# Rewrite Rules for Search Database Systems

Ronald Fagin* Benny Kimelfeld* Yunyao Li* Sriram Raghavan† Shivakumar Vaithyanathan*
fagin@almaden.ibm.com {kimelfeld, yunyaoli}@us.ibm.com sriramraghavan@in.ibm.com shiv@us.ibm.com

## ABSTRACT

The results of a search engine can be improved by consulting auxiliary data. In a search database system, the association between the user query and the auxiliary data is driven by rewrite rules that augment the user query with a set of alternative queries. This paper develops a framework that formalizes the notion of a rewrite program, which is essentially a collection of hedge-rewriting rules. When applied to a search query, the rewrite program produces a set of alternative queries that constitutes a least fixpoint (lfp). The main focus of the paper is on the lfp-convergence of a rewrite program, where a rewrite program is lfp-convergent if the least fixpoint of every search query is finite. Determining whether a given rewrite program is lfp-convergent is undecidable; to accommodate that, the paper proposes a safety condition, and shows that safety guarantees lfp-convergence, and that safety can be decided in polynomial time. The effectiveness of the safety condition in capturing lfp-convergence is illustrated by an application to a rewrite program in an implemented system that is intended for widespread use.

## Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems

## General Terms

Algorithms, Theory

## Keywords

Search database system, rewriting

## 1. INTRODUCTION

It is well known that an auxiliary database of concepts associated with the contents of a document collection can significantly improve the quality (precision and recall) of a keyword search engine [9, 10, 19, 22, 23]. The intuitive idea is that the auxiliary data enables the engine to better interpret the search query terms and retrieve documents matching the "intent" behind the query, as opposed to documents that merely contain physical matches for the query terms. For example, modern Web search engines are very good at interpreting and generating specially crafted results for queries referring to popular concepts such as movies, locations, weather, places, etc. Thus, a query such as "nyc map" brings back the map of New York City, a query such as "san jose weather" brings back the weather forecast for the city of San Jose, and so on.

At IBM Research–Almaden, we have developed a generic search engine with such "interpretation" capabilities that can be customized to different document collections by providing an appropriate auxiliary database. A central tool used for obtaining such an auxiliary database (to aid a search engine) is *information extraction* [2, 9, 12, 23, 26]. In earlier work, we developed the framework of a *search database system* [6] to formalize the approach taken by our engine. As we deployed this engine in large-scale production scenarios, we encountered the requirement to provide search administrators and domain experts with the ability to customize the process of generating interpretations.

For example, consider the scenario of intranet search. Administrators wished to express a rule that whenever a query involves the name of a person and a word such as "contact", "profile", "phone" or "email", the search engine should generate an interpretation that looks for the person's name within the company's internal employee directory. Similarly, any query of the form "download" or "install" followed by the name of some software should generate an interpretation that looks for the name of this software within the company's internal software download Web site. These examples illustrate how rules are used to direct the search engine towards specific subcollections or Web sites based on the concepts occurring in the search query. In our deployments, we encountered several other use cases for rules, such as dropping noise words (e.g., inside IBM, the query "ibm health insurance" should really be simply "health insurance") or handling under-specified queries (e.g., a vague query such as "travel"). In the latter case, the query "travel" might be augmented by a set of more specific queries. In all of these cases, the original query is augmented with a set of additional queries that should all be executed, with the results concatenated or interleaved in some manner, perhaps based on some quality measure of the queries (the details of how the result sets of the various queries are combined is beyond the scope of this paper).

To support these requirements, we enhanced our engine with support for *rewrite rules*. Rewrite rules govern how the input keyword query should be augmented by additional queries that should all be executed against the underlying search index. For exam-

*IBM Research–Almaden, San Jose, CA 95120, USA.
†IBM India Research Lab, Embassy Manyatha Business Park Bldg D4, Bangalore 560045, India


*PODS'11,* June 13–15, 2011, Athens, Greece.

ple, assume that the user gives the query "laura haas number"; let us denote this query by $\mathbf{q}$. Assume that the system has the rules $\rho_1, \rho_2, \rho_3$ that are depicted in Figure 1(b). (We shall explain the syntax and semantics of these rules later; for now, we shall simply discuss their effect when the user query is $\mathbf{q}$.) Given the query $\mathbf{q}$, the rewrite rule $\rho_1$ would fire to associate "laura haas" with the concept of a "person". Thus, in Figure 2, the rewrite rule $\rho_1$ is applied to the search query $\mathbf{q}$ in the upper-left box to obtain the query $h_1$ in the upper-right box. Here $\overline{\mathsf{person}}$ denotes the concept of a "person". This new query $h_1$ is a *hedge*, as we shall discuss later, which is why we use the letter $h$ (we do not use bold letters as we did for $\mathbf{q}$, since we reserve bold letters for sequences of keywords). The rule $\rho_2$ would associate the $\overline{\mathsf{person}}$ concept and the word "number" with a $\overline{\mathsf{person}}$ concept and a $\overline{\mathsf{phone}}$ concept that are tied together with a person-phone compound concept $\overline{\mathsf{prph}}$. This gives the query $h_2$ in the bottom-right box in Figure 2. Finally, the rule $\rho_3$ implies that a query of the form of $h_2$ should generate another query that looks for the home page of the person, as in the bottom-left box in Figure 2. We are then left with the set $\{\mathbf{q}, h_1, h_2, h_3\}$ of queries, which should all be run with their results interleaved in some manner. Note that these queries might give quite different answers. For example the original query $\mathbf{q}$ might give many pages that each contain all of the words "laura" "haas", and "number" (and might not include the home page of Laura Haas, if her home page does not contain the word "number"), whereas the query $h_3$ might give back just the home page of Laura Haas.

Rewrite rules proved to be a powerful and effective mechanism in the hands of search administrators and quickly resulted in deployments where hundreds of such rules were registered with the search engine. However, this quickly pointed us to a major problem: the application of the rules to a search query may result in a non-terminating process that produces an infinite number of alternatives. As a simple example, an administrator may add a pair of rules: a rule $\rho_1$ to transform "almaden" to its common short form "arc", and a rule $\rho_2$ to transform "ibm arc" to "ibm almaden research". Given the query "ibm almaden", the rule $\rho_1$ creates the query "ibm arc"; the rule $\rho_2$ then creates the query "ibm almaden research"; the rule $\rho_1$ then creates the query "ibm arc research"; the rule $\rho_2$ then creates the query "ibm almaden research research", and so on. This process never ends, and we get infinitely many queries of the form "ibm almaden research $\cdots$ research", where "research $\cdots$ research" represents the word "research" repeated arbitrarily many times. Lacking effective tools that would detect and help avoid such combinations, we were forced (before the research in this paper) to make the ad hoc choice of limiting rule composition by applying each rule at most once, in some prescribed order.

The goal of this paper is to lay foundations for rewrite rules in a search engine, where the focus is on understanding the above problem of non-terminating rewriting, and guaranteeing that it is avoided in the set of rewrite rules at hand. Our formalism is within the framework of the *search database system* that we developed in previous work [6]. We show that the notion of rule introduced in that work essentially corresponds to a special type of rewrite rule that is more limited in expressivity but guaranteed to terminate. A more formal discussion of this phenomenon appears in Section 5.4.

A formal definition of a search database system is given in Section 2. The essence is as follows. The *schema* of such a system is a partially ordered set of *concepts* (e.g., $\overline{\mathsf{person}}$ and $\overline{\mathsf{phone}}$, which are subconcepts of $\overline{\mathsf{prph}}$). In our examples, each concept is written with a bar over it. A *database* over the schema specifies instantiations of the concepts (e.g., "laura haas" instantiates $\overline{\mathsf{person}}$) in the documents of the corpus, as well as relationships between these instantiations according to the schema (e.g., a $\overline{\mathsf{prph}}$ association between a person and her phone number). A query over the database is a hedge (ordered forest) where nodes are (labeled by) *terms* (which are essentially the words used as keywords) and concepts from the schema, such that the parent-child relationships between concepts are consistent with the schema $\mathbf{S}$; such a query is called an $\mathbf{S}$-*hedge*. Examples of $\mathbf{S}$-hedges, where $\mathbf{S}$ is depicted in Figure 1(a), appear in the rectangular boxes of Figure 2. The *search query* phrased by the user is an $\mathbf{S}$-hedge that contains only terms, as in the upper-left box in Figure 2.

A *rewrite rule* is, conceptually, a (possibly infinite) binary relation over the $\mathbf{S}$-hedges, where inclusion of the pair $(h, h')$ in this relation means that the $\mathbf{S}$-hedge $h$ is rewritten to the $\mathbf{S}$-hedge $h'$. The exact syntax that we use for specifying a rewrite rule is described in Section 3. Essentially, a rule has the form $E \Rightarrow F$, where $E$ and $F$ are $\mathbf{S}$-hedges that contain variables (in addition to terms and concepts). In terms of standard *rewriting systems* [25], our rules can be classified into the category of *hedge rewriting* [11], *context sensitive* [20] (actually, the full context is captured in each of the two sides of a rule), *linear* [25] (i.e., each variable has at most one occurrence in each of the two sides of a rule), and *conditional* [13] (i.e., the involved hedges need to conform to the schema). Examples are the rules $E_i \Rightarrow F_i$ (for $i = 1, 2, 3$) in Figure 1(b). Under this syntax, a rewrite rule should be *consistent* in the sense that every $\mathbf{S}$-hedge is indeed rewritten into an $\mathbf{S}$-hedge. We show that consistency of a rule can be decided in polynomial time, which we realize by a nontrivial algorithm. A *rewrite program* consists of a schema $\mathbf{S}$ and a finite set $R$ of rewrite rules. When a rewrite program is applied to a search query $\mathbf{q}$, it maintains a set $H$ of $\mathbf{S}$-hedges that is initially the singleton $\{\mathbf{q}\}$. This set grows by adding to it every $\mathbf{S}$-hedge $h'$ such that an $\mathbf{S}$-hedge $h$ in $H$ can be rewritten using the rewrite rules into $h'$. We terminate when no such $h'$ can be found, which means that the set $H$ we reach is the *least fixpoint* (which we sometimes abbreviate as *lfp*) for $\mathbf{q}$. The *lfp-convergence* of a rewrite program means that this process always terminates, or in other words, the least fixpoint of every search query is finite. The notion of lfp-convergence is very similar to *weak termination* (or *quasi-termination*) of a rewriting system [8], except that we are restricted to rewritings that start with a search query (and not an arbitrary hedge).

We would like to be able to decide whether a given rewrite program is lfp-convergent. Unfortunately, it follows from known results [8] that this task is undecidable, even under strong restrictions on the rewrite program (e.g., no concepts are involved in the rewrite rules), and even if we fix *any* arbitrary schema. Due to this undecidability, the best one can hope for is a *safety condition* that is useful in practice. In Section 5, we present such a safety condition after making the case the vast literature on *proof techniques for termination* (e.g., [1, 3–5, 7, 14]) does not capture the special needs and features of lfp-convergence for our rewrite programs. We show that our safety condition indeed guarantees lfp-convergence. We show that safety can be decided in polynomial time, and we realize it by a nontrivial recursive algorithm that involves dynamic programming, linear programming, and graph theory.

We found our safety approach to be extremely useful for the management of the rewrite program used in an actual large-scale deployment of our search engine. Specifically, we were able to detect a small set of rewrite rules that cause non-termination (non-lfp-convergence), while the remaining part is lfp-convergent. Almost all of that remaining part of our program (more precisely, around 95% of the remaining part, which is around 90% of the original program) was found to be safe. (In Section 5.5 we provide more details, including some comments about how we dealt with the non-safe portion of the rules.) Hence, our experience shows that the

safety condition is highly successful in capturing lfp-convergence in a practical rewrite program.

## 2. SEARCH DATABASE SYSTEMS

In this section, we give the formal definitions for the framework of a *search database system*. We begin with some preliminaries.

### 2.1 Preliminaries: Strings, Trees and Hedges

Let $\Sigma$ be a set, which we refer to as an *alphabet*. We denote by $\Sigma^*$ the set of all strings over $\Sigma$, that is, all the finite sequences $\sigma_1 \cdots \sigma_n$, where $\sigma_i \in \Sigma$ for $1 \leq i \leq n$. The length $n$ of a string $\mathbf{s} = \sigma_1 \cdots \sigma_n$ is denoted by $|\mathbf{s}|$.

