ABSTRACT

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

We embark on the establishment of a framework for declarative cleaning of inconsistencies in IE, though principles of database theory. Specifically, building upon the formalism of document spanners for IE, we adopt the concept of prioritized repairs, which has been recently proposed as an extension of the traditional database repairs to incorporate priorities among conflicting facts. We show that our framework captures the popular cleaning policies, as well as the POSIX semantics for extraction through regular expressions. We explore the problem of determining whether a cleaning declaration is unambiguous (i.e., always results in a single repair), and whether it increases the expressive power of the extraction language. We give both positive and negative results, some of which are general, and some of which apply to policies used in practice.
and so on. The database research community has proposed principled ways to capture, manage and resolve data inconsistency [3,5,6,15–17,29,38]. Here, the identification and capturing of inconsistencies is usually done through the use of specialized constraints and dependencies [3,15,16,29,38]. A prominent formalism for specifying inconsistencies in this respect are the denial constraints [3,38] that can specify, for example, that no two persons can have the same driver’s license number, or that no single person can have multiple residential addresses. To manage and resolve data inconsistencies, the database research community has proposed a wide variety of methods, ranging from consistent query answering [3], to formalisms for the declarative specification of conflict resolution by means of prioritized repairs [38], to data cleaning and data fusion tools that introduce domain-specific operators for removing data inconsistencies [5,6,17].

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

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

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

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. [38], which extend the concept of repairs with priorities among facts that, eventually, translate into priorities among repairs. More precisely, Staworko et al. assume that in addition to denial constraints, the inconsistent database is associated with a binary relation $\succ$, called priority, over the facts (where $f_1 \succ f_2$ means that $f_1$ has priority over $f_2$). We say that a repair $J$ can be improved if we can add to $J$ a new database fact $f$ and retain consistency of $J$ by removing only facts that are inferior to $f$ (according to the priority). The idea is to restrict the set of repairs to those that are Pareto-optimal (or just optimal hereafter), where an optimal repair is one that cannot be improved.¹

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

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

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

¹Staworko et al. [38] also study global optimality as an alternative to Pareto optimality. As we discuss in Section 3, it turns out that all the results in this paper hold true even if we adopt global optimality instead of the Pareto one.
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 \( \psi(x) \rightarrow (\phi_1(x) \lor \phi_2(x)) \), which has the same semantics as a pgd, but it also specifies that \( \phi_1(x) \) and \( \phi_2(x) \) 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 [13]). Such conditions are beyond the scope of this paper, and are left as important directions for future work.

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

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

2. DOCUMENT SPANNERS

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

2.1 Strings and Spans

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

Example 2.1. In all of the examples throughout the paper, we consider the example alphabet \( \Sigma \) which consists of the lowercase and capital letters from the English alphabet (i.e., \( a, z \) and \( \wedge, \cdots, \zeta \)), the comma symbol ("", "") and the underscore symbol ("_", "_") that stands for whitespace. (We use a restricted alphabet for simplicity.) Figure 1 depicts an example document \( d \) in \( \Sigma^* \). For ease of later reference, it also depicts the index of each character in \( d \). Figure 2 shows two tables containing spans of \( d \). Observe that the spans in the left table are those that correspond to words in \( d \) that are names of US states (Georgia, Washington and Virginia). For example, the span \([21,28]\) 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 \( \Sigma^* \) and \( SVars \) are disjoint. For a finite set \( V \subseteq SVars \) of variables and a document \( d \in \Sigma^* \), a \((V,d)\)-tuple is a mapping \( \mu : V \rightarrow \text{Spans}(d) \) that assigns a span of \( d \) to each variable in \( V \). A \((V,d)\)-relation is a set of \((V,d)\)-tuples. A document spanner (or just spanner for short) is a function \( P \) that is associated with a finite set \( V \) of variables, denoted \( SVars(P) \), and that maps every document \( d \) to a \((V,d)\)-relation.

Example 2.2. Throughout our running example (which started in Example 2.1) we will define several spanners. Two of those are denoted as \([p_{an}]\) and \([p_{roc}]\), where \( SVars([p_{an}]) = \{x\} \) and \( SVars([p_{roc}]) = \{x_1, x_2, y\} \). 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 \( d \) of Figure 1.

