ is in L^2. We now prove each direction separately.

The "if" direction. We assume that $\mathcal{E}(\mathbf{d})$ is nonempty, and we need to show that $\mathbf{d}$ is in $L^=$. Suppose, by way of contradiction, that $\mathbf{d}$ is not in $L^=$. Since $\mathbf{d}$ is in L^2, it must be the case that the left and right sides of $\mathbf{d}$ are of different lengths. In particular, the right side of $\mathbf{d}$ is not among the right shifts of the left side. Let us denote by l the left side of $\mathbf{d}$ and by s_r the rightmost shift of l. (Note that l is a prefix of $\mathbf{d}$.) Suppose that s_r is the right m-shift of l. Since $\mathbf{d}$ is in L (and, in particular, both sides are of even lengths), we get that m is necessarily odd.

From the definition of our denial pgd we get that s_r is in R after the execution of line 2, since every consistent subinstance can be improved by adding the fact $R(s_r)$; this is true, since $R(s_r)$ prevails over its single conflicting tuple. (Recall that the only suffix that is prevailed over by another span is the right side of $\mathbf{d}$, but s_r is *not* the right side of $\mathbf{d}$.) Then the fact that s_r is in R implies that the right $(m-1)$-shift of l is not in R, and then the right $(m-2)$-shift of l is in R. Continuing so, we get that the right $(m-k)$-shift of l is in R if and only if k is even. In particular, the span l itself, which is the $(m-m)$-shift of l, is *not* in $R(x)$ when the program terminates. However, nonemptiness of $\mathcal{E}(\mathbf{d})$ implies that S is nonempty, which implies that l is in R when the program terminates. Hence, we get a contradiction.

The "only if" direction. We assume that $\mathbf{d}$ is in $L^=$, and we need to show that $\mathcal{E}(\mathbf{d})$ is nonempty. In this case, the right side of $\mathbf{d}$ is a right m-shift of l, where $m = |l|+1$. Let us denote the right side of $\mathbf{d}$ by r. When the program $\mathcal{E}$ terminates, r is *not* in R, since r is dominated by the empty span at the end, and that empty span is not dominated by any other span. Consequently, we get that the right $m-1$-shift of l is in R. And then, then right $m-2$-shift of l is *not* in R. Continuing so, we get that the right $(m-k)$-shift of l is in R if and only if k is odd. And since m itself is odd (as $|l|$ is even), we get that l itself is in R when the program terminates. As explained above, we get that S is nonempty, and then $\mathcal{E}(\mathbf{d})$ is nonempty. □

To complete the proof of Theorem 4.12, suppose that our denial pgd $\rho[x, y] \rightarrow R(y) \triangleright R(x)$ is REG-disposable. From Propositions 4.1 and 4.11 we get that $\mathcal{E}$ represents a regular spanner. But then we can transform it into a Boolean regular spanner by projecting x out, and then we get a Boolean spanner (i.e., NFA) that recognizes $L^=$, in contradiction to Lemma 4.13. Note that the proof gives us an explicit acyclic denial pgd in REG that is not REG-disposable, namely $\rho[x, y] \rightarrow R(y) \triangleright R(x)$.

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 through the use of a regular spanner in the premise. Moreover, we will prove that all of these cleaning updates are REG-disposable.

We start this a general result of disposability of a large class of denial pgds, which will be used in Sections 5.2 and 5.3.

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 $P \rightarrow (\varphi(\mathbf{x}) \triangleright \varphi'(\mathbf{x}))$ 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, that favors longer strings and that 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 [Chiticariu et al. 2010], along with the analogous *minimal contained* (that favors shorter strings and is expressed by the denial pgd dp_{enc} in Example 3.5), 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 P holds (e.g., some attributes overlap), then 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. 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 and* (2) *determine whether p is both transitive and acyclic.*

PROOF. Let the given denial pgd p be the following:

$$\rho[\mathbf{x}] \rightarrow \big(\varphi(\mathbf{x}) \rhd \varphi'(\mathbf{x})\big).$$

We first prove (1). Our goal is to describe a procedure to test whether p is transitive. If φ and φ' have different atomic relations, then p is transitive in a vacuous sense. Therefore, we will assume that both φ and φ' use the same relation symbol, which we denote as R. Moreover, we will assume that p has the following form:

$$\rho[\mathbf{x}] \rightarrow \big(R(x_1, \ldots, x_n) \rhd R(x'_1, \ldots, x'_n)\big).$$

Let $\rho[\mathbf{y}]$ be obtained from $\rho[\mathbf{x}]$ by renaming each variable x with a unique, fresh variable that we denote as y_x. We denote y_{x_i} by y_i and $y_{x'_i}$ by y'_i. We similarly define $\rho[\mathbf{z}]$. So now p is *not* transitive if and only if there are assignments μ^x and μ^y for $\rho[\mathbf{x}]$ and $\rho[\mathbf{y}]$, respectively, such that both of the following hold:

- $\mu^x(x'_i) = \mu^y(y_i)$ for every x'_i.
- No assignment μ^z produced by $\rho[\mathbf{z}]$ satisfies $\mu^x(x_i) = \mu^z(z_i)$ and $\mu^y(y'_i) = \mu^z(z'_i)$ for all $i = 1, \ldots, n$.

Let $\gamma^{=}[\mathbf{x}, \mathbf{y}, \mathbf{z}]$ denote the expression in REG representing the spanner defined by $x_i = z_i$ and $y'_i = z'_i$ for all $i = 1, \ldots, n$ (note that $\mathbf{y}$ includes $y_1, \ldots, y_n, y'_1, \ldots, y'_n$ and similarly for $\mathbf{z}$). We define the following:

$$\tau[\mathbf{x}, \mathbf{y}] \stackrel{\mathsf{def}}{=} \pi_{\mathbf{x},\mathbf{y}}\big(\rho[\mathbf{z}] \bowtie \gamma^{=}[\mathbf{x}, \mathbf{y}, \mathbf{z}]\big).$$

In words, $\tau(\mathbf{x}, \mathbf{y})$ consists of all the spans for $x_1, \ldots, x_n$ and $y'_1, \ldots, y'_n$, such that for some $\mathbf{z}$ we have $\rho[\mathbf{z}]$, and $x_i = z_i$ and $y'_i = z'_i$ for all $i = 1, \ldots, n$. Then τ is a regular spanner. Therefore, the complement of $\tau[\mathbf{x}, \mathbf{y}]$, denoted $\overline{\tau}[\mathbf{x}, \mathbf{y}]$ is also a regular spanner (Theorem 2.5). Observe that $\overline{\tau}[\mathbf{x}, \mathbf{y}]$ consists of all the assignments to $x_1, \ldots, x_n$ and $y'_1, \ldots, y'_n$ such that $R(x_1, \ldots, x_n) \rhd R(y'_1, \ldots, y'_n)$ is not implied by p. Let $\upsilon^{=}[\mathbf{x}, \mathbf{y}]$ denote

the expression in REG representing the spanner defined by $x'_i = y_i$ for all $i = 1, \ldots, n$. We define τ' to be the following Boolean spanner:

$$\tau' \stackrel{\text{def}}{=} \pi_{\varnothing}\big(\rho[\mathbf{x}] \bowtie \rho[\mathbf{y}] \bowtie \upsilon^{=}[\mathbf{x}, \mathbf{y}] \bowtie \overline{\tau}[\mathbf{x}, \mathbf{y}]\big)\,.$$

Then τ' is always false if and only if p is transitive. By Theorem 2.8, we can test whether τ' is false simply by an emptiness testing on an NFA.