The trees we use in this paper are directed, ordered (i.e., the children of each node form a sequence), and node labeled. Formally, for a set $\Sigma$ of labels, a *tree* (*over* $\Sigma$) is inductively defined as follows. If $\sigma \in \Sigma$ and $t_1, \ldots, t_n$ are trees over $\Sigma$ (where $n$ is a nonnegative integer), then $\sigma(t_1 \cdots t_n)$ is a tree over $\Sigma$. Intuitively, $\sigma(t_1 \cdots t_n)$ is a tree with a root labeled with $\sigma$, such that the subtrees under the root, from left to right, are $t_1, \ldots, t_n$. For convenience, we identify the symbol $\sigma$ with the tree $\sigma()$ (hence, a symbol of $\Sigma$ is a special case of a tree). A *hedge* (*over* $\Sigma$) is a sequence $h = t_1 \cdots t_n$ of trees over $\Sigma$. Special cases of a hedge include a tree, a symbol of $\Sigma$, and a string of $\Sigma^*$ (including the empty string).

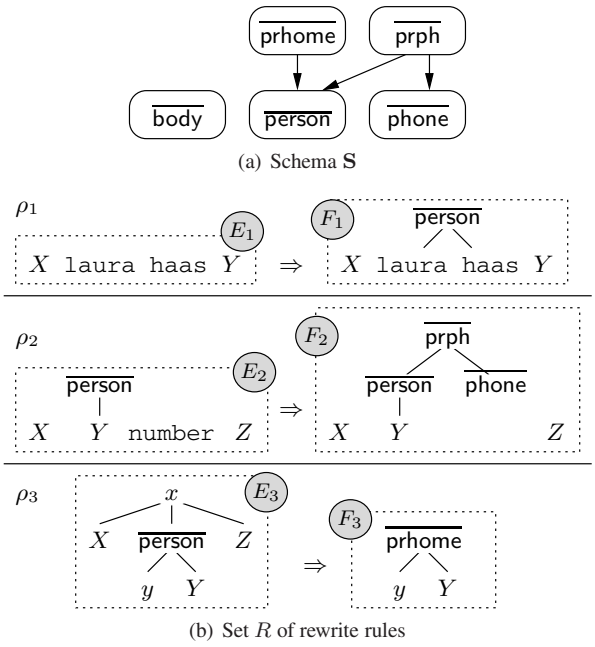
The choice of the above formalism for trees (i.e., as strings) is convenient in the sense that isomorphism is the same as equality (i.e., two trees are isomorphic if and only if they are equal). There is a straightforward translation of a tree in this formalism to the more standard formalism by means of *nodes* and *edges*,[1] and it is convenient for us to (implicitly) make this translation. In particular, we denote by $\mathrm{nodes}(h)$ the set of nodes of the hedge $h$, and we denote by $\mathrm{edges}(h)$ the set of edges of $h$. If $(v, w)$ is an edge of $h$, then $w$ is a *child* of $v$, which in turn is the *parent* of $w$. The label of the node $v$ is denoted by $\mathrm{label}(v)$. A *leaf* of a hedge is a node without children, and a non-leaf node is an *internal node*. If a hedge $h$ contains a directed path from node $v$ to node $w$, then $w$ is a *descendant* of $v$, which in turn is an *ancestor* of $w$. Note that every node is both an ancestor and a descendant of itself. For a tree $t$, the *root* of $t$, denoted $\mathrm{root}(t)$, is the unique node without a parent.

### 2.2 Concepts, Schemas and Instances

We fix an infinite set $\mathsf{Cncpt}$ of *concepts*. A *schema* is a finite *poset* (i.e., a partially ordered set) of concepts. More specifically, a schema $\mathbf{S}$ is a pair $(C, \prec)$, where $C$ is a finite subset of $\mathsf{Cncpt}$, and $\prec$ is a strict partial order over $C$ (i.e., $\prec$ is an irreflexive, asymmetric and transitive binary relation over $C$). For a concept $\gamma \in C$, the set of *subconcepts* of $\gamma$, denoted by $sc(\gamma)$, comprises all the elements $\delta \in C$, such that $\delta \prec \gamma$ and there is no $\gamma' \in C$, such that $\delta \prec \gamma' \prec \gamma$. A minimal concept $\gamma \in C$ (i.e., $\gamma$ such that $sc(\gamma) = \emptyset$) is called an *atomic concept*, abbreviated *a-concept*. If $\gamma \in C$ is not atomic, then it is a *compound concept*, abbreviated *c-concept*. We usually abuse the notation and identify $\mathbf{S}$ with $C$, avoiding any mentioning of $\prec$; for example, we may write $\gamma \in \mathbf{S}$ when we actually mean $\gamma \in C$.

EXAMPLE 2.1. Figure 1(a) shows the schema $\mathbf{S}$ that we use in our running example. The concepts of $\mathbf{S}$ are $\overline{\mathsf{person}}$, $\overline{\mathsf{phone}}$, $\overline{\mathsf{prph}}$ (which stands for "person-phone"), $\overline{\mathsf{body}}$, and $\overline{\mathsf{prhome}}$ (which stands for "personal home page"). As done here, throughout this paper we denote a specific concept by an over-lined word, in order to distinguish concepts from words of other roles (as we define below). Note that $\overline{\mathsf{prph}}$ and $\overline{\mathsf{prhome}}$ are c-concepts, with $sc(\overline{\mathsf{prph}}) = \{\overline{\mathsf{person}}, \overline{\mathsf{phone}}\}$ and $sc(\overline{\mathsf{prhome}}) = \{\overline{\mathsf{person}}\}$. The remaining three concepts of $\mathbf{S}$ are a-concepts. Intuitively, $\overline{\mathsf{prph}}$ denotes an association between a person and her phone number, and $\overline{\mathsf{prhome}}$ denotes the home page of a person. □

[1] For a specific translation see, for example, [21].




(a) Schema $\mathbf{S}$

(b) Set $R$ of rewrite rules

**Figure 1: Rewrite program $(\mathbf{S}, R)$ (running example)**

Next, we define the notion of a *database instance* over a schema. Throughout the paper, we fix two sets. First, we have a set $\mathsf{URIs}$ of *unique resource identifiers*, which are used for identifying documents. (We use the standard abbreviation *URI* for Unique Resource Identifier.) Second, we have an infinite set of *terms* that is denoted by $\mathsf{Terms}$; a term can appear in a document, and it can be used as a keyword in a search query.

Let $\mathbf{S}$ be a schema. A *database instance* (or just *instance*) $I$ (*over* $\mathbf{S}$) is a finite set of *records*, where each record $r$ is associated with a unique concept $\gamma \in \mathbf{S}$ (thus, $\gamma$ acts as a property of $r$). Every record $r$ has a URI from the set $\mathsf{URIs}$, and this URI is denoted by $uri(r)$. As for the rest of the content of a record, we distinguish between two types of records $r$, depending on whether the concept $\gamma$ associated with $r$ is atomic or compound.

- If $\gamma$ is an a-concept, then $r$ is called an *a-record*. Then, $r$ has textual data, which is a string of $\mathsf{Terms}^*$. The textual data of $r$ is denoted by $txt(r)$.

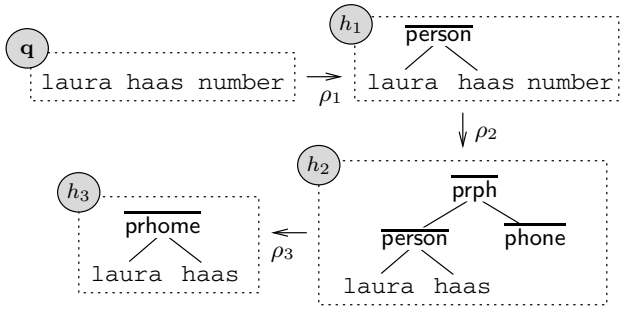



**Figure 2: S-hedges**

| rec. | concept | uri | text |
|---|---|---|---|
| $r_{\mathsf{lh}}$ | $\overline{\mathsf{person}}$ | doc1 | laura haas |
| $r_{\mathsf{wc}}$ | $\overline{\mathsf{person}}$ | doc1 | william cody |
| $r_{\mathsf{ph}}$ | $\overline{\mathsf{phone}}$ | doc1 | +1(234)567-8910 |
| $r_{\mathsf{bd}}$ | $\overline{\mathsf{body}}$ | doc1 | welcome to ibm research ... |
| $s_{\mathsf{pr}}$ | $\overline{\mathsf{person}}$ | doc2 | laura haas |
| $s_{\mathsf{ph}}$ | $\overline{\mathsf{phone}}$ | doc2 | +1(098)765-4321 |
| $s_{\mathsf{bd}}$ | $\overline{\mathsf{body}}$ | doc2 | laura's home page ... |

(a) The a-records of $I$

| rec. | concept | uri | references |
|---|---|---|---|
| $r_{\mathsf{pp}}$ | $\overline{\mathsf{prph}}$ | doc1 | $r_{\mathsf{pp}}[\overline{\mathsf{person}}] = r_{\mathsf{wc}}$, $r_{\mathsf{pp}}[\overline{\mathsf{phone}}] = r_{\mathsf{ph}}$ |
| $s_{\mathsf{pp}}$ | $\overline{\mathsf{prph}}$ | doc2 | $s_{\mathsf{pp}}[\overline{\mathsf{person}}] = s_{\mathsf{pr}}$, $s_{\mathsf{pp}}[\overline{\mathsf{phone}}] = s_{\mathsf{ph}}$ |
| $s_{\mathsf{hm}}$ | $\overline{\mathsf{prhome}}$ | doc2 | $s_{\mathsf{hm}}[\overline{\mathsf{person}}] = s_{\mathsf{pr}}$ |

(b) The c-records of $I$

**Figure 3: An instance $I$ of schema S of Figure 1(a)**

- If $\gamma$ is a c-concept, then $r$ is called a *c-record*. Then, for each subconcept $\delta \in sc(\gamma)$, the record $r$ references a record $s \in I$ with the concept $\delta$ and with the same URI as $r$ (that is, $uri(r) = uri(s)$); the record $s$ is denoted by $r[\delta]$.

EXAMPLE 2.2. We continue with our running example. The tables of Figures 3(a) and 3(b) show the a-records and c-records, respectively, of an instance $I$ over the schema $\mathbf{S}$ (Figure 1(a)). The top table (Figure 3(a)) contains the a-records and the bottom one (Figure 3(b)) contains the c-records. This instance represents two indexed pages with the URIs doc1 and doc2. Each row in the table describes one record. For example, the first row of the top table describes the a-record $r_{\mathsf{lh}}$, and the first row of the bottom table describes the c-record $r_{\mathsf{pp}}$. The column "concept" gives the concept of the record. For example, the concept of $r_{\mathsf{lh}}$ is $\overline{\mathsf{person}}$. The column "uri" gives the URI of the record; for example, $uri(r_{\mathsf{lh}}) =$ doc1. In the top table, the column "text" shows the textual data; thus, $txt(r_{\mathsf{lh}}) =$ laura haas (which consists of the two terms laura and haas). In the bottom table, the column "references" shows the reference of each subconcept.

The body records of $I$ can be produced by standard operations on Web pages (e.g., HTML-to-text conversion). Other records, however, are not explicitly specified in a Web page, and their production may require more complex (possibly heuristic) techniques. For example, the $\overline{\mathsf{person}}$ records are obtained by identifying a person name in the document. Similarly, producing a $\overline{\mathsf{phone}}$ record requires a mechanism for identifying that a substring of the document is a phone number, and producing a $\overline{\mathsf{prph}}$ record entails identifying that a recognized person and a recognized phone number are semantically associated. Producing the record $s_{\mathsf{hm}}$ with the reference to a person requires understanding that the document is the person's home page (in addition to recognition of that person). These are typical tasks in *information extraction* [17, 24], which form a primary tool for the construction of a database instance. □

Let $I$ be an instance, and consider the directed graph $G$ that has the records of $I$ as the set of nodes, and an edge $(r_1, r_2)$ from record $r_1$ to record $r_2$ whenever $r_2 = r_1[\gamma]$ for some concept $\gamma$. A record $r$ is a *descendant* of a record $r'$ if $G$ has a directed path from $r$ to $r'$. Note that every record is a descendant of itself.

