Applied Artificial Intelligence: An International Journal

Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/uaai20

WRAPPER INFERENCE FOR AMBIGUOUS WEB PAGES

Valter Crescenzi [a] & Paolo Merialdo [a]

[a] Dipartimento di Informatica e Automazione, Università degli Studi Roma Tre, Roma, Italy

Version of record first published: 28 Feb 2008.

To cite this article: Valter Crescenzi & Paolo Merialdo (2008): WRAPPER INFERENCE FOR AMBIGUOUS WEB PAGES, Applied Artificial Intelligence: An International Journal, 22:1-2, 21-52

To link to this article: http://dx.doi.org/10.1080/08839510701853093

Applied Artificial Intelligence, 22:21–52

ISSN: 0883-9514 print/1087-6545 online
DOI: 10.1080/08839510701853093

Taylor & Francis
Taylor & Francis Group

WRAPPER INFERENCE FOR AMBIGUOUS WEB PAGES

Valter Crescenzi and Paolo Merialdo □ *Dipartimento di Informatica e Automazione, Università degli Studi Roma Tre, Roma, Italy*

□ *Several studies have concentrated on the generation of wrappers for web data sources. As wrappers can be easily described as grammars, the grammatical inference heritage could play a significant role in this research field. Recent results have identified a new subclass of regular languages, called prefix mark-up languages, that nicely abstract the structures usually found in HTML pages of large web sites. This class has been proven to be identifiable in the limit, and a PTIME unsupervised learning algorithm has been previously developed. Unfortunately, many real-life web pages do not fall in this class of languages. In this article we analyze the roots of the problem and we propose a technique to transform pages in order to bring them into the class of prefix mark-up languages. In this way, we have a practical solution without renouncing to the formal background defined within the grammatical inference framework. We report on some experiments that we have conducted on real-life web pages to evaluate the approach; the results of this activity demonstrate the effectiveness of the presented techniques.*

INTRODUCTION

Due to the huge amount of data available on the web, information extraction from websites has become a relevant research field. A significant number of works have focused on the study of methods and techniques for the development of wrappers, i.e., programs that perform the extraction task (see Laender et al. (2002a) and Chang et al. (2006) for surveys on the topic). Many proposals (Baumgartner, Flesca, and Gottlob 2001; Embley, Jiang, and Ng 1999b; Freitag 1998; Kushmerick 2000; Laender, Ribeiro-Neto, and daSilva 2002b; Muslea, Minton, and Knoblock 1999; Soderland 1999) have studied the problem of semi-automatically generating wrappers for extracting data from fairly structured HTML pages. These approaches need a training phase that involves a human intervention to identify and annotate data fields to be extracted.

Recent studies have addressed the issue of making the wrapper inference process completely automatic (Arasu and Garcia-Molina 2003;

Address correspondence to Paolo Merialdo, Dipartimento di Informatica e Automazione, Università degli Studi Roma Tre, Via della Vasca Navale 79, Roma 1-00149, Italy. E-mail: merialdo@dia.uniroma3.it

Crescenzi, Mecca, and Merialdo 2004; Lerman et al. 2004; Wang and Lochovsky 2002). These studies concentrate on data-intensive websites, and their goal is to reduce and possibly eliminate the efforts required for the training process, which can represent a drawback, especially in the maintenance of wrappers over a large number of data sources. Since wrappers are essentially parsers for the HTML code of web pages, grammar inference could in principle play a fundamental role for understanding limitations and opportunities of unsupervised learning techniques. Surprisingly, most of the approaches that have been proposed in the literature do not have a formal background from the grammatical inference perspective. An exception is represented by the RoadRunner approach (Crescenzi et al. 2001), which is based on strong theoretical foundations (Crescenzi and Mecca 2004) with a clear relationship with the traditional field of grammar inference.

In RoadRunner, pages from data-intensive websites are considered as the result of an encoding process that serializes instances of a given data type into HTML pages. Then, the generation of pages can be considered as a process that produces strings of a specific class of languages whose properties depend on the encoding function and on the underlying data type. In this framework, the wrapper generation problem can be described as the problem of inferring the language associated with a data type, given the encodings of a collection of instances of that type.

RoadRunner introduces a class of languages, called prefix mark-up languages (Crescenzi and Mecca 2004), that nicely abstracts the characteristics of pages from data-intensive websites, and for which several important properties hold. Prefix mark-up languages are identifiable in the limit from positive data, i.e., it is possible to infer a grammar given a finite number of positive examples only. Also, it has been developed in algorithm, called MATCH, (Crescenzi et al., 2001), that is able to infer a wrapper from a set of sample pages: it has been proven that for pages that comply with the class of prefix mark-up languages, MATCH runs in polynomial time complexity w.r.t. the input length (Crescenzi and Mecca 2004). Therefore MATCH can represent a practical solution, yet based on a sound theoretical framework, for the extraction of data from data-intensive websites.

