

# Automatic Information Extraction from Large Websites

VALTER CRESCENZI

*Università di Roma Tre*

AND

GIANSALVATORE MECCA

*Università della Basilicata*

**Abstract.** Information extraction from websites is nowadays a relevant problem, usually performed by software modules called wrappers. A key requirement is that the wrapper generation process should be automated to the largest extent, in order to allow for large-scale extraction tasks even in presence of changes in the underlying sites. So far, however, only semi-automatic proposals have appeared in the literature.

We present a novel approach to information extraction from websites, which reconciles recent proposals for supervised wrapper induction with the more traditional field of grammar inference. Grammar inference provides a promising theoretical framework for the study of unsupervised—that is, fully automatic—wrapper generation algorithms. However, due to some unrealistic assumptions on the input, these algorithms are not practically applicable to Web information extraction tasks.

The main contributions of the article stand in the definition of a class of regular languages, called the prefix mark-up languages, that abstract the structures usually found in HTML pages, and in the definition of a polynomial-time unsupervised learning algorithm for this class. The article shows that, differently from other known classes, prefix mark-up languages and the associated algorithm can be practically used for information extraction purposes.

A system based on the techniques described in the article has been implemented in a working prototype. We present some experimental results on known Websites, and discuss opportunities and limitations of the proposed approach.

Categories and Subject Descriptors: F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Classes defined by grammars or automata*; H.2.4 [**Database Management**]: Systems—*Relational databases*

General Terms: Theory, Algorithms

Additional Key Words and Phrases: Information extraction, relational model, wrappers

## 1. Introduction

We can consider the Web as the largest “knowledge base” ever developed and made available to the public. However HTML sites are in some sense modern

---

Corresponding address: G. Mecca, Dip. di Matematica, Università della Basilicata, c. da Macchia Romana, 85100 Potenza, Italy, e-mail: crescenz@dia.uniroma3.it; mecca@unibas.it.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© 2004 ACM 0004-5411/04/0900-0731 \$5.00

legacy systems, since such a large body of data cannot be easily accessed and manipulated. The reason is that Web data sources are intended to be browsed by humans, and not computed over by applications. XML, which was introduced to overcome some of the limitations of HTML, has been so far of little help in this respect. As a consequence, extracting data from Web pages and making it available to computer applications remains a complex and relevant task.

Data extraction from HTML is usually performed by software modules called *wrappers*. Early approaches to wrapping websites were based on manual techniques [Atzeni and Mecca 1997; Hammer et al. 1997; Sahuguet and Azavant 1999; Crescenzi and Mecca 1998; Huck et al. 1998]. A key problem with manually coded wrappers is that writing them is usually a difficult and labor intensive task, and that by their nature wrappers tend to be brittle and difficult to maintain. As a consequence, the key challenge in Web information extraction is the development of techniques that allow the automatization of the extraction process to the largest extent.

This article develops a novel approach to the data extraction problem: our goal is that of fully automating the wrapper generation process, in such a way that it does not rely on any a priori knowledge about the target pages and their contents. From this attempt come the main elements of originality with respect to other works in the field, as discussed in the following section.

1.1. BACKGROUND. A number of recent proposals [Adelberg 1998; Freitag 1998; Soderland 1999; Kushmerick et al. 1997; Muslea et al. 1999; Embley et al. 1999] have attacked the problem of information extraction from websites. These works have studied the problem of (semi-)automatically generating the wrappers for extracting data from fairly structured HTML pages. These systems are not completely automatic; in fact:

- they need a *training phase*, in which the system is fed with a number of labeled examples (i.e., pages that have been labelled by a human expert to mark the relevant pieces of information);
- the algorithms assume some a-priori knowledge about the organization of data in the target pages; typically, these approaches consider that pages contain a list of records; in some cases, nesting is allowed.

The fact that the process is not completely automatic is somehow unsatisfactory for a Web information extraction system. In fact, after a wrapper has been inferred for a bunch of target pages, a small change in the HTML code can lead to the wrapper failure and, as a consequence, the labelling and training phase has to be repeated. This typically requires a new human intervention, although some work [Kushmerick 2000b; Lerman and Minton 2000; Lerman and Knoblock 2003] has been recently done to study the problem of verifying and reinducing a disrupted wrapper.

One interesting feature of these systems is that they are essentially grammar inference systems. In fact, wrappers are usually parsers for (restricted forms of) regular grammars, and generating a wrapper in these proposals essentially amounts to inferring a regular grammar for the sample pages. In this respect, grammar inference techniques could in principle play a fundamental role in this field.

Since the late Sixties—that is, way before the Web itself—grammar inference for regular languages has been a thoroughly studied topic, with an elegant theoretical background and well established techniques. One of the main contributions of these

works is the study of properties of those languages for which the inference process can be performed in a completely automatic way, and of the relative algorithms. However, despite 30 years of research, very few of the recent approaches to Web information extraction reuse theories and techniques from the grammar inference community [Chidlovskii 2000; Hong and Clark 2001; Kosala et al. 2002]. This fact, which is due to some limitations of the traditional framework, has the consequence that, in addition to their semi-automatic nature, most of the techniques that have been proposed do not have a formal background in terms of expressive power and correctness of the inference algorithms.

1.2. THE GRAMMAR INFERENCE INHERITANCE. The reason for this stands in some early negative results in grammar inference. In fact, the seminal work by Gold (Gold's Theorem [Gold 1967]) shows that not all languages can be inferred from positive examples only. A language that can be inferred by looking at a finite number of positive examples only is said to be *identifiable in the limit* [Gold 1967]. To give an example, it follows from Gold's Theorem that even regular languages cannot be identified in the limit. As a consequence, the large body of research on inductive inference that originated from Gold's works has concentrated on the problem of finding restricted classes of regular grammars for which learning from positive data is possible. This research has led to the identification of several such classes [Angluin 1982; Radhakrishnan and Nagaraja 1987], which were proven to be identifiable in the limit, and for which unsupervised algorithms were developed and formally proven to be correct.

The main limitation of traditional grammar inference techniques when applied to modern information extraction problem is that none of these classes with their algorithms can be considered as a practical solution to the problem of extracting data from Web pages. This is due to the unrealistic assumptions on the samples that need to be presented to the algorithm in order to perform the inference. In fact, these algorithms assume that the learning system is presented with a *characteristic sample*, that is, a set of samples of the target language with specific features: (i) it has to be a "finite telltale" [Angluin 1980] of the language, that is, it has somehow to describe the language in all of its features; (ii) it has to be made of the strings of minimal length among those with property (i). These assumptions make it very unlikely that a wrapper can be correctly inferred by looking at a bunch of HTML pages that have been randomly sampled from a website.

In summary, we may find two kinds of proposals in the literature: (a) practical wrapper generation techniques for Web information extraction, which however have the limitation of being supervised, that is, inherently semi-automatic; (b) fully unsupervised techniques from the grammar inference community, which are in practice of little applicability in this context.

The ROADRUNNER project introduced in this article aims at reconciling these two research communities. To describe the main contribution of this work in one sentence, we may say that ROADRUNNER extends traditional grammar inference to make it practically applicable to modern Web information extraction, thus providing fully automatic techniques for wrapping real-life websites. To show this, the article first develops the formal framework upon which our approach is based, and the main theoretical results; then, it discusses how these techniques have been implemented in a working prototype and used to conduct a number of experiments on known websites.

## 2. Overview and Contributions

The target of this research are the so-called data-intensive websites, that is, HTML-based sites with large amounts of data and a fairly regular structure. Generating a wrapper for a set of HTML pages corresponds to inferring a grammar for the HTML code and then use this grammar to parse the page and extract pieces of data.

Pages in data-intensive sites are usually automatically generated: data are stored in a back-end DBMS, and HTML pages are produced using scripts—that is, programs—based on the content of the database. To give a simple but fairly faithful abstraction of the semantics of such scripts, we can consider the page-generation process as the result of two separated activities: (i) first, the execution of a number of queries on the underlying database to generate a *source dataset*, that is, a set of tuples of a possibly nested type that will be published in the site pages; (ii) second, the serialization of the source dataset into HTML code to produce the actual pages, possibly introducing URLs links, and other material like banners or images. We call a *class of pages* in a site a collection of pages that are generated by the same script.

We may reformulate the schema finding and data extraction process studied in this paper as follows: “*given a set of sample HTML pages belonging to the same class, find the nested type of the source dataset and extract the source dataset from which the pages have been generated*”. These ideas are clarified in Figure 1, which refers to a fictional bookstore site. In that example, pages listing all books by one author are generated by a script; the script first queries the database to produce a nested dataset (Figure 1(a)) by nesting books and their editions inside authors; then, it serializes the resulting tuples into HTML pages (Figure 1(b)). When run on these pages, our system will compare the HTML codes of the two pages, infer a common structure and a wrapper, and use that to extract the source dataset. Figure 1(c) shows the actual output of the system after it is run on the two HTML pages in the example. The dataset extracted is produced in HTML format. As an alternative, it could be stored in a database.

In the article, we formalize such *schema finding problem*, and develop a fully automatic algorithm to solve it. A key contribution stands in the definition of a new class of regular languages, called the *prefix mark-up languages*, which abstract the typical structures usually found in HTML pages. For this class of languages, we formally prove a number of results, as follows:

- We show that prefix mark-up languages are identifiable in the limit, that is, that there exist unsupervised algorithms for their inference from positive examples only.
- We show that prefix mark-up languages, differently from other classes previously known to be identifiable in the limit, require for the inference a new form of characteristic sample, called a *rich set*, which is statistically very interesting, since it has a high probability of being found in a bunch of randomly sampled HTML pages; it is worth noting that the notion of rich set is a database—theoretic notion, whereas the traditional notion of characteristic sample is essentially automata—theoretic.
- We develop a fully unsupervised algorithm for prefix mark-up languages, and prove its correctness; we also show that the algorithm has polynomial time complexity, and therefore represents a practical solution to the information extraction problem from websites.

a. Source Dataset

Name	Email	Books				
		Title	Descr.	Editions		
				Details	Year	Price
John Smith	smith@..	DB Primer	This book..	1st Ed., P.back	1998	20\$
		Computer S.	An undergrad...	2nd Ed., H. Cover	2000	30\$
				1st Ed., P.back	1995	40\$
Paul Jones	null	XML at..	A compr..	1st Ed., P.back	1999	30\$
		HTML..	A useful..	null	1993	30\$
				2nd Ed., H. Cover	1999	45\$
		JavaScript	A must in..	null	2000	50\$
...	...	...	...	...	...	...

b. HTML Pages



c. Data Extraction Output

Total number of SCHEMAS found: 1  
 Schema Number 1: A(B)?(C(D)(E)F)?G? Total Time: 0' 180 ms

A	B	C	D	E	F	G
John Smith	smith@dot.com	Database Primer	First Edition, Paperback	1999	\$20	This book introduces the reader to the theory and technology of database systems. Its main topics are the relational model and the SQL query language ...
		Computer Systems	First Edition, Paperback	1995	\$40	An undergraduate level textbook on computer architectures. It starts from the Von Neumann architecture and proceeds on to parallel architectures. ...
Paul Jones	null	XML at Work	First Edition, Paperback	1999	\$30	A comprehensive description of XML, and all related standards ...
		HTML and Scripts	Second Edition, Hard Cover	1999	\$45	A useful HTML, handbook, with a tutorial on the use of scripts for the generation of page on the ...
		JavaScripts	null	2000	\$50	A must in every Webmaster's bookshelf ...

FIG. 1. Examples of HTML code generation.

Finally, we discuss how a system based on this framework has been implemented in a working prototype and used to conduct several experiments on websites.

One final comment is related to the schema that the system is able to infer from the HTML pages. As it can be seen from the Figure 1, the system infers a nested schema from the pages. Since the original database field names are generally not encoded in the pages and this schema is based purely on the content of the HTML code, it has anonymous fields (labeled by A, B, C, D, etc. in our example), which must be named manually after the dataset has been extracted; one intriguing alternative is to try some form of post-processing of the wrapper to automatically discover attribute names. We have developed a number of techniques that in many cases are able to guess a name based on the presence of particular strings in the pages (like “Our Price” or “Book Description”) [Arlotta et al. 2003]. However, this is a separate research problem; for space and simplicity reasons we don’t deal with it in this work.

The article is organized as follows: Section 3 introduces some preliminary definitions. The *Schema Finding Problem* is formalized in Section 4, along with the definition of the class of *prefix mark-up languages*. Connections with traditional

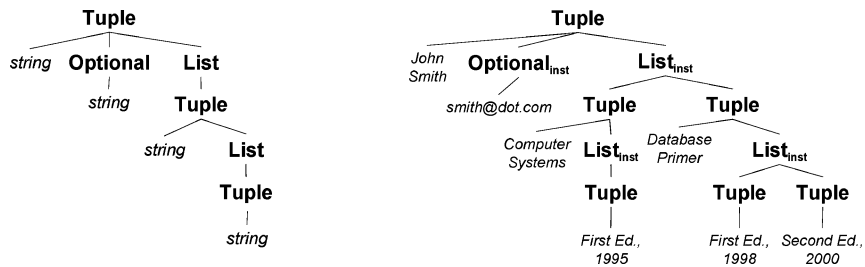


FIG. 2. Running example.

grammar inference are discussed in Section 5, where we also show that prefix mark-up languages are identifiable in the limit. The following Sections are devoted to the description of the inference algorithm for prefix mark-up languages; more specifically: we first give an informal, example-based description of the algorithm in Section 6; then, we formalize the algorithm in Section 7. Section 8 is devoted to the correctness and complexity results. Implementation and experiments are presented in Section 9. Section 10 is devoted to the final discussion and to conclusions. Related work is in Section 11.

### 3. Preliminary Definitions

This section introduces a number of preliminary definitions used throughout the article.

*Abstract Data Types and Instances.* The abstract data we consider are nested relations [Hull 1988], that is typed objects allowing nested collections and tuples. Tuples may have optional attributes. Since our algorithm works on instances serialized as HTML pages, we will only consider ordered collections, that is, (possibly nested) lists instead of sets.

The types are defined by assuming the existence of an atomic type  $U$ , called the *basic type*, whose domain, denoted by  $dom(U)$ , is a countable set of constants. There exists a distinguished constant  $null$ , and we will say that a type is *nullable* if  $null$  is part of its domain. Other nonatomic *types* (and their respective domains) can be recursively defined as follows: (i) if  $T_1, \dots, T_n$  are basic, optional or list types, amongst which at least one is not nullable, then  $[T_1, \dots, T_n]$  is a *tuple* type, with domain  $dom([T_1, \dots, T_n]) = \{[a_1, \dots, a_n] \mid a_i \in dom(T_i)\}$ , (ii) if  $T$  is a tuple type, then  $\langle T \rangle$  is a *list* type, with domain corresponding to the collection of finite lists of elements of  $dom(T)$ , (iii) if  $T$  is a basic or list type, then  $(T)?$  is an *optional* type, with domain  $dom(U) \cup \{null\}$ .

In the article, we shall use as a running example a simplified version of the books Web site discussed in Section 2. In this version, data about each author in the database is a nested tuple type, with a name, an optional email, and a list of books; for each book, the title and a nested list of editions is reported. Figure 2 shows a tree-based representation [Hull 1988] of the nested type and of one of its instances. In the instance tree, type nodes are replaced by type instance nodes; these are marked by subscripts.

*Regular Expressions and Abstract Syntax Trees.* We denote the length of a string  $s$  by  $|s|$ . Our approach is based on a close correspondence between nested

```

<HTML>
<IMG/> <B>John Smith</B>
<A><TT>smith@dot.com</TT></A>
<UL>
  <LI> <IMG/> <I>Computer Systems</I>
    <P>
      <B><BR/>First Ed., 1995<IMG/></B>
    </P>
  </LI>
  <LI> <IMG/> <I>Database Primer</I>
    <P>
      <B><BR/>First Ed., 1998<IMG/></B>
      <B><BR/>Second Ed., 2000<IMG/></B>
    </P>
  </LI>
</UL>
</HTML>

<HTML>
<IMG/> <B>Paul Jones</B>
<A></A>
<UL>
  <LI> <IMG/> <I>XML at Work</I>
    <P>
      <B><BR/>First Ed., 1999<IMG/></B>
    </P>
  </LI>
  <LI> <IMG/> <I>HTML and Scripts</I>
    <P>
      <B><BR/>First Ed., 2002<IMG/></B>
    </P>
  </LI>
  <LI> <IMG/> <I>JavaScript</I>
    <P>
      <B><BR/>First Ed., 2001<IMG/></B>
    </P>
  </LI>
</UL>
</HTML>

```

```

<HTML>
<IMG/> <B>#PCDATA</B>
<A> (<TT>#PCDATA</TT>)? </A>
<UL> (<LI> <IMG/>
  <I>#PCDATA</I>
  <P>
    (<B><BR/>#PCDATA<IMG/></B>)+
  </P>
  </LI> )+
</UL>
</HTML>

```

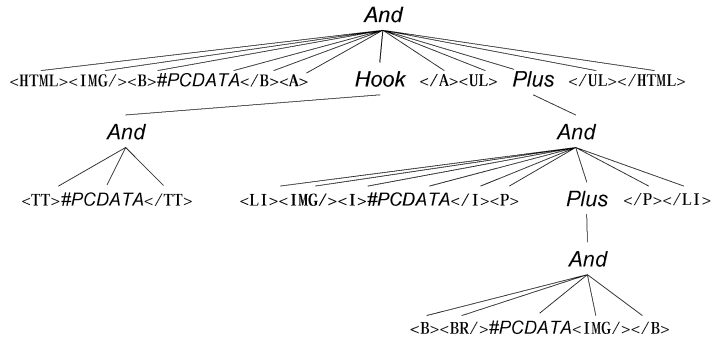


FIG. 3. Regular grammar for the running example.

types and *union-free regular expressions*. Given a special symbol #PCDATA, and an alphabet of symbols  $T$  not containing #PCDATA, a *union-free regular expression (UFRE)* over  $T$  is a string over alphabet  $T \cup \{\#PCDATA, \cdot, +, ?, (, )\}$  defined as follows: First, the empty string,  $\epsilon$  and all elements of  $T \cup \{\#PCDATA\}$  are union-free regular expressions. If  $a$  and  $b$  are UFRE, then  $a \cdot b$ ,  $(a)^+$ , and  $(a)^?$  are UFRE. The semantics of these expressions is defined as usual,  $+$  being an iterator and  $(a)^?$  being a shortcut for  $(a|\epsilon)$  (denotes optional patterns). As usual, we denote by  $L(exp)$  the language specified by the regular expression  $exp$ . With reference to our running example, Figure 3 shows a couple of HTML source strings for pages in the site, and the corresponding regular grammar.<sup>1</sup>

<sup>1</sup>In the figure, `<img/>` is a shortcut for `<img></img>`; similarly for `<br/>`.

Throughout the paper, we will often represent regular expressions by means of *Abstract Syntax Trees* (AST). Given a UFRE, its AST representation can be recursively built as follows: (i) an optional expression  $(r)?$  corresponds to a tree rooted at an *Hook* node with one subtree corresponding to  $r$ ; (ii) an iterator expression  $(r)^+$  corresponds to a tree rooted at an *Plus* node with one subtree corresponding to  $r$ ; (iii) a concatenation  $r_1 \cdot \dots \cdot r_n$  corresponds to a tree rooted at an *And* node with  $n$  subtrees, one for each  $r_i$ .<sup>2</sup> Figure 3 also shows the AST associated with the regular grammar. Throughout the article, HTML sources are considered as strings of tokens, where each token is either an HTML tag or a string. In the following, the distinction between a regular expression and its AST representation is blurred whenever it is irrelevant.

#### 4. Problem Formalization

In this section, we formalize our problem by setting a theoretical framework<sup>3</sup> based on abstract data types as defined in Section 3. We describe formal methods to encode types and instances into sequences of characters. We state the fundamental problem of recovering a type starting from a set of its instances and we show that in our framework this is related to inferring a regular grammar starting from a set of positive samples produced by encoding functions.

A key step in this process is the definition of a class of union free regular languages, called *prefix mark-up languages*, for which we show that the inference problem can be solved efficiently. However, in order to give the definition of prefix mark-up languages, we need to introduce the useful notion of *templates*, that is, partially defined types that generalize types and their instances. Prefix mark-up languages will be then defined as those languages based on encodings of templates.

4.1. TEMPLATES. Templates are built by mixing types—lists, tuples, optionals and basic—and instances—that is, list instances, optional instances, and constants. Templates allows us to generalize the existing relationship “*instance of*” between an instance and its type by means of a reflexive *subsumption* relationship between templates, which we denote by  $\preceq$ . As an example of templates, consider  $T = [c, \langle [U] \rangle]$ :  $T$  is a template that subsumes any tuple of two attributes, the first one being a ‘ $c$ ’, that is, a constant string, and the second one any list of monadic tuples. Another example is  $T' = [U, \langle [a, U], [a, U], [a, U] \rangle]$ , where  $\langle i \dots \rangle$  denote a list instance:  $T'$  subsumes any tuple of two attributes, the second one being a list of exactly three binary tuples, all having ‘ $a$ ’ as a first attribute. A type  $\sigma = [U, \langle [U] \rangle]$  and an instance  $I = [c, \langle [a] \rangle]$  are also templates, and we have that  $I \preceq T \preceq \sigma$ .