Let \( P \) be a spanner with \( SVars(P) = V \). Let \( d \in \Sigma^* \) be a document, and let \( \mu \in P(d) \) be a \((V,d)\)-tuple. We say that \( \mu \) is hierarchical if for all variables \( x, y \in SVars(P) \) 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 \( \mu \) is hierarchical for all \( d \in \Sigma^* \) and \( \mu \in P(d) \). Observe that for two variables \( x \) and \( y \) of a hierarchical spanner, it may be the case that, over the same document, one tuple maps \( x \) to a subspan of \( y \), another tuple maps \( y \) to a subspan of \( x \), and a third tuple maps \( x \) and \( y \) to disjoint spans.

2.3 Spanner Representation Systems

By a spanner representation system we refer collectively to any manner of specifying spanners through finite objects. In previous work [14] we defined several representation systems (by means of
regular expressions, special types of automata, and relational algebra). Here, we recall the definition of the regex formula system, as well as its closure under relational algebra.

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

$$\gamma := \emptyset \mid \varepsilon \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^* \mid x_1 \gamma$$

(1)

The added alternative is $$x_1 \gamma$$, where $$x \in \text{SVars}$$. We denote by $$\text{SVars}(\gamma)$$ the set of variables that occur in $$\gamma$$. We use $$\gamma^+$$ as abbreviations of $$\gamma \cdot \gamma^*$$.  

A variable regex can be matched against a document in multiple ways, or more formally, there can be multiple parse trees showing that a document matches a variable regex. Each such a parse tree naturally associates variables with spans. It is possible, however, that in a parse tree a variable is not associated with any span, or is associated with multiple spans. If every variable is associated with precisely one span, then the parse tree is said to be functional. A variable regex is called a regex formula if it has only functional parse trees on every input document. An example of a variable regex that is not a regex formula is $$x_1\{a\}^*$$, because a match against $$aa$$ assigns $$x_2$$ to two spans. We refer to Fagin et al. [14] for the full formal definition of regex formulas. By RGX we denote the class of regex formulas. A regex formula $$\gamma$$ naturally viewed as representing a spanner, and by $$[\gamma]$$ we denote the spanner that is represented by $$\gamma$$. 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.

- $$[a-zA-Z]$$ denotes the disjunction $$a \vee \cdots \vee z$$;
- $$[\_A-Z]$$ denotes the disjunction $$\_A \vee \cdots \vee \_Z$$;
- and $$\{a-zA-Z\}$$ denotes the disjunction $$[a-z] \vee [\_A-Z]$$.

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

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

$$\gamma_{\text{tok}} := (\varepsilon \vee (\Sigma^* \cdot \_)) \cdot x_1\{a-zA-Z\}^* \cdot (\varepsilon \vee (\Sigma^* \cdot \_)) \cdot \varepsilon$$

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

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

$$\gamma_{\text{Cap}} := \Sigma^* \cdot x_1\{\_A-Z\} \cdot \Sigma^* \cdot \Sigma^*$$

When applied to the document $$d$$ of Figure 1, the resulting spans include $$[1, 7], [13, 19], [13, 28]$$, 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.

$$\gamma_{\text{name}} := \Sigma^* \cdot x_1\{\text{Georgia}\text{\_, Virginia}\text{\_, Washington}\} \cdot \Sigma^*$$

When applied to the document $$d$$ of Figure 1, the resulting spans are $$[21, 28], [30, 40]$$, and $$[60, 68]$$.

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

$$\gamma_{\text{loc}} := \Sigma^* \cdot y_1x_1\{\Sigma^*\} \cdot \Sigma^* \cdot y_2x_2\{\Sigma^*\} \cdot \Sigma^*$$

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

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

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

**Example 2.4.** Using $$\gamma_{\text{tok}}$$, $$\gamma_{\text{name}}$$ and $$\gamma_{\text{loc}}$$ from Example 2.3, we define several regular spanners.