Prefix mark-up languages are generated by a particular class of encodings, called prefix mark-up encodings, which allow the constructs of the underlying data type to be identified from the encoded output data. Unfortunately, even in regular websites, there are several pages that do not fall in the class of prefix mark-up languages. This is a serious drawback that limits the opportunities of MATCH. One possible development to overcome this issue is that of studying more involved inference algorithms for broader classes of languages; however, the theoretical foundations of grammatical inference suggest that this direction is not always a practical solution.

Therefore, with the objective of elaborating a practical solution, in this article we analyze the roots of the problem on the theoretical framework of ROADRUNNER, and propose a technique to preprocess the sample pages in order to bring them in the target class of prefix mark-up languages. Our approach corresponds to defining a new a class of languages, which is broader than the class of prefix mark-up languages and which is still identifiable in the limit.

The effectiveness and the efficiency of our proposal has been evaluated in practice by means of an experimental activity. We have processed several real-life web pages with the techniques presented in this article; then we have compared the results of running the wrapper inference algorithm MATCH against the preprocessed pages and those obtained using the original pages. The results of this experience show that the proposed approach improves the effectiveness of the inference algorithm without compromising its efficiency.

This article is organized as follows: in Section 2 we recall the formal page generation model at the basis of our approach. Section 3 discusses opportunities and limitations for the class of prefix mark-up languages and introduces the main ideas on which we have built our practical solution. Section 4 illustrates the basic ideas of our approach, which are then formalized in Section 5. Section 6 discusses technical details about the implementation of the technique and discusses the results of our experimental activities on real-life web pages. Section 7 presents related work and Section 8 concludes the article.

BACKGROUND: DATA TYPES, ENCODINGS, AND MARK-UP LANGUAGES

Pages in a data-intensive website can be considered as the result of an encoding process that serializes instances of a given data type into HTML pages. In this section, we formally define data types and instances; then, we illustrate the notion of mark-up encoding, which abstracts the process to generate HTML pages from data type instances. Finally, we show that according to this framework, HTML pages can be seen as regular languages, and that the wrapper inference problem corresponds to infer a regular grammar, given a set of sample pages.

In our framework, a wrapper corresponds to a union-free regular expression (UFRE), which is defined as follows. Given a special symbol Δ, and an alphabet of symbols Σ not containing Δ, a UFRE over Σ is a string over alphabet $\Sigma \cup \{\Delta, \cdot, +, ?, (,)\}$ defined as follows. First, the empty string, ϵ and all elements of $\Sigma \cup \{\Delta\}$ are union-free regular expressions. If a and b are UFRE, then $a \cdot b$, $(a)^+$, and $(a)?$ are UFRE. The semantics of these

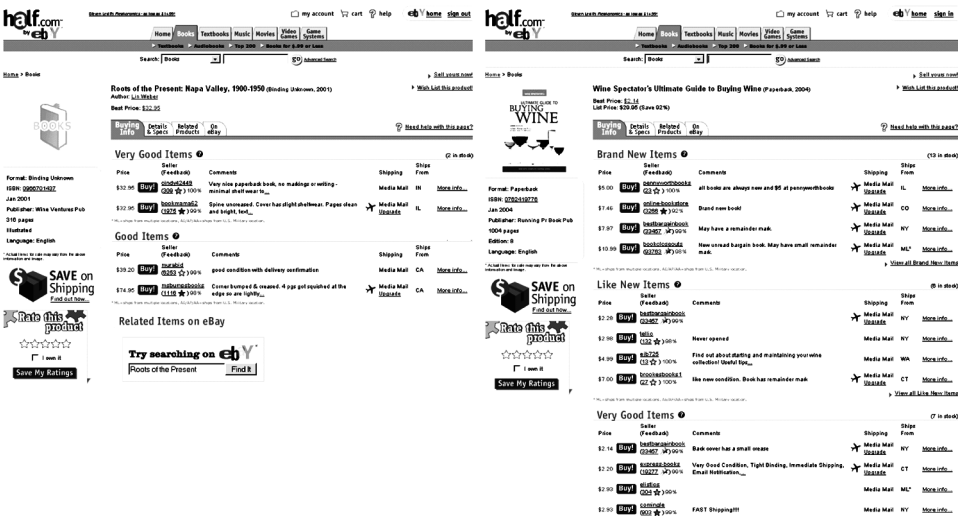

FIGURE 1 Pages from http://half.ebay.com.

expressions is defined as usual, $+$ being an iterator and $(a)?$ being a shortcut for $(a|\epsilon)$ (denotes optional patterns).