We now prove (2). We need to show a procedure to test whether p is both transitive and acyclic. Using (1) we test whether p is transitive. So we assume that p is transitive and describe a procedure to test whether p is acyclic. Since p is transitive, it is acyclic if and only if it is antireflexive (i.e., no fact is subsumed by itself). Suppose that p is the following denial pgd (note that we can assume that both atoms in p use the same relation symbol):

$$\rho[\mathbf{x}] \to (R(x_1, \ldots, x_n) \rhd R(x'_1, \ldots, x'_n)).$$

Let $\gamma^{=}[\mathbf{x}]$ denote the expression in REG representing the spanner defined by $x_i = x'_i$ for all $i = 1, \ldots, n$. Then p is antireflexive if and only if the spanner τ defined by the following Boolean spanner is always false:

$$\pi_{\varnothing}\big(\tau[\mathbf{x}] \bowtie \gamma^{=}[\mathbf{x}]\big).$$

Again, using Theorem 2.8 we translate this test into the emptiness of an NFA. □

As with Theorem 4.5, we gain insight into the complexity by the reduction of our problems to the emptiness problem for NFA's.

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. To prove that, we need the following lemma; the proof is straightforward.

LEMMA 5.2. *Let* $\mathbf{S}$ *be a schema, let* $\mathbf{d}$ *be a document, and let* I *be a* $\mathbf{d}$*-instance over* $\mathbf{S}$*. Let* p *be a denial pgd over* $\mathbf{S}$*, and suppose that* p *is transitive. Then the single optimal repair of* I *under* p *is the one that consists of all the facts* f *such that, according to* p*, no fact* f' *satisfies* $f' \rhd f$*.*

THEOREM 5.3. *If* p *is a transitive denial pgd in* REG*, then* p *is* REG*-disposable.*

PROOF. Let a given denial pgd p be the following:

$$\rho[\mathbf{w}] \to (R(x_1, \ldots, x_n) \rhd S(v_1, \ldots, v_m)).$$

Let $\mathcal{E}$ be a REG-program that has p as its single cleaning update. We need to construct a noncleaning REG-program $\mathcal{E}_r$ that is equivalent to $\mathcal{E}$. Due to Proposition 4.1, such a program $\mathcal{E}_r$ is equivalent to a regular spanner. Moreover, the definition of R and S in the program, up to the cleaning update, could be collapsed to a two single primitive updates, say:

$$R(y_1, \ldots, y_n) :- \tau_R[\mathbf{y}]$$
$$S(z_1, \ldots, z_m) :- \tau_S[\mathbf{z}].$$

Here we assume that $\mathbf{y} = y_1, \ldots, y_n$ is a sequence of n distinct variables and that $\mathbf{z} = z_1, \ldots, z_m$ is a sequence of m distinct variables. Moreover, we assume that $\mathbf{w}$, $\mathbf{y}$, and $\mathbf{z}$ are pairwise disjoint (i.e., each variable occurs in at most one sequence out of the three). In the case where R and S are the same relation symbol, we have that τ_R and τ_S represent the same spanners.

Observe that whether or not R and S are the same relation symbol, applying p as a cleaning update can change only S. Therefore, it suffices to construct a spanner $\tau[\mathbf{z}]$

that consists of precisely the content of S after the cleaning update. Then, we replace τ_S with τ in the definition of S. According to Lemma 5.2, this spanner should produce all the assignments μ produced by τ_S, such that no assignment from τ_R prevails over μ according to p.

We denote by $\mathbf{x}$ the sequence $x_1, \ldots, x_n$ and by $\mathbf{v}$ the sequence $v_1, \ldots, v_m$. Let $\gamma^{=}[\mathbf{x}, \mathbf{y}]$ denote the expression in REG representing the spanner defined by $x_i = y_i$ for all $i = 1, \ldots, n$. Similarly, let $\iota^{=}[\mathbf{v}, \mathbf{z}]$ denote the expression in REG representing the spanner defined by $v_i = z_i$ for all $i = 1, \ldots, m$. Then $\tau[\mathbf{z}]$ can be defined as follows:

$$\tau_S[\mathbf{z}] \setminus \pi_{\mathbf{z}}\Big(\rho[\mathbf{w}] \bowtie \tau_R[\mathbf{y}] \bowtie \tau_S[\mathbf{z}] \bowtie \gamma^{=}[\mathbf{x}, \mathbf{y}] \bowtie \iota^{=}[\mathbf{v}, \mathbf{z}]\Big). \tag{2}$$

In words, τ consists of all the assignments for $\mathbf{z}$ that are in τ_S and *cannot* be assignments for the right side $S(v_1, \ldots, v_m)$ of the conclusion $(R(x_1, \ldots, x_n) \rhd S(v_1, \ldots, v_m))$ of p (i.e., $\mathbf{v} = \mathbf{z}$), where the left side and right side of the conclusion of p are in τ_R (i.e., $\mathbf{x} = \mathbf{y}$) and τ_S (i.e., $\mathbf{v} = \mathbf{z}$), respectively. Since τ is a regular spanner, the claim is proved. □

We believe that the procedure introduced in the proof of Theorem 5.3 to dispose transitive denial pgds can be made effective and efficient in practice. Indeed, let p be the transitive denial pgd $\rho[\mathbf{w}] \rightarrow (R(x_1, \ldots, x_n) \rhd S(v_1, \ldots, v_m))$. Observe that the procedure of Theorem 5.3 essentially consists of removing the CLEAN(p) update and replacing the last update to relation S prior to CLEAN(p) by Equation (2) above. In particular, the number of updates in the disposed program does not increase. Furthermore, note that Equation (2) is a reasonably simple relational algebra expression (over spanners) that computes the difference of one relation and a five-way join. In practice, this can be efficiently implemented by first computing the span relations for the spanners appearing in the expression and then performing the join and difference (cf. Reiss et al. [2008]).

We illustrate the disposal procedure of Theorem 5.3 by means of the following example.

Example 5.4. Consider the following slightly modified version of the REG-program $\mathcal{E}$ from Example 3.7. The goal of the program is still to extract pairs (x, y), where x is a person and y is a location associated with x, but has only one cleaning update, whereas Example 3.7 has several. The signature is, as usual, that of Example 3.1, and we are using the notation we established in the previous examples in Section 3:

(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) $\mathsf{PerLoc}(x, y) :- \mathsf{Per}(x) \wedge \mathsf{Loc}(y) \wedge \mathsf{precede}[x, y]$ (see Example 3.3)
(4) CLEAN($\mathsf{encloses}[z, x, y] \bowtie \mathsf{encloses}[z', x', y'] \bowtie \mathsf{contains}_{\neq}[z', z]$ $\rightarrow \mathsf{PerLoc}[x, y] \rhd \mathsf{PerLoc}[x', y']$) (see Example 3.5)
(5) RETURN $\mathsf{PerLoc}(x, y)$.

The denial pgd in line (4) is transitive. By disposing the cleaning update of line (4) using the procedure of Theorem 5.3 we obtain the following program:

(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) $\mathsf{PerLoc}\ (x, y) :- \mathsf{Per}(x) \wedge \mathsf{Loc}(y) \wedge \mathsf{precede}[x, y]$ (see Example 3.3)
(4) $\mathsf{PerLoc}(x', y') :- \mathsf{PerLoc}(x', y') \setminus \pi_{x', y'}(\mathsf{encloses}[z, x, y] \bowtie \mathsf{encloses}[z', x', y'] \bowtie \mathsf{contains}_{\neq}[z', z] \bowtie \mathsf{PerLoc}(x, y) \bowtie \mathsf{PerLoc}(x', y')$
(5) RETURN $\mathsf{PerLoc}(x, y)$.