- The spanner $$\rho_{\text{tok}}$$ extracts all the tokens that are names of US states; $$\rho_{\text{tok}} := \gamma_{\text{tok}} \times [\text{Georgia}, \_\text{\_, Virginia}, \_\text{\_, Washington}]$$.
- The spanner $$\rho_{\text{Cap}}$$ extracts all the tokens beginning with a capital letter; $$\rho_{\text{Cap}} := \gamma_{\text{Cap}} \times [\_A-Z\_\text{\_, }\Sigma^*]$$.
- The spanner $$\rho_{\text{loc}}$$ extracts spans of strings like “city, state”: $$\rho_{\text{loc}} := \rho_{\text{Cap}}[x_1/x] \times \rho_{\text{tok}}[x_2/x] \times \gamma_{\text{loc}}$$.  

The results of applying the spanners $$[\rho_{\text{tok}}]$$ and $$[\rho_{\text{loc}}]$$ to the document $$d$$ of Figure 1 are in Figure 2.  

Fagin et al. [14] proved the following.

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

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

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

3.1 Signatures and Instances

A signature is a finite sequence $\mathbf{S} = \langle R_1, \ldots, R_m \rangle$ of distinct relation symbols, where each $R_i$ has an arity $a_i > 0$. In this work, the data is a document $\mathbf{d}$, and entries in the instances of a signature are spans of $\mathbf{d}$. Formally, for a signature $\mathbf{S} = \langle R_1, \ldots, R_m \rangle$ 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 $\text{Spans}(\mathbf{d})$; that is, $r_i$ is a subset of $\text{Spans}(\mathbf{d})^{a_i}$. 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 $\text{Loc}$ that stands for location;
- The unary relation symbol $\text{Per}$ that stands for person;
- The binary relation symbol $\text{PerLoc}$ that associates persons with locations.

We continue with our running example. Figure 3 shows a $\mathbf{d}$-instance over $\mathbf{S}$, where $\mathbf{d}$ is the document of Figure 1. This instance has 12 facts, and for later reference we denote them by $f_1, \ldots, f_{12}$. Note that there are quite a few mistakes in the table (e.g., the annotation of Virginia as a person by fact $f_6$); in the next section we will show how these are dealt with in the framework of this paper.

3.2 Conflicts and Priorities

The database research community has established the concept of repairs as a mechanism for handling inconsistencies in a declarative fashion [3]. Conventionally, denial constraints are specified to declare sets of facts that cannot co-exist in a consistent instance. A minimal repair of an inconsistent instance is a consistent sub-instance that is not properly contained in any other consistent sub-instance. Each minimal repair is then viewed as a possible world (and the notion of consistent query answers can then be applied).

Here, we will adapt the concept of denial constraints to our setting. In Section 5 we will illustrate the generality of these constraints w.r.t. conflict resolutions that take place in real life. However, in the world of IE the minimal repairs are not necessarily all equal. In fact, in every example we are aware of, the developer has a clear preference as to which facts to exclude when a denial constraint fires. Therefore, instead of the traditional repairs, we will use the notion of prioritized repairs of Staworko et al. [38] that extends repairing by incorporating priorities.

Let $\mathbf{S}$ be a signature, let $\mathbf{d}$ be a document, and let $I$ be a $\mathbf{d}$-instance over $\mathbf{S}$. A conflict hypergraph for $I$ is a hypergraph $H$ over the facts of $I$; that is, $H = (V, E)$ where $V$ is the set of $I$'s facts and $E$ is a collection of hyperedges (subsets of $V$). Intuitively, the hyperedges represent sets of facts that together are in conflict. A priority relation for $I$ is a binary relation $\succ$ over the facts of $I$.

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

If $f$ and $f'$ are facts of $I$, then $f \succ f'$ means intuitively that $f$ is preferred to $f'$. A repair of $I$ is a sub-instance of $I$ that does not contain any hyperedge of $H$. To accommodate priorities in cleaning, we use the notion of Pareto optimality [38]: a repair $J$ is an improvement of a repair $J'$ if there is a fact $f \in J \setminus J'$ such that $f \succ f'$ for all $f' \in J \setminus J'$; an optimal repair is a repair that has no improvement.