To introduce our formal framework, consider the two pages in Figure 1, which are taken from a popular website. Each page contains information about used copies of a given book. The book is described by several fields, such as title, author, best price, and list price. Information about used copies of the book are organized in a nested list: the outer list groups items according to their status ("brand new items," "like new," and so on). The inner list reports details of each item: price, seller feedback, comment, etc. Some of the data fields in the page, like the author of the book and the comment on an item, are optional. It is reasonable to assume that pages like those in Figure 1 have been generated by encoding in HTML data organized according to a common data type.

Data Type and Instances

To formally describe data type and instances, we consider a nested relational data model (Abiteboul and Beeri 1995; Hull 1988), that is, typed objects with nested lists (i.e., ordered sets) and tuples. Tuples have attributes (possibly optional), which in turn may be either atomic attributes or lists of tuples.

We assume the existence of an atomic type U, called the basic type, whose domain denoted by $dom(U)$, corresponds to all the strings of finite length over a data alphabet Δ. There exists a distinguished constant *null*,

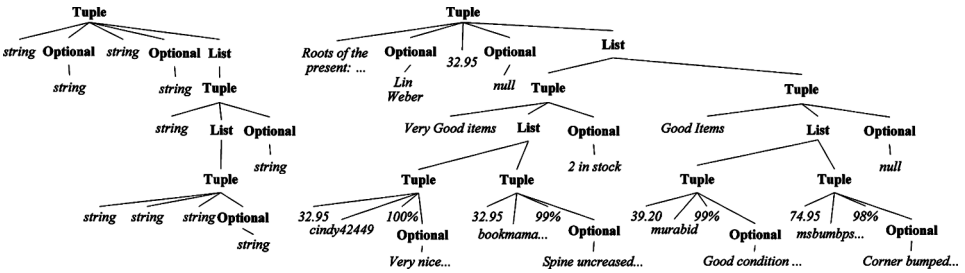

FIGURE 2 Abstract data types and instances.

and we will say that a type is *nullable* if *null* belongs to its domain. Nonatomic types (and their respective domains) are recursively defined as follows: (i) if $T_1, \ldots, T_n$ are basic, optional or list types, amongst which at least one is not nullable, then $[T_1, \ldots, T_n]$ is a tuple type, with domain $dom([T_1, \ldots, T_n]) = \{[a_1, \ldots, a_n] \,|\, a_i \in dom(T_I)\}$; (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 $dom(T)$ elements; (iii) if T is a basic or list type, then $(T)?$ is an optional type, with domain $dom(T) \cup \{null\}$.

Nested types and their instances can be suitably represented as trees (Hull 1998). Figure 2 shows the tree representation for the data type that abstracts the data of used book pages, discussed above, and an instance of this data type, namely, the instance associated to the left page of Figure 1. In the following, we will denote T_σ the tree associated with the type σ, and t_σ the tree associated with an instance of σ.

We also need to introduce a labelling system for type and instance trees to identify their nodes. The labelling system is recursively defined as follows. The root of a type tree T_σ is labelled by *root*. If a list node or an optional node is labelled α, then its child is labelled $\alpha.0$; if a tuple node with n children is labelled α, then its n children are labelled $\alpha.1, \ldots, \alpha.n$. An instance tree t_σ of T_σ is labelled similarly, but all the children of a list-instance or optional-instance node labelled α, are labelled $\alpha.0$. 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 *Very Good Items* and *Good Items* in the instance in Figure 2 are both labelled by the label *root*.5.0.1, which is also the label of the corresponding string node in the type tree.

Mark-Up Encodings

We now introduce the notion of Mark-up encodings, which aims at abstracting the creation of mark-up based documents—for example, HTML pages—from instances of data types.

Let $\Sigma \cup \overline{\Sigma}$ be an alphabet of symbols, called the schema alphabet, with $(\Sigma \cup \overline{\Sigma}) \cap \Delta = \emptyset$ and $\overline{\Sigma} = \{</\mathtt{a}> \mid <\mathtt{a}> \in \Sigma\}$. Essentially, the schema alphabet is disjoint to the data alphabet, and it is made of mark-up tags, i.e., it contains a closing tag $</\mathtt{a}>$ for every opening tag $<\mathtt{a}>$.