## 2.3 Queries

We assume the existence of a boolean *matching operator* $\propto$ of a term $\tau$ in a string $\mathbf{t}$ of terms (that is, $\propto$ is a subset of $\mathsf{Terms} \times \mathsf{Terms}^*$). For simplicity, in our examples we will always interpret $\tau \propto \mathbf{t}$ as (case-insensitive) membership of $\tau$ in the string $\mathbf{t}$. For example, america $\propto$ United States of America, but usa $\not\propto$ United States of America. The practical instantiation of this operator is irrelevant to this paper, and is discussed more in [6] (for example, in the practical implementation we are likely to have bill $\propto$ william, when the two terms refer to person names).

A *search query* is a string of terms, that is, a member $\mathbf{q}$ of $\mathsf{Terms}^*$. Every term of a search query is called a *keyword*. Let $I$ be an instance, and let $\mathbf{q}$ be a search query. A URI $u$ *matches* $\mathbf{q}$ if for every keyword $\tau$ of $\mathbf{q}$ there is an a-record $r \in I$ such that $\tau \propto txt(r)$ and $uri(r) = u$. As an example, consider the instance $I$ of Figure 3, and let the search query $\mathbf{q}$ be laura haas number. We have that laura $\propto txt(r_{\mathsf{lh}})$ and haas $\propto txt(r_{\mathsf{lh}})$; hence, doc1 matches $\mathbf{q}$ if and only if $txt(r_{\mathsf{bd}})$ contains the term number (hidden in the three dots). The same holds for doc2.

We generalize a search query by enriching it with structural constraints, in the context of a schema $\mathbf{S}$, on the matched URIs and on the way keywords are matched. We call such a query an $\mathbf{S}$-*hedge*. The formal definition is as follows.

DEFINITION 2.3. Let $\mathbf{S}$ be a schema. An $\mathbf{S}$-*hedge* is a hedge $h$ over $\mathbf{S} \cup \mathsf{Terms}$ such that for all edges $(u, v)$ of $h$ it holds that $\mathrm{label}(u)$ is a concept, and $\mathrm{label}(v)$ is either a term or a subconcept of $\mathrm{label}(u)$. □

Note that the definition of an $\mathbf{S}$-hedge implies that a node that is labeled with a term must be a leaf. The converse, however, is not true; a leaf can have any label of $\mathbf{S} \cup \mathsf{Terms}$ (i.e., a term, an a-concept, or a c-concept).

EXAMPLE 2.4. Consider again the schema $\mathbf{S}$ of our running example (Figure 1(a)). Figure 2 shows a search query $\mathbf{q}$ and three other $\mathbf{S}$-hedges $h_1$, $h_2$ and $h_3$. For now, ignore the arrows between the hedges; these will be discussed later. Note that in each of these $\mathbf{S}$-hedges, every term is a leaf, as is required by the definition. Also note that in $h_2$, the $\overline{\mathsf{phone}}$ node is also a leaf, which is allowed. Finally, consider the edge from the $\overline{\mathsf{prph}}$ node to the $\overline{\mathsf{person}}$ node of $h_3$, and observe that, indeed, $\overline{\mathsf{person}} \in sc(\overline{\mathsf{prph}})$. □

Let $I$ be an instance over a schema $\mathbf{S}$, and let $h$ be an $\mathbf{S}$-hedge. A *match* of $h$ in $I$ is a mapping $\mu : \mathrm{nodes}(h) \to I$, such that:

1. The image of $\mu$ consists of a single URI; that is, $uri(\mu(u)) = uri(\mu(v))$ for all nodes $u, v \in \mathrm{nodes}(h)$. We denote this URI by $uri(\mu)$.[2]
2. For all nodes $v \in \mathrm{nodes}(h)$, if $\mathrm{label}(v)$ is a concept, then $\mathrm{label}(v)$ is the concept of $\mu(v)$.
3. For all nodes $v \in \mathrm{nodes}(h)$, if $\mathrm{label}(v)$ is a term, then $\mathrm{label}(v) \propto txt(\mu(v))$.
4. For all edges $(u, v) \in \mathrm{edges}(h)$, if $\mathrm{label}(v)$ is a concept $\gamma$, then $\mu(u)[\gamma] = \mu(v)$.
5. For all edges $(u, v) \in \mathrm{edges}(h)$, if $\mathrm{label}(v)$ is a term, then $\mu(v)$ is a descendant of $\mu(u)$.

Note that in the definition of a match $\mu$, if an edge $(u, v)$ of $h$ is such that $\mathrm{label}(u)$ is an a-concept (and so $\mathrm{label}(v)$ is a term), then $\mu(u)$ and $\mu(v)$ must be the same; that holds due to Condition 5 and the fact that an a-record has no descendants other than itself. Also, if an edge $(u, v)$ of $h$ is such that $\mathrm{label}(u)$ is a c-concept and $\mathrm{label}(v)$ is a term (e.g., as in $h_3$ of Figure 2), then

[2] If $h$ is empty, then $uri(\mu)$ is an arbitrary URI.

$\mu(v)$ should match the text of a descendant of $\mu(u)$ (that is, the descendant $\mu(v)$), since $\mu(u)$ itself is a c-record without text.

A URI $u$ *matches* an $\mathbf{S}$-hedge $h$ if $u = uri(\mu)$ for some match $\mu$ of $h$ in $I$.

EXAMPLE 2.5. Consider again the schema $\mathbf{S}$ of our running example (Figure 1(a)) and the instance $I$ of Figure 3. We will illustrate the notion of a match by considering the $\mathbf{S}$-hedge $h_2$ of Figure 2. Since every node of $h_2$ has a unique label, we will describe a match $\mu$ by means of mappings $\mathrm{label}(v) \mapsto r$, where $v$ is a node and $r$ is a record. The following is a match $\mu$ of $h_2$ in $I$, such that $uri(\mu) = \texttt{doc2}$.

$$\overline{\mathsf{prph}} \mapsto s_{\mathsf{pp}} \quad \overline{\mathsf{person}} \mapsto s_{\mathsf{pr}} \quad \overline{\mathsf{phone}} \mapsto s_{\mathsf{ph}}$$
$$\texttt{laura} \mapsto s_{\mathsf{pr}} \quad \texttt{haas} \mapsto s_{\mathsf{pr}}$$

Consider Items 1–5 in the above definition of a match. Item 1 holds for $\mu$ since every record $r$ in the image of $\mu$ is such that $uri(r) = \texttt{doc2}$ (hence, $uri(\mu) = \texttt{doc2}$). For Item 2, observe that each mapping $\gamma \mapsto r$, where $\gamma$ is a concept, is such that $\gamma$ is the concept of $r$. For Item 3, note that both `laura` and `haas` are mapped to $s_{\mathsf{pr}}$, and $txt(s_{\mathsf{pr}}) = \texttt{laura haas}$; hence $\texttt{laura} \propto txt(s_{\mathsf{pr}})$ and $\texttt{haas} \propto txt(s_{\mathsf{pr}})$ (under our specific interpretation of $\propto$ in the examples). For Item 4, note that the nodes of $h_2$ labeled $\overline{\mathsf{prph}}$ and $\overline{\mathsf{person}}$ (which are a parent and a child, respectively) are mapped to $s_{\mathsf{pp}}$ and $s_{\mathsf{pr}}$, respectively, and indeed, $\overline{\mathsf{person}}[s_{\mathsf{pp}}] = s_{\mathsf{pr}}$; the same holds for the nodes with the labels $\overline{\mathsf{prph}}$ and $\overline{\mathsf{phone}}$. Finally, for Item 5 observe that the nodes labeled $\overline{\mathsf{person}}$, `laura` and `haas` are all mapped to the same record.

Since the match $\mu$ exists, we conclude that `doc2` matches $h_2$. Note that `doc2` also matches the $\mathbf{S}$-hedge $h_3$; specifically, a proper match $\mu'$ is the following.

$$\overline{\mathsf{prph}} \mapsto s_{\mathsf{hm}} \quad \texttt{laura} \mapsto s_{\mathsf{pr}} \quad \texttt{haas} \mapsto s_{\mathsf{pr}}$$

Note that although `doc1` contains the person `laura haas` and a phone number, `doc1` does not match $h_2$, since no phone number is associated with `laura haas` (via the concept $\overline{\mathsf{prph}}$). □

We conclude this section with two remarks. First, the notions of a *database instance* and a *match* therein do not play any role in this paper, beyond providing the background that explains the meaning of an $\mathbf{S}$-hedge within the search database system. Second, in some applications it may be desired that, for $\delta \in sc(\gamma)$, a $\gamma$-node of an $\mathbf{S}$-hedge has at most one $\delta$-child. For simplicity of presentation, we do not adopt this restriction; however, the results of this paper can be extended to allow it.

# 3. REWRITE PROGRAMS

In this section, we introduce the notion of a *rewrite program*, and give complexity results for testing properties of rewrite programs. We first give the definition of a rewrite program.

## 3.1 Definition

We fix an infinite set $\mathsf{LVars}$ of *label variables*, and an infinite set $\mathsf{HVars}$ of *hedge variables*. We assume that the sets $\mathsf{Cncpt}$, $\mathsf{Terms}$, $\mathsf{LVars}$ and $\mathsf{HVars}$ are pairwise disjoint.

DEFINITION 3.1. A *hedge expression* is a hedge $E$ over $\mathsf{Cncpt} \cup \mathsf{Terms} \cup \mathsf{LVars} \cup \mathsf{HVars}$, with the following properties.

1. Each variable of $\mathsf{LVars} \cup \mathsf{HVars}$ has at most one occurrence in $E$ (in the terminology of rewriting systems, $E$ is *linear* [25]).
2. Each node of $E$ with a label in $\mathsf{HVars} \cup \mathsf{Terms}$ is a leaf of $E$.

We denote by $\mathsf{LVars}(E)$ and $\mathsf{HVars}(E)$ the sets of label variables and hedge variables, respectively, that occur in $E$. We also denote by $\mathsf{Vars}(E)$ the union $\mathsf{LVars}(E) \cup \mathsf{HVars}(E)$. □

EXAMPLE 3.2. Figure 1(b), which is a part of our running example, shows the expressions $E_i$ and $F_i$ for $i = 1, 2, 3$. In all of the examples of this paper, hedge variables are represented by uppercase letters from the end of the Latin alphabet (e.g., $X$, $Y$ and $Z$), and label variables are represented by lower-case from the end of the Latin alphabet (e.g., $x$, $y$ and $z$). For the expression $E_3$, we have $\mathsf{LVars}(E_3) = \{x, y\}$ and $\mathsf{HVars}(E_3) = \{X, Y, Z\}$. □

An *assignment* for a hedge expression $E$ is a function $\alpha$ over $\mathsf{Vars}(E)$ that maps every label variable $x \in \mathsf{LVars}(E)$ to an element $\alpha(x) \in \mathsf{Cncpt} \cup \mathsf{Terms}$, and every hedge variable $X \in \mathsf{HVars}$ to a hedge $\alpha(X)$ over $\mathsf{Cncpt} \cup \mathsf{Terms}$. The *result* of the assignment $\alpha$ for $E$, denoted $\alpha(E)$, is the hedge that is obtained from $E$ as follows.

- The label of every node $v$ with $\mathrm{label}(v) \in \mathsf{LVars}$ is replaced with $\alpha(\mathrm{label}(v))$.
- Every node $v$ with $\mathrm{label}(v) \in \mathsf{HVars}$ is replaced with the hedge $\alpha(\mathrm{label}(v))$; if $v$ has a parent $w$ in $E$, then $w$ becomes the parent of each root node of $\alpha(\mathrm{label}(v))$.

Let $\mathbf{S}$ be a schema, let $E$ be a hedge expression, and let $\alpha$ be an assignment for $E$. We say that $\alpha$ is $\mathbf{S}$-*consistent* if $\alpha(E)$ is an $\mathbf{S}$-hedge. We also say that $E$ is $\mathbf{S}$-consistent if there exists an $\mathbf{S}$-consistent assignment for $E$.

EXAMPLE 3.3. Consider again the expressions $E_i$ and $F_i$ (for $i = 1, 2, 3$) of our running example (Figure 1(b)). Let $\alpha$ be the assignment that maps $x$ to $\overline{\mathsf{prph}}$, $y$ to `laura`, $X$ to the empty hedge, $Y$ to the (single-node) hedge `haas`, and $Z$ to the (single-node) hedge $\overline{\mathsf{phone}}$. Then $\alpha(E_3)$ is the hedge $h_2$ of Figure 2. As another example, let $E$ be the hedge expression $X$ `number`, and let $\alpha'$ be the assignment that maps $X$ to the tree $\overline{\mathsf{person}}(\texttt{laura haas})$. Then $\alpha'(E)$ is the hedge $h_1$ of Figure 2.