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

Example 3.3. Recall the instance $I$ of our running example (Figure 3). Figure 4 shows both a conflict hypergraph (which is a graph in this case) and a priority relation over $I$. Specifically, the figure has two types of edges. Dotted edges (with small arrows) define priorities, where $f_i \rightarrow f_j$ denotes that $f_i \succ f_j$. Later, we shall explain the preferences (such as $f_1 \succ f_4$). Solid edges (with bigger arrows) define both conflicts and priorities: $f_i \rightarrow f_j$ denotes that $(f_i, f_j)$ is an edge of the conflict hypergraph, and that $f_i \succ f_j$. Consider the following sets of facts:

- $J_1 = \{f_2, f_3, f_4, f_5, f_{11}\}$;

Figure 4: A conflict graph with priorities in the running example
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 $\Sigma$ be a signature, and let $R$ be a spanner representation system. A denial constraint in $R$ (over $\Sigma$), or just $R$-dc (or simply dc for short), has the form $\forall x (P \rightarrow \neg \Psi(x))$, where $\Psi$ is a sequence of variables in $SVars$, $P$ is a spanner specified in $R$ with all variables in $x$, and $\Psi$ is a conjunct of atomic formulas $\varphi(x)$ over $\Sigma$. (Note that an atomic formula $\varphi$ is an expression of the form $R(x_1, \ldots, x_n)$, where $R$ is an $n$-ary relation symbol in $\Sigma$.) We usually omit the universal quantifier, and specify a dc simply by $P \rightarrow \neg \Psi(x)$.

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

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

$$d_{doc} := \exists \text{ overlap}[x, y] \rightarrow \neg (\text{Loc}(x) \land \text{Loc}(y))$$

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

$$d_{pgd} := \exists \text{ overlap}[x, y] \rightarrow \neg (\text{Loc}(x) \land \text{Per}(y))$$

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

Example 3.5. The following pdg, denoted $p_{doc}$, 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.

$$p_{doc} := \rho[x, y] \rightarrow (\text{Loc}(x) \succ \text{Loc}(y))$$

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

$$\pi_{x, y}(\left(\Sigma^{*} \cdot x \Sigma^{*} \cdot y \Sigma^{*}\right) \sigma \left(\Sigma^{*} \cdot x \Sigma^{*} \cdot y \Sigma^{*}\right)) \lor \left(\Sigma^{*} \cdot x \Sigma^{*} \cdot y \Sigma^{*}\right)$$

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

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

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

Example 3.6. We denote by $\exists$ contains $[x, y]$ a regex formula that produces all pairs $(x, y)$ of spans where $x$ strictly contains $y$. Let $\exists$ contains $[x, y]$ 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 pdg, denoted $d_{doc}$, 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).

$$\exists \text{ contains}[z, x, y] \rightarrow \exists \text{ contains}[z', x, y']$$

Example 3.7. Consider again the $d$-instance $I$ of Figure 3. The reader can verify that dc's $d_{doc}$ and $d_{pgd}$ from Example 3.4, the pdgs $p_{doc}$ and $p_{pgd}$ in Example 3.5 and the denial pdg $d_{doc}$ of Example 3.6, together define the conflicts and priorities discussed in Example 3.3 (Figure 4).
S is a signature, U is a finite sequence u₁, ..., uₘ of updates, and ϕ is an atomic formula over S (representing the result of the program). There are two types of updates uᵢ:

1. **CQ updates.** These updates are conjunctive queries of the form \( R(y₁, ..., yₙ) := \alpha₁ \land \cdots \land \alphaₖ \), where \( R \) is a relation symbol of \( S \) of arity \( a \), and each \( \alphaᵢ \) is either an atomic formula over \( S \) or a spanner in \( \mathcal{R} \). The \( \alphaᵢ \) are called atoms. We make the requirement that each \( yᵢ \) occurs in at least one atom.

2. **Cleaning updates.** A cleaning update is an update of the form \( \text{CLEAN}(δ₁, ..., δₖ) \), where each \( δᵢ \) 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 \((S, U, ϕ)\) using only \( U \) along with a special return statement that specifies \( ϕ \). We then assume that \( S \) consists of precisely the relation symbols that occur in the program.