Templates and the subsumption relation can be formally defined as follows: basic, list, tuple and optional templates reflect the corresponding definitions previously given for types; a number of new definitions are needed for partially specified cases.

<sup>2</sup>It is worth noting that since the concatenation of expressions is an associative operator, one could end up with trees which differ only for the nesting and the arity of *And* nodes. In the following, these ambiguities are removed by only considering trees with the minimum number of *And* nodes such that all *Token* nodes have got an *And* node as parent. Considering for example  $(a(b)?c)^+$ , the chosen representation would be  $Plus(And(a, Hook(And(b)), c))$ .

<sup>3</sup>The problem formalized in this article is an extension of that defined in Grumbach and Mecca [1999].

*Definition 4.1 (Templates)*

- Every element  $u \in \text{dom}(U)$  is a nonnullable *constant template*; *null* is a *null-template*; it is nullable;
- $U$ , the basic type, is a *basic template*; it is not nullable;
- if  $T_1, \dots, T_n$  are nontuple templates, amongst which at least one is nonnullable, then  $[T_1, \dots, T_n]$  is a *tuple template*; it is not nullable;
- if  $T$  is a tuple template, then  $\langle T \rangle$  is a *list template*; it is not nullable;
- if  $T$  is either basic, constant, list or list-instance template, then  $(T)?$  is an *optional template*; it is nullable;
- if  $T_1, \dots, T_k$  are tuple templates, then  $T = \langle_i T_1, \dots, T_k \rangle$  is a *list-instance template*;  $k \geq 1$  is called the *cardinality* of  $T$ ; it is not nullable;
- if  $T$  is either basic, constant, list or list-instance template, then  $(_i T)?$  is an *optional-instance template*; it is not nullable.

It is easy to see that: (i) every type is a template, constructed using only tuple, list, optional and basic subtemplates; (ii) every instance of a type is itself a template, made of tuple, list-instance, optional-instance, constant and *null* subtemplates. Note that the tree representation of types and instances extends immediately to templates. In the following, we blur the distinction among types, instances and the corresponding templates.

*Definition 4.2 (Subsumption)*

- Every constant and null template subsumes itself;
- The basic template  $U$  subsumes every constant template;
- A tuple template,  $[T_1, \dots, T_n]$  subsumes any tuple template in  $\{[t_1, \dots, t_n] \mid t_i \preceq T_i\}$ ;
- A list template,  $\langle T \rangle$ , subsumes any list template  $\langle S \rangle$  such that  $S \preceq T$ , and any list-instance template  $\langle_i T_1, \dots, T_m \rangle$  such that  $T_j \preceq T, j = 1, \dots, m$ ;
- An optional template  $(T)?$  subsumes the *null-template*, any optional template  $(S)?$  such that  $S \preceq T$ , and any optional-instance template in  $\{(_i t)? \mid t \preceq T\}$ ;
- a list-instance template  $T = \langle_i T_1, \dots, T_n \rangle$  subsumes any template in the set  $\{\langle_i t_1, \dots, t_n \rangle \mid t_j \preceq T_j, j = 1 \dots n\}$ ;
- an optional-instance template  $(_i T)?$  subsumes any optional-instance template in  $\{(_i t)? \mid t \preceq T\}$ ;

Figure 4 shows an example of a template. The template in Figure 4 is subsumed by the type and subsumes the instance shown in Figure 2. It represents books by an author whose name is fixed (John Smith), which has an optional email, and exactly two books; the first book has only one edition, published in 1995.

We denote by  $\mathcal{T}$  the universe of all templates. The relation  $\preceq$  defines a partial order on the set  $\mathcal{T}$ . We say that two templates  $T_1, T_2$  are *homogeneous* if they admit a common ancestor, that is, there exist a template  $T \in \mathcal{T}$  such that  $T_1 \preceq T$  and  $T_2 \preceq T$ . Intuitively, two templates are homogeneous if they represent objects that are subsumed by the same type.

4.2. RICHNESS OF A COLLECTION OF INSTANCES. We introduce a labeling system of template trees to identify the nodes. It is recursively defined as follows: The

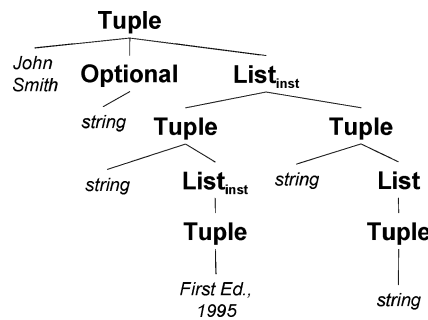


FIG. 4. Example of a template.

root is labeled by the string *root*. If a list node or an optional node is labeled  $\alpha$ , then its child is labeled  $\alpha.0$ ; and if a tuple node with  $n$  children is labeled  $\alpha$ , then its children are labeled  $\alpha.1, \dots, \alpha.n$ . Instances are labeled similarly, with the children of a list, list-instance or optional-instance node labeled  $\alpha$ , all labeled  $\alpha.0$ . In this way, each node in an instance tree has the same label as the corresponding node in the type tree. For example, the two nodes “Computer Systems” and “Database Primer” in the instance in Figure 2 are both labeled by the label *root.3.0.1*, which is also the label of the corresponding *string* node in the type tree.

Instances can sometimes underuse their type. An instance underuses its type for example when list types are used to model list that always have the same cardinality (and would have been more accurately typed with a tuple), or when some basic attribute always has the same value (and could have been omitted), and finally when an optional attribute is either never or always *null* (either could have been considered nonoptional or omitted). The concept of *rich* collection of instances defines this notion formally.

*Definition 4.3 (Rich Collection of Instances).* A collection of instances  $I$  is *rich* with respect to a common type  $\sigma$  if it satisfies the following three properties: (i) *basic richness*: for each leaf node labeled  $\alpha$ , there are at least two distinct objects labeled  $\alpha$ ; (ii) *list richness*: for each label  $\alpha$  of a list node, there are at least two lists of distinct cardinalities labeled  $\alpha$ , and (iii) *optional richness*: for each label  $\alpha$  of an optional node, there is at least one *null* and one *non-null* object labeled  $\alpha.0$ .

Intuitively, a collection of instances is rich with respect to a common type, if it makes full use of it, and we will see in the sequel that it contains enough information to recover the type.

**4.3. WELL-FORMED MARK-UP ENCODINGS OF ABSTRACT DATA.** To produce HTML pages, abstract data needs to be encoded into strings, that is, concrete representations. Our concrete representations are simply strings of symbols over finite alphabets, and therefore this concept can be formalized by introducing the notion of *encoding*, that is a function *enc* from the set of all abstract instances to strings over those alphabets.

We introduce the *well-formed mark-up encodings*, which abstract mark-up based languages like HTML or XML—which are used in practice to encode information on many data-intensive websites. To reflect the distinction between tags—which are intuitively used to encode the structure—and strings—which encode the data—the encoding uses two different alphabets. Essentially, the mark-up encodings map

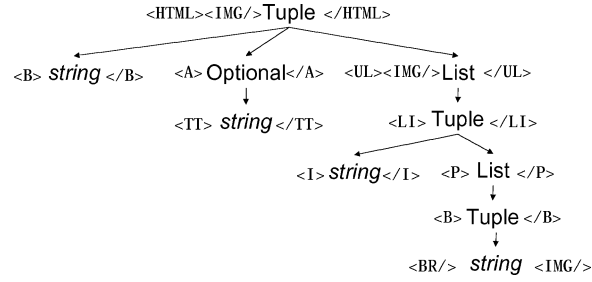


FIG. 5. A sample mark-up encoding.

database objects into strings in which the constants are encoded as strings over a data alphabet,  $\Delta$ , and the structure is encoded by a schema alphabet, made of tags. The schema alphabet will contain a closing tag  $\langle /a \rangle$  for each opening tag  $\langle a \rangle$ .

*Definition 4.4 (Alphabets).* We fix a *data alphabet*  $\Delta$  and a *schema alphabet*,  $\Sigma \cup \bar{\Sigma}$ , with  $\bar{\Sigma} = \{ \langle /a \rangle \mid \langle a \rangle \in \Sigma \}$ ,

In the encoding process, we first need to formalize how abstract data items in the instances are encoded as strings. To this end, we need a *data encoding function*,  $\delta$ , that is a 1-1 mapping from  $\text{dom}(U)$  to  $\Delta^+$  according to which constants are encoded as words in  $\Delta^+$ . However, in the following, for the sake of simplicity, we will assume that  $\text{dom}(U)$  is a set of strings over  $\Delta$ , that is,  $\text{dom}(U) \subseteq \Delta^+$ , and that  $\delta$  is the identity function; we will therefore omit to mention  $\delta$  explicitly.

Second, we need to describe how the schema structure is encoded using tags. For this, we introduce a *tagging function*,  $\text{tag}$ , that works on a type tree and essentially associates a pair of delimiters with each node; these delimiters will then be used to serialize instances of the type. This process is illustrated in Figure 5.

We require that encodings produced by these functions are *well formed*, that is, tags are properly nested and balanced so that every occurrence of a symbol  $\langle a \rangle$  in  $\Sigma$  is “closed” by a corresponding occurrence of a symbol  $\langle /a \rangle$  in  $\bar{\Sigma}$ . For instance, if  $\Sigma = \{ \langle a \rangle, \langle b \rangle, \langle c \rangle \}$  and  $\Delta = \{ 0, 1 \}$ , these are well-formed strings:

$\langle a \rangle \langle b \rangle 11 \langle /b \rangle \langle /a \rangle$ ,  $\langle a \rangle \langle b \rangle \langle /b \rangle 100 \langle c \rangle \langle /c \rangle \langle /a \rangle$

while these are not:

$\langle a \rangle \langle b \rangle 0 \langle /a \rangle \langle /b \rangle$ ,  $\langle a \rangle \langle b \rangle \langle /c \rangle 0 \langle /b \rangle \langle /a \rangle$ ,  $\langle a \rangle \langle b \rangle \langle c \rangle \langle /c \rangle 01 \langle /a \rangle$ .

We formalize this concept by saying that well-formed strings need to belong to the language defined by a context-free grammar  $G_{\text{tag}}$ . Let  $\Delta$  denote a place holder for data encodings and  $S$  be the starting nonterminal symbol. The productions of the grammar are as follows:

$$\begin{aligned} S &\rightarrow aX_D\bar{a} \mid XX_DX \\ X &\rightarrow a\bar{a} \mid aX\bar{a} \mid XX && \text{(for all } a \in \Sigma). \\ X_D &\rightarrow aX_D\bar{a} \mid XX_D \mid X_DX \mid \Delta \end{aligned}$$

We are now ready to formalize the notion of *well-formed tagging function* and that of *well-formed mark-up encoding*.

*Definition 4.5 (Well-Formed Tagging Function).* Given a type  $\sigma$ , and the corresponding labeled tree  $T_\sigma$ , let  $\mathcal{L}$  denote the set of labels in  $T_\sigma$ . A *well-formed*

*tagging function* for the type  $\sigma$  associates with each label  $\alpha \in \mathcal{L}$  two strings over the schema alphabet, called  $start(\alpha)$  and  $end(\alpha)$ , as follows:

$$\begin{aligned} tag : \mathcal{L} &\rightarrow (\Sigma \cup \overline{\Sigma})^+ \times (\Sigma \cup \overline{\Sigma})^+ \\ \alpha &\mapsto (start(\alpha), end(\alpha)) \text{ such that } start(\alpha) \cdot \Delta \cdot end(\alpha) \in L(G_{tag}). \end{aligned}$$

*Definition 4.6 (Well-Formed Mark-Up Encodings).* Given a type  $\sigma$  a *well-formed mark-up encoding*  $enc$  based on a structure  $(\Sigma, \Delta, tag)$ —where  $\Sigma$  and  $\Delta$  are two disjoint finite alphabets, and  $tag$  a well-formed tagging function for  $\sigma$ —is a function recursively defined on the tree of any template  $T$  subsumed by  $\sigma$ , as follows:

- for a constant leaf node  $a \in dom(U)$  with label  $\alpha$ ,  $enc(a) = start(\alpha) \cdot a \cdot end(\alpha)$ ;<sup>4</sup>
- for a *null* template with label  $\alpha$ ,  $enc(null) = start(\alpha)end(\alpha)$ ;
- for a list-instance node  $\langle_i a_1, \dots, a_n \rangle$  with label  $\alpha$ ,  $enc(\langle_i a_1, \dots, a_n \rangle) = start(\alpha) \cdot enc(a_1) \cdot \dots \cdot enc(a_n) \cdot end(\alpha)$ ;
- for an optional-instance node  $\langle_i a \rangle?$  with label  $\alpha$ ,  $enc(\langle_i a \rangle?) = start(\alpha) \cdot enc(a) \cdot end(\alpha)$ ;
- for a basic leaf node  $U$  labeled  $\alpha$ ,  $enc(U) = start(\alpha) \cdot \Delta^+ \cdot end(\alpha)$ ;
- for an optional node  $\langle T \rangle?$  with label  $\alpha$ ,  $enc(\langle T \rangle?) = start(\alpha) \cdot (enc(T))? \cdot end(\alpha)$ ;
- for a tuple node  $[T_1, \dots, T_n]$  with label  $\alpha$ ,  $enc([T_1, \dots, T_n]) = start(\alpha) \cdot enc(T_1) \cdot \dots \cdot enc(T_n) \cdot end(\alpha)$ ;
- for a list node  $\langle T \rangle$  with label  $\alpha$ ,  $enc(\langle T \rangle) = start(\alpha) \cdot (enc(T))^+ \cdot end(\alpha)$ .

For example, the encoding shown in Figure 5 for the type  $\sigma$  in Figure 2 produces the HTML code in Figure 3.

Note that the notion of well-formed mark-up encoding is defined for templates, and therefore also for types and for instances. It can be seen that a well-formed mark-up encoding  $enc$  applied to all instances of a type  $\sigma$  generates a language of strings. It is also easy to see that these languages are regular languages, and that they belong to the language defined by the regular expression  $enc(\sigma)$  obtained by applying  $enc$  to  $\sigma$ . The following Proposition summarizes a close correspondence between the theoretical framework based on data and encoding functions, and the theory of regular languages.

**PROPOSITION 4.7.** *Given a mark-up encoding  $enc$  based on  $(\Sigma, \Delta, tag)$ , then,  $enc(\sigma)$  is a regular language and for each instance  $I$  of type  $\sigma$ ,  $enc(I) \in L(enc(\sigma))$ .*

We have therefore identified a subset of regular languages, which we call *well-formed mark-up languages*, that is, those obtained by applying well-formed mark-up functions to templates.

*Definition 4.8 (Well-Formed Mark-Up Language).* Any regular language  $enc(T)$  obtained by applying a well-formed mark-up encoding function  $enc$  to a template  $T$ .

**4.4. SCHEMA FINDING PROBLEM.** Now we can define formally the problem we are interested in. Intuitively, our problem takes as input a set of encoded instances

<sup>4</sup>Recall that we assume that  $dom(U) \subseteq \Delta^+$ , and therefore that leafs are encoded as themselves.

of a type  $\sigma$ , and tries to recover the original type by finding the encoding function. In order to have representative inputs, we assume that the input is rich with respect to the type.

*Definition 4.9 (Schema Finding Problem for Mark-Up Encodings)*

*Input.*  $\Sigma$ ,  $\Delta$  as defined above, and a finite collection  $W$  of strings of  $(\Sigma \cup \bar{\Sigma} \cup \Delta)^*$  which are the encodings of a rich set of instances of a type  $\sigma$ .

*Output.* The type  $\sigma$ , an encoding function  $enc$  and a finite collection  $C$  of instances of type  $\sigma$ , such that  $enc(C) = W$ .

Unfortunately, the following example shows that the schema finding problem for mark-up encodings does not admit a unique solution in general. Consider first a type  $\sigma = \langle [U] \rangle$  and a set of instances of  $\sigma$ :  $I_1 = \langle_i [0], [1] \rangle$  and  $I_2 = \langle_i [0], [1], [2], [3] \rangle$ . Suppose we fix the following encoding:

$$enc(\sigma) = \langle z \rangle (\langle a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle \Delta^+ \langle b \rangle \langle /b \rangle \langle a \rangle \langle /a \rangle)^+ \langle /z \rangle.$$

Then consider the other type  $\sigma' = \langle [U, U] \rangle$ , a rich set of instances of  $\sigma'$ :  $J_1 = \langle_i [0, 1] \rangle$ ,  $J_2 = \langle_i [0, 1], [2, 3] \rangle$ , and the following encoding  $enc'$ :

$$enc(\sigma') = \langle z \rangle (\langle a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle \Delta^+ \langle b \rangle \langle /b \rangle \langle a \rangle \langle /a \rangle \langle a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle \Delta^+ \langle b \rangle \langle /b \rangle \langle a \rangle \langle /a \rangle)^+ \langle /z \rangle.$$

Observe that  $\mathcal{I} = \{I_1, I_2\}$  and  $\mathcal{J} = \{J_1, J_2\}$  are both rich collections of instances of  $\sigma$  and  $\sigma'$ , respectively. However,  $enc(\mathcal{I}) = enc'(\mathcal{J})$ . In fact, it can be seen that:

$$\begin{aligned} enc(I_1) = enc'(J_1) &= \langle z \rangle \langle a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle 0 \langle b \rangle \langle /b \rangle \langle a \rangle \langle /a \rangle \\ &\quad \langle a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle 1 \langle b \rangle \langle /b \rangle \langle a \rangle \langle /a \rangle \langle /z \rangle \\ enc(I_2) = enc'(J_2) &= \langle z \rangle \langle a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle 0 \langle b \rangle \langle /b \rangle \langle a \rangle \langle /a \rangle \\ &\quad \langle a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle 1 \langle b \rangle \langle /b \rangle \langle a \rangle \langle /a \rangle \\ &\quad \langle a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle 2 \langle b \rangle \langle /b \rangle \langle a \rangle \langle /a \rangle \\ &\quad \langle a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle 3 \langle b \rangle \langle /b \rangle \langle a \rangle \langle /a \rangle \langle /z \rangle. \end{aligned}$$

This shows that the problem stated above may in general admit multiple solutions. Intuitively, this is due to the fact that the encoding functions may be ambiguous. To avoid this problem, we further restrict the class of encodings allowed. Our goal is to avoid the ambiguousness of delimiters from which multiple solutions may be derived. We define a special subclass of mark-up encodings called *prefix mark-up encodings* which force delimiters of list and optional nodes to be somehow identifiable.

*Definition 4.10 (Prefix Mark-Up Encodings).* A *prefix mark-up encoding* is a *well-formed mark-up encoding* based on a structure  $(\Sigma, \Delta, tag)$  where the tagging function  $tag$  satisfy the following additional conditions:

- wrapping delimiters.* All delimiters of nonleaf nodes are such that there is at least one symbol of  $\Sigma$  in the start delimiter that is closed by a symbol of  $\bar{\Sigma}$  in the end delimiter;
- point of choice delimiters.* Symbols of delimiters which mark optional and list nodes do not occur inside delimiters of their child node.

The two tagging functions shown in the counterexample of the previous section contradict these conditions. Based on this definition, we can identify a new class of

regular languages, a proper subset of the class of well-formed mark-up languages defined above, which we call the *prefix mark-up languages*.

*Definition 4.11 (Prefix Mark-Up Languages).* Any regular language  $enc(T)$  obtained by applying a prefix mark-up encoding function  $enc$  to a template  $T$ .

We can now formally define the *Schema Finding Problem*.

*Definition 4.12 (Schema Finding Problem)*

*Input.*  $\Sigma$ ,  $\Delta$  as defined above, and a finite collection  $W$  of strings of  $(\Sigma \cup \bar{\Sigma} \cup \Delta)^*$  which are prefix mark-up encodings of a rich set of instances of a type  $\sigma$ .

*Output.* The type  $\sigma$ , a prefix mark-up encoding function  $enc$  and a finite collection  $C$  of instances of type  $\sigma$ , such that  $enc(C) = W$ .

### 5. Schema Finding as a Grammar Inference Problem

The schema finding problem introduced in the previous section is essentially a (regular) grammar inference problem. In fact, it is possible to see that:

**PROPOSITION 5.1.** *Given a nested tuple type  $\sigma$  and a prefix mark-up encoding  $enc$ , it is possible to derive  $\sigma$  from  $enc(\sigma)$  in linear time.*

As a consequence, given a set of encoded instances, the problem amounts to finding the regular expression which encodes the corresponding common type; from that, we can easily recover the type and the original instances.

Grammar inference is a well known and extensively studied problem (for a survey of the literature see, e.g., Pitt [1989]). Gold gave a simple and widely accepted model of inductive learning also called “learning from text” which goes on as follows [Gold 1967]: (a) consider a target language  $L$  we want to infer, for example a regular language like  $a(bc)^+(d)?$ ; (b) assume a learner is given a (possibly infinite) sequence of positive samples of  $L$ , that is, a sequence of strings belonging to  $L$ , like, for example  $abcbcd$ ,  $abcd$ ,  $abcbbc$  . . .; (c) after each sample, the learner produces as output a new guess on the target language. Intuitively, each new sample adds new knowledge about the language, and therefore can help in identifying the solution. A class of languages is *identifiable in the limit* if, for each language  $L$  in the class, the learner converges towards the right hypothesis after a finite number of positive examples.