For simplicity, we have omitted in line (4) the terms corresponding to $\gamma^{=}[\mathbf{x}, \mathbf{y}]$ and $\iota^{=}[\mathbf{v}, \mathbf{z}]$ in Equation (2) since these are vacuously satisfied in this example.

Note that, strictly speaking, the disposed program above is not an extraction program since line (4) is not a CQ update (its right-hand side is an algebraic expression featuring difference and joins over atomic expressions instead of a conjunction of spanners and atomic expressions). Nevertheless, it does showcase how the program would be executed in practice. If desired, one can always transform the right-hand side of line (4) into a CQ update by expanding the definitions of PerLoc, Per, and Loc.

5.2. JAPE Controls

JAPE [Cunningham et al. 2000] is an instantiation of the Common Pattern Specification Language (CPSL) [Appelt and Onyshkevych 1998], 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 [Chiticariu et al. 2010] (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 5.5. 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 can be expressed in REG. For example, the Appelt control is presented in Example 3.4 with R being the relation symbol Loc. We can also show the following.

THEOREM 5.6. *Each of the denial pgds that correspond to the four JAPE controls is* REG*-disposable.*

In the remainder of this section, we prove Theorem 5.6. In the case of Once, the proof of Theorem 5.6 is by using Theorem 5.3 (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 5.7 below that is of independent interest. We first need a definition.

Let P be a unary spanner. Define the *Kleene star* of P, denoted 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 $a_1 \leqslant \ldots \leqslant a_n$, 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 5.7. *Let P be a unary spanner. If P is regular, then P^* is regular.*

PROOF. Suppose that $\mathsf{SVars}(P) = \{x\}$. We define P^+ exactly like P^* above, except that we add the requirement that $n > 1$. Then P^* is the union $P^+ \cup \epsilon(x)$, where $\epsilon(x)$ is the spanner that assigns to x every empty span. Clearly, $\epsilon(x)$ is a regular spanner. And since the regular spanners are closed under union, it suffices to prove that P^+ is regular whenever P is regular. We will do so in the remainder of the proof.

Let $A = (Q, q_0, q_f, \delta)$ be a vset-automaton such that $[\![A]\!] = P$. We will show how to construct a vset-automaton B, such that $[\![B]\!] = P^+$. For clarification of the proof, we will use the variable y (rather than x) in B.

The idea of the proof is to have B simulating multiple parallel runs ρ of A, where each ρ produces one unique mapping of x to one of the $[a_i, a_{i+1}\rangle$ in the definition of P^*. The crux of the proof is in the observation that, for such a simulation, it is enough for B to keep track on just the *set* of current states the runs are in (hence, bounded memory suffices). We formalize this intuition next.

States. The states of B are triples $\langle prX, inX, pstX\rangle$, where:

- prX is the set of states of the simulated runs that have not opened x yet.
- inX stores the state of the simulated run, if any, that has opened x but has not closed it yet.
- $pstX$ is the set of states of the simulated runs that have closed x already.

The initial state of B is the triple $\langle\{q_0\}, \varnothing, \varnothing\rangle$, and the final state of B is $\langle\varnothing, \varnothing, \{q_f\}\rangle$.

Transitions. We list the transitions of B as ones of various types.

- *Read transitions.* These are transitions (r, σ, r') where $r = \langle prX, inX, pstX\rangle$, $r' = \langle prX', inX', pstX'\rangle$, and the following hold.
  - For each (R, R') among the pairs (prX, prX') and $(pstX, pstX')$ we have the following. For every $q \in R$ there is $q' \in R'$ such that $(q, \sigma, q') \in \delta$, and for every $q' \in R'$ there is $q \in R$ with $(q, \sigma, q') \in \delta$.
  - Either $inX = inX' = \varnothing$ or it is the case that for some $(q, \sigma, q') \in \delta$ we have $inX = \{q\}$ and $inX' = \{q'\}$.
- *Ordinary epsilon transitions.* These transitions have the form (r, ϵ, r') where $r = \langle prX, inX, pstX\rangle$, $r' = \langle prX', inX', pstX'\rangle$, and the following hold.
  - For each (R, R') among the pairs (prX, prX') and $(pstX, pstX')$ we have the following. For every $q \in R$ there is $q' \in R'$ such that either $q = q'$ or $(q, \epsilon, q') \in \delta$, and for every $q' \in R'$ there is $q \in R$ such that either $q = q'$ or $(q, \epsilon, q') \in \delta$.
  - Either $inX = inX'$ or it is the case that for some $(q, \epsilon, q') \in \delta$ we have $inX = \{q\}$ and $inX' = \{q'\}$.

- *Transitions that open y.* These are transitions $(r, y\vdash, r')$ where $r = \langle prX, inX, pstX\rangle$, $r' = \langle prX', inX', pstX'\rangle$, and the following hold.
  - $inX = pstX = pstX' = \varnothing$.
  - There is a transition $(q, x\vdash, q')$ in Q such that $q \in prX, prX'$ is either prX or $prX\setminus\{q\}$ and $inX = \{q'\}$.
- *Epsilon transitions that switch x.* These are transitions of the form (r, ϵ, r') where $r = \langle prX, inX, pstX\rangle$, $r' = \langle prX', inX', pstX'\rangle$, and the following hold.
  - There is a transition $(q, \dashv x, q')$ in Q such that $inX = \{q\}$ and $pstX' = pstX \cup \{q'\}$.
  - There is a transition $(p, x\vdash, p')$ in Q such that $p \in prX$, prX' is either prX or $prX \setminus \{p\}$ and $inX = \{p'\}$.
- *Transitions that close y.* These are transitions $(r, y\vdash, r')$ where $r = \langle prX, inX, pstX\rangle$, $r' = \langle prX', inX', pstX'\rangle$, and the following hold.
  - $prX = prX' = inX' = \varnothing$.
  - There is a transition $(q, \dashv x, q')$ in Q such that $inX = \{q\}$ and $pstX' = pstX \cup \{q'\}$.

That completes the description of the vset-automaton B. The fact that $[\![B]\!] = P^+$ follows fairly easily from the construction, as follows.

Let $\mathbf{d}$ be a document. Let $[a, a'\rangle \in P^+(\mathbf{d})$ be given. Then there are indices $a_1 \leqslant \cdots \leqslant a_n$ where $n > 1$, $a_1 = a$, $a_n = a'$, and $[a_i, a_{i+1}\rangle$ is in $P(\mathbf{d})$ for all $i = 1, \ldots, n-1$. By selecting an accepting run ρ_i for each $[a_i, a_{i+1}\rangle$, we can build an accepting path for B by simulating a parallel run of all the ρ_i. Now let $[a, a'\rangle \in [\![B]\!](\mathbf{d})$ be given. Let ρ be a run that produces $[a, a'\rangle$. We can construct accepting runs ρ_i of A that match the triples $\langle prX, inX, pstX\rangle$ in ρ in the sense that ρ simulates a parallel run of these accepting runs. In particular, it then follows that $\rho(y)$ can be represented as a concatenation of the $\rho_i(x)$. □

LEMMA 5.8. *The denial pgd for the* AppeIt *control is* REG*-disposable.*

PROOF. Let $\mathcal{E}$ be a REG-program that has AppeIt as its single denial pgd. Suppose that the denial pgd is the following:

$$\mathsf{Irl}[x, y] \to R(x) \rhd R(y).$$

Here $\mathsf{Irl}[x, y]$ is expression in REG stating that (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.