A mark-up encoding *enc* is a function that associates a regular expression to a given type σ. Namely, *enc* associates every node of T_σ labelled α with a pair of delimiters, denoted $start(\alpha)$ and $end(\alpha)$, with $start(\alpha), end(\alpha) \in (\Sigma \cup \overline{\Sigma})^+$, i.e., nonempty strings over the schema alphabet, and then recursively processes the nodes of T_σ as follows:

- for a basic leaf node U labelled α, $enc(U) = start(\alpha) \cdot \Delta^+ \cdot end(\alpha)$
- for an optional node $(T)?$ with label α, $enc((T)?) = start(\alpha) \cdot (enc(T))? \cdot end(\alpha)$
- for a tuple node $[T_1, \ldots, T_n]$ with label α, $enc([T_1, \ldots, T_n]) = start(\alpha) \cdot enc(T_1) \cdot \cdots \cdot enc(T_n) \cdot end(\alpha)$
- for a list node $\langle T \rangle$ with label α, $enc(\langle T \rangle) = start(\alpha) \cdot (enc(T))^+ \cdot end(\alpha)$.

We say that a mark-up encoding is well-formed if tags in the encoding of any node of the type tree are properly nested and balanced so that every occurrence of a symbol $<\mathtt{a}>$ in Σ is "closed" by a corresponding occurrence of a symbol $</\mathtt{a}>$ in $\overline{\Sigma}$. Formally, we say a string is well-formed if it belongs to the language defined by the context-free grammar G_{tag} defined by the following productions, being S the starting nonterminal symbol:

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

Then, a mark-up encoding is a well-formed mark-up encoding if $start(\alpha) \triangle\ end(\alpha)$, is a well-formed string with $\triangle$ denoting a placeholder for data encodings.

Figure 3 shows an encoding for the data type of our running example; it is easy to verify that this encoding is well-formed.[1]

Mark-up encodings are used to model the construction of pages starting from the instances of a data type. Let t_σ be the tree representation

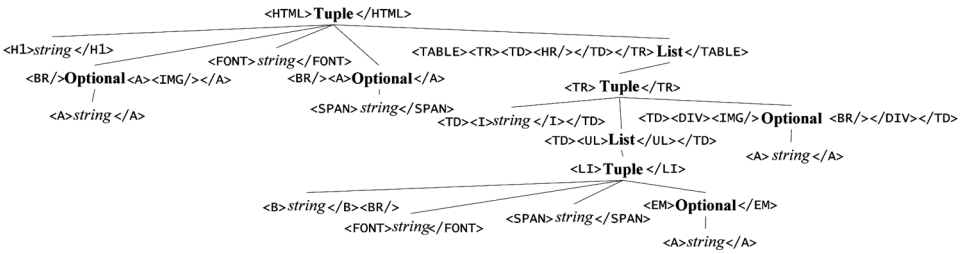

FIGURE 3 A well-formed mark-up encoding.

of an instance I of the data type σ, and let *enc* be an encoding over σ; *enc* applies over the labelled nodes of t_σ as follows:

- for a constant leaf node $a \in dom(U)$ with label $\alpha, enc(a) = start(\alpha) \cdot a \cdot end(\alpha)$
- for a *null* node with label $\alpha, enc(null) = start(\alpha) end(\alpha)$
- for a list-instance node $\langle a_1, \ldots, a_n \rangle$ with label $\alpha, enc(\langle a_1, \ldots, a_n \rangle) = start(\alpha) \cdot enc(a_1) \cdot \ldots \cdot enc(a_n) \cdot end(\alpha)$
- for a tuple-instance node $[a_1, \ldots, a_n]$ with label $\alpha, enc([a_1, \ldots, a_n]) = start(\alpha) \cdot enc(a_1) \cdot \ldots \cdot enc(a_n) \cdot end(\alpha)$
- for an optional-instance node $(a)?$ with label $\alpha, enc((a)?) = start(\alpha) \cdot enc(a) \cdot end(\alpha)$.

The production of a web page can be seen as the serialization of an instance obtained with a visit of the corresponding instance tree.