Unfortunately, not all languages are inferrable in the limit. Gold himself produced the first negative results on the inductive inference of grammars (Gold’s Theorem [Gold 1967]). To recall the theorem, let us fix an alphabet  $T$ . We call a *superfinite class of languages* over  $T$  any class of languages that contains all finite languages over  $T$  plus at least one infinite language. The theorem says that a superfinite class of languages cannot be identified in the limit from positive examples alone. To give an example, from his theorem it follows that even regular languages cannot be identified in the limit. As a consequence, the large body of research on inductive inference that originated from Gold’s seminal works has concentrated on the problem of finding restricted classes of regular grammars for which learning from positive data is possible.

In the early ’80s Angluin has posed several milestones on this subject by finding necessary and sufficient conditions for a class of languages to be inferrable from positive examples [Angluin 1980] based on the fundamental notion of a *characteristic*

*sample*. According to this theorem, a language class  $\mathcal{L}$  is inferrable if and only if every language  $L \in \mathcal{L}$  has a characteristic sample. Intuitively, the characteristic sample of a language  $L$  is a sort of finite *fingerprint* that discriminates  $L$  from any other language of the class.

*Definition 5.2 (Characteristic Sample).* A characteristic sample of a language  $L$  in a class  $\mathcal{L}$  is a subset  $\chi(L) \subseteq L$  such that for every other language  $L' \in \mathcal{L}$  such that  $\chi(L) \subseteq L'$ ,  $L'$  is not a proper subset of  $L$ , that is,  $L$  is the minimal language from  $\mathcal{L}$  containing  $\chi(L)$ .

Based on her work, subsequent works introduced several classes of languages identifiable in the limit; prominent examples are the class of reversible grammars [Angluin 1982] and the class of terminal distinguishable languages [Radhakrishnan and Nagaraja 1987], which were proven to be identifiable in the limit, and for which unsupervised algorithms were developed and formally proven to be correct.

Recently, Fernau [2003] has introduced the notion of *f-distinguishable language* to generalize Angluin's work [Angluin 1982]. *f-distinguishable languages* are identifiable in the limit from positive examples only and generalize many previously known classes, including reversible languages and terminal distinguishable languages. The classes of languages in that family are parametric with respect to a given *distinguishing function*.

*Definition 5.3 (f-Distinguishing Function [Fernau 2003, Definition 1]).* Let  $F$  be a finite set. A mapping  $f : T^* \rightarrow F$  is called a *distinguishing function* if  $f(w) = f(z)$  implies  $f(wu) = f(zu)$  for all  $u, w, z \in T^*$ .

A language is called *f-distinguishable* if it is recognized by a *f-distinguishable automaton*:

*Definition 5.4 (f-Distinguishable Automaton [Fernau 2003, Definition 3]).* Let  $A = (Q, T, \delta, q_0, Q_F)$ <sup>5</sup> be a finite automaton, and  $f : T^* \rightarrow F$  a distinguishing function.  $A$  is called *f-distinguishable* if

- (1)  $A$  is deterministic
- (2) For all states  $q \in Q$  and all  $x, y \in T^*$  with  $\delta^*(q_0, x) = \delta^*(q_0, y)$ , we have  $f(x) = f(y)$ . (In other words, for every  $q \in Q$ , if we define  $f(q)$  as the value of  $f(x)$  for some  $x$  with  $\delta^*(q_0, x) = q$ , then  $f(q)$  is well-defined.)
- (3) For all  $q_1, q_2 \in Q$ ,  $q_1 \neq q_2$ , with either (a)  $q_1, q_2 \in Q_F$  or (b) there exist  $q_3 \in Q$  and  $a \in T$  with  $\delta(q_1, a) = \delta(q_2, a) = q_3$ , we have  $f(q_1) \neq f(q_2)$

Given a distinguishing function  $f$ , Fernau has shown [Fernau 2003, Theorem 8] that the corresponding class of *f-distinguishable languages* *f-DL* is identifiable in the limit.

5.1. IDENTIFIABILITY IN THE LIMIT OF PREFIX MARK-UP LANGUAGES. A first fundamental result about prefix mark-up languages is that it is possible to show that they are identifiable in the limit from positive examples only. In order to show this, we need to slightly extend the setting introduced by Fernau. In fact, the class of

<sup>5</sup> As usual:  $Q$  is a set of *states*,  $T$  is a set of *terminal symbols*,  $\delta : Q \times T \rightarrow Q$  is the *transition function*,  $q_0$  is the *initial state* and  $Q_F$  is the set of *final states*.

languages considered by Fernau is such that (a) all languages are regular languages; (b) the countable union of all languages in the class is still a regular language. On the contrary, prefix mark-up languages are regular languages but condition (b) does not hold. To see this, consider that the union of all prefix mark-up languages contains a well-known non-regular language, that is, the language of balanced parenthesis.

To take this into account, let us introduce the notion of *extended distinguishing function*.

*Definition 5.5 (Extended Distinguishing Function).* Let  $\mathcal{L}$  be a class of regular languages over  $T$ . A mapping  $f : T^* \rightarrow F$  is called an *extended distinguishing function for  $\mathcal{L}$*  if:

- $f(w) = f(z)$  implies  $f(wu) = f(zu)$  for all  $u, w, z \in T^*$ ;
- for every language  $L \in \mathcal{L}$ , the set of strings  $\{f(u) \mid uw \in L\}$  is a finite set.

With respect to Definition 5.3, we are removing the hypothesis that  $f$  has a finite codomain, and replacing it with a weaker one, that is, that  $f$  assumes a finite set of values when applied to prefixes of strings in each single language of the class.

An *extended  $f$ -distinguishable automaton* is defined as in Definition 5.4, with the only difference that it is based on an extended  $f$ -distinguishable function. A language is called *extended  $f$ -distinguishable* if it is recognized by an extended  $f$ -distinguishable automaton.

We shall now introduce a function,  $f_\pi$ , for prefix mark-up languages. Let us now consider the following set of reductions of string over  $\Sigma \cup \bar{\Sigma} \cup \Delta$ :

$$\begin{aligned} a\Delta\bar{a} &\rightarrow \epsilon \text{ for } a \in \Sigma \\ d &\rightarrow \Delta \text{ for } d \in \Delta^+ \\ \Delta\Delta &\rightarrow \Delta \end{aligned}$$

By applying these reductions, a string over  $\Sigma \cup \bar{\Sigma} \cup \Delta$  is reduced to a string over  $\Sigma \cup \bar{\Sigma} \cup \{\Delta\}$ . Intuitively, the reductions remove from a string any well-formed substring. A word is said to be *reduced* or *irreducible* if it cannot be further reduced. Let  $\rho(w)$  denote the unique irreducible word obtained from  $w$ . We now show that by appropriately choosing a distinguishing function  $f_\pi$ , it follows that the  $f_\pi$ -DL contains the class of prefix mark-up languages.

*Definition 5.6 ( $f_\pi$ ).* Let  $w$  be any string over  $\Sigma \cup \bar{\Sigma} \cup \Delta$ . Function  $f_\pi$  is defined as follows:

$$\begin{aligned} f_\pi : (\Sigma \cup \bar{\Sigma} \cup \Delta)^* &\rightarrow (\Sigma \cup \bar{\Sigma} \cup \{\Delta\})^* \\ \epsilon &\mapsto \epsilon \\ wa &\mapsto \rho(w) \cdot a, & a \in \Sigma \\ wa &\mapsto \rho(w \cdot a), & a \in \Delta. \end{aligned}$$

As an example, consider the following prefixes of strings in  $ab\Delta^+\bar{b}\bar{a}$ :

$$\begin{aligned} f(a) &= a & f(ab01) &= ab\Delta & f(ab01\bar{b}b1) &= ab\Delta \\ f(ab) &= ab & f(ab01\bar{b}) &= ab\Delta\bar{b} & f(ab01\bar{b}b1\bar{b}) &= ab\Delta\bar{b} \\ f(ab0) &= ab\Delta & f(ab01\bar{b}b) &= ab & f(ab01\bar{b}b1\bar{b}b) &= ab. \end{aligned}$$

THEOREM 5.7.  $f_\pi$  is an extended distinguishing function for the class of prefix mark-up languages.

THEOREM 5.8 (IDENTIFIABILITY IN THE LIMIT). The class of prefix mark-up languages is identifiable in the limit.

The proof of these theorems is in Appendix A. From Theorem 5.8, it also follows that prefix mark-up languages have a characteristic sample.

COROLLARY 5.9. The class of prefix mark-up languages have characteristic samples.

5.2. LIMITATIONS OF TRADITIONAL GRAMMAR INFERENCE FOR INFORMATION EXTRACTION. Given the availability of fully unsupervised algorithms for their learning, the classes of languages that are inferrable in the limit represent natural candidates for automatic wrapper generation on the Web. However, there is a number of shortcomings associated with this approach that seriously limit its practical applicability for many of the known classes of languages. In this discussion, we will mainly focus on reversible grammars [Angluin 1982], but the same argument holds for any class of  $f$ -distinguishable languages.

The  $k$ -reversible languages are a family of classes of languages (based on the value of  $k$ , we have respectively 0-reversible languages, 1-reversible languages, 2-reversible languages etc.); each of these classes is a subset of the regular languages. We omit the formal definition. However, to give an intuition, given a value for  $k$ , the class of  $k$ -reversible languages is the class of regular languages such that the corresponding automaton is *reversible with lookahead of  $k$  symbols*; this means that the reversed automaton—obtained from the direct one by exchanging initial and final states and inverting all transitions—is such that any nondeterministic transition can be made deterministic by looking ahead the next  $k$  symbols in the input.

The main limitation of the learning algorithm developed for  $k$ -reversible languages is that, in order to produce a correct result, the algorithm assumes that the inference system is given a characteristic sample of the language; in automata-theoretic terms, it is a set of samples with two main characteristics: (a) it is a set of samples that has the property of “covering” the whole automaton, that is, touching all states and traversing all transitions of the language automaton; (b) among all the sample sets that have this property, it is the one made of strings of the minimal length.

Let us fix a  $k$ -reversible language with automaton  $A = \{Q, T, \delta, q_0, Q_F\}$ ; then, a characteristic sample for the language is any set of strings of the form:

$$\chi(A) = \{u(q)v(q) \mid q \in Q\} \\ \cup \{u(q)av(q) \mid q \in Q, a \in T\},$$

where  $u(q)$  and  $v(q)$  are words of minimal length with  $\delta(q_0, u(q)) = q$  and  $\delta^*(q, v(q)) \in Q_F$ .

The strongest assumption in this definition is that the strings need to have minimal length. This makes quite unlikely in practice to find a characteristic sample in a collection of random samples. Consider our running example. It can be shown that the language in Figure 3 is a 1-reversible language. A characteristic sample for this language is made of the strings in Figure 6. The sample contains four strings: Sample  $a$  is a representative of the strings of minimal-length in the language

<i>Sample a</i>	<i>Sample b</i>
<pre> &lt;HTML&gt; &lt;IMG/&gt; &lt;B&gt;Wally Wood&lt;/B&gt; &lt;A&gt;&lt;/A&gt; &lt;UL&gt;   &lt;LI&gt; &lt;IMG/&gt; &lt;I&gt;Linux Programming&lt;/I&gt;     &lt;P&gt;       &lt;B&gt;&lt;BR/&gt;First Ed., 1998&lt;IMG/&gt;&lt;/B&gt;     &lt;/P&gt;   &lt;/LI&gt; &lt;/UL&gt; &lt;/HTML&gt; </pre>	<pre> &lt;HTML&gt; &lt;IMG&gt; &lt;B&gt;Jack Kirby&lt;/B&gt; &lt;A&gt;&lt;TT&gt;kirby@dot.edu&lt;/TT&gt;&lt;/A&gt; &lt;UL&gt;   &lt;LI&gt; &lt;IMG/&gt; &lt;I&gt;J2EE 1.4&lt;/I&gt;     &lt;P&gt;       &lt;B&gt;&lt;BR/&gt;First Ed., 2002&lt;IMG/&gt;&lt;/B&gt;     &lt;/P&gt;   &lt;/LI&gt; &lt;/UL&gt; &lt;/HTML&gt; </pre>
<i>Sample c</i>	<i>Sample d</i>
<pre> &lt;HTML&gt; &lt;IMG&gt; &lt;B&gt;Stan Lee&lt;/B&gt; &lt;A&gt;&lt;/A&gt; &lt;UL&gt;   &lt;LI&gt; &lt;IMG/&gt; &lt;I&gt;Superpowers&lt;/I&gt;     &lt;P&gt;       &lt;B&gt;&lt;BR/&gt;First Ed., 2001&lt;IMG/&gt;&lt;/B&gt;     &lt;/P&gt;   &lt;/LI&gt;   &lt;LI&gt; &lt;IMG/&gt; &lt;I&gt;Microsoft .NET&lt;/I&gt;     &lt;P&gt;       &lt;B&gt;&lt;BR/&gt;First Ed., 2002&lt;IMG/&gt;&lt;/B&gt;     &lt;/P&gt;   &lt;/LI&gt; &lt;/UL&gt; &lt;/HTML&gt; </pre>	<pre> &lt;HTML&gt; &lt;IMG&gt; &lt;B&gt;John Romita&lt;/B&gt; &lt;A&gt;&lt;/A&gt; &lt;UL&gt;   &lt;LI&gt; &lt;IMG/&gt; &lt;I&gt;Security&lt;/I&gt;     &lt;P&gt;       &lt;B&gt;&lt;BR/&gt;First Ed., 2001&lt;IMG/&gt;&lt;/B&gt;     &lt;/P&gt;   &lt;P&gt;       &lt;B&gt;&lt;BR/&gt;Second Ed., 2002&lt;IMG/&gt;&lt;/B&gt;     &lt;/P&gt;   &lt;/LI&gt; &lt;/UL&gt; &lt;/HTML&gt; </pre>

FIG. 6. A characteristic sample for the running example.

(no e-mail, one book with one edition); Sample *b* derives from Sample *a* with an addition to exercise the portion of the automaton related with the optional (e-mail, one book with one edition); Sample *c* derives Sample *a* and exercises the portion of the automaton related with the external plus (no e-mail, two books each with one edition); finally, Sample *d* derives from Sample *a* and exercises the internal plus (no e-mail, one book with two editions).

It can be seen how requiring that the input to the learning algorithm includes a characteristic sample defined as shown above is a serious drawback. In fact, a simple probabilistic argument shows that the probability of finding such strings of minimal length in a collection of random HTML pages in a site is quite low. Just to give an example, pages like the ones shown in Figure 3—or any other page with a different distribution of cardinalities of email, books and editions—would not help to identify the correct language unless also the samples in Figure 6 are inspected by the algorithm.

As a consequence, in the next sections we develop a new algorithm for inferring prefix mark-up languages that has the nice property of being based on a more natural notion of characteristic sample. This notion is that of a *rich collection of instances*, that is, a database-theoretic notion, which essentially requires that the samples used in the inference make full use of the underlying type.

Our experience shows that there is usually a good probability of finding a rich collection of instances by looking at a few samples. To see this, consider this simple argument (for simplicity, we focus on lists only): suppose we are given random instances with a probability  $p$  that, for a given label  $\alpha$ , two instances have all lists with label  $\alpha$  of equal cardinality. Then, the probability of finding two different cardinalities among all lists labeled  $\alpha$  in a collection of  $n$  instances is

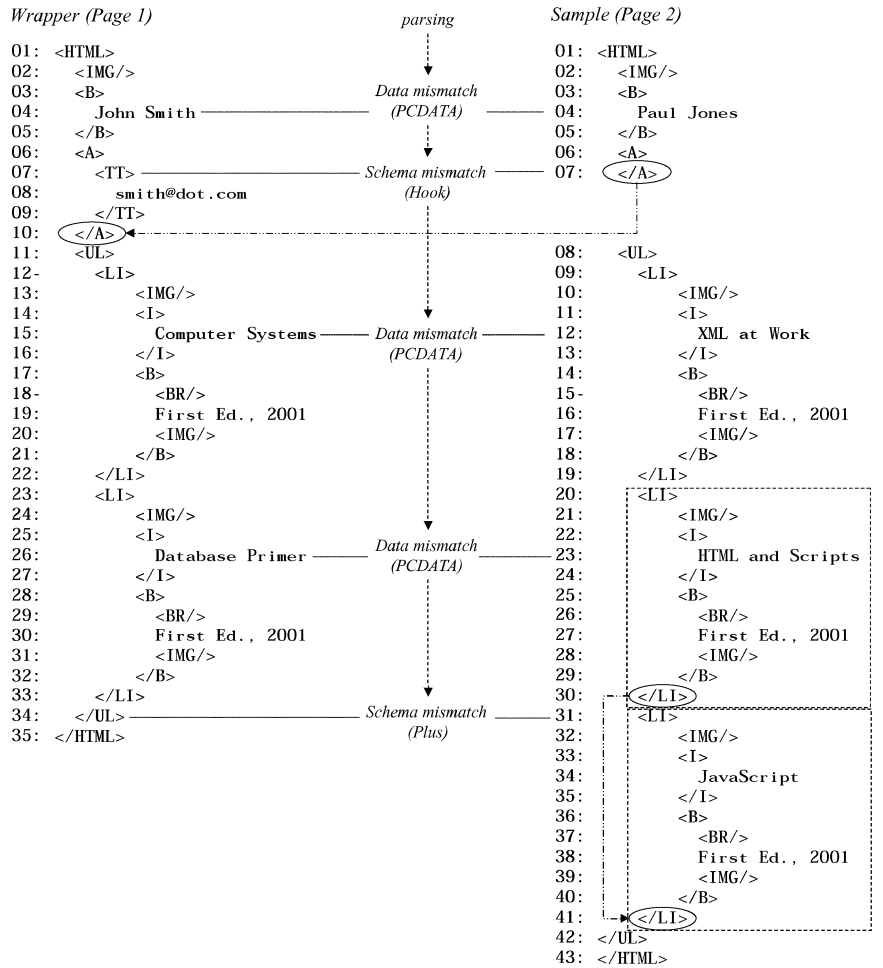


FIG. 7. One simple matching.

$(1 - p^{n-1})$ . Assuming that the probabilities are independent for different labels, then, the probability that a collection of  $n$  instances with  $k$  list labels is list-rich, is  $(1 - p^{n-1})^k$ . So, for example if  $p = \frac{1}{2}$ , the probability that a type with 5 lists in its type is found after looking at 10 random samples is 99%.

### 6. The Matching Technique

This section is devoted to the informal presentation of algorithm *match* [Crescenzi et al. 2001] for solving the schema finding problem for prefix mark-up encodings. We assume that HTML sources are preprocessed to transform them into lists of tokens. Each token is either an HTML tag or a string. Tags will be represented by symbols of the schema alphabet  $\Sigma \cup \bar{\Sigma}$ , strings will be represented by symbols of the data alphabet  $\Delta$ . Figure 7 shows a simple example in which two HTML sources have been transformed into lists of 35 and 43 tokens, respectively.

The algorithm is quite complex because it makes heavy use of recursion, and for the sake of presentation, an informal description based on an HTML running

example will precede a more precise description. The following sections are organized as follows: First, Sections 6.1–6.1.3 illustrate the key ideas behind the matching technique; subtleties of the algorithm are discussed in Section 6.2. A formal description of the algorithm is given in Section 7.

**6.1. MISMATCHES.** The matching algorithm works on two objects at a time: (i) a *sample*, that is, a list of tokens corresponding to one of the sample pages, and (ii) a *wrapper*, that is, a prefix mark-up language represented as an abstract syntax tree. Given two HTML pages (called page 1 and page 2), to start we take one of the two, for example page 1, as an initial version of the wrapper; then, the wrapper is progressively refined trying to find a common language for the two pages.

The algorithm consists in *parsing* the sample by using the wrapper. A crucial notion, in this context, is the one of *mismatch*: a mismatch happens when some token in the sample does not comply with the grammar specified by the wrapper. Mismatches are very important, since in the hypothesis that the input instance and template are homogeneous, they help to discover essential information about the wrapper. Whenever one mismatch is found, the algorithm tries to solve it by generalizing the wrapper. This is done by applying suitable *generalization operators*. The algorithm succeeds if a common wrapper can be generated by solving all mismatches encountered during the parsing.

There are essentially two kinds of mismatches that can be generated during the parsing. The simplest case is that of *data mismatches*, that is, mismatches that happen when different strings occur in corresponding positions of the wrapper and of the sample. If the two pages are encodings of two homogeneous instances, these differences may be due only to different values of a basic attribute. This case is solved by applying the operator *addPCDATA*, which introduces a #PCDATA leaf in the wrapper.