We need to construct a noncleaning REG-program $\mathcal{E}_r$ that is equivalent to $\mathcal{E}$. Due to Proposition 4.1, such a program $\mathcal{E}_r$ is equivalent to a regular spanner. Moreover, the definition of R in the program, up to the cleaning update, can be collapsed to a single primitive update, say, $R(z) :- \rho[z]$. We will then assume that our program $\mathcal{E}$ is then the following:

(1) $R(z) :- \rho[z]$
(2) CLEAN($\mathsf{Irl}(x, y) \to R(x) \rhd R(y)$)
(3) RETURN $R(x)$.

In the remainder of this proof we will show that $\mathcal{E}$ is a regular spanner. This would then imply that the first two updates can be replaced with a single update $R(z) :- \rho'[z]$, where ρ' is in REG.

In what follows, we use P to denote the spanner $[\![\rho]\!]$.

Define Q to be the spanner where for each document $\mathbf{d}$, we have that $Q(\mathbf{d})$ consists of those spans $[a, b\rangle$ such that there is c where $b < c$ and $[a, c\rangle$ is in $P(\mathbf{d})$. That is, $Q(\mathbf{d})$ consists of those spans that are strict prefixes of spans in $P(\mathbf{d})$. Then Q is a regular spanner. Moreover, by the closure of regular spanners under complement (Theorem 2.5), the complement of Q is a regular spanner. Let us denote the complement of Q by Q'.

Now define M (for "maximal") to be the spanner where for each document $\mathbf{d}$, we have that $M(\mathbf{d})$ consists of those spans $[a, b\rangle$ in $P(\mathbf{d}) \cap Q'(\mathbf{d})$. Intuitively, $M(\mathbf{d})$ consists of

the right-maximal spans in $P(\mathbf{d})$ (which means that they are maximal when we ignore possible extensions to the left). Then by closure under intersection (a special case of closure under join), M is a regular spanner.

Next, define T to be the spanner where for each document $\mathbf{d}$, we have that $T(\mathbf{d})$ consists of those spans $[a, b\rangle$ such that there are c and d where $a \leqslant c < b$ and $[c, d\rangle$ is in $P(\mathbf{d})$. Then T is a regular spanner. Let us denote the complement of T by N (for "no beginning"). By closure under complement, N is also a regular spanner. Intuitively, N consists of the spans that do not contain any starting point for a span in $P(\mathbf{d})$.

Next, define K to be the spanner where for each document $\mathbf{d}$; we have that $K(\mathbf{d})$ consists of those spans $[1, b\rangle$ that are in $(M \cup N)^*$. Thus, these spans are prefixes of $\mathbf{d}$ (since they are of the form $[1, b\rangle$), and they begin with either a span in $M(\mathbf{d})$ or a span in $N(\mathbf{d})$, followed by a span in $M(\mathbf{d})$ or a span in $N(\mathbf{d})$, and so on, some number of times. Intuitively, this gives us a sequence of right-maximal strings in $P(\mathbf{d})$, possibly preceded by or followed by spans that do that do not contain any starting point for a span in $P(\mathbf{d})$. By closure of regular spanners under union and the Kleene star (Lemma 5.7), we know that K is a regular spanner.

Finally, define S to be the spanner where for each document $\mathbf{d}$, we have that $S(\mathbf{d})$ consists of those spans $[a, b\rangle$ in $M(\mathbf{d})$ that are preceded by a span in $K(\mathbf{d})$. Then S is a regular spanner.

Observe that $S(\mathbf{d})$ consists of those spans in the result of Appelt cleaning. That is, S is equivalent to $\mathcal{E}$. In particular, $\mathcal{E}$ is regular, as claimed. □

LEMMA 5.9. *The denial pgd for the Brill control is* REG*-disposable.*

PROOF. The proof is the same as for Appelt (Lemma 5.8), except that we remove part (2) in the definition of $\mathsf{lrl}[x, y]$. That is, instead of $\mathsf{lrl}[x, y]$ we use $\mathsf{lrl}'[x, y]$ stating that x and y overlap and x starts earlier than y. □

LEMMA 5.10. *The denial pgd for the First control is* REG*-disposable.*

PROOF. The proof is the same as for Appelt, except that we define M to consist of the *minimal* spans (w.r.t. span containment) rather than *maximal* spans in P. More precisely, M should now be defined as the spanner where for each document $\mathbf{d}$, we have that $M(\mathbf{d})$ consists of those spans $[a, b\rangle$ such that $[a, b\rangle$ is in $P(\mathbf{d})$ and there is no c where $c < b$ and $[a, c\rangle$ is in $P(\mathbf{d})$. □

We can now prove Theorem 5.6. The Appelt, Brill, and First controls are covered by Lemmas 5.8, 5.9, and 5.10, respectively. The *Once* control is transitive, and hence, we get REG-disposability by applying Theorem 5.3.

We now make a few comments on the proof of Theorem 5.6. The most interesting part of the proof is Lemma 5.8, which is a surprising result with a somewhat magical proof. This proof of Lemma 5.8 depends on Lemma 5.7, which is an interesting result in its own right.

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 view definitions $R_1, \ldots, R_n$ (ordered by decreasing priority), and we apply the cleaning of Appelt simultaneously to all of these (that is, as if applying it to the union of these), and 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 5.6 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 CLEAN(true $\rightarrow R_i(x) \triangleright R_j(x)$) 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 [Fowler 2003; Institute of Electrical and Electronic Engineers and The Open Group 2013], which is followed by all POSIX compliant tools such as `sed` and `awk`. Formalizations of this policy have been proposed by Vansummeren [2006] and Okui and Suzuki [2010], and multiple efficient algorithms for implementing the policy are known [Cox 2009; Laurikari 2001; Okui and Suzuki 2010]. We show in particular that the POSIX disambiguation policy can be expressed in our framework as a REG-program that uses only 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., Vansummeren [2006] 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 the Appendix. If γ is one of $\varnothing$, 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 $\mathbf{d_1}$ (respectively, $\mathbf{d_2}$) is recursively matched under the POSIX disambiguation policy by γ_1 (respectively, γ_2) to construct a unique parse tree for γ. If γ is a Kleene closure δ^* and $\mathbf{d}$ is nonempty, then POSIX views γ as 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 POSIX disambiguation policy.

Example 5.11. Consider $\gamma = x\{(0 \vee 01)^*\} \cdot y\{1^*\}$ and $\mathbf{d} = 0011$. 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 $y\{1^*\}$, to match the remainder of $\mathbf{d}$. As such, $x\{(0 \vee 01)^*\}$ will match the prefix 001 and $y\{1^*\}$ will match the corresponding suffix 1. We hence bind x to the span $[1, 3\rangle$ and y to $[3, 4\rangle$.

[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*.

[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.

Next, consider $\gamma = (x\{0\} \cdot y\{1\}) \vee (x\{(0 \vee 01)^*\} \cdot y\{1^*\})$ and $\mathbf{d} = 01$. Since the first disjunct matches $\mathbf{d}$, we bind x to the span $[1, 2\rangle$ and y to the span $[2, 3\rangle$ under the POSIX disambiguation policy.