Note that by construction, encoded instances and encoded types are well-formed strings. Then, every occurrence o of a symbol in the schema alphabet is in correspondence with one balancing occurrence $\overline{o}$.[2] Given a well-formed string s, we denote $subStr_{(o)}(s)$ ($subStr_{]o[}(s)$) the substring of s that is (strictly) enclosed between o and $\overline{o}$. For example, given the string $s = <\texttt{TR}><\texttt{TD}><\texttt{A}></\texttt{A}></\texttt{TD}></\texttt{TR}>$, then $subStr_{]<\texttt{TD}>[}(s) = <\texttt{TD}><\texttt{A}></\texttt{A}></\texttt{TD}>$, and $subStr_{(<\texttt{A}>)}(s) = <\texttt{A}></\texttt{A}>$.

```
<HTML>
  <H1>Δ+</H1>
  <BR/>(<A>Δ+</A>)? <A><IMG/></A>
  <FONT>Δ+</FONT>
  <BR/><A>(<SPAN>Δ+</SPAN>)? </A>
  <TABLE>
      <TR><TD><HR/></TD></TR>
    ( <TR>
        <TD><I>Δ+</I></TD>
        <TD><UL>
          ( <LI>
              <B>Δ+</B><BR/>
              <FONT>Δ+</FONT>
              <SPAN>Δ+</SPAN>
              <EM> (<A>Δ+</A>)? </EM>
            </LI> )+
          </UL></TD>
          <TD><DIV><IMG/> (<A>Δ+</A>)?
           <BR/></DIV></TD>
      </TR> )+
  </TABLE>
</HTML>
```

```
<HTML>
 <H1>Roots of the present: ...</H1>
 <BR/><A>Lin weber</A><A><IMG/></A>
 <FONT>32.95</FONT>
 <BR/><A></A>
 <TABLE>
  <TR><TD><HR/></TD></TR>
  <TR><TD><I>Very Good Items</I></TD>
      <TD><UL>
        <LI>
          <B>32.95</B><BR/>
          <FONT>cindy42449</FONT>
          <FONT>100%</FONT>
          <EM><A>Very nice...</A></EM>
        </LI>
        <LI>
          <B>32.95</B><BR/>
          <FONT>bookmama</FONT>
          <SPAN>99%</SPAN>
          <EM><A>Spine uncreased...</A></EM>
        </LI>
      </UL></TD>
      <TD><DIV><IMG/><A>2 in stock</A>
      <BR/></DIV></TD>
  </TR>
  <TR><TD><I>Good Items</I></TD>
      <TD><UL><LI>...</LI></UL></TD>
      <TD><DIV><IMG/><BR/></DIV></TD>
  </TR>
 </TABLE>
</HTML>
```

```
<HTML>
 <H1>Wine Spectator's ...</H1>
 <BR/><A><IMG/></A>
 <FONT>2.14</FONT>
 <BR/><A>29.95 (Save ...)</A>
 <TABLE>
   <TR><TD><HR/></TD></TR>
   <TR><TD><I>Brand New Items</I></TD>
     <TD><UL>
       <LI>
         <B>5.00</B><BR/>
         <FONT>pennyworthbook</FONT>
         <FONT>100%</FONT>
         <EM><A>all book are...</A></EM>
       </LI>
       <LI>
         <B>7.46</B><BR/>
         <FONT>online-bookstore</FONT>
         <SPAN>92%</SPAN>
         <EM><A>Brand new book!</A></EM>
       </LI>
       <LI>...</LI>
     </UL></TD>
     <TD><DIV><IMG/><A>13 in stock</A>
     <BR/></DIV></TD>
   </TR>
   <TR><TD><I>Like New Items</I></TD>
    <TD><UL><LI>...</LI></UL></TD>
    <TD><DIV><IMG/><A>8 in stock</A>
    <BR/></DIV></TD>
   </TR>
   <TR><TD><I>Very Good Items</I></TD>
    <TD><UL><LI>...</LI></UL></TD>
    <TD><DIV><IMG/><A>7 in stock</A>
    <BR/></DIV></TD>
   </TR>
 </TABLE>
</HTML>
```

FIGURE 4 Encodings of types ($enc(\sigma)$) and instances ($enc(I)$).

Well-Formed Mark-Up Languages

A well-formed mark-up encoding *enc* applied to all the instances of a type σ generates a language of strings. These languages, which are called well-formed mark-up languages, are regular and the corresponding regular expression $enc(\sigma)$ can obtained by applying *enc* to σ (Crescenzi and Mecca 2004).

This is illustrated in Figure 4: the encoding shown in Figure 3 for the type σ of Figure 2 (left) produces the regular language specified by the regular expression in Figure 4 (left). When it is applied to the instance shown in Figure 2 (right) it produces one of the strings of the language, namely, the HTML code in Figure 4 (middle).

In this framework, the generation of a wrapper from a set of sample pages corresponds to infer a regular language given a set of sample strings.