Mismatches that involve either two different tags, or one tag and one string on the wrapper and on the sample are more complex. These mismatches are due to the presence of iterators (i.e., lists) and optional patterns. We generalize zero or one repetition by introducing an optional, one or more repetitions by introducing a list. They are solved by applying the operators *addPlus* and *addHook*, which respectively add a *Plus* and a *Hook* node on the wrapper. In light of this, the matching of a wrapper and a sample can be considered as a search problem in a particular state space. States in this space correspond to different versions of the wrapper. The algorithm moves from one state to another by applying operators *addPCDATA*, *addPlus*, *addHook*. A final state is reached whenever the current version of the wrapper can be used to correctly parse the given sample.

These ideas are clarified in the following with the help of the running example shown in Figure 7. For the sake of simplicity, with respect to Figure 2, we have simplified the original type and the HTML sources by assuming that all books involved in the matching have a single edition; this allows us to simplify the discussion by ignoring the inner level of nesting in the original type due to multiple editions for a given book.

**6.1.1. Applying Operator *addPCDATA*: Discovering Attributes.** Figure 7 shows several examples of data mismatches during the first steps of the parsing. Consider, for example, strings ‘John Smith’ and ‘Paul Jones’ at token 4. To solve this data mismatch, we apply operator *addPCDATA*, that is, we generalize the wrapper by replacing string ‘John Smith’ by #PCDATA. The same happens a few steps after for ‘Database Primer’ and ‘XML at Work’.

6.1.2. *Applying Operator addHook: Discovering Optionals.* Schema mismatches are used to discover both lists and optionals. This means that whenever one of these mismatches is found, the algorithm needs to choose which operator to apply. Let us for now ignore the details of this choice, and concentrate first on the discovery of optionals, that is, the application of operator *addHook*. Lists will be discussed in the following section.

Consider again Figure 7. The first schema mismatch happens at token 7 due to the presence of the email in the wrapper and not in the sample, that is, the mismatch is due to an optional node which has been instantiated in different ways. To apply operator *addHook* and generalize the wrapper, we need to carry out the following steps:

- (1) *Optional Pattern Location by Cross-Search.* With respect to the running example, given the mismatching tags at token 7— `<TT>` and `</A>`—we know that: (a) assuming the optional pattern is located on the wrapper, after skipping it we should be able to proceed by matching the `</A>` on the sample with some successive occurrence of `</A>` tag on the wrapper; (b) on the contrary, assuming the pattern is located on the sample, we should proceed by matching token 7 on the wrapper with an occurrence of tag `<TT>` on the sample. A simple cross-search of the mismatching tags leads to the conclusion that the optional pattern is located on the wrapper (the sample does not contain any `<TT>` tag).
- (2) *Wrapper Generalization.* Once the optional pattern has been identified, we may generalize the wrapper accordingly and then resume the parsing. In this case, the wrapper is generalized by introducing pattern `(<TT>smith@dot.com</TT>)?`, and the parsing is resumed by comparing tokens `</UL>` (11 and 8 respectively).

6.1.3. *Applying Operator addPlus: Discovering Iterators.* Let us now concentrate on the task of discovering iterators. Consider again Figure 7; it can be seen that the two HTML sources contain, for each author, one list of book titles. During the parsing, a tag mismatch between tokens 34 and 31 is encountered; it is easy to see that the mismatch comes from different cardinalities in the book lists (two books on the wrapper, three books on the sample). To solve the mismatch, we need to identify these repeated patterns that we call *squares* by applying operator *addPlus* to generalize the wrapper accordingly; then, the parsing can be resumed. In this case three main steps are performed:

(1) *Square Location by Delimiter Search.* After a schema mismatch, a key hint we have about the square is that, since we are under an iterator (+), both the wrapper and the sample contain at least one occurrence of the square. Let us call  $o_w$  and  $o_s$  the number of occurrences of the square in the wrapper and in the sample, respectively (2 and 3 in our example). If we assume that occurrences match each other, we may conclude that before encountering the mismatch the first  $\min(o_w, o_s)$  square occurrences have been matched (2 in our example).

As a consequence, we can identify the last token of the square by looking at the token immediately before the mismatch position. This last token is called *end delimiter* (in the running example, this corresponds to tag `</LI>`). Also, since the mismatch corresponds to the end of the list on one sample and the beginning of a new occurrence of the square on the other one, we also have a clue about how the square starts, that is about its *start delimiter*; however, we don't know exactly where the list with the higher cardinality is located, that is, if in the wrapper or in the sample; this means that we don't know which one of the mismatching tokens

corresponds to the start delimiter (`</UL>` or `<LI>`). We therefore need to explore two possibilities: (i) candidate square of the form `</UL> . . . </LI>` on the wrapper, which is not a real square; or (ii) candidate square of the form `<LI> . . . </LI>` on the sample. We check both possibilities by searching first the wrapper and then the sample for occurrences of the end delimiter `</LI>`; in our example, the search fails on the wrapper; it succeeds on the sample. We may therefore infer that the sample contains one candidate square occurrence at tokens 31 to 41.

(2) *Candidate Square Matching*. To check whether this candidate occurrence really identifies a square, we try to match the candidate square occurrence (tokens 31–41) against some upward portion of the sample. This is done backwards, that is, it starts by matching tokens 41 and 30, then moves to 40 and 29 and so on. The search succeeds if we manage to find a match for the whole square, as it happens in Figure 7.

(3) *Wrapper Generalization*. It is now possible to generalize the wrapper; if we denote the newly found square by  $s$ , we do that by searching the wrapper for contiguous repeated occurrences of  $s$  around the mismatch point, and by replacing them by  $(s)^+$ .

Once the mismatch has been solved, the parsing can be resumed. In the running example, after solving this last mismatch the parsing is completed. We can therefore conclude that the parsing has been successful and we have generated a common wrapper for the two input HTML pages.

6.2. RECURSION. In Figure 7, the algorithm succeeds after solving several data mismatches and two simple schema mismatches. In general, the number of mismatches to solve may be high, mainly because the mismatch solving algorithm is inherently recursive: when trying to solve one mismatch by finding an iterator, during the candidate square matching step more mismatches can be generated and have to be solved.

To see this, consider Figure 8, which shows the process of matching two pages of our running example with the list of editions nested inside the list of books. The wrapper (page 1) is matched against the sample (page 2). After solving a couple of data mismatches, the parsing stops at token 25, where a schema mismatch is found. It can be solved by looking for a possible iterator, following the usual three steps: (i) the candidate square occurrence on the wrapper is located (tokens 25–42) by looking for an occurrence of the possible end delimiter (`</LI>` at token 24); then (ii) the candidate is evaluated by matching it against the upward portion of the wrapper (tokens 25–42 against the portion preceding token 25); and finally, (iii) the wrapper is generalized. Let us concentrate on the second step: remember that the candidate is evaluated by matching it backwards, that is, starting from comparing the two occurrences of the end delimiter (tokens 42 and 24), then move to tokens 41 and 23 and so on.

This comparison has been emphasized in Figure 8 by duplicating the wrapper portions that have to be matched. Since they are matched backwards, tokens are listed in reverse order. Differently from the previous example—in which the square had been matched by a simple alignment—it can be seen that, in this case, new mismatches are generated when trying to match the two fragments. These mismatches are called *internal mismatches*. The first internal mismatch in our example involves tokens 35 and 17: it depends on the nested structure of the page, and will lead to the discovery of the list of editions inside the list of books.

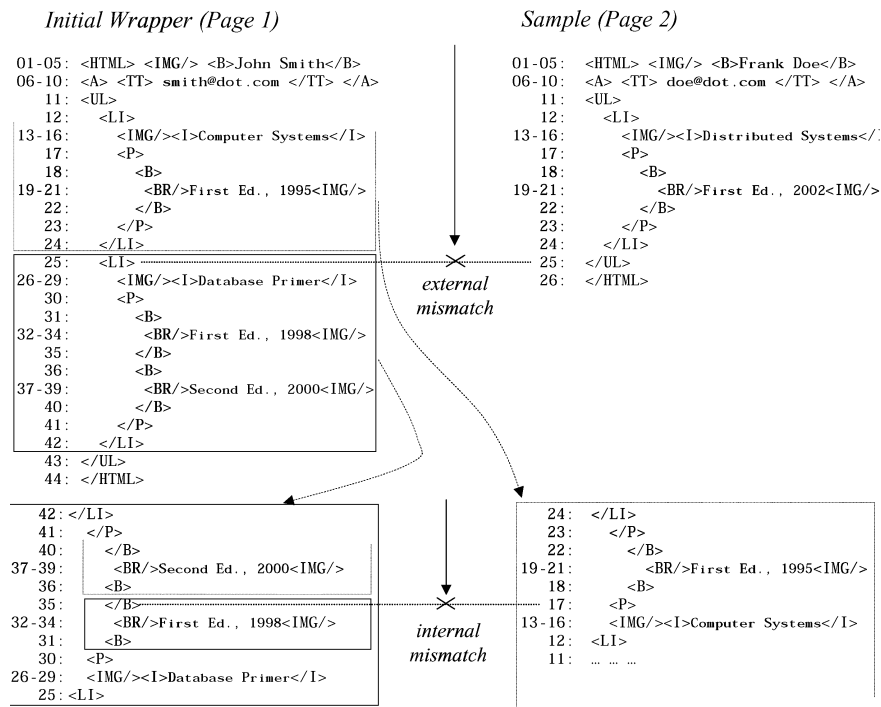


FIG. 8. A more complex matching.

These internal mismatches have to be processed exactly in the same way as the external ones. This means that the matching algorithm needs to be recursive, since, when trying to solve some external mismatch, new internal mismatches may be raised, and each of these requires to start a new matching procedure, based on the same ideas discussed above. The only difference is that these recursive matchings do not work by comparing one wrapper and one sample, but rather two different portions of the same object, that is, either wrapper or sample.<sup>6</sup>

It can be seen that this recursive nature of the problem makes the algorithm quite involved. In fact, during the search in the state space, in order to be able to apply *addPlus* operators it is necessary to trigger a new search problem, which corresponds to matching candidate squares. In this respect, the state space of this new problem may be considered at a different level: its initial state coincides with the candidate square of the operator while the final state, if any, is the square which will be used to generalize the wrapper in the upper level. The search in this new space may in turn trigger other instances of the same search problem. These ideas are summarized in Figure 9, which shows how the search is really performed by working on several state spaces, at different levels.

As a search space, the algorithm sketched in this section might be subjected to backtracking. This is due to the fact that, in general, to solve a mismatch the

<sup>6</sup> As it can be seen from this example, internal mismatches may lead to matchings between portions of the wrapper; since the wrapper is in general one regular expression, this would require matching two regular expressions, instead of one expression and one sample. We will discuss how to solve this problem in Section 7.

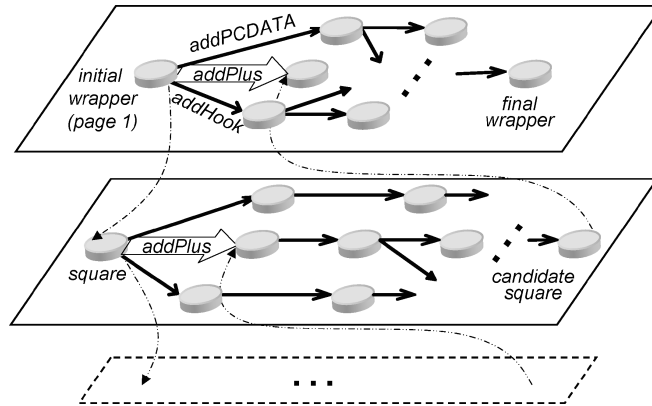


FIG. 9. Matching as a search problem in a state space.

algorithm needs to choose among several alternatives, which are not guaranteed to lead to a correct solution. When, going ahead in the matching, these choices prove to be wrong, it is necessary to backtrack and resume the parsing from the next alternative until the wrapper successfully parses the sample. However, in the following sections we will show that prefix mark-up languages can be inferred without backtracking in the search space. This makes the complexity of the algorithm polynomial with respect to the length of the input samples.

## 7. The Algorithm

This section formalizes the matching algorithm informally described above.

**7.1. PRELIMINARY DEFINITIONS.** To give a more precise definition of the algorithm, we first need to introduce some notation. In the following, we will refer to both the wrapper and the sample as abstract syntax trees.

*Regions.* To formalize operations on abstract syntax trees, we introduce a syntax to refer to regions. We call a *region* in a tree  $\tau$  a (possibly empty) list of contiguous subtrees of  $\tau$ . We will denote a region in a tree  $\tau$  by the roots  $n, n'$  of the delimiting subtrees, with the usual syntax for intervals:  $\tau[n, n']$  is the region made by the contiguous subtrees delimited by subtrees rooted at nodes  $n$  and  $n'$ , including the two delimiting subtrees.  $\tau[n, n')$  does not include the subtree rooted at  $n'$ ; similarly for  $\tau(n, n']$  and  $\tau(n, n')$ . When there is no ambiguity, the reference to the tree will be omitted. So, for example, with respect to the tree in Figure 3, the region  $[<B>, Hook]$  denotes the region delimited by token  $<B>$  and by the subtree rooted at the *Hook* node, delimiting subtrees included (i.e., the subexpression going from  $<B>$  to  $</TT>$ ).

We also introduce a notation for regions with only one delimiting subtree. A region including all subtrees rooted at successive siblings of a node  $n$  (included) will be denoted  $[n, \dots]$ . A region including all subtrees rooted at siblings preceding a node  $n$  (included) will be denoted  $[\dots, n]$ . Similarly for  $(n, \dots]$  and  $[\dots, n)$ . So, for example, with respect to the tree in Figure 3,  $[\dots, Hook)$  denotes the region delimited by token  $(<HTML>)$  and by the subtree rooted at the *Hook* node (excluded).

*Tree Operators.* We now introduce some edit operators on trees. The first one, the *substitution operator*,  $\text{subst}(\tau, r, \tau')$ , replaces a region  $r$  in a tree  $\tau$  by a new

subtree  $\tau'$ . Then, we have two *insert operators* that insert new subtrees in an existing tree.  $\text{insBefore}(\tau, n, \tau')$  inserts subtree  $\tau'$  in  $\tau$  immediately before the subtree rooted at node  $n$ .  $\text{insAfter}(\tau, n, \tau')$  inserts subtree  $\tau'$  in  $\tau$  immediately after the subtree rooted at node  $n$ . For example, if  $\tau$  is the abstract tree of Figure 3 and  $\tau' = \text{Hook}(\text{And}(\langle \text{HR}/\rangle, \langle \text{TT}\rangle, \text{PCDATA}, \langle / \text{TT}\rangle))$  then:

$$\text{subst}(\tau, [\langle \text{A}\rangle, \langle / \text{UL}\rangle], \tau') = \text{And}(\langle \text{HTML}\rangle, \langle \text{IMG}/\rangle, \langle \text{B}\rangle, \text{PCDATA}, \langle / \text{B}\rangle, \langle \text{HR}/\rangle, \langle \text{TT}\rangle, \text{PCDATA}, \langle / \text{TT}\rangle, \langle / \text{HTML}\rangle)$$

We also define two *search operators* on trees, that are used to search subtrees in an abstract syntax both forward and backward. If  $\tau$  and  $\tau'$  are trees, and  $n$  is a node in  $\tau$ ,  $\text{search}_k^{\rightarrow}(\tau, n, \tau')$  returns the  $k$ th occurrence in tree  $\tau$  of the subtree  $\tau'$  following node  $n$  (included);  $\text{search}_k^{\leftarrow}(\tau, n, \tau')$  returns the  $k$ th occurrence in tree  $\tau$  of subtree  $\tau'$  preceding node  $n$  (included). If that occurrence does not exist, the search functions are undefined and return  $\perp$ .

*Mismatches.* Let us also formalize the notion of mismatch. A *mismatch* is defined as a quadruple  $m = (w, s, (n, t))$ , where  $w$  is a wrapper, that is, a prefix mark-up language represented through its abstract syntax tree,  $s$  is a sample, that is, a sequence of tokens,  $n$  is a node in the abstract syntax tree, and  $t$  is a token in the sample.

We are now ready to precisely define the generalization operators.

**7.2. OPERATOR *addPCDATA*.** Operator *addPCDATA* replaces a constant token by #PCDATA. It can be formalized as a function that receives a *data* mismatch  $m = (w, s, (n, t))$  and returns an abstract syntax tree representation of a regular expression, as follows:<sup>7</sup>

```
AST addPCDATA(mismatch  $m = (w, s, (n, t))$ ,  $d \in \{\rightarrow, \leftarrow\}$ )
begin
  if ( $m$  is a data mismatch) return  $\text{subst}(w, w[n, n], \text{PCDATA})$ ;
  return  $\perp$ ;
end
```

**7.3. OPERATOR *addHook*.** Section 6.1.2 has shown the intuition behind this operator that should solve mismatches due to the presence of a pattern on the sample but not on the wrapper or vice versa. The pseudo-code of operator *addHook* is shown in Figure 10. The figure contains both the case in which the optional is located on the wrapper ( $\text{addHook}^w$ ) and on the sample ( $\text{addHook}^s$ ).

Function  $\text{findHookSquare}$  finds candidate squares by searching for occurrences of the delimiter. It discards ill-formed candidates by using function  $\text{isWFMarkUp}$ ; function  $\text{isWFMarkUp}$  is used to check if the language associated with a region is a mark-up language, that is, if the region represents a piece of well-formed mark-up interleaved with some data.

Predicate  $\text{checkSquare}$  is needed to avoid the mistake of applying an *addHook* operator whenever *addPlus* would be the right choice. In fact, whenever *addPlus* is

<sup>7</sup> Note that parameter  $d$  is not strictly necessary in function *addPCDATA*; however, to simplify the notation in the following sections, we choose to have the same signature for all operators.

<pre> AST <math>\overrightarrow{addHook}^w</math>(mismatch <math>m = (w, s, (n, t))</math>) begin   if (<math>m</math> is a data mismatch) return <math>\perp</math>;   Let <math>csquare</math> be <math>\overrightarrow{findHookSquare}(w, n, t)</math>;   if (<math>csquare = \perp</math>) return <math>\perp</math>;   if (not checkSquare(<math>m, csquare</math>)) return <math>\perp</math>;   return subst(<math>w, csquare, Hook(And(csquare))</math>); end </pre>	<pre> AST <math>\overrightarrow{addHook}^s</math>(mismatch <math>m = (w, s, (n, t))</math>) begin   if (<math>m</math> is a data mismatch) return <math>\perp</math>;   Let <math>csquare</math> be <math>\overrightarrow{findHookSquare}(s, t, n)</math>;   if (<math>csquare = \perp</math>) return <math>\perp</math>;   if (not checkSquare(<math>m, csquare</math>)) return <math>\perp</math>;   return insBefore(<math>w, n, Hook(And(csquare))</math>); end </pre>
<pre> AST <math>\overrightarrow{findHookSquare}(tree, from, wanted)</math> begin   Let <math>k</math> be 0;   do <math>k = k+1</math>;     Let <math>occ</math> be <math>\overrightarrow{search}_k(tree, from, wanted)</math>;     if (<math>occ = \perp</math>) return <math>\perp</math>;     Let <math>csquare</math> be <math>tree[from, occ]</math>;     while (not isWFMarkup(<math>csquare</math>));     return <math>csquare</math>; end </pre>	<pre> boolean checkSquare(<math>m = (w, s, (n, t)), square</math>) begin   return (last token of square is not equal to   token immediately before <math>t</math>) end  boolean checkSquare(<math>m = (w, s, (n, t)), square</math>) begin   return (first token of square is not equal to   token immediately after <math>t</math>) end </pre>

FIG. 10.  $addHook$  operator.

applicable, also  $addHook$  is applicable. Such situations can be sketched as follows:

$$\begin{array}{l}
 w : \dots \overbrace{start(\alpha) \cdot start(\beta) \dots end(\beta) \dots start(\beta) \dots end(\beta) \dots start(\beta) \dots end(\beta)}^{g \text{ times}} \cdot end(\alpha) \dots \\
 s : \dots \underbrace{start(\alpha) \cdot start(\beta) \dots end(\beta) \dots start(\beta) \dots end(\beta)}_{h < g \text{ times}} \dots end(\alpha) \dots
 \end{array}$$

Obviously, there could be more occurrences of the pattern on the sample than on the wrapper and still this discussion would symmetrically hold; by applying  $addPlus^w$  we obtain the correct generalization:

$$\dots start(\alpha) \cdot (start(\beta) \dots end(\beta))^+ \cdot end(\alpha) \dots$$

if, on the contrary,  $addHook^w$  is applied, it would produce:

$$\dots \overbrace{start(\alpha) \cdot start(\beta) \dots end(\beta)}^{h \text{ times}} \cdot \underbrace{start(\beta) \dots end(\beta) \cdot start(\beta) \dots end(\beta)}_{g - h \text{ times}} \cdot end(\alpha) \dots$$

but this version of the wrapper is useless. In fact, observe that by definition of prefix mark-up encoding, symbols of  $end(\beta)$  cannot mark an optional pattern and also occur immediately before it, because otherwise they would appear in the delimiters of an optional node and in those of its child. These configurations of the wrapper are discarded by checkSquare.