Recall the schema $\mathbf{S}$ of our running example (Figure 1(a)). Note that the above assignment $\alpha$ is $\mathbf{S}$-consistent, since the hedge $h_2$ is an $\mathbf{S}$-hedge. Hence, $E_3$ is $\mathbf{S}$-consistent. It is easy to verify that each of the expressions of Figure 2 is $\mathbf{S}$-consistent.

The reader may wonder what role the variable $y$ plays in the expression $E_3$. The answer is that $y$ guarantees that the string under $\overline{\mathsf{person}}$ is nonempty, since every $\mathbf{S}$-consistent assignment $\alpha$ maps $y$ to a term. In contrast, an assignment for $E_2$ may place nothing under the $\overline{\mathsf{person}}$ node, which happens when $Y$ is mapped to the empty hedge. □

The following theorem states that there is a polynomial-time algorithm for testing whether a given hedge expression is $\mathbf{S}$-consistent. Our specific algorithm for realizing this result applies dynamic programming, based on a bottom-up traversal of the given expression.

THEOREM 3.4. *Testing whether a hedge expression $E$ is $\mathbf{S}$-consistent, given $\mathbf{S}$ and $E$, is in polynomial time.*

Next, we define a rewrite rule.

DEFINITION 3.5. A *rewrite rule* is an expression of the form $E \Rightarrow F$, where $E$ and $F$ are hedge expressions satisfying $\mathsf{Vars}(F) \subseteq \mathsf{Vars}(E)$. □

Consider rule $E \Rightarrow F$, and let $h$ be an $\mathbf{S}$-hedge for some schema $\mathbf{S}$. An assignment $\alpha$ for $E$, such that $\alpha(E) = h$, gives rise to the production of the hedge $\alpha(F)$.

EXAMPLE 3.6. Figure 1(b) shows the rewrite rules of our running example. Specifically, there are rewrite rules $E_i \Rightarrow F_i$, denoted $\rho_i$, for $i = 1, 2, 3$. Recall that the expressions $E_i$ and $F_i$ were discussed in Example 3.2. For $\rho_3$ we have that $\mathsf{Vars}(F_3) = \{y, Y\}$

and $\mathsf{Vars}(E_3) = \{X, Y, Z, x, y\}$; hence, $\mathsf{Vars}(F_3) \subseteq \mathsf{Vars}(E_3)$, as required for a rewrite rule. Note that the variables $x$, $X$ and $Z$ appear in the left-hand side $E_3$ of rule $\rho_3$, but not in the right-hand side $F_3$. In particular, if $\rho_3$ is fired, then $x$ is necessarily instantiated with either $\overline{\mathsf{prhome}}$ or $\overline{\mathsf{prph}}$, and rule $\rho_3$ says that we can then ignore the contents of $X$ and $Z$ and simply look for the relevant home page. □

In terms of standard *rewriting systems* [25], our rules hold the following properties.

- *Hedge rewriting* [11] (i.e., they define a transformation of a hedge into another hedge).[3]
- *Context sensitive* [20]; actually, our rules are applied in a very special context: the full hedge (context) is captured in each of the two sides of a rule.
- *Linear* [25] (that is, every variable can have at most one occurrence in each side of the rule).
- *Conditional* [13]; this property means that the rule is applied to a hedge $h$ only when $h$ satisfies some condition, and here are condition is conformance to the schema $\mathbf{S}$.

In addition, to the best of our knowledge the label variable (which is needed to express, e.g., non-emptiness of the assigned hedges) has not been used in hedge rewriting.

Among the rewrite rules that are used in our implemented system (developed at IBM Research–Almaden), many rules are what we call *replacement rules*. Intuitively, when such a rule is applied to an $\mathbf{S}$-hedge $t_1 \cdots t_n$, it replaces a sub-hedge $h = t_k \cdots t_{k+i}$ with a new hedge $h'$ in a context-free manner. Formally, a replacement rule has the form $X\ h\ Y \Rightarrow X\ h'\ Y$, where $X$ and $Y$ are hedge variables (as implied by the notation), and $h$ and $h'$ are hedges that contain no variables (that is, $h$ and $h'$ are hedges over $\mathsf{Cncpt} \cup \mathsf{Terms}$). As an example, in Figure 1(b) the rule $\rho_1$ is a replacement rule, but neither $\rho_2$ nor $\rho_3$ is a replacement rule (specifically, note that in each side of $\rho_2$ the hedge between $X$ and $Z$ contains a variable $Y$, and that cannot happen in a replacement rule). Furthermore, a replacement rule $X\ h\ Y \Rightarrow X\ h'\ Y$ is a *semi-Thue* rule [25] if $h$ and $h'$ are search queries (i.e., consist of only terms).

Next, we consider consistency of a rewrite rule with a schema.

DEFINITION 3.7. Let $\mathbf{S}$ be a schema. A rewrite rule $E \Rightarrow F$ is $\mathbf{S}$-consistent if the following hold.

1. $E$ is $\mathbf{S}$-consistent (i.e., there exists an $\mathbf{S}$-consistent assignment for $E$).
2. $\alpha(F)$ is an $\mathbf{S}$-hedge for all $\mathbf{S}$-consistent assignments $\alpha$ for $E$. □

EXAMPLE 3.8. Consider again the schema $\mathbf{S}$ and the rules $\rho_i$ (for $i = 1, 2, 3$) of our running example (Figure 1). Recall from Example 3.3 that each $E_i$ (and each $F_i$) is $\mathbf{S}$-consistent. We claim that each $\rho_i$ is $\mathbf{S}$-consistent, and we will prove that for $\rho_2$ (the proofs for $\rho_1$ and $\rho_3$ are similar or simpler). Let $\alpha$ be an $\mathbf{S}$-consistent assignment for $E_2$. Then $\alpha(F_2)$ is an $\mathbf{S}$-hedge if and only if $\alpha(Y)$ is a hedge that can be put under $\overline{\mathsf{person}}$; since $\overline{\mathsf{person}}$ is an a-concept, a $\overline{\mathsf{person}}$ node can have only terms as children. But, it is guaranteed that $\alpha(Y)$ consists of only terms, because $\alpha$ is $\mathbf{S}$-consistent for $E_2$, and in $E_2$ the $Y$ node is a child of the $\overline{\mathsf{person}}$ node.

Consider now the rule $E \Rightarrow F$, which we denote by $\rho$, of Figure 4. Clearly, both $E$ and $F$ are $\mathbf{S}$-consistent. Nevertheless, $\rho$ itself is not $\mathbf{S}$-consistent, since there is an $\mathbf{S}$-consistent assignment $\alpha$ that maps $x$ to $\overline{\mathsf{prhome}}$, and yet, $\overline{\mathsf{phone}}$ (which occurs in $F$ under $x$) is not a subconcept of $\overline{\mathsf{prhome}}$. □

[3]To the best of our knowledge, the vast majority of the literature on rewriting systems considers *term rewriting* and *string rewriting* that apply transformations to trees and strings, respectively.

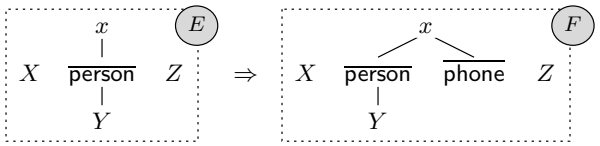



**Figure 4: A rule $\rho$ that is not S-consistent**

Observe that if $\rho$ is the replacement rule $X\ h\ Y \Rightarrow X\ h'\ Y$, then $\rho$ is $\mathbf{S}$-consistent if and only if both of its sides are $\mathbf{S}$-consistent, which is equivalent to saying that both $h$ and $h'$ are $\mathbf{S}$-hedges.

The following theorem states that there is a polynomial-time algorithm for testing whether a given rule $E \Rightarrow F$ is $\mathbf{S}$-consistent. We have an interesting algorithm for realizing this result. Intuitively, our algorithm attempts to find labels that, if used instead of certain variables, would make $F$ inconsistent while leaving $E$ consistent; for these consistency tests, we use the algorithm of Theorem 3.4.

THEOREM 3.9. *Testing for the $\mathbf{S}$-consistency of a rewrite rule $\rho$, given $\mathbf{S}$ and $\rho$, is in polynomial time.*

Finally, we define a *rewrite program*.

DEFINITION 3.10. A *rewrite program* is pair $(\mathbf{S}, R)$, where $\mathbf{S}$ is a schema, and $R$ is a finite set of $\mathbf{S}$-consistent rewrite rules. □

EXAMPLE 3.11. For the schema $\mathbf{S}$ and the set $R$ of our running example (Figure 1), the pair $(\mathbf{S}, R)$ is a rewrite program. Recall from Example 3.8 that each of the rules of $R$ is $\mathbf{S}$-consistent (hence, $(\mathbf{S}, R)$ indeed satisfies the definition of a rewrite program). □

## 3.2 Semantics of a Rewrite Program

A rewrite program $(\mathbf{S}, R)$ is used within the execution of a search engine, having an instance $I$ over $\mathbf{S}$, and the rewrite program is applied as follows (at least conceptually). Given a search query $\mathbf{q}$ submitted by the user, each rule of $R$ that is applicable to $\mathbf{q}$ is applied to $\mathbf{q}$, generating new $\mathbf{S}$-hedges $h$. We then further apply the rules of $R$ to the generated $\mathbf{S}$-hedges $h$, and continue doing so until no new $\mathbf{S}$-hedge can be produced. (Later on, we will discuss the question of whether or not this process indeed terminates.) We then use the set of generated $\mathbf{S}$-hedges, first for finding the URIs (of the instance $I$) that match the final set of hedges, and second for *ranking*, which we discuss briefly at the beginning of Section 4.

We now formally define the application of a rewrite program $(\mathbf{S}, R)$ to a search query $\mathbf{q}$. For $\mathbf{S}$-hedges $h_1$ and $h_2$, we denote by $h_1 \Rightarrow_R h_2$ the fact that there is a rule $E \Rightarrow F$ of $R$ and an assignment $\alpha$ for $E$, such that $h_1 = \alpha(E)$ and $h_2 = \alpha(F)$. The $(\mathbf{S}, R)$-*graph* is a special case of the well-known notion of an *abstract rewrite system*, and is formally defined as follows.

DEFINITION 3.12. The $(\mathbf{S}, R)$-*graph*, denoted $\mathcal{G}(\mathbf{S}, R)$, is the infinite directed graph $(V, E)$, where $V$ is the set of all $\mathbf{S}$-hedges, and $E$ contains all the edges $(h_1, h_2)$, such that $h_1 \Rightarrow_R h_2$. □

We view $(\mathbf{S}, R)$ as a logic program that gets as input a single search string $\mathbf{q} \in \mathsf{Terms}^*$, and produces a set of $\mathbf{S}$-hedges, such that this set forms a *least fixpoint* (which we often abbreviate as *lfp*). More precisely, let $h$ be an $\mathbf{S}$-hedge (e.g., a search query), and let $H$ be a (possibly infinite) set of $\mathbf{S}$-hedges. We say that $H$ is a *fixpoint* for $h$ if $h \in H$, and furthermore, for all $\mathbf{S}$-hedges $h_1$ and $h_2$ with $h_1 \Rightarrow_R h_2$, if $h_1 \in H$ then $h_2 \in H$ as well. We say that $H$ is a *least fixpoint* for $h$ if $H$ is a fixpoint for $\mathbf{q}$, and $H \subseteq H'$ for all fixpoints $H'$ for $\mathbf{q}$.

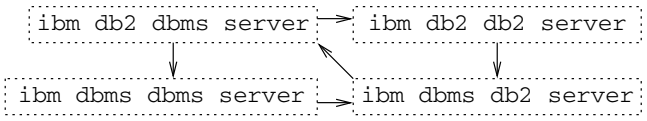



**Figure 5: The subgraph of $\mathcal{G}(\mathbf{S}, R')$ that is reachable from** `ibm db2 dbms server`

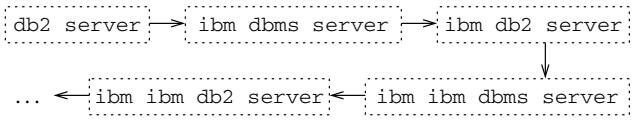



**Figure 6: The infinite subgraph of $\mathcal{G}(\mathbf{S}, R'')$ reachable from** `ibm db2 server`

By applying standard observations from fixpoint logic, it easily follows that a least fixpoint for a hedge $h$ always exists, and is unique: it is exactly the set of $\mathbf{S}$-hedges that are reachable from $h$ in $\mathcal{G}(\mathbf{S}, R)$. We denote by $lfp_R^{\mathbf{S}}(h)$ the least fixpoint for $h$. When $R$ and $\mathbf{S}$ are clear from the context, we may write just $lfp(h)$ instead of $lfp_R^{\mathbf{S}}(h)$.