**Example 3.8.** We now define the REG-program \( E \) 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 \( x \). The signature is, as usual, that of Example 3.1. The sequence \( U \) of updates is the following. Note that we are using the notation we established in the previous examples.

1. \( \text{Loc}(x) := ρ_{loc}[x] \) (see example 2.4)
2. \( \text{Per}(y) := ρ_{cap}[y] \) (see example 2.4)
3. \( \text{CLEAN}(d_{loc}, d_{po}, p_{loc}, p_{po}) \) (see Examples 3.4 and 3.5)
4. \( \text{PerLoc}(x, y) := \text{Per}(x) \land \text{Loc}(y) \land \text{precede}[x, y] \) (see Example 3.4)
5. \( \text{CLEAN}(d_{po}) \) (see Example 3.6)
6. \( \text{RETURN PerLoc}(x, y) \)

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 \( E = (S, U, ϕ) \) be an \( \mathcal{R} \)-program with \( U = \langle u₁, ..., uₘ \rangle \), and let \( d \in \Sigma^* \) be a document. Let \( I₀ \) be the singleton \( \{I₀\} \), where \( I₀ \) is the empty instance over \( S \). For \( i = 1, ..., m \), we denote by \( Iᵢ \) the result of executing the updates \( uᵢ, ..., uₘ \) as we describe below. Since the cleaning operation can result in multiple instances (optimal repairs), each \( Iᵢ \) is a set of \( d \)-instances, rather than a single one. For \( i > 0 \) we define the following.

1. If \( uᵢ \) is the CQ update \( R(x₁, ..., xₙ) := \alpha₁ \land \cdots \land \alphaₖ \), then \( Iᵢ \) is obtained from \( Iᵢ₋₁ \) by adding to each \( I ∈ Iᵢ₋₁ \) all the facts (over \( R \)) that are obtained by evaluating the CQ over \( I \).
2. If \( uᵢ \) is the cleaning \( \text{CLEAN}(δ₁, ..., δₖ) \), then \( Iᵢ \) is obtained from \( Iᵢ₋₁ \) by replacing each \( I ∈ Iᵢ₋₁ \) with all the optimal repairs of \( I \), defined by the conflict hypergraph and priorities implied by all the \( δᵢ \).

Recall that a spanner is a function that maps a document into a \((V, d)\)-relation (see Section 2). An extraction program acts similarly, except that a document is mapped into a set of \((V, d)\)-relations (since it branches into multiple repairs). Later, we are going to investigate cases where the extraction program produces precisely one \((V, d)\)-relation, and then we will view the extraction program simply as a spanner. Next, we define the output of an extraction program \( E = (S, U, ϕ) \), where \( U = \langle u₁, ..., uₘ \rangle \) and \( ϕ = R(x₁, ..., xₙ) \). Let \( V = \{x₁, ..., xₙ\} \) be the set of variables in \( ϕ \). From a \( d \)-instance \( I \) over \( S \) (that does not involve any variables) we naturally obtain a \((V, d)\)-relation (that involves the variables in \( V \)):

The \((V, d)\)-relation consisting of all the assignments \( μ : V → \text{Spans}(d) \) such that \( R(μ(x₁), ..., μ(xₙ)) \) is a fact in \( I \). We denote this relation by \( I[ϕ] \). The result \( E(d) \) of evaluating the program \( E \) over the document \( d \) is the set of all the \((V, d)\)-relations \( I[ϕ] \) with \( I ∈ Iₘ \), where \( Iₘ \) is the result (as defined earlier) of the last update \( uₘ \).

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

1. In \( I₁ \), the relation \( \text{Loc} \) is as shown in Figure 3, and the other two relations are empty.
2. In \( I₂ \), the relations \( \text{Loc} \) and \( \text{Per} \) are as shown in Figure 3, and \( \text{PerLoc} \) is empty.
3. In \( I₃ \), the relations \( \text{Loc} \) and \( \text{Per} \) are as shown in Figure 5, and \( \text{PerLoc} \) is empty. The cleaning operation is described throughout Sections 3.2 and 3.3.
4. In \( I₄ \), the relations \( \text{Loc} \) and \( \text{Per} \) are as in \( I₃ \), and \( \text{PerLoc} \) is as shown in Figure 3.
5. \( I₅ \) is the instance shown in Figure 5.