7.4. OPERATOR  $addPlus$ . Section 6.1.3 has already presented the main ideas to solve mismatches by means of iterators. Here, a precise description of the corresponding bidirectional operator  $addPlus$  is given. Operator  $addPlus$  is the source of most of the complexity of the matching technique. First, it is mutually recursive with algorithm  $match$  (which is about to be described in Section 7.5), because when trying to apply an iterator the algorithm needs to look for a repeated pattern by matching two portions of the same object; second, also bidirectionality arise from

<pre> AST <math>\overrightarrow{addPlus}^w</math>(mismatch <math>m=(w, s, (n, t))</math>) begin   if (<math>m</math> is a data mismatch) return <math>\perp</math>;   //Square Location by Delimiter Search   Let <math>ldel</math> be <math>\overrightarrow{lastDelim}(m)</math>;   if (<math>ldel=\perp</math>) return <math>\perp</math>;   Let <math>csquare^w=w[n, l_i]</math> be   findPlusSquare<math>\overrightarrow{w, n, ldel}</math>;   if (<math>csquare^w=\perp</math>) return <math>\perp</math>;   //Candidate Square Evaluation   if (<math>\overrightarrow{match}(w[. . n], csquare^w)</math>) begin     Let <math>i</math> be 0;     Let <math>square_i</math> be <math>\overleftarrow{match}</math>.getResult();     Let <math>f_i</math> be <math>\overrightarrow{match}</math>.getLastMatchingNode();     //Candidate Square Matching - Leftward     while (<math>\overrightarrow{match}(square_i, w[. . f_i])</math>) do       <math>i = i - 1</math>;     Let <math>square_i</math> be <math>\overleftarrow{match}</math>.getResult();     Let <math>f_i</math> be <math>\overrightarrow{match}</math>.getLastMatchingNode();   end   //Candidate Square Matching - Rightward   Let <math>f</math> be <math>f_i</math>;   Let <math>square_f</math> be <math>square_i</math>;   Let <math>j</math> be 1;   while (<math>\overrightarrow{match}(square_j, w[l_j . .])</math>) do     <math>j = j + 1</math>;     Let <math>square_j</math> be <math>\overrightarrow{match}</math>.getResult();     Let <math>l_j</math> be <math>\overrightarrow{match}</math>.getLastMatchingNode();   end   Let <math>l</math> be <math>l_j</math>;   Let <math>square</math> be <math>square_j</math>;   Let <math>squareExt</math> be <math>w[f, l]</math>;   //Wrapper Generalization   return <math>\text{subst}(w, squareExt, Plus(square))</math>; end return <math>\perp</math>; end </pre>	<pre> AST <math>\overrightarrow{addPlus}^s</math>(mismatch <math>m=(w, s, (n, t))</math>) begin   if (<math>m</math> is a data mismatch) return <math>\perp</math>;   //Square Location by Delimiter Search   Let <math>ldel</math> be <math>\overrightarrow{lastDelim}(m)</math>;   if (<math>ldel=\perp</math>) return <math>\perp</math>;   Let <math>csquare^s=s[t, l_i]</math> be   findPlusSquare<math>\overrightarrow{s, t, ldel}</math>;   if (<math>csquare^s=\perp</math>) return <math>\perp</math>;   //Candidate Square Evaluation   if (<math>\overrightarrow{match}(w[. . n], csquare^s)</math>) begin     Let <math>i</math> be 0;     Let <math>square_i</math> be <math>\overleftarrow{match}</math>.getResult();     Let <math>f_i</math> be <math>\overrightarrow{match}</math>.getLastMatchingNode();     //Candidate Square Matching - Leftward     while (<math>\overrightarrow{match}(square_i, w[. . f_i])</math>) do       <math>i = i - 1</math>;     Let <math>square_i</math> be <math>\overleftarrow{match}</math>.getResult();     Let <math>f_i</math> be <math>\overrightarrow{match}</math>.getLastMatchingNode();   end   //Candidate Square Matching - Rightward   Let <math>f</math> be <math>f_i</math>;   Let <math>square_f</math> be <math>square_i</math>;   Let <math>j</math> be 1;   while (<math>\overrightarrow{match}(square_j, s[l_j . .])</math>) do     <math>j = j + 1</math>;     Let <math>square_j</math> be <math>\overrightarrow{match}</math>.getResult();     Let <math>l_j</math> be <math>\overrightarrow{match}</math>.getLastMatchingToken();   end   Let <math>square</math> be <math>square_j</math>;   Let <math>squareExt</math> be <math>w[f, n]</math>;   //Wrapper Generalization   return <math>\text{subst}(w, squareExt, Plus(square))</math>; end return <math>\perp</math>; end </pre>
<pre> AST region <math>\overrightarrow{lastDelim}</math>(mismatch (<math>w, s, (n, t)</math>)) begin   Let <math>d</math> be the last data token preceding <math>t</math>;   if (<math>d</math> does not exist) return <math>\perp</math>;   Let <math>lct</math> be the last closing tag of region <math>s[d, t)</math>   for which there is no opening tag;   if (<math>lct</math> does not exist) return <math>\perp</math>;   if (not isWellFormed(<math>s[lct, t)</math>)) return <math>\perp</math>;   return <math>s[lct, t)</math>; end  boolean isWellFormed(a region <math>s</math>) { check well-formedness of <math>s</math> } </pre>	<pre> AST <math>\overrightarrow{findPlusSquare}(tree, from, wanted)</math> begin   Let <math>k</math> be 0;   do <math>k = k+1</math>;     Let <math>occ</math> be <math>\overrightarrow{search}_k(tree, from, wanted)</math>;     if (<math>occ = \perp</math>) return <math>\perp</math>;     Let <math>csquare</math> be <math>tree[from, occ]</math>;   while (not isWFMarkup(<math>csquare</math>));   return <math>csquare</math>; end </pre>

FIG. 11.  $\overrightarrow{addPlus}$ .

this operator because during the evaluation of the candidate square, the matching direction needs to be reversed.

For the sake of readability it is assumed that the operator is being applied in the direction ‘ $\rightarrow$ ’ but the same description symmetrically holds in the opposite direction.<sup>8</sup> We briefly comment on the code in Figure 11. The Figure contains both the case in which the candidate square is located on the wrapper ( $\overrightarrow{addPlus}^w$ ) and

<sup>8</sup> In the following, whenever the discussion holds for both directions, we will omit to specify the direction.

on the sample  $\overrightarrow{addPlus^s}$ . The two functions perform the following steps:

*Square Location by Delimiter Search.* Given a mismatch  $m = (w, s, (n, t))$ , the *last delimiter* is the sequence of tokens used to mark a candidate square both on  $w$  and on  $s$ . It encompasses tokens from the wrapping delimiter to the last matching token before the mismatch point. The delimiter is located by function `lastDelim`. Function `findPlusSquare` finds candidate squares by searching for occurrences of the delimiter. Ill-formed candidates are discarded.

*Candidate Square Evaluation.* The candidate square is evaluated by performing a first *internal* matching. If it succeeds, there are at least two occurrences of the same pattern to collapse.

*Candidate Square Matching.* Once a first version of the square has been found, the operator tries to locate its extension around the mismatch position. First the algorithm tries to locate its left border by iteratively consuming occurrences of the pattern on the left hand side of the mismatch position, then it tries to locate the right border of the extension. The latter step depends on whether the square has been located on the wrapper or on the sample. If it has been located on the sample, the square extension on the wrapper ends immediately before the mismatch point  $n$ . On the contrary, if the square has been located on the wrapper, it means that the last occurrence on the wrapper of the repeated pattern needs to be located by collapsing more square occurrences on the right of the mismatching point.

*Wrapper Generalization.* Finally, the wrapper can be generalized by inserting the new *Plus* node in its abstract syntax tree.

7.5. ALGORITHM `match`. Our generalization procedure, called `match`, receives  $n$  sample strings  $enc(I_1), enc(I_2), \dots, enc(I_n)$ , which are supposed to be encodings of instances of a nested type according to some prefix mark-up encoding  $enc$ . It works as follows: (i) it takes  $enc(I_1)$  as a first version  $w_1$  of the wrapper; (ii) then, it progressively matches each of the samples with the current wrapper in order to produce a new wrapper; to do this, it uses a binary matching function, which we also call `match`, as follows:

$$\begin{aligned} w_1 &= enc(I_1) \\ w_2 &= match(w_1, enc(I_2)) \\ &\dots \\ w_n &= match(w_{n-1}, enc(I_n)) \end{aligned}$$

We define  $match(enc(I_1), enc(I_2), \dots, enc(I_n)) = w_n$ . It can be seen that all of the complexity of the matching stands in the binary algorithm  $match(w, s)$  which is used to progressively match the current wrapper with each new sample.

Please note that, as informally discussed in the previous Section, `match` is inherently recursive; this means that we can identify two different cases in the use of `match`: in some cases `match` works on the current wrapper (a regular expression) and sample (a string); however, in other cases it works on two inner portions of either the wrapper or the sample; when it works to match portions of the wrapper it tries to match two regular expressions with each other. In order to generalize these two cases, we define `match` as a function that takes as input two abstract syntax trees representing encodings of two templates (the second of which is possibly simply

```

boolean match(AST regions  $w$ ,  $w'$ , direction  $d \in \{\rightarrow, \leftarrow\}$ )
begin
  Let  $samples$  be charSample( $w'$ );
  for each  $s \in samples$  do
     $w = solveMismatches(w, s, d)$ ;
    save  $parse.getLastMatchingNode()$  as  $LastMatchingNode$ ;
    save  $parse.getLastMatchingToken()$  as  $LastMatchingToken$ ;
    save  $w$  as  $Result$ ;
    return ( $w \neq \perp$ );
  end

AST solveMismatches(AST region  $w$ , sample  $s$ , direction  $d$ )
begin
  while (not  $parse(w, s, d)$ ) do
    Let  $m$  be  $parse.getMismatch()$ ;
     $w = applyOperator(m)$ ;
  end
  return  $w$ ;
end

AST applyOperator(mismatch  $m=(w, s, (n, t))$ , direction  $d$ )
begin
  Let  $op$  be the first operator defined on  $m$ 
  amongst ( $addPCDATA$ ,  $addPlus^w$ ,  $addPlus^s$ ,  $addHook^w$ ,  $addHook^s$ );
  if (none operator is defined) return  $\perp$ ;
  else return  $op(m, d)$ ;
end

```

FIG. 12. The algorithm match.

the encoding of an instance), and returns a new abstract syntax tree that generalizes the inputs. Figure 12 shows the pseudo-code of algorithm match.

Initially, the second template encoding is represented by a characteristic sample of the corresponding prefix mark-up language (Definition 5.2). This is generated by function  $charSample(w')$ . Such characteristic sample is composed of several (possibly one) instance encodings, each of which is iteratively matched with the wrapper by function  $solveMismatches$ . This function essentially implements the search in the state space of Figure 9. The selection of which operator to apply is demanded to  $applyOperator$ , which receives a mismatch and chooses the right operator.

Mismatches are raised during the parsing performed by function  $parse$  (whose pseudo-code is reported in Figure 13). The  $parse$  algorithm receives a wrapper and a sample represented as abstract syntax trees. It matches the wrapper with the sample by visiting the trees. If they do not match, it returns the mismatch point that prevented the parsing from succeeding. The parsing is *bidirectional* in the sense that it can be performed in both directions.<sup>9</sup>

<sup>9</sup> In Figure 13, collections are iterated over in both directions using enumerators, in the spirit of `java.util.Iterator` and `java.util.Enumeration`.

<pre> boolean parse(AST region <math>w</math>, sample <math>s</math>, <math>d \in \{\rightarrow, \leftarrow\}</math>) begin   Let <math>m</math> be the current mismatch;   Let <math>wEn</math> be an Enumerator over <math>w</math>'s nodes in the direction <math>d</math>;   Let <math>sEn</math> be an Enumerator over <math>s</math> in the direction <math>d</math>;   return (align(<math>wEn, sEn</math>) and not <math>wEn.hasNext()</math>); end </pre>	
<pre> boolean visit(<math>n:Token</math>, Enumerator <math>en</math>) begin   Let <math>t = en.next()</math>;   if (<math>n</math> matches <math>t</math>) return true;   else begin     save (<math>w, s, (n, t)</math>) as mismatch <math>m</math>;     return false;   end end </pre>	<pre> boolean visit(<math>n:PCDATA</math>, Enumerator <math>en</math>) begin   Let <math>t = en.next()</math>;   if (<math>t</math> is a data token) return true;   else begin     save (<math>w, s, (n, t)</math>) as mismatch <math>m</math>;     return false;   end end </pre>
<pre> boolean visit(<math>n:Hook</math>, Enumerator <math>en</math>) begin   visitChildren(<math>m, en</math>);   return true; end </pre>	<pre> boolean visit(<math>And(t_1, \dots, t_h)</math>, Enumerator <math>sEn</math>) begin   Let <math>oldIndex</math> be the position of <math>sEn</math> over <math>s</math>;   Let <math>wEn</math> be an Enumerator over <math>t_1, \dots, t_h</math>   in the same direction as <math>sEn</math>;   if (not align(<math>wEn, sEn</math>)) begin     roll back <math>sEn</math> to <math>oldIndex</math>;     return false;   end   return (not <math>wEn.hasNext()</math>); end </pre>
<pre> boolean visit(<math>n:Plus</math>, Enumerator <math>en</math>) begin   Let <math>result</math> be false;   while (visitChildren(<math>n, en</math>)) do     <math>result = true</math>;     discard current mismatch <math>m</math>;   end end </pre>	<pre> boolean align(Enumerators <math>wEn, sEn</math>) begin   while (<math>wEn.hasNext()</math> and <math>sEn.hasNext()</math>) do     if (not visit(<math>wEn.next(), sEn</math>)) return false;   end   return true; end </pre>
<pre> getLastMatchingToken() {returns the last matching token of the sample} getLastMatchingNode() {returns the last matching node of the wrapper} getMismatch() {returns mismatch <math>m</math>} </pre>	

FIG. 13. Algorithm for matching a wrapper and a sample.

## 8. Correctness and Complexity

It is possible to prove the following fundamental result about the algorithm.

**THEOREM 8.1 (CORRECTNESS).** *Given a set of strings, called  $\{enc(I_1), enc(I_2), \dots, enc(I_n)\}$ , of a rich set of instances of a type  $\sigma$  according to a prefix mark-up encoding  $enc$ , then:*

$$match(enc(I_1), enc(I_2), \dots, enc(I_n)) = enc(\sigma).$$

Since the proof is quite long and requires the introduction of a number of technical notions, for readability reasons we have moved it to Appendix B. From Theorem 8.1 it immediately follows that:

**COROLLARY 8.2.** *Any encoding of a rich set of instances of a type  $\sigma$  according to a prefix mark-up encoding  $enc$  is a characteristic sample for  $enc(\sigma)$ .*

Theorem 8.1 and Corollary 8.2 represent the main contributions of this article. In essence, they suggest that algorithm *match* can be effectively used for information extraction purposes on the Web.

In fact, on the one side, the algorithm is totally unsupervised; this means that, given a class of HTML pages that comply with a prefix mark-up grammar, *match* can infer the proper wrapper by simply looking at the pages, without needing any training or labeling phase; this greatly simplifies the maintenance task: in case, after the wrapper has been generated, some change in the HTML code does prevent it from working properly, to fix the wrapper it suffices to run *match* again in order to rebuild the wrapper.

On the other side, the new notion of characteristic sample—that is, the notion of rich set—is statistically much more probable than the traditional one; in fact, rich sets do not need to be made of strings of minimal length, and this significantly augments the probability of finding a rich set in a collection of random samples.

Finally, we can prove that algorithm *match* runs in polynomial time.

**THEOREM 8.3 (COMPLEXITY).** *Given two templates  $T$  and  $S$  subsumed by a common type  $\sigma$ , and a prefix mark-up encodings  $enc$ ,  $match(enc(T), enc(S))$  runs in polynomial time with respect to the maximum length of the input encodings.*

The proof is in Appendix B, and is given along with the proof of Theorem 8.1.

## 9. Implementation and Experiments

To validate the algorithm *match* described above, we have developed a prototype of the wrapper generation system and used it to run a number of experiments on HTML sites. The system has been completely written in Java.

Before feeding samples to our algorithm, we have run a preliminary cleaning step, to fix errors and make the code compliant with XHTML; this step is based on *JTidy*,<sup>10</sup> a Java library for HTML cleaning. All experiments have been conducted on a machine equipped with an Intel Pentium III processor working at 450 MHz, with 128 MBytes of RAM, running Linux (kernel 2.4) and Sun Java Development Kit 1.4.

To perform our experiments, we have selected several classes of pages from real-life Websites. We report our results in Figure 14. Let us first discuss Table A in Figure 14, which refers to experiments we have conducted independently. Then, we present some comparison with other information extraction systems for which experimental results are available in the literature, namely Wien [Kushmerick et al. 1997; Kushmerick 2000a] and Stalker [Muslea et al. 1999, 2001], two wrapper generation systems based on a machine learning approach (Table B).

Table A in Figure 14 reports a list of results relative to several well known data-intensive Websites. For each class we have downloaded a number of samples (usually between 10 and 20) and let the algorithm run on the samples in a fully unsupervised way. The samples were selected with the requirement that the pages obey a prefix mark-up grammar. The table contains the following elements: (i) *class*: a short description of each class, and the number of samples considered for that

---

<sup>10</sup><http://www.w3.org/People/Raggett/tidy/>.

Table A

classes				results				
n.	site	description	#s	time	nest	pcd	opt	lists
1	buy.com	product subcategories	20	1"107ms	2	16	0	4
2	buy.com	product information	10	0"735ms	1	14	3	2
3	rpmfind.net	packages by name	30	4"827ms	3	5	2	3
4	rpmfind.net	packages by distribution	20	1"963ms	2	8	1	3
5	uefa.com	clubs by country	20	0"434ms	1	5	2	1
6	uefa.com	players in the national team	20	0"260ms	2	2	1	2

Table B

n.	site		schema			comparative results		
	name (URL)	#s	nest	opt	ord	ROADRUNNER	Wien	Stalker
7	Okra (discontinued)	20	1	no	no	√	0"0"700ms	√
8	La Weekly (laweekly.com)	28	1	yes	no	√	0"0"391ms	no
9	Address Finder (iaf.net)	10	1	yes	yes	no		no

FIG. 14. Experimental results.

class; (ii) *results*: some elements about the results obtained, and in particular about the schema of the extracted dataset, namely: level of nesting (*nest*), number of attributes (*pcd*), number of optionals (*opt*) and number of lists (*lists*). In all cases the system was able to correctly infer the correct grammar. Computing times are generally in the order of a few seconds; our experience also shows that the matching usually converges after examining a small number of samples (i.e., after the first few matchings—usually less than 5—the wrapper remains unchanged).

One interesting issue is related to the actual coverage that prefix mark-up languages give when applied to websites. Note that Kushmerick in his paper about the wrapper generation system Wien [Kushmerick 2000a] proposes various classes of grammars for information extraction and reports a survey of coverage results for these grammars based on a collection of randomly selected resources. These results were recently updated in Muslea et al. [2001], where a comparison between Wien and Stalker is reported.

In order to make a comparison, we downloaded from RISE,<sup>11</sup> a repository of information sources from data extraction projects, the dataset used in Kushmerick [2000a] and Muslea et al. [2001] and conducted similar experiments using our system. Note, however, that Wien and Stalker are essentially machine learning systems, and therefore rely on a preliminary training phase based on manually annotated samples; as a consequence, our experimental methodology is quite different from theirs. More specifically, our experiments were conducted as follows: (i) all tests were run in a fully automatic way; (ii) we ran our algorithm on all the available samples of each class; (iii) we considered a test successful if the algorithm was able to infer a single prefix mark-up grammar for all the samples, and a wrapper capable of extracting data from 100% of the samples; a test was considered failed otherwise. Also, we did not compare the wrapper inferred by our system with those learned by Wien and Stalker.<sup>12</sup> In essence, we considered this mainly as a coverage test, that is, we aimed at finding how many of the original Wien sites obeyed a

<sup>11</sup> <http://www.isi.edu/~muslea/RISE>.

<sup>12</sup> On the one side we had not enough information available to do this comparison, and on the other side the two frameworks are too different to give a meaningful report.

prefix mark-up grammar, and in how many cases we could find a rich set in the random samples available.

The results were as follows: (i) for 16 sites, the algorithm was actually able to infer a prefix mark-up grammar for the pages, derive a wrapper and use it to gather information;<sup>13</sup> (ii) for 13 sites, the algorithm was unable to derive a single prefix mark-up grammar for the pages, and therefore generated no wrapper; (iii) in one single case<sup>14</sup> our tools were unable to generate well-formed XHTML code from the original HTML, since it contained some unrecoverable errors; therefore, we were unable to run the algorithm on the pages.

Overall, the prefix mark-up learning algorithm was able to wrap 55% of the sites. In all cases the algorithm was very efficient, running in less than a few seconds. Corresponding results reported in Muslea et al. [2001] for the other two systems are as follows: (i) using its six different grammar classes, Wien was able to wrap 18 of the 30 sites, and failed on 12, with an overall percentage of 60% (in Kushmerick [2000a] the overall coverage of the union of all six grammar classes was reported to be 70%). (ii) Stalker had perfect extraction results on 20 sites; on 8 sites it generated imperfect rules (of which 4 of high quality); it failed on 2 sites.