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 5.12. *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.*

We discuss the proof at the end of this section but illustrate its idea by means of the following example.

Example 5.13. Reconsider $\gamma = x\{(0 \vee 01)^*\} \cdot y\{1^*\}$ from Example 5.11. The POSIX disambiguation policy for γ is expressed by the following REG-program:

(1) $R(x, y) :- x\{(0 \vee 01)^*\} \cdot y\{1^*\}$
(2) CLEAN$(\Upsilon_{x_1,y_1,x_2,y_2} \bowtie \mathsf{prefix}[x_1, x_2] \rightarrow R(x_2, y_2) \rhd R(x_1, y_1))$.

Here $\Upsilon_{x_1,y_1,x_2,y_2}$ denotes the universal spanner over variables x_1,y_1,x_2,y_2; and $\mathsf{prefix}[x_1, x_2]$ is the spanner that returns all pairs x_1, x_2 where the span of x_1 is a strict prefix of the span of x_2 (cf. Example 2.6).

In particular, in line (1) this program computes, in relation $R(x, y)$, all possible matches of γ to the input document. The cleaning operation in line (2) disambiguates R cf. the POSIX policy by stating that R-tuple (x_1, y_1) is in conflict with (x_2, y_2) if x_1 is a strict prefix of x_2. In that case, (x_2, y_2) is to be preferred.

Since every transitive denial pgd is REG-disposable by Theorem 5.3, we immediately obtain from Lemma 5.12 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 2.7 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 5.14. *For every regex formula γ, the spanner* $\mathsf{posix}[\gamma]$ *is definable in* RGX.

In the remainder of this section, we prove Lemma 5.12. We do so by establishing a more general result. To state it, we require the notion of a *submatch*. Intuitively, a submatch of regex formula γ is a tuple that specifies how γ matches on a substring of input $\mathbf{d}$ according to the POSIX disambiguation policy, together with the information which substring (i.e., span) was matched.

Formally, let γ be a regex formula, let V be the set of all variables occurring in γ, and let x be a variable not occurring in V. We say that $(V \cup \{x\}, \mathbf{d})$-tuple μ specifies a POSIX *submatch of γ on $\mathbf{d}$ with respect to x* if $\mu(x) = [i, j\rangle$ for some i, j and the $\mu(y)$ for $y \in V$ specify how γ would be matched under the POSIX policy when it is applied to $\mathbf{d}_{[i,j\rangle}$. That is, there must be a tuple $\nu \in \mathsf{posix}[\gamma](\mathbf{d}_{[i,j\rangle})$ such that for every $y \in V$ we have $\mu(y) = [i + k, i + l\rangle$ with $\nu(y) = [k, l\rangle$.

We will now prove the following.

LEMMA 5.15. *Let γ be a regex formula and let x be a variable not occurring in γ. There exists an unambiguous extraction* REG*-program* $\mathrm{SubMatches}_\gamma$ *such that, for every* $\mathbf{d}$*,*

the unique $\mathbf{d}$*-relation* R *returned by* $\text{SubMatches}_\gamma(\mathbf{d})$ *consists of all submatches of* γ *on* $\mathbf{d}$ *w.r.t.* x*. Moreover, all cleaning updates in* SubMatches_γ *are of the form* $\textsc{Clean}(p)$ *with* p *a transitive denial pgd.*

Lemma 5.12 readily follows from Lemma 5.15 since we can now compute $\mathsf{posix}[\gamma]$ by means of the following program. Assume that $\text{SubMatches}_\gamma = (\mathbf{S}, U, R(x, \mathbf{y}))$ with $\mathbf{y}$ an enumeration of the span variables in γ. Let Result be a new relation symbol of arity $|V|$.

(1) Do all updates of U
(2) $\text{Fullspan}(x) :- x\{\Sigma^*\}$
(3) $\text{Result}(\mathbf{y}) :- R(x, \mathbf{y}) \wedge \text{Fullspan}(x)$
(4) $\textsc{Return}(\text{Result}(\mathbf{y}))$

Here Fullspan computes the span $[1, |\mathbf{d}| + 1\rangle$. By joining this with R, we select from R the submatch of γ on $\mathbf{d}$ with respect to the entire document, that is, $\mathsf{posix}[\gamma](\mathbf{d})$.

Proof of Lemma 5.15. We prove Lemma 5.15 by induction on γ. Let V be the set of variables mentioned in γ and let $\mathbf{y}$ be a sequence of variables that enumerates V.

- Case $\gamma = \varnothing$. Then SubMatches_γ is the program
  (1) $\text{Result}(x) :- \varnothing$
  (2) $\textsc{Return}(\text{Result}(x))$
- Case $\gamma = \epsilon$.
  (1) $\text{Result}(x) :- \Sigma^* \cdot x\{\epsilon\} \cdot \Sigma^*$
  (2) $\textsc{Return}(\text{Result}(x))$
- Case $\gamma = \sigma \in \Sigma$. Then SubMatches_γ is the program
  (1) $\text{Result}(x) :- \Sigma^* \cdot x\{\sigma\} \cdot \Sigma^*$
  (2) $\textsc{Return}(\text{Result}(x))$
- Case $\gamma = \gamma_1 \vee \gamma_2$. First observe that, since γ is a regex formula, and thus functional on V, it must hold that $V = \mathsf{SVars}(\gamma_1) = \mathsf{SVars}(\gamma_2)$. By induction hypothesis, we have programs
$$\text{SubMatches}_{\gamma_1} = (\mathbf{S}_1, U_1, R_1(x, \mathbf{z}_1))$$
$$\text{SubMatches}_{\gamma_2} = (\mathbf{S}_2, U_2, R_2(x, \mathbf{z}_2))$$
  for γ_1 and γ_2, respectively. We may assume w.l.o.g. that $\mathbf{S}_1$ and $\mathbf{S}_2$ are disjoint. Then SubMatches_γ is the program
  (1) Do all updates of U_1
  (2) Do all updates of U_2
  (3) $\textsc{Clean}(p)$
  (4) $\text{Result}(x, \mathbf{y}) :- R_1(x, \mathbf{y}) \cup R_2(x, \mathbf{y})$
  (5) $\textsc{Return}(\text{Result}(x, \mathbf{y}))$
  where Result a new relation symbol of arity $|V| + 1$ and p is the denial pgd
$$p \stackrel{\text{def}}{=} (\Sigma^* \cdot x\{\gamma_1[\mathbf{z}_1]\} \cdot \Sigma^*) \bowtie (\Sigma^* \cdot x\{\gamma_2[\mathbf{z}_2]\} \cdot \Sigma^*) \rightarrow R_1(x, \mathbf{z}_1) \rhd R_2(x, \mathbf{z}_2).$$
  Here $\mathbf{z}_1$ and $\mathbf{z}_2$ are disjoint sequences of variables, all distinct from x. Since R_1 and R_2 are distinct relation symbols, transitivity of p vacuously holds. Hence, by induction hypothesis, all of SubMatches_γ's cleaning updates are of the form $\textsc{Clean}(p')$ with p' a transitive denial pgd.
  Correctness of SubMatches_γ follows readily from correctness of $\text{SubMatches}_{\gamma_1}$ and $\text{SubMatches}_{\gamma_2}$ (by induction hypothesis) and the fact that $\textsc{Clean}(p)$ removes all tuples from R_2 where submatch on the same subspan x can be found in R_1.
- Case $\gamma = \gamma_1 \cdot \gamma_2$. First observe that, since γ is a regex formula, and thus functional on V, it must hold that $V_1 = \mathsf{SVars}(\gamma_1)$ and $V_2 = \mathsf{SVars}(\gamma_2)$ are disjoint. Let $\mathbf{y}_1$ and