We can now formalize the application of a rewrite program to a search query (we informally discussed this notion at the beginning of this section). Given a search query $\mathbf{q}$ submitted by the user, *applying* $(\mathbf{S}, R)$ *to* $\mathbf{q}$ means generating the set $lfp(\mathbf{q})$ (and terminating if $lfp(\mathbf{q})$ is finite).

EXAMPLE 3.13. Consider the rewrite program $(\mathbf{S}, R)$ of our running example (Figure 1). Consider also the search query $\mathbf{q}$ of Figure 2 and the $\mathbf{S}$-hedges $h_i$ (for $i = 1, 2, 3$) of Figure 2. When taking the arrows entering the $h_i$ in Figure 2 as directed edges, the figure actually shows a path in (hence, a subgraph of) $\mathcal{G}(\mathbf{S}, R)$. Note that the rules $\rho_i$ labeling the edges are not a part of $\mathcal{G}(\mathbf{S}, R)$; they are given only for convenience (i.e., to show the rule that gives rise to the edge). The reader can easily verify that $lfp(\mathbf{q}) = \{\mathbf{q}, h_1, h_2, h_3\}$. □

EXAMPLE 3.14. Let $\mathbf{S}$ be a schema. Consider the following set $R'$ of three replacement rules, saying that a user can use `ibm db2` to refer to `ibm dbms`, `dbms server` to refer to `db2 server`, and `dbms db2` to refer to `db2 dbms`.

$$X\ \texttt{ibm db2}\ Y \Rightarrow X\ \texttt{ibm dbms}\ Y$$
$$X\ \texttt{dbms server}\ Y \Rightarrow X\ \texttt{db2 server}\ Y$$
$$X\ \texttt{dbms db2}\ Y \Rightarrow X\ \texttt{db2 dbms}\ Y$$

Figure 5 shows the subgraph of $\mathcal{G}(\mathbf{S}, R')$ that is reachable from the search query `ibm db2 dbms server`.

Now consider the set $R''$ that consists of the following two replacement rules:

$$X\ \texttt{db2}\ Y \Rightarrow X\ \texttt{ibm dbms}\ Y$$
$$X\ \texttt{dbms server}\ Y \Rightarrow X\ \texttt{db2 server}\ Y$$

Note that the rules of $R''$ are the first two rules of $R'$, except that the left-hand side of the first rule is `db2` instead of `ibm db2`. Figure 6 shows the subgraph of $\mathcal{G}(\mathbf{S}, R'')$ that is reachable from `db2 server`. Note that this subgraph is an infinite path that contains every string of the form $\texttt{ibm}^n$ `db2 server` for $n \in \mathbb{N}$, where $\texttt{ibm}^n$ is the string `ibm` repeated $n$ times. □

We emphasize that, unlike the usual interest in rewriting systems, in our semantics there is no (a priori) importance to whether or not a hedge $h$ is a *normal form* (or *irreducible*, which means that no rule can be applied to $h$ for producing a different hedge).

# 4. LFP-CONVERGENCE

Recall that when applying a program $(\mathbf{S}, R)$ to a search query $\mathbf{q}$, we would like to produce the set $lfp(\mathbf{q})$. From a practical point of view, having the set $lfp(\mathbf{q})$ at hand is important not only to identify the set of matching documents, but also for the crucial task of *ranking* (which is beyond the scope of this paper). Essentially, in our system hedges are ranked by reasoning about their properties (e.g., the concepts involved) and the set of rules involved in producing the hedges. For example, some rules are marked as "improving" the query, and when ranking matching documents such information is incorporated along with other measures (e.g., the extent to which keywords are matched, and the global quality of documents). Therefore, it is critical that the production of $lfp(\mathbf{q})$ will terminate on every search query $\mathbf{q}$ (and if that is not the case, then the program violates the intent of the developer phrasing the rules). A rewrite program with this property is said to be *lfp-convergent*. The formal definition follows.

DEFINITION 4.1. A rewrite program $(\mathbf{S}, R)$ is *lfp-convergent* if $lfp(\mathbf{q})$ is finite for all search queries $\mathbf{q} \in \mathsf{Terms}^*$. □

EXAMPLE 4.2. Consider again the rewrite program $(\mathbf{S}, R)$ in Example 3.11 for our running example, and consider the sets $R'$ and $R''$ defined in Example 3.14. In Section 5, we will show that the programs $(\mathbf{S}, R)$ and $(\mathbf{S}, R')$ are both lfp-convergent. However, the program $(\mathbf{S}, R'')$ is not lfp-convergent, since $lfp(\mathbf{q})$ is infinite when $\mathbf{q}$ is the search query `ibm db2 server` (as explained in Example 3.14). □

According to Definition 4.1, a rewrite program $(\mathbf{S}, R)$ is lfp-convergent if $lfp(h)$ is finite whenever the $\mathbf{S}$-hedge $h$ is a search query. A closely related notion is *weak termination* (or *quasi-termination*) [8], which in our setting means that $lfp(h)$ is finite for *every* $\mathbf{S}$-hedge $h$ (rather than for only those that are search queries). For clarity of presentation, here we call weak termination *strong lfp-convergence*. For instance, we will later show that the rewrite program $(\mathbf{S}, R)$ of our running example is strongly lfp-convergent. Clearly, strong lfp-convergence implies lfp-convergence; the other direction, though, is not true, as the following example illustrates.

EXAMPLE 4.3. Let $\mathbf{S}$ be the schema of our running example (Figure 1(a)), and consider the set $R$ that consists of the following single replacement rule:

$$X\ \overline{\texttt{person}}\ Y \Rightarrow X\ \overline{\texttt{person}}\ \texttt{facebook}\ Y$$

Let $h$ be the single-node $\mathbf{S}$-hedge $\overline{\texttt{person}}$. Clearly, $lfp(h)$ is infinite (as it includes the $\mathbf{S}$-hedge $\overline{\texttt{person}}\ \texttt{facebook}^n$ for all $n \in \mathbb{N}$), and hence, the system $(\mathbf{S}, R)$ is not strongly lfp-convergent. However, the one rule of $R$ is not applicable to a search query $\mathbf{q}$ (since $\mathbf{q}$ does not contain the concept $\overline{\texttt{person}}$). That implies that $lfp(\mathbf{q}) = \{\mathbf{q}\}$ for all search queries $\mathbf{q}$, and in particular, $(\mathbf{S}, R)$ is lfp-convergent.

More interesting is the set $R'$ with the following two rules:

$$\texttt{locate number}\ X \Rightarrow \overline{\texttt{phone}}\ \overline{\texttt{person}}\ X$$
$$\overline{\texttt{phone}}\ x\ Y \Rightarrow x\ \overline{\texttt{phone}}\ \texttt{whitepages}\ Y$$

Unlike the set $R$ in the previous example, both rules of $R'$ can fire. For example, the first rule rewrites `locate number laura` as $\overline{\texttt{phone}}\ \overline{\texttt{person}}$ `laura`, which in then rewritten by the second rule as $\overline{\texttt{person}}\ \overline{\texttt{phone}}$ `whitepages laura`. It is easy to see that for a search query $\mathbf{q}$, either none of the two rules is applicable

(and then $lfp(\mathbf{q}) = \{\mathbf{q}\}$), or each of the two rules is applied exactly once (and then $lfp(\mathbf{q})$ has exactly three hedges). Therefore, $(\mathbf{S}, R')$ is lfp-convergent. However, $(\mathbf{S}, R')$ is not strongly lfp-convergent. For example, $lfp(h)$ is infinite for $h =$ phone phone; to see that, note that using just the second rule, $h$ is transformed into phone phone whitepages, which is transformed into phone phone whitepages whitepages, and so on, so that $lfp(h)$ contains phone phone whitepages$^n$ for all $n \in \mathbb{N}$. □

Ideally, we would like to be able to verify that a rewrite program is lfp-convergent, and warn the developer if not. A typical use case of such verification is when a new rule is added to the program; if the program is non-lfp-convergent following this addition, then the developer is urged to modify (or avoid) the new rule, or to change rules that already exist in the program. One may desire a closely related verification at runtime: once a query $\mathbf{q}$ is submitted by a user, verify that $lfp(\mathbf{q})$ is finite before applying the program.

Unfortunately, the following theorem, which follows easily from the work of Guttag et al. [8] (specifically, their results on weak termination), implies that neither of the above two types of verification is possible in the general case. Specifically, the theorem states that lfp-convergence (or strong lfp-convergence) of a given rewrite program $(\mathbf{S}, R)$, as well as finiteness of $lfp(\mathbf{q})$ for a given search query $\mathbf{q}$, is undecidable. Furthermore, undecidability remains even if only replacement rules are allowed, and even if we fix *any* schema $\mathbf{S}$.

THEOREM 4.4. *Let $\mathbf{S}$ be a fixed schema. The following problems are undecidable.*

1. *Given a set $R$ of $\mathbf{S}$-consistent replacement rules, determine whether $(\mathbf{S}, R)$ is (strongly) lfp-convergent.*
2. *Given a set $R$ of $\mathbf{S}$-consistent replacement rules and a search query $\mathbf{q}$, determine whether $lfp(\mathbf{q})$ is finite.*

In the next sections, we devise tractable *safety conditions* that are sufficient (but not necessary) for lfp-convergence.

# 5. SAFETY CONDITION

Theorem 4.4 shows that it is impossible to determine automatically whether a given rewrite program is lfp-convergent. Still, we would like to have a tractable *safety condition*, which guarantees lfp-convergence, and which is robust enough to capture lfp-convergence in practical rewrite programs.

There is a vast literature on proof techniques for *termination* of rewrite systems (e.g., [1, 3, 4, 14]), especially for tree rewriting. In our terminology, usual termination means that the $(\mathbf{S}, R)$-graph has no infinite paths (e.g., cycles). These techniques find a *well-founded* order on the set of trees, such that the order is respected by the rules [18]. But recall that there is a major difference between termination and lfp-convergence (or weak termination). As a simple example, the program $(\mathbf{S}, R')$ of Example 3.14 is clearly non-terminating (see Figure 5), but is nevertheless (strongly) lfp-convergent as we show later. Examples like that are abundant in practical rewrite programs, since they include replacements among synonyms and similar terms, and such rules cause cycles.

There are also proof techniques for weak termination [5,7]. However, it is not at all clear to us how these existing techniques can be translated into efficient (i.e., polynomial-time) safety checking (rather than just proof techniques). More importantly, to the best of our knowledge none of the existing techniques for weak termination takes into account the special (key) properties of our rewrite programs, notably linearity and being an extreme case of context sensitive (i.e., the whole context is represented). Finally, to the best of our knowledge, weak termination has not been studied for hedge rewriting. Therefore, in this section we propose a tractable safety condition. In Section 5.5, we will discuss the performance of the safety condition in our implemented search database system, where we found the condition to be highly successful in capturing lfp-convergence.

## 5.1 Definitions and Notation

We first need some definitions and notation. The principal new notions are those of an *invocation path*, an *invocation cycle*, and a *guarding potential*.

### 5.1.1 Invocation Paths and Cycles

Let $\mathbf{S}$ be a schema, and let $E_1$ and $E_2$ be two hedge expressions. We say that $E_1$ and $E_2$ are $\mathbf{S}$-*unifiable* if there are $\mathbf{S}$-consistent assignments $\alpha_1$ and $\alpha_2$ for $E_1$ and $E_2$, respectively, such that $\alpha_1(E_1) = \alpha_2(E_2)$. For example, an easy observation is that every two $\mathbf{S}$-consistent hedge expressions that start and end with hedge variables (e.g., expressions of the form $X\ E\ Y$ where $E$ is $\mathbf{S}$-consistent) are $\mathbf{S}$-unifiable.[4] In particular, if $E_1 \Rightarrow F_1$ and $E_2 \Rightarrow F_2$ are replacement rules, then every two expressions in $\{E_1, F_1, E_2, F_2\}$ are $\mathbf{S}$-unifiable. Additional examples are given next.