Table B in Figure 14 reports a more detailed comparison of results between the three wrapper generators based on other selected sites. Also in this case, the test samples have been downloaded from RISE. Table B contains the following elements: (i) *site* from which the pages were taken, and number of samples; (ii) description of the target *schema*, that is, level of nesting (*nest*), whether the pages contain optional elements (*opt*), and whether attributes may occur in different orders (*ord*); (iii) *results*: results obtained by the three systems, with computing times; times for Wien and Stalker refer to CPU times used during the learning.

A few things are worth noting here with respect to the expressive power of the various systems. (i) While Wien and Stalker generated their wrappers by examining a number of labeled examples, and therefore the systems had a precise knowledge of the target schema, ROADRUNNER did not have any a priori knowledge about the organization of the pages. (ii) Even considering the radical differences in the three approaches, and that computing times refer to different machines, by comparing the times needed by ROADRUNNER with those reported in the literature [Kushmerick 2000a; Muslea et al. 1999, 2001], it appears that inferring the grammar takes considerably less time than training Wien and Stalker. (iii) Differently from ROADRUNNER and Stalker, Wien is unable to handle optional fields, and therefore fails on samples 8 and 9. (iv) Stalker has considerably more expressive power since it can handle disjunctive patterns; this allows for treating attributes that appear in various orders, like in Address Finder (10); being limited to union-free patterns, ROADRUNNER fails in cases like this.

## 10. Discussion and Conclusions

This article reconsiders grammar inference techniques in the new context of information extraction, and tries to bridge the gap between these two fields; it shows that

---

<sup>13</sup> Sources n. 2, 3, 4, 5, 10, 13, 14, 15, 16, 19, 20, 22, 25, 26, 27, 30.

<sup>14</sup> Source n. 28.

the formal setting of traditional grammar inference can be reused, provided that the central notion of a characteristic sample is revisited. In our opinion, the main contributions can be summarized as follows:

- we introduce a new, database theoretic notion of a characteristic sample, which has considerably higher probability of being found in a collection of randomly sampled HTML pages;
- we define and formally study a new class of regular grammars, called prefix mark-up; we believe that this class represents an interesting compromise between expressibility and complexity; in fact, the hypotheses assumed in the grammar allow for efficient inference in the limit;<sup>15</sup>
- finally, we develop a polynomial algorithm for inferring prefix mark-up languages; the algorithm is fully unsupervised, that is, it does not require any form of training.

We therefore believe that this work lays the foundation for the development of new and more effective techniques aimed at extracting information from the Web. However, despite these promising results, the overall problem is far from being solved. In fact, it is possible to see that the technique proposed in this article still has a number of limitations.

First, it is clearly targeted at sites with fairly structured pages; these sites are becoming more and more frequent, but there is still a wealth of information available in less structured formats; also, we assume that the HTML code is either well formed, or can be made well formed using known tools like JTidy; this latter hypothesis is not always the case, as shown by our coverage test, in which one of the 30 Wien sites could not be handled since JTidy was unable to clean the HTML.

Second, the class of prefix mark-up languages makes some assumptions on the way data items are delimited in the page; more specifically, we assume that delimiters of different non-terminals are sufficiently different from one another to eliminate ambiguities among grammar productions; this is a less strong hypothesis than it may seem, since our schema alphabet is very rich: in fact, when we compare delimiting tags we not only take into account the tag name (es: `br` or `td`), but also the possible attributes (es: `class="left"` or `bgcolor="white"`), and, more important, the depth in the DOM tree (es: in the fragment `<b><b></b></b>` the first and second `<b>` are considered as different symbols, since they have equal name but different depth in the DOM); still, as shown in the coverage tests, only approximately the 50% of the sites obeyed a prefix markup grammar.

Finally, the wrappers produced by our algorithm need a post-processing phase; this is needed to perform at least two tasks: (i) annotate with more semantic labels the extracted attributes, which are initially anonymous; (ii) inspect the target schema associated with the wrapper and somehow correct it; in fact, the generated schema may in some cases differ from the expected one; to see one example of this, consider that RoadRunner generates a grammar for the *whole* HTML pages—from the first

---

<sup>15</sup>In previous papers and demonstrations [Crescenzi et al. 2001, 2002] the inference algorithm was reported to be exponential; however, the algorithm described in those papers was not conceived for prefix mark-up languages. It incorporated several extensions: for example, it was able to handle tagging functions in which tags are made both of tags and constant strings and had been complemented with heuristics for dealing with less regular data.

<html> tag to the last </html> tag; as a consequence, it often extracts varying portions of a class of pages—like commercials, banners, navigation links—that might be considered as irrelevant.

It is easy to see that these limitations do not usually arise in machine learning systems. These systems can more easily be used to deal with less-structured sources, ideally also plain ASCII files; moreover, the presence of the user-assisted training phase also solves the problem of identifying and labeling the relevant pieces of information in a page.

To conclude, our future research will be devoted to build on this work, in order to overcome the limitations discussed above. We have identified two main directions of investigation: (i) extending the class of grammars with some controlled form of disjunction; (ii) developing automatic techniques for labeling attributes in a page. Some preliminary results in this respect are reported in Arlotta et al. [2003].

## 11. Related Works

Early approaches to structuring and wrapping websites were based on manual techniques. Different approaches have been pursued to this end: they range from the adoption of specialized procedural languages [Atzeni and Mecca 1997], to the definition of declarative specification languages that work on the text [Hammer et al. 1997; Gupta et al. 1998], or on GUI-based tools [Sahuguet and Azavant 1999; Liu et al. 2000; Baumgartner et al. 2001] that help the user to rapidly prototype a wrapper. Another interesting research direction has been concerned with the extension of context free or regular grammars in order to make them more flexible and better suited to this task [Crescenzi and Mecca 1998; Huck et al. 1998].

The first attempts to automate the wrapper generation process heavily relied on the use of heuristics. For example, Ashish and Knoblock [1997], develop a practical approach to identify attributes in a HTML page; the technique is based on the identification of specific formatting tags (like the ones for headings, boldface, italics etc.) in order to recognize semantically relevant portions of a page.

More recently, other proposals have attacked the problem of automating the wrapper generation process under a machine learning perspective. Some of these works concentrate on free-text [Freitag 1998; Soderland 1999] available on the Web, others on fairly structured HTML pages.

One example in this category is Wien [Kushmerick et al. 1997; Kushmerick 2000a]. The authors develop a machine-learning approach to wrapper induction. The starting point for these work is the identification of several simple classes of wrapper-specification formalisms, that are easily learnable and yet sufficiently expressive. Wrappers are developed for multiple-record HTML documents. Positive examples consist of occurrences of a target attribute in the page—that is, occurrences of a country name in a page listing country names and codes. These examples may be either fed to the system by the user or by some specialized *extractors*. Roughly speaking, the formalisms studied in the article allow us to specify the wrapper by associating a left and right delimiting string with each relevant attribute in the page. Results are established about the relative expressiveness of the various formalisms, and the complexity of the learning. The goal is to restrict the wrapper-specification formalisms in such a way to reduce the complexity of the learning; this obviously has a trade-off in terms of expressiveness; in this respect,

Kushmerick [2000a] surveys 30 sites from different domains, 70% of which can be wrapped using the formalisms discussed in the article.

A similar approach is pursued in Stalker [Muslea et al. 1999, 2001] and SoftMealy [Hsu and Dung 1998]. Since these approaches also allow for disjunctions, missing attributes, and can handle various permutations in the order of attributes, they are strictly more expressive than Kushmerick et al. [1997] and Kushmerick [2000a]. Given an arbitrarily nested, multiple-record document, Stalker relies on a hierarchical description of the page content, corresponding to the nesting of lists of records inside the page (i.e., a list of restaurants, with name and address and a list of dishes, with prices, etc.). The wrapper-specification formalism consists in annotating each node in this hierarchy with a *landmark*, that is, a simple regular expression that can be used to locate occurrences of that attribute inside the page. The actual wrapper is a deterministic finite state automaton that, based on the landmarks, implements the extraction rules needed to wrap the page. The authors discuss how these automaton and their extraction rules may be efficiently learned based on few examples provided by the user. Experimental results reported in the paper confirm the augmented expressibility of the formalism.

Another example of wrapper-generating systems from labeled examples comes from Embley et al. [1999]. These works have a conceptual-modeling background, and base the data-extraction process on the use of domain-specific ontologies. The target pages are multiple-record HTML pages coming from specific domains—like, for example, car ads or movie reviews—for which an ontology is available. The ontology provides a concise description of the conceptual model of data in the page and also allows for recognizing attribute occurrences in the text. It thus allows for labeling a number of examples in the target page, and trying to infer the wrapper on those occurrences. An interesting contribution of this research is that the ontology can be used to infer wrappers around different sites of the same domain, making them, in some sense, also more resilient to changes in the target site. Experimental results on real-life HTML pages are reported in the article to show the effectiveness of this approach.

One final approach to wrapper generation from labeled examples is represented by NoDoSe [Adelberg 1998]. In this work, the wrapper is derived by making the system interact with the user through a graphical interface. The user manually starts the semi-automatic structuring phase by defining the logical schema in the page in some data model; differently from Soderland [1999], Hsu and Dung [1998], and Embley et al. [1999], which concentrate on flat records only, NoDoSe uses a rich object-oriented data model that can handle arbitrarily nested tables; then, s/he labels a few occurrences of the relevant attributes in a page, and then asks the system to generalize those examples and infer the wrapper; if the system fails, it will stop and ask for more examples to refine the patterns. With respect to the system described in this article, NoDoSe adopts a richer class of extraction rules, which for example can specify that occurrences of one attribute start after a given pattern, or at a given offset in the line. In fact, NoDoSe is not specifically targeted at HTML, and can be used to write wrappers on a much wider class of textual documents, including formatted ASCII files. The availability of user inputs during the wrapper generation process helps to handle the increased complexity due to the more expressive formalism. A similar approach has also been used in Ribeiro-Neto et al. [1999].

Recently, some works have appeared that use grammar inference techniques for information extraction, in the spirit of this article. For example, Fernau [2000a]

concentrates on XML documents, which do not suffer from the high ambiguity of HTML pages, and therefore are suitable for easier inference. Some other works try to apply grammar inference techniques to information extraction from HTML code. For example, Chidlovskii [2000] defines a wrapper-generation algorithm based on the inference of  $k$ -reversible grammars; however, its approach is not fully automatic, and suffers from some of the limitations of traditional grammar inference techniques discussed earlier in the paper. Hong and Clark [2001] use stochastic context-free grammars to infer wrappers for Web sources; their approach is based on domain specific knowledge provided to the wrapper generator. Another related work is Kosala et al. [2002]. In that paper, tree automata are used to infer tree languages for HTML pages; also, in this case, a preliminary annotation phase is used to feed the grammar generator with samples.

## Appendixes

### A. Proof: Identifiability in the Limit of Prefix Mark-Up Encodings

Let us first show that Theorem 5.7 holds, that is, that  $f_\pi$  is an extended distinguishing function.

Consider Figure 15(A)–(C), which show the “building blocks” needed to represent type nodes in the automata of prefix mark-up languages:

- (A) basic node  $U$  whose encoding is  $start(\alpha)\Delta^+end(\alpha)$ ;
- (B) optional node  $(T)?$  whose encoding is  $start(\alpha)(start(\beta)enc(T)end(\beta))?end(\alpha)$ ;
- (C) list node  $(T)^+$  whose encoding is  $start(\alpha)(start(\beta)enc(T)end(\beta))^+end(\alpha)$ .

Let us note that any prefix mark-up encoding  $enc$  of a template  $T$  can easily be transformed into an automaton recognizing  $L(enc(T))$  by properly nesting and concatenating these elementary automata. Then observe that any automaton obtained in this way is deterministic because symbols of  $start(\beta)$  do not occur in  $end(\alpha)$  by definition of prefix mark-up encoding. Therefore, such an automaton represents the canonical automaton for the prefix mark-up language. Recall that the canonical automaton of a regular language is the minimal deterministic automaton for the language.

We now introduce the following lemmas.

LEMMA A.1. *Function  $f_\pi$  is such that for all strings  $w, u, z \in (\Sigma \cup \bar{\Sigma} \cup \Delta)^*$ ,  $f_\pi(w) = f_\pi(u)$  implies that  $f_\pi(wz) = f_\pi(uz)$ .*

PROOF. We shall prove the claim by induction on the length of  $z$ .

*Basis Case:*  $|z| = 0$ ; The claim holds.

*Induction:* Suppose now  $z = z'a$ , with  $|z'| = n - 1$ ; we know that  $f_\pi(wz') = f_\pi(uz')$ ; to show that  $f_\pi(wz) = f_\pi(uz)$ , let us consider the two different cases: (i)  $a \in \Sigma$ ; In this case, we have that  $f_\pi(wz) = f_\pi(wz'a) = \rho(wz'a)$ ; similarly:  $f_\pi(uz) = f_\pi(uz'a) = \rho(uz'a)$ ; but we know that  $f_\pi(wz') = f_\pi(uz') \Rightarrow \rho(wz') = \rho(uz')$ , and this proves the claim. (ii)  $a \in \Delta$ ; In this case, we have that  $f_\pi(wz) = f_\pi(wz'a) = \rho(wz'a)$ ; similarly:  $f_\pi(uz) = f_\pi(uz'a) = \rho(uz'a)$ ; since  $\rho$  is such that  $\rho(wz'a) = \rho(\rho(wz')a)$ , and  $\rho(uz'a) = \rho(\rho(uz')a)$  and we know that  $\rho(wz') = \rho(uz')$ , also in this case the claim holds.  $\square$

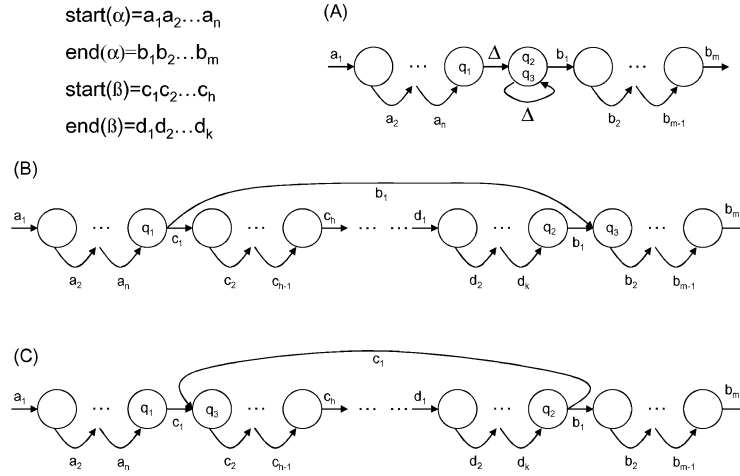


FIG. 15. Building blocks of the canonical automaton of prefix mark-up languages.

LEMMA A.2. *Given a prefix mark-up language  $L$ , let  $A = (Q, T, \delta, q_0, Q_F)$  be the canonical automaton of  $L$ . Then, function  $f_\pi$  is such that for all states  $q \in Q$  and all  $x, y \in T^*$  with  $\delta^*(q_0, x) = \delta^*(q_0, y)$ , we have  $f(x) = f(y)$ .*

PROOF. Let us prove this by induction. Since  $x$  and  $y$  lead to some state  $q$  in the automaton  $A$ , they need to be prefixes of some strings in  $L$ . Our induction will be on the maximum length of a prefix of a string in  $L$ .

*Base Case:* For prefixes of maximum length 0, the claim holds trivially (there is only one initial state).

*Induction:* Suppose now the claim holds for prefixes up to length  $n$ ; we need to show that it also holds for prefixes up to length  $n + 1$ . Consider two prefixes  $x, y$  of length at most  $n + 1$ , and assume  $\delta^*(q_0, x) = \delta^*(q_0, y)$ .

By the inductive hypothesis and Lemma A.1, the claim holds for states that have only one incoming transition. We will therefore concentrate only on states with more than one incoming transition. By looking at Figure 15, it is possible to see that there are three different cases in which different prefixes lead to the same state.

- (1) There is a *PCDATA* in the language (Figure 15(A), state  $q_3$ ); by the inductive hypothesis, we know that  $f_\pi$  is well defined in state  $q_1$ ; the two prefixes  $x$  and  $y$  that lead to state  $q_3$  can be written as follows:  $x_1 \cdot d, d \in \Delta$ , and  $y_1 \cdot d_1, \dots, d_n, d_1, d_2, \dots, d_n \in \Delta$ , with  $x_1$  and  $y_1$  of length at most  $n$ ; we know that  $f_\pi(x_1) = f_\pi(y_1)$ ; therefore, we have:  $f_\pi(y) = f_\pi(y_1 \cdot d_1, \dots, d_n) = \rho(y_1 \cdot d_1, \dots, d_n) = \rho(y_1) \cdot \Delta = f_\pi(y_1) \cdot \Delta = f_\pi(x_1) \cdot \Delta = \rho(x_1 \cdot d) = f_\pi(x_1 \cdot d) = f_\pi(x)$ ;
- (2) There is a *hook* in the language (Figure 15(B), state  $q_3$ ); by the inductive hypothesis, we know that  $f_\pi$  is well defined in state  $q_1$ ; the two prefixes  $x$  and  $y$  that lead to state  $q_3$  can be written as follows:  $x_1 \cdot b_1$ , and  $y_1 \cdot \text{start}(\beta), \dots, \text{end}(\beta) \cdot b_1$ ; also in this case we know that  $f_\pi(x_1) = f_\pi(y_1)$ ; therefore we have:  $f_\pi(y) = f_\pi(y_1 \cdot \text{start}(\beta) \cdot \dots \cdot \text{end}(\beta) \cdot b_1) = \rho(y_1 \cdot \text{start}(\beta) \cdot \dots \cdot \text{end}(\beta)) \cdot b_1 = \rho(y_1) \cdot b_1 = f_\pi(y_1) \cdot b_1 = f_\pi(x_1) \cdot b_1 = f_\pi(x_1 \cdot b_1) = f_\pi(x)$ ;
- (3) There is a *plus* in the language (Figure 15(C), state  $q_1$ ); in this case, the claim can be shown similarly to case (2) above.  $\square$

We are now ready to give the proof of Theorem 5.7 holds, that is, that  $f_\pi$  is an extended distinguishing function.

PROOF. To show this, we first need to show that for all strings  $w, u, z$ ,  $f_\pi(w) = f_\pi(u)$  implies that  $f_\pi(wz) = f_\pi(uz)$ . But this is stated in Lemma A.1.

Second, we need to show that the set of strings  $\{f(u) | uw \in L\}$  is a finite set. But this follows immediately from Lemma A.2, since  $f_\pi$  can only assume as many values as the states in the automaton of  $L$ .  $\square$

In order to prove identifiability in the limit, we also need the following Lemma.

LEMMA A.3. *Given a prefix mark-up language  $L$ , let  $A = (Q, T, \delta, q_0, Q_F)$  be the automaton of  $L$ . Then,  $A$  is an extended  $f_\pi$ -distinguishable automaton.*

PROOF. Consider Definition 5.4. We have already shown that the automaton in Figure 15 is deterministic, and therefore satisfies condition 1 in the definition. Condition 2 is stated in Lemma A.2. Condition 3 must be checked on the states marked with  $q_1, q_2$ , and  $q_3$  in Figure 15. It must be the case that  $f_\pi(q_1) \neq f_\pi(q_2)$ . In fact, by definition of  $f_\pi$ , in the cases of Figure 15(B)–(C),  $f_\pi(q_2)$  ends with  $a_n$  and  $f_\pi(q_1)$  ends with  $d_k$ . Since they occur respectively in  $start(\alpha)$  and in  $end(\beta)$  this is sufficient to prove that  $f_\pi(q_1) \neq f_\pi(q_2)$ . In the case of Figure 15(A),  $f_\pi(q_2)$  ends with a symbol of  $T$ , while  $f_\pi(q_1)$  ends with  $\Delta$ .  $\square$

We are now ready to prove Theorem 5.8, that is, that the class of prefix mark-up languages is identifiable in the limit.

PROOF. The proof is based on Theorem 8 in Fernau [2000b] (also in Fernau [2003]). The only difference between our setting and the setting of Fernau [2003] is that our extended  $f$ -distinguishable function does not need to have a finite codomain on  $T^*$ , while  $f$ -distinguishable functions do. As a consequence, consider the *canonical automaton*  $A_f$  of an extended distinguishing function  $f$ , defined as:  $A_f = (F, T, \delta_f, f(\epsilon), F)$ , where: (i)  $F$  is the codomain of  $f$  over  $T^*$ , and (ii)  $\delta_f(q, a) = f(w \cdot a)$ , for any string  $w \in T^*$  such that  $f(w) = q$ . It can be seen that  $A_f$  has in general an infinite number of states.

However, for every prefix mark-up language  $L$ , by definition of extended distinguishing function, we know that  $f_\pi$  has a finite set of values over prefixes of  $L$ . As a consequence, it can be seen that the *f-canonical automaton*  $A(L, f_\pi)$  [Fernau 2000b, Definition 4] is a finite one. Therefore, the proof in Theorem 8 in Fernau [2000b] still holds for extended  $f$ -distinguishable languages.  $\square$

### B. Proof: Correctness and Complexity of the Matching Algorithm

In order to give a proof of the correctness Theorem, we need to introduce a number of preliminary definitions.