$\mathbf{y}_2$ be enumerations of V_1 and V_2, respectively. Let z_1 and z_2 be two new variables, distinct from x, and any variable in V. By induction hypothesis, we have programs

$$\text{SubMatches}_{\gamma_1} = (\mathbf{S}_1, U_1, R_1(x, \mathbf{y}_1))$$
$$\text{SubMatches}_{\gamma_2} = (\mathbf{S}_2, U_2, R_2(x, \mathbf{y}_2))$$

for γ_1 and γ_2, respectively. We may assume w.l.o.g. that $\mathbf{S}_1$ and $\mathbf{S}_2$ are disjoint. Assume that P is a new ternary relation symbol and Result a new relation symbol of arity $|V| + 1$. Then SubMatches_γ is the program
(1) Do all updates of U_1
(2) Do all updates of U_2
(3) $P(x, z_1, z_2) :- \Sigma^* \cdot x\{z_1\{\gamma_1\} \cdot z_2\{\gamma_2\}\} \cdot \Sigma^*$
(4) CLEAN(p)
(5) $\text{Result}(x, \mathbf{y}) :- R_1(z_1, \mathbf{y}_1) \wedge R_2(z_2, \mathbf{y}_2) \wedge P(x, z_1, z_2)$
(6) RETURN(Result($x, \mathbf{y}$))
with p the denial pgd

$$\Upsilon_{x,z_1,z_2,z_1',z_2'} \bowtie \mathsf{prefix}[z_1', z_1] \rightarrow P(x, z_1, z_2) \rhd P(x, z_1', z_2').$$

Here $\Upsilon_{x,z_1,z_2,z_1',z_2'}$ denotes the universal spanner over variables x, z_1, z_2, z_1', and z_2'; and $\mathsf{prefix}[z_1', z_2]$ is the spanner that returns all pairs z_1', z_2' where the span of z_1' is a strict prefix of the span of z_1 (cf. Example 2.6).

Essentially, line (3) computes $P(x, z_1, z_2)$ to hold for all tuples $([i, j\rangle, v_1, v_2)$ such that
- $\mathbf{d}_{[i,j\rangle}$ is matched by γ,
- v_1 is the span of the prefix $\mathbf{d}_1$ of $\mathbf{d}_{[i,j\rangle}$ matched by γ_1; and
- v_2 is the span of the corresponding suffix $\mathbf{d}_2$ of $\mathbf{d}_{[i,j\rangle}$ of (which is matched by γ_2).

Then line (4) selects from P the single tuple that maximizes the length of the prefix delimited by v_1. Finally, line (5) joins the submatches of γ_1 and γ_2 based on the identified prefix and suffix.

Correctness of SubMatches_γ then follows from these observations and the correctness of $\text{SubMatches}_{\gamma_1}$ and $\text{SubMatches}_{\gamma_2}$ (by induction hypothesis).

Moreover, it is readily verified that p is transitive. Hence, by induction hypothesis, all of SubMatches_γ cleaning updates are of the form CLEAN(p') with p' a transitive denial pgd.

- Case $\gamma = \delta^*$. It is readily verified that, since γ is a regex formula (and hence is functional on V), it must be the case that $V = \varnothing$. Therefore, construction of SubMatches_γ is trivial: It is readily verified that it suffices to let SubMatches_γ be the program
(1) $\text{Result}(x) :- \Sigma^* \cdot x\{\gamma\} \cdot \Sigma^*$
(2) RETURN(Result(x))
with Result a unary relation symbol.
- Case $\gamma = z\{\delta\}$. Let $W = \mathsf{SVars}(\delta)$. Observe that, since γ is a regex formula (and hence functional on V), we have $z \notin W$. Let $\mathbf{w}$ be an enumeration of W. We assume w.l.o.g. that $\mathbf{y} = z, \mathbf{w}$. By induction hypothesis we have a program

$$\text{SubMatches}_\delta = (\mathbf{S}_1, U_1, R_1(z, \mathbf{w}))$$

that computes the POSIX submatches of δ with respect to z. Then SubMatches_γ is the following program. Let Result be a new relation symbol of arity $|V| + 1$.
(1) Do all updates of U_1
(2) $\text{Result}(x, z, \mathbf{w}) :- R_1(z, \mathbf{w}) \wedge \Sigma^* \cdot x\{z\{\Sigma^*\}\} \cdot \Sigma^*$.
(3) RETURN(Result($x, \mathbf{y}$))
Here the spanner $\Sigma^* \cdot x\{z\{\Sigma^*\}\} \cdot \Sigma^*$ computes those pairs of spans (x, z) with $x = z$ (equality on spans).

Correctness of SubMatches_γ follows immediately from the induction hypothesis. □

5.4. Core Spanners

Recall that REG is the closure of RGX under union, projection, and natural join. The class Core of *core spanners* [Fagin et al. 2015] is obtained by adding to this list of operators the *string-equality selection*, denoted $\zeta^=$. Formally, given an expression ρ in REG and two variables $x, y \in \mathsf{SVars}(\rho)$, the spanner defined by $\zeta^=_{x,y}(\rho)$ 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 do not extend to the core spanners.

THEOREM 5.16. *If p is the maximal-container denial pgd or one of the JAPE denial pgds, then p is not* Core*-disposable.*

We prove Theorem 5.16 in the remainder of this section. Assume that the alphabet Σ contains the symbols 0, 1, and $\#$. Define L to be the language of all the strings $\mathbf{s}\#\mathbf{t}$ where $\mathbf{s}$ and $\mathbf{t}$ are in $\{0,1\}^+$ and $\mathbf{s}$ is *not* a substring of $\mathbf{t}$.

In Fagin et al. [2015], we show the following.

LEMMA 5.17 [FAGIN ET AL. 2015]. *No spanner in* Core *recognizes L.*

We can now prove Theorem 5.16 by proving a more general result. Let p be a denial pgd of the form

$$\rho[x, y] \rightarrow R(x) \triangleright S(y),$$

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 $R(a) \triangleright S(b)$ is implied by p. Note that all the denial pgds in Theorem 5.16 favor strict containers. We have the following.

THEOREM 5.18. *If p is a pgd with unary relation symbols, and p favors strict containers, then p is not* Core*-disposable.*

PROOF. Suppose that p is the denial pgd $\rho[x, y] \rightarrow S(x) \triangleright S(y)$. Let $\mathcal{E}$ be the following program:

(1) $R(x) :- \{0,1\}^+ \cdot x\{\#\} \cdot \{0,1\}^+$
(2) $S(x) :- R(x)$
(3) $S(x) :- \zeta^=_{y,z}(x\{y\{\Sigma^*\} \cdot \# \cdot \Sigma^* \cdot z\{\Sigma^*\}\} \cdot \Sigma^*)$
(4) CLEAN($\rho[x, y] \rightarrow S(x) \triangleright S(y)$)
(5) $T() :- S(x) \wedge R(x)$
(6) RETURN($T()$)

Let $\mathbf{d}$ be a document of the form $\mathbf{s}\#\mathbf{t}$ where $\mathbf{s}$ and $\mathbf{t}$ are in $\{0,1\}^+$. When evaluating $\mathcal{E}$ on $\mathbf{d}$, after line 2 each of the relations R and S consists of the span of $\#$. In line 3, we add to S the span of the entire document *if and only if* it is the case that $\mathbf{s}$ is a substring of $\mathbf{t}$. Line 4 applies a cleaning update to S, and in line 5, we have that T is true if and only if $S(x)$ still contains the span of $\#$. The return value is T.