EXAMPLE 5.1. Consider again the rewrite program $(\mathbf{S}, R)$ of our running example (Figure 1). The expressions $F_2$ and $E_3$ are $\mathbf{S}$-unifiable, since there are assignments $\alpha_2$ and $\alpha_3$ for $F_2$ and $E_3$, respectively, such that $\alpha_2(E_2) = \alpha_3(F_3) = h_2$, where the $\mathbf{S}$-hedge $h_2$ is depicted in Figure 2. Due to the remark preceding this example, every pair in $\{E_1, F_1, E_2, F_2\}$ is $\mathbf{S}$-unifiable (since each of them starts and ends with a hedge variable). It is easy to show that for all $E' \in \{E_1, F_1, E_2, F_3\}$, the expressions $E_3$ and $E'$ are not $\mathbf{S}$-unifiable; similarly, for all $E' \in \{E_1, F_1, E_2, F_2, E_3\}$, the expressions $F_3$ and $E'$ are not $\mathbf{S}$-unifiable. □

Next, we define the notions of an *invocation path* and an *invocation cycle*.

DEFINITION 5.2. Let $(\mathbf{S}, R)$ be a rewrite program. An *invocation path* is a (finite or infinite) sequence $E_1 \Rightarrow F_1, E_2 \Rightarrow F_2 \ldots$ of rules of $R$, such that $F_i$ and $E_{i+1}$ are $\mathbf{S}$-unifiable for all $i$. A finite invocation path $E_1 \Rightarrow F_1, \ldots, E_n \Rightarrow F_n$ is an *invocation cycle* if $F_n$ and $E_1$ are $\mathbf{S}$-unifiable. □

Note that in an invocation path $E_1 \Rightarrow F_1, E_2 \Rightarrow F_2 \ldots$, the rules $E_i \Rightarrow F_i$ are not necessarily distinct. It is not hard to show that a cycle in the graph $\mathcal{G}(\mathbf{S}, R)$ (which was defined in Definition 3.12) implies the existence of an invocation cycle. However, the converse is not necessarily true. As a simple example of the failure of the converse, let $R$ consist of the single rewrite rule ibm $X \Rightarrow X$, which might be a rule in an IBM intranet system where ibm is a noise word, and let $p$ consist of this single rule. It is straightforward to verify that $p$ is an invocation cycle of $(\mathbf{S}, R)$, but that there is no cycle in $\mathcal{G}(\mathbf{S}, R)$.

If $p$ is an invocation path, we denote by $\mathrm{rules}(p)$ the set of all rewrite rules $E_i \Rightarrow F_i$ that participate in $p$. In the sequel, we may discuss an invocation path without mentioning the underlying rewrite program $(\mathbf{S}, R)$.

EXAMPLE 5.3. We consider again the rewrite program $(\mathbf{S}, R)$ of our running example (Figure 1). Recall that $\mathbf{S}$-unifiability among the $E_i$ and the $F_i$ is discussed in Example 5.1. It follows from

[4]This is true since if $\alpha$ and $\alpha'$ are assignments for $E$ and $E'$, respectively, then there are assignments for $X\ E\ Y$ and $X'\ E'\ Y'$ that produce $\alpha(E)\alpha(E')$.

the observations in Example 5.1 that the following are invocation paths:

- $p_1 : E_1 \Rightarrow F_1, E_2 \Rightarrow F_2, E_2 \Rightarrow F_2$
- $p_2 : E_1 \Rightarrow F_1$
- $p_3 : E_2 \Rightarrow F_2, E_3 \Rightarrow F_3$
- $p_4 : E_1 \Rightarrow F_1, E_2 \Rightarrow F_2, E_1 \Rightarrow F_1, E_2 \Rightarrow F_2, \cdots$

Note that $p_1$, $p_2$ and $p_3$ are finite invocation paths, while $p_4$ is infinite. Also note that $p_1$ is an invocation cycle, since $F_2$ and $E_1$ are $\mathbf{S}$-unifiable. Similarly, $p_2$ is an invocation cycle, since $F_1$ and $E_1$ are $\mathbf{S}$-unifiable. Finally, $p_3$ is not an invocation cycle, since $F_3$ and $E_2$ are not $\mathbf{S}$-unifiable. □

Let $E$ be a hedge expression. We denote by $\|E\|$ the number of nodes $v$ of $E$, such that $\mathrm{label}(v)$ is not a hedge variable (i.e., $\mathrm{label}(v)$ is a term, a concept, or a label variable). Note that a term or a concept can be counted multiple times if it has multiple occurrences in $E$. Next, we define when a finite invocation path is *expanding*.

DEFINITION 5.4. An invocation path $E_1 \Rightarrow F_1, \ldots, E_n \Rightarrow F_n$ is said to be *expanding* if $\sum_{i=1}^{n} \|E_i\| < \sum_{i=1}^{n} \|F_i\|$; otherwise, $E_1 \Rightarrow F_1, \ldots, E_n \Rightarrow F_n$ is *nonexpanding*. □

For instance, consider the paths $p_1$ and $p_3$ of Example 5.3. For $p_1$ we have $\|E_1\| + \|E_2\| + \|E_2\| = 2 + 2 + 2 = 6$ and $\|F_1\| + \|F_2\| + \|F_2\| = 3 + 3 + 3 = 9$; hence $p_1$ is expanding. For $p_3$ we have $\|E_2\| + \|E_3\| = 2 + 3 = 5$ and $\|F_2\| + \|F_3\| = 3 + 2 = 5$; hence, $p_3$ is nonexpanding. Another example is given next.

EXAMPLE 5.5. Let $\mathbf{S}$ be a schema, and consider the set $R$ that consists of the following three rules:

- $X$ `home` $\Rightarrow X$ `home page`
- $X$ `home page` $\Rightarrow X$ `personal info page`
- $X$ `page` $\Rightarrow X$

Denote the first rule by $E_1 \Rightarrow F_1$, the second by $E_2 \Rightarrow F_2$, and the third by $E_3 \Rightarrow F_3$. Note that $F_1$ and $E_2$ are $\mathbf{S}$-unifiable, and so are $F_2$ and $E_3$, and $F_3$ and $E_1$. Also $\mathbf{S}$-unifiable are $F_1$ and $E_3$, and $F_3$ and $E_i$ for all $i \in \{1, 2, 3\}$. Let $p_1$ be the invocation path $E_1 \Rightarrow F_1$. Since $\|E_1\| = 1$ and $\|F_1\| = 2$, the path $p_1$ is expanding. Now, consider the following two invocation cycles:

- $p : E_1 \Rightarrow F_1, E_2 \Rightarrow F_2, E_3 \Rightarrow F_3$
- $p' : E_1 \Rightarrow F_1, E_3 \Rightarrow F_3$

For $p$ we have $\|E_1\|+\|E_2\|+\|E_3\| = 4$ and $|F_1\|+\|F_2\|+\|F_3\| = 5$; hence, $p$ is expanding. But for $p'$ we have $\|E_1\| + \|E_3\| = \|F_1\| + \|F_3\| = 2$, and hence, $p'$ is nonexpanding. □

### 5.1.2 Guarding Potentials

We call a member of $\mathsf{Cncpt} \cup \mathsf{Terms}$ a *constant*. Furthermore, if $v$ is a node of a hedge expression $E$, and $\mathrm{label}(v)$ is a constant, then we call $v$ a *constant node*. We denote by $\mathrm{nodes}^{\mathsf{ct}}(E)$ the set of all constant nodes of $E$; that is, $\mathrm{nodes}^{\mathsf{ct}}(E) = \{v \in \mathrm{nodes}(E) \mid \mathrm{label}(v) \in \mathsf{Cncpt} \cup \mathsf{Terms}\}$.

In this paper, a *potential function* (or just *potential* for short) is a numerical nonnegative function $\pi$ over the constants; that is, $\pi : \mathsf{Cncpt} \cup \mathsf{Terms} \to [0, \infty)$. If $E$ is a hedge expression and $\pi$ is a potential, then we define:

$$\pi(E) \stackrel{\text{def}}{=} \sum_{v \in \mathrm{nodes}^{\mathsf{ct}}(E)} \pi(\mathrm{label}(v))$$

Let $\pi$ be a potential, let $E \Rightarrow F$ be a rule, and let $R$ be a set of rewrite rules. We say that $\pi$ is *(1) nonincreasing on $\rho$* if $\pi(E) \geq \pi(F)$, *(2) decreasing on $\rho$* if $\pi(E) > \pi(F)$, *(3) nonincreasing on $R$* if $\pi$ is nonincreasing on every rule of $R$, *(4) weakly decreasing on $R$* if $\pi$ is nonincreasing on $R$ and decreasing on one or more rules of $R$, *and (5) positive-nonincreasing on $R$* if $\pi$ is nonincreasing on $R$ and $\pi(c) > 0$ for all the constants $c$ that appear in $R$. Lastly, we define a *guarding potential*.

DEFINITION 5.6. A *guarding potential* for a set $R$ of rules is a potential $\pi$ that is either weakly decreasing on $R$ or positive-nonincreasing on $R$. A *guarding potential* for an invocation cycle $p$ is a guarding potential for $\mathrm{rules}(p)$. □

EXAMPLE 5.7. Let $\mathbf{S}$ be a schema, and let $R$ be the set of rules from Example 5.5. Consider again the invocation cycles $p$ and $p'$ of Example 5.5. Define the potential $\pi$ by $\pi(\texttt{home}) = 1$ and $\pi(c) = 0$ for every other constant $c$. Note that $\pi$ is nonincreasing on $\mathrm{rules}(p)$; for example, $\pi(E_1) = \pi(F_1) = 1$ and $\pi(E_3) = \pi(F_3) = 0$. Moreover, $\pi(E_2) = 1$ and $\pi(F_2) = 0$ (since `home` occurs in $E_2$ but not in $F_2$); hence, $\pi$ is weakly decreasing on $\mathrm{rules}(p)$. Thus, $\pi$ is a guarding potential for $p$.

Note that $\pi$ is not a guarding potential for $p'$, since $\pi$ is neither weakly decreasing nor positive-nonincreasing on $\mathrm{rules}(p')$. Actually, there is no guarding potential for $p'$: if $\pi'$ is such a potential, then $\pi'(\texttt{page}) > 0$ must hold; but then, $\pi'(E_1) < \pi'(F_1)$, as opposed to the fact that a guarding potential is nonincreasing. □

EXAMPLE 5.8. A common use of rewrite rules is to apply substitution among acronyms. Such a substitution is done by a set of replacement rules as in the following example:

- $X$ `nyc` $Y \Rightarrow X$ `new york city` $Y$
- $X$ `new york city` $Y \Rightarrow X$ `big apple` $Y$
- $X$ `big apple` $Y \Rightarrow X$ `nyc` $Y$

Note that the three rules are replacement rules, which implies that every two of the hedge expressions are $\mathbf{S}$-unifiable (for the underlying schema $\mathbf{S}$). Consider an invocation cycle $p$ that contains all three rules. An easy observation is that there is no potential that is weakly decreasing on $\mathrm{rules}(p)$. However, positive-nonincreasing on $p$ is the potential $\pi$ defined by the following assignments:[5]

- $\pi(\texttt{nyc}) = 1$
- $\pi(\texttt{new}) = \pi(\texttt{york}) = \pi(\texttt{city}) = \frac{1}{3}$
- $\pi(\texttt{big}) = \pi(\texttt{apple}) = \frac{1}{2}$

Observe that $\pi$ is indeed positive-nonincreasing on $p$, and hence, $\pi$ is a guarding potential for $p$. □

## 5.2 The Safety Condition

We can now present our safety condition for a rewrite program.

SAFETY CONDITION. *Every expanding invocation cycle has a guarding potential.*

A program $(\mathbf{S}, R)$ that satisfies the safety condition is said to be *safe*; otherwise it is *unsafe*. The following theorem states the correctness of the safety condition, that is, that its satisfaction guarantees lfp-convergence. Actually, even strong lfp-convergence is guaranteed. (Recall from Theorem 4.4 that strong lfp-convergence is undecidable, as is usual lfp-convergence.) The proof shows a contradiction to the existence of a simple infinite path of $\mathcal{G}(\mathbf{S}, R)$ which, by an argument similar to the proof of König's lemma [15], must be present in the absence of strong lfp-convergence.