We call an *instance node* any list-instance, optional-instance or constant node in a template. Any list, optional or basic node is called a *type node*.

We denote by  $\mathcal{T}$  the universe of all templates. Recall that we say that two templates  $T_1, T_2$  are *homogeneous* if there exist a template  $T \in \mathcal{T}$  such that  $T_1 \preceq T$  and  $T_2 \preceq T$ . It is now easy to see that, for each maximal set of homogeneous templates,  $\mathcal{T}_H$ ,  $(\mathcal{T}_H, \preceq)$  is a join-semilattice. Recall that an ordered set  $(X, \preceq)$  is a join-semilattice if every couple of elements of  $X$  admits a least upper bound.

Given a template  $T$ , we denote by  $\mathcal{H}(T)$  the class of all templates homogeneous to  $T$ . In the following, unless explicitly specified, we will always refer to join-semilattices of homogeneous templates. Given a finite collection  $S$  of homogeneous templates,  $\text{LUB}(S)$  will denote the least upper bound of elements in  $S$  in the corresponding join-semilattice.

Consider the relationship between a type and its instances. Since a type subsumes all of its instances, given a set of instances  $\mathcal{I} = \{I_1, \dots, I_n\}$  of some type  $\sigma$ ,  $\sigma$  is a common upper bound of  $\mathcal{I}$  in the join-semilattice of templates homogeneous to  $\sigma$ ,  $\mathcal{H}(\sigma)$ . More specifically, we have the following lemma.

**LEMMA B.1.** *Given a set of instances  $\mathcal{I} = \{I_1, \dots, I_n\}$  of type  $\sigma$ ,  $\mathcal{I}$  is rich for  $\sigma$  iff  $\sigma = \text{LUB}(\mathcal{I})$ .*

**PROOF.** Let us prove the two directions separately.

$\mathcal{I}$  is rich for  $\sigma \Rightarrow \sigma = \text{LUB}(\mathcal{I})$ . The proof is by contradiction. Assume there is some  $\sigma' = \text{LUB}(\mathcal{I})$ . Assume that  $\sigma' \neq \sigma$ . Still,  $\sigma'$  belongs to  $\mathcal{H}(\sigma)$ , the class of templates homogeneous to  $\sigma$ . Therefore, it must be the case that  $\sigma' \preceq \sigma$ . Since  $\sigma$  is a fully instantiated type,  $\sigma'$  cannot be fully instantiated; thus, it must exist at least one instance node; assume the node is labeled  $\alpha$ . Let us also assume that it is a list-instance node of cardinality  $k$ , but the same discussion would hold for any other kind of instance node. In that case, since  $I_i \preceq \sigma'$  for any  $i$ , nodes labeled  $\alpha$  in  $\mathcal{I}$  all have the same cardinality  $k$ , so  $\mathcal{I}$  cannot be rich with respect to  $\sigma$ .

$\sigma = \text{LUB}(\mathcal{I}) \Rightarrow \mathcal{I}$  is rich for  $\sigma$ . Similarly, let us assume that  $\mathcal{I}$  is not rich for  $\sigma$ . Then, it must exist a type node for which  $\mathcal{I}$  is not rich. Let us assume that node is a list node labeled  $\alpha$ , but the same discussion would hold for any other type node. In that case all list-instance nodes labeled  $\alpha$  in  $\mathcal{I}$  have the same cardinality  $k$ . For each  $I_h$ ,  $h = 1 \dots n$ , let  $I_h^{\alpha_j}$  ( $j = 1 \dots k$ ) be the subtemplate of  $I_h$  rooted at the  $j$ th child of node labeled  $\alpha$ . We can build  $\sigma' \neq \sigma$  such that  $\sigma'$  is obtained from  $\sigma$  by replacing the subtree rooted at node  $\alpha$  with a list-instance node  $\text{LUB}(I_1^{\alpha_j}, \dots, I_n^{\alpha_j})$ . Observe that for each  $h = 1 \dots n$ , it is the case that  $I_h \preceq \sigma'$ ,  $\sigma \neq \sigma'$ , and  $\sigma' \preceq \sigma$ ; this contradicts the hypothesis that  $\sigma$  is  $\text{LUB}(\mathcal{I})$ .  $\square$

There is a close relationship between template subsumption,  $\preceq$ , and the familiar concept of containment,  $\subseteq$ , between regular expressions, as stated by the following theorem.

**LEMMA B.2.** *Given a type  $\sigma$ , and a prefix mark-up encoding  $\text{enc}$ , let us call  $\text{enc}(\mathcal{H}(\sigma))$  the image of  $\mathcal{H}(\sigma)$  according to  $\text{enc}$ . Then,  $(\text{enc}(\mathcal{H}(\sigma)), \subseteq)$  is a join-semilattice, and, for each set of templates  $T_1, \dots, T_n \in \mathcal{H}(\sigma)$  it is the case that  $\text{LUB}_{\subseteq}(\text{enc}(T_1), \dots, \text{enc}(T_n)) = \text{enc}(\text{LUB}_{\preceq}(T_1, \dots, T_n))$ .*

**PROOF.** We prove that  $(\text{enc}(\mathcal{H}(\sigma)), \subseteq)$  is a join semi-lattice by showing that  $T \preceq T' \Leftrightarrow \text{enc}(T) \subseteq \text{enc}(T')$ . Let us prove the two directions separately:

$T \preceq T' \Rightarrow \text{enc}(T) \subseteq \text{enc}(T')$ , Follows directly from Definitions 4.1, 4.2, and 4.6, respectively, of template, subsumption relationship, and well-formed mark-up encodings. Proposition 4.7 is just a special case of this assertion.

$\text{enc}(T) \subseteq \text{enc}(T') \Rightarrow T \preceq T'$ . We shall prove equivalently,  $T \not\preceq T' \Rightarrow \text{enc}(T) \not\subseteq \text{enc}(T')$ .

First, note that any regular expression  $enc(R)$  such that  $R \in \mathcal{H}(\sigma)$  is unambiguous [Bruggemann-Klein and Wood 1998]: every symbol occurring in a given string  $enc(I) \in L(enc(R))$  ( $I$  is an instance of  $\sigma$ ) can be unambiguously associated to the terminal symbol it unifies with. Therefore, we can unambiguously label every symbols  $x \in \Sigma \cup \bar{\Sigma} \cup \Delta$  of  $enc(I)$  with the corresponding type label  $\alpha$ . Finally, let us distinguish different terminal symbol occurrences: if the node labeled  $\alpha$  is not a leaf ( $x \in \Sigma \cup \bar{\Sigma}$ ), we denote by  $x_i$  the  $i$ th occurrence of  $x$  within  $start(\alpha)end(\alpha)$ ; if it is a leaf ( $x \in \Sigma \cup \bar{\Sigma} \cup \Delta$ ) we denote by  $x_i$  the  $i$ th occurrence within  $start(\alpha) a end(\alpha)$  where  $a \in \Delta^+$  is the encoding of corresponding constant subtemplate labeled  $\alpha$ .

We show the thesis by structural induction on the common type  $\sigma$ .

- $\sigma$  is a basic type. Trivial.
- $\sigma = (\sigma')?$  is an optional type and  $\sigma'$  is a nonnullable type. If  $T \not\leq T'$  there are four possibilities: (i)  $T = ({}_i S)?$  and  $T' = (S')?$  such that  $S \not\leq S'$ , the thesis follows directly from the inductive hypothesis because by Definition 4.6 of markup encodings of a template we have that  $L(enc(T)) = L(enc(S))$ ,  $L(enc(T')) = L(enc(S')) \cup \{enc(null)\}$  and  $enc(null) \notin L(enc(T))$ ; (ii)  $T = (S)?$  and  $T' = (S')?$  such that  $S \not\leq S'$ , the thesis follow from the case (i) by observing that from  $({}_i S)? \leq (S)?$  it follows that  $L(enc({}_i S?)) \subseteq L(enc((S)?))$  as proved above; (iii)  $T = ({}_i S)?$  and  $T' = ({}_i S')?$  such that  $S \not\leq S'$ , the thesis directly follows from the inductive hypothesis because  $L(enc(T)) = L(enc(S))$  and  $L(enc(T')) = L(enc(S'))$ ; (iv)  $T = (S)?$  and  $T' = ({}_i S')?$ , this case can be reduced to case (iii) exactly as done for case (ii) with respect to case (i);
- $\sigma = [\sigma_1, \dots, \sigma_n]$  is tuple type and  $\sigma_1 \cdots \sigma_n$  are nontuple types. If  $T \not\leq T'$ , it must exist a subtemplate  $T_h \leq \sigma_h$  of  $T$ , and a subtemplate  $T'_h \leq \sigma_h$  of  $T'$  such that  $T_h \not\leq T'_h$ . Observe that any string of  $L(enc(T))$  can be unambiguously decomposed in  $start(root)w_1w_2 \cdots w_n end(root)$  such that  $w_j \in L(enc(T_j))$ ,  $j = 1 \cdots n$ , and similarly any string of  $L(enc(T'))$  can be unambiguously decomposed in  $start(root)w'_1w'_2 \cdots w'_n end(root)$  such that  $w'_j \in L(enc(T'_j))$ ,  $j = 1 \cdots n$ . Consider  $w_h$  and  $w'_h$ : since by inductive hypothesis  $L(enc(T_h)) \not\subseteq L(enc(T'_h))$ , it follows the thesis.
- $\sigma = \langle \sigma' \rangle$  is a list type and  $\sigma'$  is a tuple type. There are several possibilities to consider: (i)  $T = \langle {}_i T_1, \dots, T_k \rangle$  and  $T' = \langle S' \rangle$  such that there exist a  $T_j \not\leq S'$ . Consider that any string of  $L(enc(T))$  can be unambiguously decomposed in  $start(root)w_1w_2 \cdots w_k end(root)$  such that  $w_j \in L(enc(T_j))$ ,  $j = 1 \cdots k$ . Similarly, any string of  $L(enc(T'))$  can be written as  $start(root)w'_1w'_2 \cdots w'_n end(root)$  where  $n > 0$  and  $w'_j \in L(enc(S'))$ ,  $j = 1 \cdots n$ . Let  $x_o$  be a symbol occurrence (labeled  $root.0$ ) of  $start(root.0)end(root.0)$ :  $x_o$  occurs exactly  $k$  times in any string of  $L(enc(T))$ . We have to show that there exist strings  $L(enc(T))$  that do not occur in  $L(enc(T'))$  even if  $n = k$ . In that case, according to the inductive hypothesis  $L(enc(T_j)) \not\subseteq L(enc(S'))$ , and it suffices to choose any string  $w_j \in L(enc(T_j))$  such that  $w_j \notin L(enc(S'))$  to prove the thesis. (ii)  $T = \langle S \rangle$  and  $T' = \langle S' \rangle$  such that  $S \leq S'$ . Consider a new template  $T'' = \langle {}_i S, \dots, S \rangle$ , that is, a list-instance of cardinality  $k$  such that  $T'' \leq T = \langle S \rangle$ . Then, it follows that  $L(enc(T'')) \subseteq L(enc(T'))$  and therefore this case can be reduced to case (i). (iii)  $T = \langle {}_i T_1, \dots, T_k \rangle$  and  $T' = \langle {}_i T'_1, \dots, T'_h \rangle$  such that  $k \neq h$ , consider any symbol occurrence  $x_o$  in  $start(root.0)end(root.0)$  labeled  $root.0$ . There are exactly  $k$  and  $h$  occurrences of  $x_o$  in any string of  $L(enc(T))$  and  $L(enc(T'))$

respectively. So it cannot be the case that  $L(\text{enc}(T)) \subseteq L(\text{enc}(T'))$ . (iv)  $T = \langle S \rangle$  and  $T' = \langle {}_i T'_1, \dots, T'_h \rangle$ , this case can be reduced to case (iii), exactly as done for case (ii) with respect to case (i). (v)  $T = \langle {}_i T_1, \dots, T_k \rangle$  and  $T' = \langle {}_i T'_1, \dots, T'_k \rangle$  such that for a given  $h$ ,  $T_h \not\subseteq T'_h$ . Observe that any string of  $L(\text{enc}(T))$  can be unambiguously decomposed in  $\text{start}(\text{root})w_1w_2 \cdots w_k \text{end}(\text{root})$  such that  $w_j \in L(\text{enc}(T_j))$ ,  $j = 1 \cdots k$ , and similarly any string of  $L(\text{enc}(T'))$  can be written as  $\text{start}(\text{root})w'_1w'_2 \cdots w'_k \text{end}(\text{root})$  such that  $w'_j \in L(\text{enc}(T'_j))$ ,  $j = 1 \cdots k$ . Since by inductive hypothesis  $L(\text{enc}(T_h)) \not\subseteq L(\text{enc}(T'_h))$ , it follows the thesis.  $\square$

Lemma B.2 and Proposition 5.1 (which states that a nested tuple type can be recovered in linear time from any of its encodings) suggest that we can solve the schema finding problem by working on strings and join-semilattices of regular expressions. Given a set of encoded instances of a nested tuple type  $\sigma$ ,  $\mathcal{E} = \{\text{enc}(I_1), \dots, \text{enc}(I_n)\}$  according to some mark-up encoding  $\text{enc}$ , the strategy to solve the schema finding problem is: (i) to find the regular expression encoding  $\sigma$  as the least upper bound  $e_\sigma = \text{LUB}_\subseteq(\mathcal{E})$ ; (ii) from the regular expression, to construct  $\sigma$ ; (iii) based on the grammar defined by  $e_\sigma$ , from each  $\text{enc}(I_j)$  to derive a representation of an instance  $I_j$  of  $\sigma$ .

Let us consider the case in which inputs are encoded using a *prefix* mark-up encoding function. Recall that, for every join-semilattice, the least upper bound operator is associative, and therefore, given a set of elements,  $\mathcal{E}$ , we can progressively compute the least upper bound of the set, independently of the order for elements in  $\mathcal{E}$ , based on the following iterative algorithm:

$$\begin{cases} \text{lub}_1 &= e_1 \\ \text{lub}_{i+1} &= \text{LUB}(\text{lub}_i, e_{i+1}), \quad \text{for } i = 2, \dots, k \end{cases}$$

The equations above suggest a strategy to solve the schema finding problem with a prefix mark-up encoding, assuming we know how to compute least upper bounds of two prefix mark-up languages.<sup>16</sup> This means that, in order to prove Theorem 8.1, it suffices to prove that algorithm `match` correctly computes least upper bounds of prefix mark-up languages. Before getting to the proof of this fundamental result, we need to prove a number of preliminary lemmas.

Some of these are concerned with the important notion of *proper mismatch*, defined as follows.

*Definition B.3 (Proper Mismatch).* Let  $T$  be a template,  $I$  an instance of  $T$ , and  $\text{enc}$  a mark-up encoding. A mismatch  $(\text{enc}(T), \text{enc}(I), (n, t))$  produced by  $\overrightarrow{\text{parse}}(\text{enc}(T), \text{enc}(I))$  [respectively,  $\overleftarrow{\text{parse}}(\text{enc}(T), \text{enc}(I))$ ] is called *proper* with respect to a template node labeled  $\alpha$  if:

- (i) it is a data mismatch, and  $(n, t)$  are encodings of constant templates nodes with the same label  $\alpha$  as shown in Figure 16

<sup>16</sup>Note that computing upper bounds of regular expression implies testing containment. The containment problem for regular expression is complete for PSPACE [Papadimitriou 1994]. However, in this context, we deal with rather simplified regular expressions, for which we prove that the least upper bound can be computed in PTIME. This is due to the very limited use of union—essentially only as a part of iterators and optionals—and to the fact that subexpressions are clearly marked using tags.

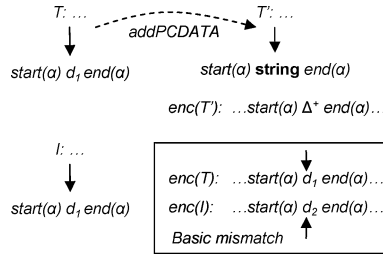


FIG. 16. Proper basic mismatches.

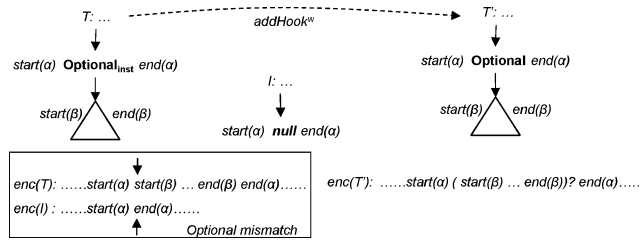


FIG. 17. Proper optional mismatches.

- (ii) it is a schema mismatch, and  $(n, t)$  are respectively the first symbol of  $end(\alpha)$  and the first symbol of  $start(\alpha.0)$  [respectively, the last symbol of  $start(\alpha)$  and the last symbol of  $end(\alpha.0)$ ] as shown in Figure 18 and Figure 17 (where  $\beta = \alpha.0$ )

Based on the type of the node labeled  $\alpha$  associated with the proper mismatch, we can unambiguously classify a proper mismatch as either *basic*, or *optional* or *list* depending on whether the template node labeled  $\alpha$  is subsumed respectively by a basic, an optional or a list node. Proper mismatches are fundamental in our setting. In fact, it can be easily seen that, from Definitions 4.5 and 4.6, it follows that:

PROPOSITION B.4. *Let  $T$  be a template,  $I$  an instance homogeneous to  $T$ , and  $enc$  a prefix mark-up encoding. If  $I \not\preceq T$ , then  $parse(enc(T), enc(I))$  returns a proper mismatch.*

Let us formalize the concept of solving mismatches:

Definition B.5 (*Solving Mismatches*). Let  $T$  be a template,  $I$  an instance homogeneous to  $T$ , and  $enc$  a prefix mark-up encoding. Let  $m$  be a proper mismatch returned by  $parse(enc(T), enc(I))$  and let  $\alpha$  be the label of the node it is associated to. We say that the template  $T'$  such that  $T \preceq T'$  solves  $m$  if  $parse(enc(T'), enc(I))$  does not return a proper mismatch associated with the same label  $\alpha$ .

A first result we can prove is the following:

LEMMA B.6. *Let  $T$  be a template,  $I$  an instance homogeneous to  $T$ , and  $enc$  a prefix mark-up encoding. Let  $m$  be a proper mismatch, either basic or optional, as obtained by invoking  $parse(enc(T), enc(I))$ .  $applyOperator(m)$  returns  $enc(T')$  such that  $T'$  is the minimal template which solves  $m$ .*

PROOF. The proof is a case by case analysis and is developed referring to a left to right parsing.

Let  $\alpha$  be the label of the node associated with the proper mismatch, and to start let us suppose that it is a basic mismatch as shown in Figure 16. By definitions of the operators  $addPCDATA$  (Section 7.2),  $addHook$  (Figure 10), and  $addPlus$  (Figure 11), it results immediately that  $addPCDATA$  is applied if and only if the proper mismatch is a basic mismatch. In that case,  $addPCDATA$  substitutes to the data token  $d_1$  in  $enc(T)$  a  $PCDATA$  node. From the templates point of view,  $applyOperator$  returns the encoding of the template  $T'$  obtained from  $T$  by replacing the constant template labeled  $\alpha$  with a basic template. This generalization solves  $m$  and is minimal, because by Definition 4.2 of the  $\leq$  relationship, two different constant templates are subsumed only by the basic template.

Optional mismatches are more involved. Let us start by showing that in the case of optional mismatches,  $addPlus$  is not defined. The situation is depicted in Figure 17 (the same discussion would symmetrically hold if the *null*-template had been located on the wrapper): the operators  $addPlus$  defined in Figure 11 would look for a last delimiter by calling  $lastDelim(m)$ . Then observe that by Definition 4.10 of prefix mark-up encodings,  $start(\alpha)$  includes a wrapping delimiter, that is an open tag which is not closed within  $start(\alpha)$ . This tag would prevent  $lastDelim$  from returning any delimiter because even if it can manage to locate a “last closing tag”  $lct$ , it would proceed  $start(\alpha)$ . Therefore,  $s(lct, t)$  encompasses the open tag that is not closed and therefore it is not well formed. The  $addPlus$  operators return  $\perp$ .

Next we show that  $addHook$  is not defined. The first symbol of  $start(\beta)$  is searched in the region of the sample following the mismatch. Since  $end(\alpha)$  cannot encompass symbols of  $start(\beta)$ , the candidate square would start by  $end(\alpha)$  itself, and therefore would not be well formed.

In order to complete the case of optional mismatches, we have to show that  $addHook$  is defined, and its invocation of  $findHookSquare$  selects the right occurrence of the first symbol of  $end(\alpha)$ , that is the one marking the optional. There are two cases according to Definition 4.1 of template: the node labeled  $\beta$  is subsumed by either a basic or a list node. In the former case then  $start(\beta) \cdots end(\beta)$  is of the form  $start(\beta) d end(\beta)$  where  $d$  may be either a data token or a  $PCDATA$ , and therefore the first symbol of  $end(\alpha)$  cannot occur in it. In the latter case  $start(\beta)$  contains a wrapping delimiter that prevents any other occurrence before the right one from delimiting a well-formed candidate square.