Since p favors strict containers, the cleaning of line 4 will have the following effect on S:

- If $\mathbf{s}$ is a substring of $\mathbf{t}$, then S contains the spans of both $\#$ and $\mathbf{d}$; then the cleaning will remove the former and leave the latter.
- If $\mathbf{s}$ is a not substring of $\mathbf{t}$, then S contains only the span of S; then the cleaning will not change S.

We then conclude that if $\mathbf{s}$ is a substring of $\mathbf{t}$, then T is false when $\mathcal{E}$ terminates (because R and S are disjoint), and then $\mathcal{E}(\mathbf{d})$ is false; otherwise (if $\mathbf{s}$ is not a substring of $\mathbf{t}$), then

$\mathcal{E}(\mathbf{d})$ is true. In other words, $\mathcal{E}$ recognizes L. But if the cleaner is Core-disposable, then $\mathcal{E}$ is a core spanner, in contradiction to Theorem 5.17. □

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 deciding if a given pgd 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.

There are several more directions for interesting future work. One is the challenge of devising a sufficient condition for unambiguity of cleaners, featuring low complexity, and robustness to realistic needs. Another direction is towards different types of repairing. Our focus has been on *deletion repairs* (where one repairs by deleting tuples), since we were mainly concerned about elimination of redundant extractions, in accordance to the cleaning policies of rule based IE systems. It would be interesting to incorporate other notions of repairing, such as repairing via value substitution [Beskales et al. 2014; Bohannon et al. 2005]. 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.

APPENDIX

This appendix contains the formal definition of the POSIX disambiguation policy.

We follow Okui and Suzuki [2010] and formalize the POSIX disambiguation policy by means of a partial order on the set of γ-parses.

Actually, like Okui and Suzuki, to comply with the POSIX requirement that "*a subexpression repeated by '*' shall not match an empty string unless it is the only match for the repetition . . .*" we are required to consider only so-called *canonical* γ-parses [Okui and Suzuki 2010].

To define canonical γ-parses, let $|t|$ denote the length of the string matched by parse t (i.e., $|t|$ is the number of occurrence of symbols in Σ in t). For example, for $t = \cdot(a, \cdot(\epsilon, b))$, since t is parsing ab, we have $|t| = 2$.

A γ-parse for $\mathbf{d}$ is *canonical* if for each subtree of the form $^*(t_1 \dots t_n)$ with $n \geqslant 2$ it holds that $|t_i| > 0$, for $1 \leqslant i \leqslant n$. For example, let $\gamma = (a^*)^*$. Then the γ-parse $^*(^*())$ for $\mathbf{d} = \epsilon$ is canonical but $^*(^*(), ^*())$ is not. Similarly, the γ-parse $^*(a(), ^*())$ for $\mathbf{d} = a$ is not canonical.

Observe that, in general, if t is a γ-parse for document $\mathbf{d}$, then we can always convert t into a canonical γ-parse by removing, for every subtree of the form $^*(t_1 \dots t_n)$ with $n \geqslant 2$, the subtrees t_i with $|t_i| = 0$, where $2 \leqslant i \leqslant n$. (Recall that our trees are unranked, so we are free to remove children from *-labeled nodes.) Clearly, the $(V, \mathbf{d})$ tuple defined by this canonical parse is the same as that of t. As such, we may restrict ourselves without loss of generality to considering canonical parses only.

Let γ be a variable regex and let t_1 and t_2 be two canonical γ-parses. We define parse t_1 to be *POSIX preferable* to t_2, denoted $t_1 \gg t_2$, inductively as follows.

- If $|t_1| > |t_2|$, then $t_1 \gg t_2$.
- If $\gamma = \gamma_1 \vee \gamma_2$, $t_1 = \vee_i(t_1')$, $t_2 = \vee_j(t_2')$, and $|t_1| = |t_2|$, then $t_1 \gg t_2$ if $i < j$ or if $i = j$ and $t_1' \gg t_2'$.
- If $\gamma = \gamma_1 \cdot \gamma_2$, $t_1 = \cdot(t_1^1, t_1^2)$, $t_2 = \cdot(t_2^1, t_2^2)$, and $|t_1| = |t_2|$, then $t_1 \gg t_2$ if $t_1^1 \gg t_2^1$, or if $t_1^1 = t_2^1$ and $t_1^2 \gg t_2^2$.
- If $\gamma = \delta^*$, $t_1 = {}^*(t_1^1, \ldots, t_1^k)$, $t_2 = {}^*(t_2^1, \ldots, t_2^l)$, and $|t_1| = |t_2|$, then $t_1 \gg t_2$ if there exists some i with $1 \leqslant i \leqslant \min(k, l)$ such that $t_1^j = t_2^j$ for all j with $1 \leqslant j < i$ and $t_i^1 \gg t_i^2$.
- If $\gamma = x\{\delta\}$, $t_1 = x(t_1')$, $t_2 = x(t_2')$, and $|t_1| = |t_2|$, then $t_1 \gg t_2$ if $t_1' \gg t_2'$.

Example A.1. Reconsider regex formula $\gamma = x\{(0 \vee 01)\} \cdot y\{(1 \vee \epsilon)\}$ from Example 5.11. Let t_1 and t_2 be the γ-parses for $\mathbf{d} = 01$ shown in Figure 3 on the left and right, respectively. Then $t_1 \gg t_2$. Indeed, the left child subtree of t_1 is POSIX preferable to the left child of t_2 since it matches a longer string.

Let $\mathsf{Parses}(\gamma, \mathbf{d})$ denote the set of all canonical γ-parses for $\mathbf{d}$. The following is now straightforward to obtain ([Okui and Suzuki 2010]).

PROPOSITION A.2 ([OKUI AND SUZUKI 2010]). *$\gg$ is a strict total order on* $\mathsf{Parses}(\gamma, \mathbf{d})$ *for all regex formulas γ and documents* $\mathbf{d}$.

As a consequence, for each γ and $\mathbf{d}$ with $\mathsf{Parses}(\gamma, \mathbf{d})$ nonempty there is a unique maximally POSIX preferred parse $t \in \mathsf{Parses}(\gamma, \mathbf{d})$, in the sense that $t \gg t'$ for all $t' \neq t$ in $\mathsf{Parses}(\gamma, \mathbf{d})$. We write $\mathsf{posix}[\gamma]$ for the spanner represented by the regex formula γ under the POSIX disambiguation policy; this is the spanner where $\mathsf{SVars}(\mathsf{posix}[\gamma])$ is the set $\mathsf{SVars}(\gamma)$ and where $\mathsf{posix}[\gamma](\mathbf{d})$ is the span relation $\{\mu^t \mid t \text{ is } \gg\text{-maximal canonical } \gamma\text{-parse for } \mathbf{d}\}$. Here μ^t denotes the $\mathbf{d}$-tuple defined by functional γ-parse t for $\mathbf{d}$. Note that $\mathsf{posix}[\gamma](\mathbf{d})$ is hence either empty or a singleton.

ACKNOWLEDGMENTS

We are extremely grateful to Phokion G. Kolaitis for suggestions that had significant impact on the article. We also thank Martin Kutrib and Frank Neven for insightful discussions on multihead automata.