PREFIX MARK-UP LANGUAGES: OPPORTUNITIES AND LIMITATIONS

The class of well-formed mark-up languages presented in the previous section is not identifiable in the limit with positive data only (Crescenzi and Mecca 2004). In this section, we discuss the class of prefix mark-up languages, a subclass of mark-up languages that is identifiable in the limit, and then represents a promising opportunity for the development of techniques for the automatic inference of web wrappers.

Prefix Mark-Up Languages

The class of prefix mark-up languages is defined as the class of languages obtained by applying to the instances of an abstract data type σ a prefix mark-up encoding (Crescenzi and Mecca 2004), that is a mark-up encoding function *enc* for which the following conditions, called the prefix constraints, hold:

- *Wrapping delimiters*: all delimiters of nonleaf nodes are such that there is at least one symbol of Σ in the start delimiter which is closed by a symbol of $\overline{\Sigma}$ in the end delimiter;
- *Point-of-choice delimiters*: symbols of delimiters that mark optional, and list nodes do not occur inside delimiters of their child node.

The mark-up encoding shown in Figure 3 is not prefix. The point-of-choice delimiters constraint is violated because the delimiter of the outer list node includes a <TR> tag, which occurs also inside the delimiter of its child (tuple) node. Likewise, the wrapping delimiters condition is not satisfied, because in the first optional node of the root tuple there is no opening tag in the start delimiter that is closed in the end delimiter. Pages generated by the encoding in Figure 3 will not fall in the class of prefix mark-up languages.

The definition of prefix mark-up languages poses constraints on the delimiters of the underlying data types that ensure this class of languages to be identifiable in the limit with positive sample only (Crescenzi and Mecca 2004). In the web context, this has important consequences as it means that for pages that belong to this class of languages, a wrapper can be generated automatically even taking as input only a set of sample pages.

Another important feature that makes wrapper inference feasible for prefix mark-up languages is that they are associated with a simple and natural characteristic sample. In grammar inference, the notion of characteristic sample associated with a class of languages is used to define the properties that the set of samples must exhibit to identify exactly one language in the class (Angluin 1982). Prefix mark-up languages are associated with an interesting notion of characteristic sample, which is obtained by encoding a rich set of instances.

Definition 1 (Rich Set of Instances). *A set of instances* $\mathcal{I} = \{I_1, \ldots, I_n\}$ *of a schema* σ*, is a rich set of instances if in* $\mathcal{I}$ *every attribute occurs with at least two different values; every optional is instantiated at least once, and is null at least once, as well; every list appears at least with two different cardinalities.*

This notion of characteristic sample (Crescenzi and Mecca 2004) has a practical impact since it is likely that even taking a small set of sample pages randomly, one can obtain the characteristic sample needed to feed the inference process.

In Crescenzi et al. (2001), we describe an algorithm called MATCH, which is able to infer a grammar from a set of sample pages. It has been proven that for prefix mark-up languages, MATCH runs in polynomial time with respect to the length of the input encodings (Crescenzi and Mecca 2004). Unfortunately, although prefix mark-up languages exhibit these nice features, experience says that many HTML pages found on the web do not fall in this class of languages, whereas they usually belong to the broader class of well-formed mark-up languages.

To overcome this issue, one possible direction is that of studying more involved inference algorithms for broader classes of languages; however, the theoretical foundations of grammatical inference suggest that this direction is not always a practical solution, as the more expressive the class of inferable languages is, the more complex the characteristic samples become, and too sophisticated and "unnatural" characteristic samples cannot be asked of the final user of a wrapper generator system.

Therefore, with the objective of elaborating a practical solution, our approach is that of preprocessing the sample pages in order to bring them in the target class of prefix mark-up languages.

BRINGING PAGES INTO PREFIX MARK-UP LANGUAGES

Our proposal for bringing pages into prefix mark-up languages is based on a transformation, $trans f_{ext}^{\mathcal{P}}$, that works on a set of sample pages $\mathcal{P}$. Assuming that pages in $\mathcal{P}$ might have been generated by a well-formed mark-up encoding that does not satisfy the wrapping and point-of-choice delimiters conditions, the goal of $trans f_{ext}^{\mathcal{P}}$ is that of repairing the violations of the prefix constraints directly on pages. Hence, $transf_{ext}^{\mathcal{P}}$ will be applied as a preprocessing step over the set of pages that feed the wrapper inference process.