THEOREM 5.9. *Every safe rewrite program $(\mathbf{S}, R)$ is strongly lfp-convergent.*

A simple example of a safe (hence, lfp-convergent) rewrite program is the program $(\mathbf{S}, R')$ of Example 3.14. There, $\|E\| = \|F\| = 2$ for each rewrite rule $E \Rightarrow F$, and therefore, every invocation cycle is nonexpanding. Next, we give additional examples.

[5]Here and in other places, we define a potential only on the constants used in the rewrite program at hand; the rest of the potential is irrelevant, and can be defined arbitrarily.

EXAMPLE 5.10. Let $\mathbf{S}$ and $R$ be as defined in Example 5.5. We will now prove that $(\mathbf{S}, R)$ is safe. Later, we will give an efficient algorithm for testing safety; for now, the proof will be ad hoc. Consider the invocation cycles $p$ and $p'$ that are given in Example 5.5 and discussed in Example 5.7. In Example 5.7 we showed that $p$ has a guarding potential $\pi$, and in the same way it is shown that $\pi$ is a guarding potential for *every* invocation cycle that includes the rule $E_2 \Rightarrow F_2$. Hence, it suffices to consider invocation cycles $p'_0$ that use only $E_1 \Rightarrow F_1$ and $E_3 \Rightarrow F_3$. Let $p'_0$ be such a cycle. Suppose that $p'_0$ is $E'_1 \Rightarrow F'_1, \ldots, E'_n \Rightarrow F'_n$ (where each $E'_i \Rightarrow F'_i$ is either $E_1 \Rightarrow F_1$ or $E_3 \Rightarrow F_3$). Every occurrence of $E_1 \Rightarrow F_1$ adds 1 to $\sum_{i=1}^{n} \|E'_i\|$ and 2 to $\sum_{i=1}^{n} \|F'_i\|$. Every occurrence of $E_3 \Rightarrow F_3$ adds 1 to $\sum_{i=1}^{n} \|E'_i\|$ and 0 to $\sum_{i=1}^{n} \|F'_i\|$. An easy observation is that if $E'_1 = E_1$, then every occurrence of $E_1 \Rightarrow F_1$ is followed by an occurrence of $E_3 \Rightarrow F_3$, and if $E'_1 = E_3$, then every occurrence of $E_1 \Rightarrow F_1$ is preceded by an occurrence of $E_3 \Rightarrow F_3$. Hence, in both cases, $p'_0$ contains at least as many occurrences of $E_3 \Rightarrow F_3$ as of $E_1 \Rightarrow F_1$, which implies that $\sum_{i=1}^{n} \|E'_i\| \geq \sum_{i=1}^{n} \|F'_i\|$ (i.e., $p'_0$ is nonexpanding). □

EXAMPLE 5.11. Consider again the rewrite program $(\mathbf{S}, R)$ of our running example (Figure 1). Let $p$ be the invocation cycle $E_1 \Rightarrow F_1$ (consisting of one occurrence of $\rho_1$). Note that $p$ is expanding, since $\|E_1\| = 2$ and $\|F_1\| = 3$. Moreover, the bag of constants of $E_1$ is strictly contained in the bag of constants of $F_1$, and this easily implies that there is no guarding potential for $p$. Hence, $(\mathbf{S}, R)$ is unsafe. However, as we discuss next, we will later show that $(\mathbf{S}, R)$ is lfp-convergent, and moreover, its lfp-convergence can be verified by a simple relaxation of the safety condition. □

The rule $\rho_1$ of our running example (Figure 1) belongs to an important class of rewrite rules, which we call *Context-Free Grammar* (*CFG*) rules and discuss later in Section 5.4. We will show how to relax our safety condition to accommodate these rules. We will also show that the program $(\mathbf{S}, R)$ of Example 5.11 satisfies the relaxed safety condition, and hence, is (strongly) lfp-convergent.

Recall that safety is defined due to the inability to test for lfp-convergence. Of course, it would be useless unless safety checking is tractable. This tractability is not at all clear since, on the face of it, infinitely many invocation cycles need to be inspected. In the next section, we describe an algorithm that shows the surprising result that safety can be tested in polynomial time.

## 5.3 Algorithm for Safety Checking

In this section, we present a polynomial-time algorithm for testing the safety of a rewrite program. Our algorithm operates on the *expression graph*, which we now define.

Let $(\mathbf{S}, R)$ be a rewrite program, and let $E \Rightarrow F$ be a rule in $R$. We call $E$ a *left expression* of $R$, and we call $F$ a *right expression* of $R$. The *expression graph* of $(\mathbf{S}, R)$, denoted $\mathcal{X}(\mathbf{S}, R)$, is the directed and edge-weighted graph $G$ that is defined as follows. First, for each left expression $E$ of $R$ there is a unique node $v^l_E$ in $G$, and similarly, for each right expression $F$ of $R$ there is a unique node $v^r_F$ in $G$. Second, $G$ has an edge $e$ from $v^l_E$ to $v^r_F$ if $E \Rightarrow F$ is a rule of $R$; in this case, the weight of $e$ is $\|F\| - \|E\|$. Finally, $G$ has an edge $e$ from $v^r_F$ to $v^l_E$ if $F$ and $E$ are $\mathbf{S}$-unifiable; in this case, the weight of $e$ is zero.

EXAMPLE 5.12. The left side of Figure 7 depicts the graph $\mathcal{X}(\mathbf{S}, R)$ of the rewrite program $(\mathbf{S}, R)$ of our running example (Figure 1). Note that there are three rightward edges that correspond to the three rules of $R$. The leftward edges go from every $F_i$ to every $E_j$ such that $F_i$ and $E_j$ are unifiable. Recall that unifiability among these rules is discussed in Example 5.1. The graph specifies the weights only for rightward edges, while the weights of the leftward edges are zero by definition. As an example, the weight on the edge from $v^l_{E_3}$ to $v^r_{F_3}$ is $-1$, since $\|F_3\| - \|E_3\| = 2 - 3 = -1$.

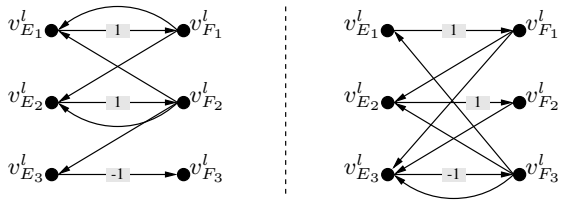



**Figure 7: The expression graphs $\mathcal{X}(\mathbf{S}, R)$ of the rewrite programs $(\mathbf{S}, R)$ of the running example of Figure 1 (left) and Example 5.5 (right)**

The graph $\mathcal{X}(\mathbf{S}, R)$ in the right-hand side of Figure 7 is that of the program $(\mathbf{S}, R)$ of Example 5.5. □

We denote our algorithm for checking the safety of a rewrite program $(\mathbf{S}, R)$ by IsSafe$(\mathbf{S}, R)$, and the pseudo-code is given in Figure 8. Our algorithm operates on the expression graph. Of course, the expression graph can be constructed in polynomial time if $\mathbf{S}$-unifiability can be decided in polynomial time. Hence, the following lemma is needed.

LEMMA 5.13. *Testing whether two hedge expressions are $\mathbf{S}$-unifiable, given the expressions and $\mathbf{S}$, is in polynomial time. Hence, the expression graph can be constructed in polynomial time.*

The first step of the algorithm is based on the observation that $(\mathbf{S}, R)$ has an expanding invocation cycle if and only if $\mathcal{X}(\mathbf{S}, R)$ has a positive cycle (i.e., a cycle such that the weights on the edges sum up to a positive number). We abbreviate "positive-weight cycle" by simply "positive cycle." This means that lack of positive cycles in $\mathcal{X}(\mathbf{S}, R)$ implies safety. So, in lines 1–2 the algorithm tests whether $\mathcal{X}(\mathbf{S}, R)$ has a positive cycle, and if not, it terminates by returning `true`; otherwise, the algorithm continues to the next steps. Note that testing whether a graph has a positive cycle is in polynomial time (e.g., using the *Bellman-Ford* algorithm).

The next step of the algorithm is based on the observation that safety can be tested independently in each *strongly connected component* of $\mathcal{X}(\mathbf{S}, R)$. More formally, let $(\mathbf{S}, R)$ be a rewrite program, and let $U$ be a subset of the nodes of $\mathcal{X}(\mathbf{S}, R)$. Recall that $U$ is a *strongly connected component* (abbreviated *SCC*) if $U$ is a maximal set of nodes in which $\mathcal{X}(\mathbf{S}, R)$ contains a path from every node to every other node. Note that the SCCs are pairwise disjoint. For example, in the left graph of Figure 7 the SCCs are the set $\{v^l_{E_1}, v^r_{F_1}, v^l_{E_2}, v^r_{F_2}\}$ and the singletons $\{v^l_{E_3}\}$ and $\{v^r_{F_3}\}$. In the right graph of Figure 7, $\mathcal{X}(\mathbf{S}, R)$ has exactly one SCC (namely, the one containing all six nodes), and in that case we say that $\mathcal{X}(\mathbf{S}, R)$ is *strongly connected*.

Observe that every cycle of $\mathcal{X}(\mathbf{S}, R)$ must be entirely contained in a single connected component. This observation implies that $(\mathbf{S}, R)$ is safe if and only if $(\mathbf{S}, \mathrm{rules}(U))$ is safe for all SCCs $U$, where $\mathrm{rules}(U)$ is the set of all the rules $E \Rightarrow F$ of $R$ with both $v^l_E$ and $v^r_F$ in $U$. So, if $\mathcal{X}(\mathbf{S}, R)$ has two or more SCCs, then in Line 5 the algorithm terminates by recursively testing safety of each SCC.

In lines 6–11, the algorithm looks for a guarding potential $\pi$ for $R$. Note that for the sake of safety checking, we care about the values of $\pi$ only on the finite set of constants that actually appear in $R$; hence, we view $\pi$ as a finite function (with a finite representation). In lines 6–7, the algorithm tests whether $R$ has a positive-nonincreasing potential, and if so, it terminates by returning `true`. Correctness of this step follows from the fact that if $\pi$ is a positive

**Algorithm** IsSafe$(\mathbf{S}, R)$

1: **if** $\mathcal{X}(\mathbf{S}, R)$ has no positive cycles **then**
2: **return** `true`
3: let $U_1, \ldots, U_k$ be the SCCs of $\mathcal{X}(\mathbf{S}, R)$
4: **if** $k > 1$ **then**
5: **return** $\bigwedge_{i=1}^{k}$ IsSafe$(\mathbf{S}, \text{rules}(U_i))$
6: **if** $R$ has a positive-nonincreasing potential **then**
7: **return** `true`
8: **if** $R$ has a weakly decreasing potential **then**
9: $\pi \leftarrow$ a weakly decreasing potential for $R$
10: $R' \leftarrow$ all rules $E \Rightarrow F$ in $R$, such that $\pi(E) = \pi(F)$
11: **return** IsSafe$(\mathbf{S}, R')$
12: **return false**

**Figure 8: Algorithm for Safety Checking**

guarding potential for $R$, then $\pi$ is a positive guarding potential for every invocation cycle of $R$.

In lines 8–11 the algorithm considers the case where $R$ has a weakly decreasing potential $\pi$. In that case, let $R' \subseteq R$ be the set of rules $E \Rightarrow F$ with $\pi(E) = \pi(F)$, and let $R'' \subseteq R$ be the set of rules $E \Rightarrow F$ with $\pi(E) > \pi(F)$. Note that $R = R' \cup R''$. In Line 11 the algorithm returns the result of the recursive safety check on $(\mathbf{S}, R')$. Correctness of this step is due to the fact that $\pi$ is a guarding potential for every invocation cycle that contains one or more rules of $R''$, and hence, $(\mathbf{S}, R)$ is safe if and only if $(\mathbf{S}, R')$ is safe. For a later use, we record this conclusion as a proposition.

PROPOSITION 5.14. *Let $(\mathbf{S}, R)$ be a rewrite program, let $\pi$ be a potential that is weakly decreasing on $R$, and let $R'$ be the set of rules $E \Rightarrow F$ of $R$ with $\pi(E) = \pi(F)$. Then $(\mathbf{S}, R)$ is safe if and only if $(\mathbf{S}, R')$ is safe.*

Regarding the complexity of lines 6–11, the following lemma states that in polynomial time we can find a guarding potential or decide that none exists. The proof is by a straightforward translation of the problem of finding a guarding potential to that of finding a feasible solution to a linear program (LP), where each constant $c$ gives rise to a variable $X_c$ of the LP.[6]

LEMMA 5.15. *Deciding whether a set of rewrite rules has a guarding potential, and finding one if so, are in polynomial time.*

If the tests of lines 1, 4, 6, and 8 are all **false**, then $\mathcal{X}(\mathbf{S}, R)$ has a positive cycle (which means that $R$ has an expanding invocation cycle), $\mathcal{X}(\mathbf{S}, R)$ is strongly connected, and $R$ does not have a guarding potential. In this case, the algorithm terminates by returning **false** (declaring that $(\mathbf{S}, R)$ is unsafe). We now discuss the justification of this step.