REFERENCES

Jitendra Ajmera, Hyung-Il Ahn, Meena Nagarajan, Ashish Verma, Danish Contractor, Stephen Dill, and Matthew Denesuk. 2013. A CRM system for social media: Challenges and experiences. In *WWW*. 49–58.

Douglas E. Appelt and Boyan Onyshkevych. 1998. The common pattern specification language. In *Proceedings of the TIPSTER Text Program: Phase III*. 23–30.

Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. 1999. Consistent query answers in inconsistent databases. In *PODS*. 68–79.

Edward Benson, Aria Haghighi, and Regina Barzilay. 2011. Event discovery in social media feeds. In *ACL*. 389–398.

Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. 2013. Data cleaning and query answering with matching dependencies and matching functions. *Theor. Comput. Syst.* 52, 3 (2013), 441–482.

George Beskales, Ihab F. Ilyas, Lukasz Golab, and Artur Galiullin. 2014. Sampling from repairs of conditional functional dependency violations. *VLDB J.* 23, 1 (2014), 103–128.

Jens Bleiholder and Felix Naumann. 2008. Data fusion. *ACM Comput. Surv.* 41, 1 (2008).

Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. 2005. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 143–154.

Laura Chiticariu, Vivian Chu, Sajib Dasgupta, Thilo W. Goetz, Howard Ho, Rajasekar Krishnamurthy, Alexander Lang, Yunyao Li, Bin Liu, Sriram Raghavan, and others. 2011. The SystemT IDE: An

integrated development environment for information extraction rules. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 1291–1294.

Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, and Shivakumar Vaithyanathan. 2010. SystemT: An algebraic approach to declarative information extraction. In *ACL*. 128–137.

Laura Chiticariu, Yunyao Li, and Frederick R. Reiss. 2013. Rule-based information extraction is dead! Long live rule-based information extraction systems!. In *EMNLP*. 827–832.

Rus Cox. 2009. Regular Expression Matching: the Virtual Machine Approach. Digression: POSIX Submatching. Retrieved from http://swtch.com/ rsc/regexp/regexp2.html.

Hamish Cunningham. 2002. GATE, a general architecture for text engineering. *Comput. Humanities* 36, 2 (2002), 223–254.

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

Gerald DeJong. 1982. An overview of the FRUMP system. In *Strategies for Natural Language Processing*, Wendy G. Lehnert and Martin H. Ringle (Eds.). Lawrence Erlbaum Associates, 149–176.

Maximilian Dylla, Iris Miliaraki, and Martin Theobald. 2013. A temporal-probabilistic database model for information extraction. *Proc. VLDB* 6, 14 (2013), 1810–1821.

Ronald Fagin, Benny Kimelfeld, Yunyao Li, Sriram Raghavan, and Shivakumar Vaithyanathan. 2011. Rewrite rules for search database systems. In *PODS*. 271–282.

Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2014. Cleaning inconsistencies in information extraction via prioritized repairs. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'14)*. ACM, New York, NY, 164–175. DOI:http://dx.doi.org/10.1145/2594538.2594540

Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document spanners: A formal approach to information extraction. *J. ACM* 62, 2 (2015), 12. DOI:http://dx.doi.org/10.1145/2699442

Wenfei Fan. 2008. Dependencies revisited for improving data quality. In *PODS*. 159–170.

Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. 2011a. Dynamic constraints for record matching. *VLDB J.* 20, 4 (2011), 495–520.

Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. 2011b. Interaction between record matching and data repairing. In *SIGMOD Conference*. 469–480.

David A. Ferrucci and Adam Lally. 2004. UIMA: An architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.* 10, 3–4 (2004), 327–348.

Glenn Fowler. 2003. An interpreation of the POSIX Regex Standard (2003). http://gsf.cococlyde.org/download/re-interpretation.tgz.

Dayne Freitag. 1998. Toward general-purpose learning for information extraction. In *COLING-ACL*. 404–408.

Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*. 149–158.

Ralph Grishman and Beth Sundheim. 1996. Message understanding conference-6: A brief history. In *COLING*. 466–471.

Markus Holzer, Martin Kutrib, and Andreas Malcher. 2008. Multi-head finite automata: Characterizations, concepts and open problems. In *CSP (EPTCS)*, Vol. 1. 93–107.

Institute of Electrical and Electronic Engineers and The Open Group. 2013. The Open Group Base Specifications Issue 7. (2013). IEEE Std 1003.1, 2013 Edition.

John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*. 282–289.

Ville Laurikari. 2001. *Efficient Submatch Addressing for Regular Expressions*. Master's thesis. Helsinki University of Technology.

T. R. Leek. 1997. *Information Extraction Using Hidden Markov Models*. Master's thesis. UC San Diego.

Bin Liu, Laura Chiticariu, Vivian Chu, H. V. Jagadish, and Frederick Reiss. 2010. Automatic rule refinement for information extraction. *Proc. VLDB* 3, 1 (2010), 588–597.

Shuai Ma, Wenfei Fan, and Loreto Bravo. 2014. Extending inclusion dependencies with conditions. *Theor. Comput. Sci.* 515 (2014), 64–95.

Andrew McCallum, Dayne Freitag, and Fernando C. N. Pereira. 2000. Maximum entropy Markov models for information extraction and segmentation. In *ICML*. 591–598.

Feng Niu, Christopher Ré, AnHai Doan, and Jude W. Shavlik. 2011. Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. *Proc. VLDB* 4, 6 (2011), 373–384.

Satoshi Okui and Taro Suzuki. 2010. Disambiguation in regular expression matching via position automata with augmented transitions. In *CIAA (Lecture Notes in Computer Science)*, Michael Domaratzki and Kai Salomaa (Eds.), Vol. 6482. 231–240.

Hoifung Poon and Pedro Domingos. 2007. Joint inference in information extraction. In *AAAI*. AAAI Press, 913–918.

Frederick Reiss, Sriram Raghavan, Rajasekar Krishnamurthy, Huaiyu Zhu, and Shivakumar Vaithyanathan. 2008. An algebraic approach to rule-based information extraction. In *ICDE*. 933–942.

Ellen Riloff. 1993. Automatically constructing a dictionary for information extraction tasks. In *AAAI*. 811–816.

Sudeepa Roy, Laura Chiticariu, Vitaly Feldman, Frederick R. Reiss, and Huaiyu Zhu. 2013. Provenance-based dictionary refinement in information extraction. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 457–468.

Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. 2007. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*. 1033–1044.

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

Stephen Soderland, David Fisher, Jonathan Aseltine, and Wendy G. Lehnert. 1995. CRYSTAL: Inducing a conceptual dictionary. In *IJCAI*. 1314–1321.

Slawek Staworko, Jan Chomicki, and Jerzy Marcinkowski. 2012. Prioritized repairing and consistent query answering in relational databases. *Ann. Math. Artif. Intell.* 64, 2–3 (2012), 209–246.

Stijn Vansummeren. 2006. Type inference for unique pattern matching. *ACM Trans. Program. Lang. Syst.* 28, 3 (2006), 389–428.

Hua Xu, Shane P. Stenner, Son Doan, Kevin B. Johnson, Lemuel R. Waitman, and Joshua C. Denny. 2010. Application of information technology: MedEx: A medication information extraction system for clinical narratives. *JAMIA* 17, 1 (2010), 19–24.

Huaiyu Zhu, Sriram Raghavan, Shivakumar Vaithyanathan, and Alexander Löser. 2007. Navigating the intranet with high precision. In *WWW*. 491–500.

Received January 2015; revised September 2015; accepted November 2015