Finally, Figure 17 shows how  $addHook$  computes the encoding of a template  $T'$  which solves the mismatch by generalizing the optional-instance node labeled  $\alpha$  with an optional node. From the point of view of the encodings, consider the last return statement of  $addHook$ 's code in Figure 10: it introduces an *Hook* over the candidate square just located. Observe that the generalization is minimal because by Definition 4.2 of the  $\leq$  relationship, an optional-instance template and a null-template are both subsumed only by an optional template.  $\square$

We are now ready to prove our fundamental result about the correctness of the algorithm. To be more precise, we shall prove both the correctness result (i.e., Theorem 8.1) and the complexity result (i.e., Theorem 8.3). In fact, the two proofs share a number of commonalities that make a single treatment more convenient. Let us assume that `subst`, `insAfter`, `insBefore`, `search`, `isWFMarkUp`, `isWellFormed`, `findHookSquare`, `findPlusSquare`, `checkSquare`, `charSample`,  $addPCDATA$ ,  $addHook$ , and

parse run in PTIME with respect to the length of the input encodings.<sup>17</sup> We have the following Lemma, from which both Theorem 8.1 and Theorem 8.3 immediately descend.

LEMMA B.7. *Given two homogeneous templates  $T$  and  $S$ , and a prefix mark-up encoding  $enc$ ,  $match(enc(T), enc(S))$  computes  $enc(LUB(T, S))$  in polynomial time with respect to the maximum length of the input encodings.*

PROOF. We shall proceed by induction on the invocation nesting level  $m$  of  $match$ . Contextually we prove a few claims. The first one completes Lemma B.6 for list mismatches.

CLAIM B.8. *Consider two homogeneous templates  $T$  and  $S$  and the invocation of  $match$  on  $enc(T)$ ,  $enc(S)$ . Every  $applyOperator$  invocation is either on a proper mismatch, or it does not apply any operator. In the former case  $applyOperator$  returns the minimal generalization which solves the mismatch.*

Next Claim is focused on function  $solveMismatches$  of Figure 12.

CLAIM B.9. *Let  $R$  be a template,  $I$  an instance homogeneous to  $R$ , and  $enc$  a prefix mark-up encoding:  $solveMismatches(enc(R), enc(I)) = enc(LUB(R, I))$ .*

Finally, a separate Claim is devoted to the complexity. Let  $n$  denote the maximum length of the input encodings:

CLAIM B.10.  *$match(enc(T), enc(S))$  runs in PTIME with respect to  $n$ .*

*Basis Case  $m = 0$ .*  $match$  is not recursively called. This means that every  $applyOperator$  invocation does not apply the  $addPlus$  operator. Let us start by observing that  $match$  computes a characteristic sample  $\chi$  of  $w' = enc(S)$ , that is, a set of encodings of instances  $\mathcal{I} = \{I_1, \dots, I_n\}$  homogeneous to  $S$  (and therefore to  $T$ ) and such that  $\chi = LUB_{\subseteq}(enc(I_1), \dots, enc(I_n))$ . By Lemma B.2, it follows that  $S = LUB(I_1, \dots, I_n)$ .

Then we show that Claim B.9 holds when  $w = enc(T)$  and  $s$  is the encoding  $enc(I)$  of a generic instance homogeneous to  $T$ : The while loop stops when  $s \in L(w)$ , that is  $I \preceq T$ . Otherwise,  $parse(enc(T), enc(I))$  detects a mismatch that by Proposition B.4 is proper, and by hypothesis is either a basic or an optional mismatch. By Lemma B.6, we know that  $applyOperator$  solves that mismatch by returning the encoding  $enc(T^1)$  of a minimal generalization of  $T$ . Since  $T^1$  and  $I$  are homogeneous as well as  $T$  and  $I$ , next iterations of the while loop computes further generalizations, until after  $k$  iterations,  $applyOperator$  produces an  $enc(T^k)$  such that  $s \in L(enc(T^k))$ . Since the generalizations performed by  $applyOperator$  are minimal, this may happen only when  $w = enc(LUB(T, I)) = enc(T^k)$ . This proves Claim B.9 while Claim B.8 trivially holds as consequence of Lemma B.6 because every mismatch is either basic or optional by hypotheses.

Finally observe that by the associative property of the LUB,  $match$  returns:

$$\begin{aligned} w &= enc(LUB(\dots LUB(LUB(LUB(T, I_1), I_2)), \dots, I_n)) \\ &= enc(LUB(T, LUB(I_1, \dots, I_n))) = enc(LUB(T, S)). \end{aligned}$$

This proves the correctness for the basis case.

<sup>17</sup> In the prototype implementation all these functions runs in linear time,  $parse$  in linear time but only for prefix mark-up encodings,  $charSample$  in quadratic time.

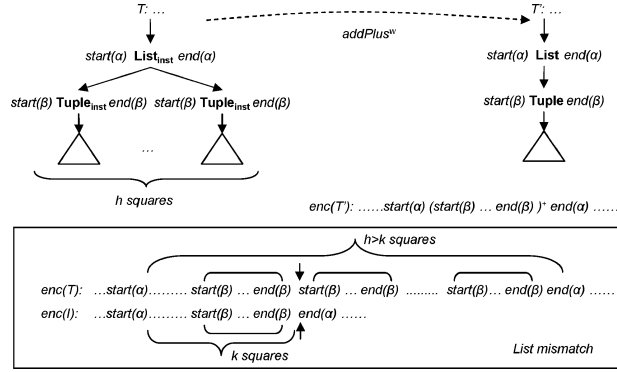


FIG. 18. Proper list mismatches.

To discuss the complexity, note that  $\text{charSample}(S)$  produces  $\mathcal{O}(n)$  instances whose encodings are  $\mathcal{O}(n)$  long. The number of external mismatches solved by  $\text{applyOperator}$  for each instance is  $\mathcal{O}(n)$ . Overall, there are  $\mathcal{O}(n^2)$   $\text{applyOperator}$  invocations and operator applications. First, observe that from the discussion above it is easy to show that undefined operators return  $\perp$  only by invoking functions that run in PTIME by assumption. Finally, Claim B.10 follows from the fact that by inductive hypothesis and by Claim B.8, the only operators invoked are  $\text{addPCDATA}$  and  $\text{addHook}$ , which runs in PTIME by assumption.

*Inductive Case  $m > 0$ .* Consider  $\text{match}(\text{enc}(T), \text{enc}(S))$  when the nesting level of  $\text{match}$  invocations is up to  $m > 0$ . This means that the  $\text{addPlus}$  operator may be applied and list mismatches have to be solved. By inductive hypothesis  $\text{match}(\text{enc}(T'), \text{enc}(S'))$  computes  $\text{enc}(\text{LUB}(T', S'))$  for any invocation whose nesting level is up to  $m - 1$  in polynomial time.

We prove Claim B.8 by exploiting the inductive hypothesis and so we complete Lemma B.6 for list mismatches. Refer to the situation depicted in Figure 18 but a similar discussion would symmetrically hold in the opposite direction and in the case of  $\text{addPlus}^s$ . Let us call  $\text{enc}(T_j^w) = \text{start}(\beta) \cdots \text{end}(\beta)$  the  $j$ th square occurrence on the wrapper; for all  $j$ , these are all encodings of homogeneous templates whose root node is labeled  $\beta$ . We start by showing that in the case of list mismatches, the  $\text{addHook}$  operators are not defined.  $\text{addHook}^s$  is not defined because it searches the first symbol of  $\text{start}(\beta)$  that follows the mismatch on the sample: the candidate squares are not well-formed; in fact, they start with  $\text{end}(\alpha)$ .  $\text{addHook}^w$  would select as candidate square the sequence of  $h - k$  patterns of the form  $\text{start}(\beta) \cdots \text{end}(\beta)$  but then it fails since  $\text{checkSquare}(m, \text{csquare})$  would detect that the candidate square ends with the same symbol preceding the mismatch point, namely, the last symbol of  $\text{end}(\beta)$ . Finally observe that  $\text{addPlus}^s$  is not defined because the candidate squares are not well formed: they start with  $\text{end}(\alpha)$ .

In order to complete the proof of Claim B.8, we have to show that  $\text{addPlus}^w$  is defined and solves the mismatch with a minimal generalization. By Definition 4.10 of prefix mark-up encodings, we know that  $\text{end}(\beta)$ , which marks a tuple node, includes a wrapping delimiter such that  $\text{lastDelim}(m)$  and  $\text{csquare}^w$  are defined, and that the latter corresponds to  $\text{enc}(T_{h+1}^w)$  on the wrapper.

During the candidate square evaluation we know by inductive hypothesis that  $\text{square}_0$  equals  $\text{enc}(\text{LUB}(T_k^w, T_{k+1}^w))$ . Then, occurrences of the candidate square are

consumed both on the left and on the right. When occurrences on the left are consumed—that is,  $square_{-1}$ ,  $square_{-2}$ ,  $\dots$ ,  $square_{-k+1}$ —we have, by inductive hypothesis that:

$$\begin{aligned} square_{-1} &= enc(\text{LUB}(T_{k-1}^w, T_k^w, T_{k+1}^w)), \dots, square_{-k+1} \\ &= enc(\text{LUB}(T_1^w, \dots, T_{k-1}^w, T_k^w, T_{k+1}^w)). \end{aligned}$$

The matching on the left is stopped when the call to  $\overleftarrow{\text{match}}(square_{-k+1}, w[\dots f_{-k+1}])$  fails, where  $w[\dots f_{-k+1}]$  is the region of the wrapper up to  $start(\alpha)$  included. However, that call does not generate any operator application and Claim B.8 is still valid. In fact, whichever is the sample generated by  $\text{charSample}(w[\dots f_{-k+1}])$ , a *non-proper* schema mismatch occurs between the last token of  $end(\beta)$  and the last token of  $start(\alpha)$ ; this  $\overleftarrow{\text{mismatch}}$  is not proper because the two template sections are not homogeneous.  $\overleftarrow{\text{addPlus}}$  is not defined because  $\overleftarrow{\text{lastDelim}}(m)$  looks for a last delimiter in an empty region. Finally,  $\overleftarrow{\text{addHook}}$  is not defined since  $\beta$  marks a tuple node and  $end(\beta)$  contains a wrapping delimiter.  $\overleftarrow{\text{addHook}}$  looks for an occurrence of the last token of  $start(\alpha)$  in  $square_{-k+1}$  (the search is performed from the end to the beginning backwards), but the corresponding candidate square cannot be well formed because in any case it starts<sub>s</sub> after  $start(\beta)$  and ends with  $end(\beta)$ . Finally, the only candidate squares  $\overleftarrow{\text{addHook}}$  can locate are not well formed because they end with  $start(\alpha)$ .

After the left matching has been concluded,  $\overrightarrow{\text{addPlus}}$  proceeds by matching the square on the right. In this case we have:

$$\begin{aligned} square_2 &= enc(\text{LUB}(T_1^w, \dots, T_{k-1}^w, T_k^w, T_{k+1}^w, T_{k+2}^w)), \dots, square_{h-k} \\ &= enc(\text{LUB}(T_1^w, \dots, T_h^w)). \end{aligned}$$

Again, the last recursive call on the right is  $\overrightarrow{\text{match}}(square_{h-k}, w(f_{h-k} \dots))$ , but it can be shown with an argument symmetric to the one above, that it does not apply any operator. Therefore, the final square  $square$  equals  $enc(\text{LUB}(T_1^w, \dots, T_h^w))$  and the wrapper generalization step computes as a result  $enc(T')$ , where  $T \preceq T'$  in such a way to solve  $m$ , as depicted in Figure 18. The generalization is also minimal, because by Definition 4.2 of the subsumption relationship, two list-instance templates of different cardinality can be subsumed only by a list template, and in that case, the one with a child tuple template equals to  $\text{LUB}(T_1^w, \dots, T_h^w)$  is the least general. This proves that list mismatches are correctly solved by  $\text{applyOperator}$ ; as a consequence, Claim B.8 holds, and Claim B.9 and the correctness thesis follow.

As far as the complexity is concerned, the square location step and the wrapper generalization step are performed in PTIME since all invoked functions run in PTIME by assumption. Then, observe that the candidate square evaluation make a recursive call of  $\overleftarrow{\text{match}}$  whose actual parameters are  $\mathcal{O}(n)$  long. During the candidate square matching, there may be other  $\mathcal{O}(n)$  of such recursive invocations. Finally, there are two failing recursive match invocations on *non-proper* mismatches to stop the candidate square matching on the left and on the right. Overall,  $\overrightarrow{\text{addPlus}}$  operator makes  $\mathcal{O}(n)$  recursive invocations of  $\text{match}$  that by inductive hypothesis runs in PTIME with respect to  $n$ . This proves that  $\overrightarrow{\text{addPlus}}$  runs in PTIME with respect to  $n$  and Claim B.10 follows exactly like for the basis case.  $\square$

ACKNOWLEDGMENTS. The authors wish to thank Prof. Henning Fernau for his feedback on *f-distinguishable functions* and Paolo Merialdo for the many interesting discussions on the subject of this article.

## REFERENCES

- ADELBERG, B. 1998. NoDoSE—A tool for semi-automatically extracting structured and semistructured data from text documents. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'98)* (Seattle, Wash.). ACM, New York.
- ANGLUIN, D. 1980. Inductive inference of formal languages from positive data. *Inf. Cont.* 45, 117–135.
- ANGLUIN, D. 1982. Inference of reversible languages. *J. ACM* 29, 3, 741–765.
- ARLOTTA, L., CRESCENZI, V., MECCA, G., AND MERIALDO, P. 2003. Automatic annotation of data extracted from large Web sites. In *Proceedings of the 6th Workshop on the Web and Databases (WebDB'03) (in conjunction with SIGMOD'03)*. ACM, New York, 7–12.
- ASHISH, N., AND KNOBLOCK, C. 1997. Wrapper generation for semistructured Internet sources. In *Proceedings of the Workshop on the Management of Semistructured Data (in conjunction with ACM SIGMOD 1997)*. ACM, New York.
- ATZENI, P., AND MECCA, G. 1997. Cut and paste. In *Proceedings of the 16th ACM SIGMOD International Symposium on Principles of Database Systems (PODS'97)* (Tucson, AZ). ACM, New York, 144–153.
- BAUMGARTNER, R., FLESCA, S., AND GOTTLÖB, G. 2001. Visual web information extraction with lixto. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'2001)* (Roma, Italy, Sept. 11–14). 119–128.
- BRUGEMANN-KLEIN, A., AND WOOD, D. 1998. One-unambiguous regular languages. *Info. Comput.* 142, 2 (May), 182–206.
- CHIDLOVSKII, B. 2000. Wrapper generation by *k*-reversible grammar induction. In *Proceedings of the International Workshop on Machine Learning and Information Extraction (ECAI'00)*, 61–72.
- CRESCENZI, V. 2002. On automatic information extraction from large web sites. Ph.D. dissertation, Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, Rome (Italy).
- CRESCENZI, V., AND MECCA, G. 1998. Grammars have exceptions. *Info. Syst.* 23, 8, 539–565. (Special Issue on Semistructured Data.)
- CRESCENZI, V., MECCA, G., AND MERIALDO, P. 2001. Roadrunner: Towards automatic data extraction from large Web sites. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'2001)* (Rome, Italy, Sept. 11–14). 109–119.
- CRESCENZI, V., MECCA, G., AND MERIALDO, P. 2002. ROADRUNNER: Automatic data extraction from data-intensive web sites. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'2002)* (Madison, Wisco.). ACM, New York.
- EMBLEY, D. W., CAMPBELL, M. D., JIANG, Y. S., LIDDLE, S. W., NG, Y. K., QUASS, D., AND SMITH, R. D. 1999. Conceptual-model-based data extraction from multiple-record web pages. *Data Knowl. Eng.* 31, 3, 227–251.
- FERNAU, H. 2000a. Learning XML grammars. In *Proceedings of the 2nd Machine Learning and Data Mining in Pattern Recognition MLDM'01*. Lecture Notes in Computer Science and Lecture Notes in Artificial Intelligence, vol. 2123. Springer-Verlag, New York, 73–87.
- FERNAU, H. 2000b. On learning function distinguishable languages. Tech. Rep. WSI-2000-13, Wilhelm-Schickard-Institut für Informatik.
- FERNAU, H. 2003. Identification of function distinguishable languages. *Theoret. Comput. Sci.* 290, 1679–1711.
- FREITAG, D. 1998. Information extraction from html: Application of a general learning approach. In *Proceedings of the 15th Conference on Artificial Intelligence (AAAI-98)*. 517–523.
- GOLD, E. M. 1967. Language identification in the limit. *Inf. Cont.* 10, 5, 447–474.
- GRUMBACH, S., AND MECCA, G. 1999. In search of the lost schema. In *Proceedings of the 7th International Conference on Data Base Theory (ICDT'99)* (Jerusalem, Israel). Lecture Notes in Computer Science, Springer-Verlag, New York, 314–331.
- GUPTA, A., HARINARAYAN, V., AND RAJARAMAN, A. 1998. Virtual database technology. In *Proceedings of the 14th International Conference on Data Engineering* (Orlando, Fla., Feb. 23–27). IEEE Computer Society, Los Alamitos, Calif., 297–301.
- HAMMER, J., GARCIA-MOLINA, H., CHO, J., ARANHA, R., AND CRESPO, A. 1997. Extracting semistructured information from the Web. In *Proceedings of the Workshop on the Management of Semistructured Data (in conjunction with ACM SIGMOD 1997)*. ACM, New York.

- HONG, T. W., AND CLARK, K. L. 2001. Using grammatical inference to automate information extraction from the Web. In *Proceedings of the European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2001)*. 216–227.
- HSU, C., AND DUNG, M. 1998. Generating finite-state transducers for semistructured data extraction from the web. *Info. Syst.* 23, 8, 521–538.
- HUCK, G., FRANKHAUSER, P., ABERER, K., AND NEUHOLD, E. J. 1998. Jedi: Extracting and synthesizing information from the web. In *Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS'98)*. 32–43.
- HULL, R. 1988. A survey of theoretical research on typed complex database objects. In *Databases*, J. Paredaens, Ed. Academic Press, Orlando, Fla. 193–256.
- KOSALA, R., VAN DEN BUSSCHE, J., BRUYNOOGHE, M., AND BLOCCKEEL, H. 2002. Information extraction in structured documents using tree automata induction. In *Proceedings of the European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2002)*. 299–310.
- KUSHMERICK, N. 2000a. Wrapper induction: Efficiency and expressiveness. *Artif. Intel.* 118, 15–68.
- KUSHMERICK, N. 2000b. Wrapper verification. *WWW J.* 3, 2, 79–94.
- KUSHMERICK, N., WELD, D. S., AND DOORENBOS, R. 1997. Wrapper induction for information extraction. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'97)*.
- LERMAN, K., AND MINTON, S. 2000. Learning the common structure of data. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00)*.
- LERMAN, K., MINTON, S. N., AND KNOBLOCK, C. A. 2003. Wrapper maintenance: A machine learning approach. *J. Artif. Intel. Res.* 18, 149–181.
- LIU, L., PU, C., AND HAN, W. 2000. Xwrap: An xml-enabled wrapper construction system for web information sources. In *Proceedings of the 16th IEEE International Conference on Data Engineering (ICDE'00)* (San Diego, Calif.), IEEE Computer Society Press, Los Alamitos, Calif. 611–621.
- MUSLEA, I., MINTON, S., AND KNOBLOCK, C. A. 1999. A hierarchical approach to wrapper induction. In *Proceedings of the 3rd Annual Conference on Autonomous Agents*. 190–197.
- MUSLEA, I., MINTON, S., AND KNOBLOCK, C. 2001. Hierarchical wrapper induction for semistructured sources. *J. Autonom. Agents Multi-Agent Syst.* 4, 93–114.
- PAPADIMITRIOU, C. H. 1994. *Computational Complexity*. Addison-Wesley, Reading, Mass.
- PITT, L. 1989. Inductive inference, DFAs and computational complexity. In *Analogical and Inductive Inference*. Lecture Notes in Artificial Intelligence, vol. 397, K. P. Jantke, Ed. Springer-Verlag, Berlin, 18–44.
- RADHAKRISHNAN, V., AND NAGARAJA, G. 1987. Inference of regular grammars via skeletons. *IEEE Trans. Syst., Man and Cybernet.* 17, 6, 982–992.
- RIBEIRO-NETO, B. A., LAENDER, A. H. F., AND SOARES DA SILVA, A. 1999. Extracting semistructured data through examples. In *Proceedings of the 1999 ACM International Conference on Information and Knowledge Management (CIKM'99)*. ACM, New York. 94–101.
- SAHUGUET, A., AND AZAVANT, F. 1999. Web ecology: Recycling HTML pages as XML documents using W4F. In *Proceedings of the 2nd Workshop on the Web and Databases (WebDB'99)* (in conjunction with SIGMOD'99). ACM, New York.
- SODERLAND, S. 1999. Learning information extraction rules for semistructured and free text. *Mach. Learn.* 34, 1–3, 233–272.

RECEIVED FEBRUARY 2003; REVISED JANUARY 2004; ACCEPTED APRIL 2004