If we reach Line 12, then $R$ has an expanding invocation cycle $p$, and $R$ does not have a guarding potential. But it may be the case that $p$ has a guarding potential, since $p$ can use only *some* of the rules of $R$. As an example, suppose that $R$ is the set that contains the replacement rules $X\ \mathtt{a}\ Y \Rightarrow X\ \mathtt{b}\ \mathtt{c}\ Y$ and $X\ \mathtt{b}\ \mathtt{c}\ \mathtt{b}\ \mathtt{c}\ Y \Rightarrow X\ \mathtt{a}\ \mathtt{a}\ \mathtt{a}\ Y$, which we denote by $\rho_1$ and $\rho_2$, respectively. The rule $\rho_1$ forms an expanding cycle, but it clearly has a guarding potential (e.g., $\pi(\mathtt{a}) = 3$ and $\pi(\mathtt{b}) = \pi(\mathtt{c}) = 1$). So, on the face of it, it may seem like Line 12 can be wrong in declaring that $(\mathbf{S}, R)$ is unsafe. Nevertheless, by using the fact that $\mathcal{X}(\mathbf{S}, R)$ is strongly connected (due to lines 4–5), we can show that $(\mathbf{S}, R)$ is necessarily unsafe. For instance, in the above example an expanding cycle that has no guarding potential is $\rho_1, \rho_1, \rho_2$. Actually, in this example not only is $(\mathbf{S}, R)$ unsafe, it is non-lfp-convergent, since as the reader can verify, $lfp(\mathtt{a}\ \mathtt{a})$ is infinite.

The next lemma verifies the correctness of the algorithm returning **false** (declaring that $(\mathbf{S}, R)$ is unsafe), if the tests of lines 1, 4, 6, and 8 are all **false** (i.e., $\mathcal{X}(\mathbf{S}, R)$ has a positive cycle, $\mathcal{X}(\mathbf{S}, R)$ is strongly connected, and $R$ does not have a guarding potential).

LEMMA 5.16. *Let $(\mathbf{S}, R)$ be a rewrite program, such that* (1) *$\mathcal{X}(\mathbf{S}, R)$ has a positive cycle,* (2) *$\mathcal{X}(\mathbf{S}, R)$ is strongly connected,* and (3) *$R$ has no guarding potential. Then $(\mathbf{S}, R)$ is unsafe.*

We get the following theorem, stating the correctness and efficiency of IsSafe$(\mathbf{S}, R)$, and the resulting tractability of safety.

THEOREM 5.17. IsSafe$(\mathbf{S}, R)$ *returns* `true` *if and only if the program $(\mathbf{S}, R)$ is safe, and it terminates in polynomial time. Thus, safety checking for a rewrite program is in polynomial time.*

## 5.4 Weak Safety

As we describe later, our safety condition very well captures lfp-convergence in our implemented search database system. But to properly handle rules of our past research [6], a weakened safety condition is needed. Specifically, in our previous paper [6], we applied a different kind of rule to a search query in order to obtain an *interpretation*, which is essentially a special type of an $\mathbf{S}$-hedge. Those rules are captured by a special type of rewrite rules, which we call *CFG* (*Context-Free Grammar*) *rules*.[7] Unfortunately, our safety condition does not capture lfp-convergence of programs that constitute such rules. But we can handle CFG rules by a slight relaxation of our safety condition, as we briefly describe next.

For a set $R$ of rewrite rules, we denote by $R{\downarrow}$ the set that is obtained from $R$ by removing from each rule every node but the leaves of the expressions. A potential $\pi$ is a *weakly guarding potential* for a set $R$ of rules if $\pi$ is a guarding potential for $R$, or it is the case that $R{\downarrow}$ is a rewrite program (i.e., in each rule, every variable occurs in the left side) and $\pi$ is a guarding potential for $R{\downarrow}$. Then, we relax our safety condition by replacing "guarding potential" with "weakly guarding potential." That is, the weaker safety condition is "every expanding invocation cycle has a weakly guarding potential." We can show that the weaker safety condition is correct (i.e., guarantees strong lfp-convergence) and tractable (i.e., can be decided in polynomial time). One can easily show that every rewrite program that consists of only CFG rules satisfies the weaker safety condition. Furthermore, the rewrite program $(\mathbf{S}, R)$ of our running example satisfies the weak safety condition as well (and hence, is strongly lfp-convergent).

## 5.5 Practical Evaluation

We considered a rewrite program that is used in a search database system. This system was developed at IBM Research–Almaden, and is used internally within IBM. The rewrite program has 380 rules, and it is blatantly non-lfp-convergent. For example, 22 rules $\rho$ are such that $\{\rho\}$ is non-lfp-convergent, like $X\ \mathtt{medical}\ Y \Rightarrow X\ \mathtt{medical\ plans}\ Y$. Therefore, instead of using the desired

[6] Linear programming has also been used, in a similar flavor, by Korovin and Voronkov [16] for a related problem of finding a special type of a *Knuth-Bendix order* [14] (for the sake of ordinary termination).

[7] To be precise, we note that [6] required each term of an $\mathbf{S}$-hedge to have a concept parent. We do not view this requirement as practically important, and we indeed avoid it here.

least-fixpoint semantics, in the past (before the research of this paper) each rule was applied at most once, in some arbitrary order.

Following the research of this paper, we essentially used the algorithm IsSafe in an extended way, where the goal was to extract a significant part of the program that is lfp-convergent, and mark the remaining part (which includes the above 22 rules) as needing modification.[8] So, in the end we were left with an lfp-convergent rewrite program that is obtained from $(\mathbf{S}, R)$ by removing from $R$ a subset $Z$ of marked rules.

More specifically, recall that IsSafe recursively partitions and reduces sets $R$ of rules (e.g., by applying Proposition 5.14), until none of the tests of lines 1, 4, 6, and 8 is true (hence, $\mathcal{X}(\mathbf{S}, R)$ has no positive cycles and is strongly connected, and $R$ has no guarding potential). For each such set $R$ (where none of the tests is true) we manually tested whether $R$ is lfp-convergent or not.[9] In all but two cases (that together consist of 13 rules, which are discussed below), we found that $R$ is indeed non-lfp-convergent, in which case we moved one or more selected rules from $R$ to $Z$, and continued.

At the end of the day, with the above process we extracted a safe program of 336 (out of 380) rules. Among the 44 remaining rules, 31 were in $Z$, marked as requiring modification, and we were left with an unsafe yet lfp-convergent program $(\mathbf{S}, R)$ such that $R$ contains only 13 rules.

Following is an example of a rule $\rho$ among the 13 rules, such that $\{\rho\}$ is unsafe, but is nevertheless strongly lfp-convergent.

$$X\,\texttt{publishing external}\,Y \Rightarrow$$
$$X\,\texttt{external web publishing process}\,Y$$

We handled this and the other 12 rules by extending the notion of a guarding potential from constants to strings (for example, we assign values to $\pi(\texttt{publishing external})$ rather than just to $\pi(\texttt{publishing})$ and $\pi(\texttt{external})$). A general discussion on this extension is beyond the scope of this paper.

Overall, we found our safety approach highly successful. In particular, we found that safety captures lfp- convergence very well in our system. Furthermore, it allowed us to extract a significant lfp-convergent part of our of program by eliminating problematic rules for justifiable reasons (they cause non-lfp-convergence).

# 6. CONCLUSIONS

We presented a framework that incorporates rewrite rules in a search database system. Specifically, a rewrite program is applied to a search query in a least-fixpoint manner. Our focus has been on lfp-convergence. Because lfp-convergence is undecidable, even under strong restrictions on the program, we presented a safety condition and showed that it guarantees lfp-convergence and that it is tractable. We also presented a weakened (less restrictive) version of the safety condition that better suits the CFG rules used in [6]. We believe that our safety condition (and the weakened one) well capture lfp-convergence in practice. As described in Section 5.5, we found this condition very successful when we applied it to a system being developed at IBM.

Several extensions of this work are desired to capture practical needs, and they will be the subject of future work. One such extension is the generalization of a guarding potential to strings and hedges (rather than just to constants), thereby broadening the notion of safety. Another extension will allow *regular expressions* in a rewrite rule, in order to qualify a restricted domain for a variable.

[8]There are simple ways of applying such modification, but a discussion on them is beyond the scope of this paper.

[9]To obtain small sets $R$, we used additional partitioning techniques that are not discussed in this paper.


# Acknowledgments

The authors are grateful to members of the PODS 2011 Program Committee and their subreferees, and to Balder ten Cate, for their invaluable help with understanding and referencing the literature of rewriting systems.


# 7. REFERENCES


[1] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

[2] J. Bear, D. J. Israel, J. Petit, and D. L. Martin. Using information extraction to improve document retrieval. In *TREC*, pages 367–377, 1997.

[3] A. B. Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Sci. Comput. Program.*, 9(2):137–159, 1987.

[4] N. Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

[5] N. Dershowitz. Termination. In *RTA*, volume 202 of *Lecture Notes in Computer Science*, pages 180–224. Springer, 1985.

[6] R. Fagin, B. Kimelfeld, Y. Li, S. Raghavan, and S. Vaithyanathan. Understanding queries in a search database system. In *PODS*, pages 273–284. ACM, 2010.

[7] O. Fissore, I. Gnaedig, and H. Kirchner. A proof of weak termination providing the right way to terminate. In *ICTAC*, volume 3407 of *Lecture Notes in Computer Science*, pages 356–371. Springer, 2004.

[8] J. V. Guttag, D. Kapur, and D. R. Musser. On proving uniform termination and restricted termination of rewriting systems. *SIAM J. Comput.*, 12(1):189–214, 1983.

[9] M. A. Hearst. Direction-based text interpretation as an information access refinement. In P. S. Jacobs, editor, *Text-Based Intelligent Systems: Current Research and Practice in Information Extraction and Retrieval*, pages 257–274. Erlbaum, Hillsdale, 1992.

[10] P. S. Jacobs. Introduction: Text power and intelligent systems. In P. S. Jacobs, editor, *Text-Based Intelligent Systems: Current Research and Practice in Information Extraction and Retrieval*, pages 1–8. Erlbaum, Hillsdale, 1992.

[11] F. Jacquemard and M. Rusinowitch. Closure of hedge-automata languages by hedge rewriting. In *RTA*, volume 5117 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2008.

[12] E. Kandogan, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar semantic search: a database approach to information retrieval. In *SIGMOD Conference*, pages 790–792. ACM, 2006.

[13] S. Kaplan. Conditional rewrite rules. *Theor. Comput. Sci.*, 33:175–193, 1984.

[14] D. Knuth and P. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[15] D. König. *Theorie der Endlichen und Unendlichen Graphen: Kombinatorische Topologie der Streckenkomplexe*. Akad. Verlag, Leipzig, 1936.

[16] K. Korovin and A. Voronkov. Orienting rewrite rules with the Knuth-Bendix order. *Inf. Comput.*, 183(2):165–186, 2003.

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

[18] D. S. Lankford. On proving term rewriting systems are Noetherian. Technical report, Mathematics Department, Louisiana Tech. University, Ruston, 1979.

[19] D. D. Lewis. Text representation for intelligent text retrieval: A classification-oriented view. In P. S. Jacobs, editor, *Text-Based Intelligent Systems: Current Research and Practice in Information Extraction and Retrieval*, pages 179–197. Erlbaum, Hillsdale, 1992.

[20] S. Lucas. Context-sensitive computations in confluent programs. In *PLILP*, volume 1140 of *Lecture Notes in Computer Science*, pages 408–422. Springer, 1996.

[21] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[22] B. Pang and L. Lee. Using very simple statistics for review search: An exploration. In *Proceedings of COLING: Companion volume: Posters*, pages 73–76, 2008.

[23] Y. Qiu and H.-P. Frei. Concept based query expansion. In *SIGIR*, pages 160–169. ACM, 1993.

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

[25] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

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