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

**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 \( d \) into a \((V, d)\)-relation, for a set \( V \) of variables, while an extraction program maps \( d \) into a \((V, d)\)-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 \( E \) is unambiguous if \( E(d) \) is a singleton \((V, d)\)-relation for every document \( d \). We may view an unambiguous extraction program \( E \) simply as a specification of spanner.

Let \( \mathcal{R} \) be a spanner representation system. An \( \mathcal{R} \)-program is said to be non-cleaning if it does not contain cleaning updates (hence, it consists of only CQ updates). Clearly, if an extraction program is non-cleaning, then it is unambiguous. The following proposition states that in the case where \( \mathcal{R} \) is REG or RGX, the non-cleaning \( \mathcal{R} \)-programs do not have any expressive power beyond the regular spanners. The proof is straightforward from the definitions.
Proposition 4.1. Let $P$ be a spanner. The following are equivalent: (1) $P$ is representable by a non-cleaning REG-program, (2) $P$ is representable by a non-cleaning RGX-program, and (3) $P$ is regular.

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

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

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

Let $I$ be a $d$-instance over a signature $S$. Let $H$ and $>$ be a conflict hypergraph and a priority relation over $I$, respectively. Staworko et al. [38] give the following sufficient condition for the extraction programs.

If every cleaning update of $P$ is minimum generating and is decidable, then only the priority relation implied by the pgds of $u$ is acyclic. We say that $u$ is minimum generating if, for every document $d$ and $d$-instance $I$ over $S$, the priority relation implied by the pgds of $u$ is acyclic. The semantics of a 2NFA($k$) is as follows. Let $s \in \Theta^*$ be a string, and denote it as $s_1, \ldots, s_n$. The machine $A$ does not read $s$ directly, but rather the augmentation $\vdash s \dashv$ (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 $\vdash s \dashv$. In the initial configuration, the state is $s_0$ and all the heads are on $s_0$. The transition $(q, t_1, \ldots, t_k) \delta(q, a_1, \ldots, a_k)$, where $t_i \in \{l, r, h\}$ and $a_i \in \Theta \cup \{\vdash, \dashv\}$, means that if the current state is $q$ and the $i$th 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 $C$ is one without possible next configurations (because $\delta(q, a_1, \ldots, a_k) = \emptyset$). A halting configuration is accepting if its state $q$ is in $F$; otherwise, the halting configuration is rejecting. We say that $A$ accepts a string $s$ if there is a run (legal sequence of configurations) on $\vdash s \dashv$ that starts with the initial configuration and ends with an accepting configuration.

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

In private communication with the authors, Martin Kutrib independently proved this result.
LEMMA 4.7. For 2DFA(2), the finite mortality problem is co-recursively enumerable but not recursively enumerable. In particular, it is undecidable.

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

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

4.2 Disposability

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

Proposition 4.8. Let R be a spanner representation system and let E be an R-program. If every cleaning update of E is R-disposable, then E is equivalent to a non-cleaning R-program.

We say that a denial pgd p is R-disposable if the cleaning update that consists of only p is 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 4.5).

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

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

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

5. SPECIAL CLEANING STRATEGIES

In this section, we discuss several classes of cleaning strategies that are used in practice. We will show that the strategies in each class are expressible through the use of a regular spanner in the premise. Moreover, we will prove that all of these cleaning updates are REG-disposable.

5.1 Transitive Denial Pgd

A denial pgd is transitive if the relationship “fact f is in conflict with and has priority over fact f’” is transitive. More formally, let p be a denial pgd P → (φ(x) ⊳ ϕ(x)) over a schema S. Let I be a d-instance over S, and let f and f’ be two facts in I. By p |= f ⊳ f’ we denote the fact that there is a span assignment for x that is true on P, and that maps ϕ(x) and ϕ’(x) to f and f’, respectively. We say that p is transitive if for every d-instance I over S and for every three facts f1, f2 and f3 in I, if p |= f1 ⊳ f2 and p |= f2 ⊳ f3 then p |= f1 ⊳ f3.