To argument the correctness of the transformation, we present $transf_{ext}^{\mathcal{P}}$ contextually with its intensional counterpart, $transf_{int}$, an abstract transformation that operates on the encodings in order to fix violations of prefix constraints. In fact, $trans f_{ext}^{\mathcal{P}}$ can be considered correct if it does not corrupt the underlying data type, that is, if it produces strings that can still be considered as the result of the application of a "fixed" encoding on the original data type instances. Therefore, we show that every transformation performed by $trans f_{ext}^{\mathcal{P}}$ on pages has a univocal correspondence with a transformation performed by $trans f_{int}$ on the encoding. If $trans f_{int}$ fixes violations of the prefix constraints on the originating encoding, then $trans f_{ext}^{\mathcal{P}}$ achieves its goal of transforming pages into a language of the class of prefix mark-up languages.

Overall, our approach can be summarized as follows. We have a set of pages, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ that represent the set of positive samples for the wrapper inference process. According to our model, we assume that these pages have been generated by applying a well-formed mark-up encoding *enc* over a set of instances $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ of a given data type $\sigma : p_i = enc(I_i),\ i = 1, \ldots, n$. If *enc* does not satisfy the prefix constraints, pages in $\mathcal{P}$ do not belong to a language of the class of prefix mark-up languages. We introduce a transformation $trans f_{int}$ that operates on *enc*, and a transformation $trans f_{ext}^{\mathcal{P}}$ that works over pages of the sample set $\mathcal{P}$ such that if $trans f_{int}(enc)$ is a prefix mark-up encoding, then $trans f_{ext}^{\mathcal{P}}(p_i) = trans f_{int}(enc) \cdot (I_i),\ i = 1, \ldots, n$, and every $trans f_{ext}^{\mathcal{P}}(p_i)$ belongs to a prefix mark-up language.

The approach corresponds to define a new class of languages, for which our transformation succeeds. The new class is more expressive than the class of prefix mark-up languages, and is still identifiable in the limit with a characteristic sample that slightly differs from that of the class of prefix mark-up languages.

Our transformations are based on two complementary techniques: segmentation and disambiguation. The former aims at introducing wrapping delimiters, the latter solves ambiguities due to violations of the point-of-choice delimiters. We first present the two techniques separately, then

we show how they can actually cooperate to deal with the recursive structure of the underlying data types.

We introduce some notations to distinguish the symbols of the schema alphabet $\Sigma \cup \overline{\Sigma}$ from their occurrences in the encoding and in the pages. Given a string $s = a_1 \cdot a_2 \cdot \ldots \cdot a_n$ over an alphabet $\Sigma \cup \overline{\Sigma}$, $\texttt{occur}(s)$ denotes the set of all occurrences of the alphabet symbol in s: $\texttt{occur}(s) = \{a_1, a_2, \ldots, a_n\}$. Conversely, $\texttt{symbol}(a_i)$ denotes the alphabet symbol of which a_i is occurrence; then $\texttt{symbol}(a_i) \in \Sigma \cup \overline{\Sigma}$, $i = 1, \ldots, n$.

The segmentation and the disambiguation techniques manipulate the occurrences of symbols in the encoding and in the pages. We associate every occurrence o with an annotation, i.e., a sequence of symbols over the schema alphabet. We assume that our transformations initialize the annotation of every occurrence (in pages as well as in the encoding) with its corresponding symbol in the schema alphabet. Annotations are then manipulated by the transformations by means of a function, $\texttt{annotate}(o, s)$, that simply appends a string s to the annotation of the occurrence o it applies to. In the following, for the sake of readability, we blur the distinction between symbol and annotation associated to an occurrence; we assume $\texttt{symbol}(o)$ returns the annotation associated with o, and we refer to the annotated alphabet to indicate the set of annotation symbols that can be dynamically built over the original schema alphabet.

The Segmentation Technique

To describe our repairing transformations, we shall refer again to the encoding in Figure 3 and to the pages it produces, such as the ones in Figure 4. The wrapping delimiters condition imposes that all delimiters of nonleaf nodes must have at least one opening tag in the start delimiter which is closed in the end delimiter. In our example, in the encoding of Figure 3 the wrapping delimiters condition does not hold in the delimiters of the optional node of the root tuple. However, observe that such an optional node occurs before the unique occurrence of `IMG` (which appears in the delimiter of the optional itself) and after the unique occurrence of `/H1` (in the delimiter of the first attribute). These properties can be used to split the tuple into segments: each segment is bounded by a pair of unique occurrences and includes a (possibly empty) sequence of tuple attributes. Every segment can be marked by a "virtual" delimiter, and occurrences within each segment can be annotated by means of a symbol identifying the segment they belong to.