An example of a transitive denial pgd is the maximal container denial pgd, which has the form contains(x, y) → (R(x) ⊳ R(y)), where R is a relation symbol, and x and y are disjoint sequences of variables that contain x and y, respectively. This denial pgd is among the standard collection of “consolidation” strategies provided by SystemT [7], along with the analogous minimal contained (that favors shorter strings, and is expressed by the denial pgd dpconc in Example 3.6), which is also transitive. Another example of a transitive denial pgd that captures a popular strategy is that of rule priority, and it has the form P → (R(x) ⊳ S(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), 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; (2) Determine whether p is both transitive and acyclic.

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

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

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

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

5.2 JAPE Controls

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

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

- Under the Appelt control, R(x) ⊳ 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 and is also provided by SystemT [7] (as a “consolidator”). This control also involves rule priority, which we ignore for now and discuss later.
• The Brill control is similar to Appelt, with the exclusion of option (2); that is, \( R(x) \triangleright R(y) \) 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, \( R(x) \triangleright R(y) \) if and only if \( x \) is that remaining fact and \( x \neq y \).

**Example 5.3.** Suppose that \( \Sigma = \{0, 1\} \), and that \( R \) is defined by the following regex formula:

\[
\Sigma^* \cdot x \{1^+ 0^+ 1^+\} \cdot \Sigma^*
\]

Now, consider the following two documents:

\[
d^1 = 100110100101 \quad \text{and} \quad d^2 = 000110100101
\]

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

• By applying the Appelt control, the spans that remain in \( R \) for \( d^1 \) are \{1, 6\} and \{7, 11\}, and the spans that remain in \( R \) for \( d^2 \) are \{4, 8\} and \{10, 13\}.

• By applying the Brill control, the spans that remain in \( R \) for \( d^1 \) are \{1, 5\}, \{1, 6\} and \{7, 11\}, and the spans that remain in \( R \) for \( d^2 \) are \{4, 8\} and \{10, 13\}.

• By applying the First control, the spans that remain in \( R \) for \( d^1 \) are \{1, 5\}, \{5, 8\} and \{10, 13\}, and the spans that remain in \( R \) for \( d^2 \) are \{4, 8\} and \{10, 13\}.

• By applying the Once control, the span that remains in \( R \) for \( d^1 \) is \{1, 5\}, and the span that remains in \( R \) for \( d^2 \) is \{4, 8\}.

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 \( \Pi ^5.2 \) (since the corresponding denial pgd is transitive in a vacuous sense).

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

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

Let \( P \) be a unary spanner. Define the Kleene star of \( P \), denoted \( P^* \), to be the spanner \( Q \) with \( V S a r s ( P ) = V S a r s ( Q ) \), where for each document \( d \), we have that \( Q(d) \) consists of those spans \([a, a']\) such that the following holds: there are indices \( a_1 \leq \ldots \leq a_n \) where \( n \geq 1 \), \( a_1 = a \), \( a_{n-1} = a' \), and \( [a_i, a_{i+1}] \) is in \( P(d) \) for all \( i \). Observe that for all documents \( d \), every empty span \([a, a']\) is in \( P^* \); this is obtained by letting \( a' = a \) and \( n = 1 \).

**Lemma 5.5.** Let \( P \) be a unary spanner. If \( P \) is regular, then \( P^* \) is regular.

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

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

### 5.2.1 Rule Priority

In its general form, Appelt involves rule priority. In JAPE, the priority of a rule is determined by an explicit numerical priority (e.g., Priority:20) and, in the case of ties, the position of the rule in the program definition. In our terminology, we have \( n \) unary 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 \( R_1 \times \cdots \times R_{i-1} \times R_{i+1} \times \cdots \times R_n \).

We can extend Theorem 5.4 to include priorities, as follows. We first define \( R \) as the union of the \( R_i \) (which we can do in an extraction program). Next, we apply Appelt, as previously defined, to \( R \), and then join each \( R_i \) with \( R \) using \( R_i(x) \leftarrow R_i(x) \land R(x) \). Finally, we apply the (transitive) cleaners \( \text{CLEAN}(\text{true} \rightarrow R_i(x) \triangleright R_j(x)) \) for \( 1 \leq i < j \leq n \), one by one, in order of increasing \( i \).

### 5.3 POSIX Disambiguation

Recall from Section 2.3 that a regex formula \( \gamma \) defines a spanner by considering all possible ways that the input document \( d \) can be matched by \( \gamma \); that is, it considers all possible parse trees of \( \gamma \) on \( d \). Each such parse tree generates a new \((V, d)\)-tuple, where \( V = \text{SVars}(\gamma) \), 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 \((V, d)\)-tuple per matched input document \( d \) instead of multiple such tuples.

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

\footnote{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.}
be expressed in our framework as a \( \text{REG} \)-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. [39] for a description of this policy) are left for future work.

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

**Example 5.6.** Consider \( \gamma = x(001) \cdot y(1\epsilon) \) and \( d = 01 \). Under the POSIX disambiguation policy, subexpression \( x(001) \) will match as much of \( d \) as possible while still allowing the rest of the expression, namely \( y(1\epsilon) \), to match the remainder of \( d \). As such, \( x(001) \) will match \( d \) entirely, and \( y(1\epsilon) \) will match the empty string. We hence bind \( x \) to the span \([1, 3]\) and \( y \) to \([3, 3]\).

In contrast, when \( \gamma = x(01) \cdot y(1\epsilon) \) or \( x(01) \cdot y(1\epsilon) \) and \( d = 01 \), under the POSIX disambiguation policy we bind \( x \) to the span \([1, 2]\) and \( y \) to the span \([2, 3]\).

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

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

The proof proceeds by induction on \( \gamma \), showing how to encode the POSIX policy by means of transitive denial pgds. Since every transitive denial pgd is REG-disposable by Theorem 5.2, we immediately obtain from Lemma 5.7 that the spanner \( \text{posix}[\gamma] \) is regular, for every regex formula \( \gamma \). Moreover, since it is easily verified that \( \text{posix}[\gamma] \) is hierarchical, it follows by Theorem 2.5 that \( \text{posix}[\gamma] \) is itself definable in \( \text{RGX} \) by a regex formula \( \delta \). 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 \( \gamma \)-parse constructed by the POSIX disambiguation policy.

\[ \text{posix}[\gamma] \text{ is definable in } \text{RGX}. \]

### 5.4 Core Spanners

Recall that REG is the closure of \( \text{RGX} \) under union, projection, and natural join. The class \( \text{Core} \) of core spanners [14] is obtained by adding to this list of operators the string-equality selection, denoted \( \gamma^".\) Formally, given an expression \( \rho \) in REG and two variables \( x, y \in \Sigma \), the spanner defined by \( \gamma^" \) selects all those tuples from \( \rho \) 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 5.8.** For every regex formula \( \gamma \), the spanner \( \text{posix}[\gamma] \) is definable in \( \text{RGX} \).

To prove Theorem 5.9, we use the following definition. Let \( p \) be a denial pgd of the form \( \rho[x, y] \rightarrow R(x) \triangleright S(y) \), 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 nor a suffix of \( a \), then \( R(a) \triangleright S(b) \) is implied by \( p \). Note that all the denial pgds in Theorem 5.9 favor strict containers. It also applies to POSIX, which we did not define formally as a denial pgd. Our proof is by showing that if \( p \) favors strict containers, then \( p \) is not Core-disposable. In particular, if a denial pgd that favors strict containers is Core-disposable, then there is a Boolean core spanner that recognizes the language of all the strings \( s \# t \) where \( s, t \in \{0, 1\}^* \) and \( s \) is not a substring of \( t \). Then, we use the result of Fagin et al. [14] stating that no such core spanner exists. Note that Theorem 5.9 does not mention any rule priority denial pgd (Section 5.1), since this rule is underspecified. Nevertheless, we can show that the cleaning update consisting of the denial pgd true \( \rightarrow (R(x) \triangleright S(x)) \), where \( R \) and \( S \) are distinct unary relation symbols, is not Core-disposable.

### 6. CONCLUSIONS

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

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

7. REFERENCES