The introduction of the virtual delimiters associated to segments can solve violations of the wrapping delimiters condition. In our example, since our optional node falls in a segment that embeds one attribute, it would be enclosed by a "virtual" delimiter that will work as a wrapping delimiter.

The above transformation, which is performed by $trans f_{int}$, can repair missing wrapping delimiters in the root tuple of the encoding. To introduce its extensional counterpart, which is actually performed by $trans f_{ext}^{\mathcal{P}}$ on pages, we observe that symbols that in the encoding occur exactly once in the delimiters of the attributes of the root tuple will appear exactly once in all the pages produced by that encoding; interestingly, the inverse also holds, if we assume instances from a characteristic sample: symbols that occur exactly once in every encoded instance are originated by symbols that occur exactly once in the delimiters of child nodes of the root tuple. In other words, by segmenting the pages based on symbols that occur once in all the pages, we can obtain on them the same effects of segmenting directly the encoding. Therefore, $trans f_{ext}^{\mathcal{P}}$ segments pages based on symbols that occur exactly once in all the pages, in the hope of repairing the effects of a missing wrapping delimiter.

Once pages are segmented, it is possible to inspect each segment, in order to trigger further segmentations. In fact, occurrences of tags that are not unique on the original page, could become unique within the more focused context of a segment. It is easy to verify that the effects of a recursive segmentation on pages still correspond to a recursive segmentation of the encoding. Then, our transformation on pages $trans f_{ext}^{\mathcal{P}}$, and analogously its intensional counterpart $trans f_{int}$, performs a recursive segmentation that run until new segments are found.

Figure 5 illustrates the intensional and the extensional segmentation functions ($segment_{int}$ and $segment_{ext}$, respectively). The former operates on an encoding, the latter on a set of pages (that is, encoded instance of a given type). They both perform the actual segmentation by calling the same function, `segment`, shown in Figure 6.

Function `segment` works on well-formed strings: based on a set of elements, U, which occur in the input string s the segmentation process: (i) splits the well-formed input string s into segments, i.e., well-formed substrings of s; (ii) annotates each segment with a suitable identifier;

Function $segment_{int}(enc, o)$
input: an encoding enc over a data type σ,
o an occurrence of the schema alphabet in enc

output: an encoding enc over a data type σ
begin
Let $str = subStr_{[o]}(enc(\sigma))$
Let $U = \{u | u \in \text{occur}(str)$ and symbol(u) occurs exactly once in str outside of any optional or iterator substring$\}$
$segment(str, U)$
return enc
end

Function $segment_{ext}(\mathcal{P}, \mathcal{O})$
input: a set of sample pages $\mathcal{P} = \{p | p = enc(I)\}$,
a set of occurrences of schema alphabet symbols $\mathcal{O} = \{o | o \in occur(p)\}$
output: a set of pages $\mathcal{P}$
begin
Let $\mathcal{S}$ be a set of strings $\{str | str = subStr_{[o]}(p))\}$
Let $\mathcal{U} = \{U | U = \{u | u \in occur(p_i)$ and symbol(u) occurs exactly once in every element of $\mathcal{S}\}$
for each string $str_i, i = 1, \ldots, n$
$segment(str_i, U_i)$
return $\mathcal{P}$
end

FIGURE 5 Intensional and extensional segmentation.

Function $segment(str, U)$
input: str a well-formed string $str = c_0 \cdot \ldots \cdot c_n$
over a (possibly annotated) alphabet $\Sigma \cup \overline{\Sigma}$,
a set of occurrences $U = \{u | u \in occur(str)\}$
output: a string over a (possibly annotated) alphabet of symbols in $\Sigma \cup \overline{\Sigma}$
begin
do begin
Let $U_i = \{u | u \in (U \cap \text{occur}(subStr_{[c_i]}(str)))\}$
Let $k = 0$
for each $c_i, i = 1, \ldots, n$ begin
if $U_i \neq U_{i-1}$ begin
$k = k + 1$
if $((c_i \in \Sigma) \wedge (c_{i-1} \in \overline{\Sigma}))$
$insert(str, i, \text{“} < /v > \cdot < v > \text{”})$
if $((c_i \in \Sigma) \wedge (s_{i-1} \in \Sigma))$
$insert(str, i, \text{“} < v > \cdot < v > \text{”})$
if $(c_i \in \overline{\Sigma}) \wedge (s_{i-1} \in \overline{\Sigma})$
$insert(str, i, \text{“} < /v > \cdot < /v > \text{”})$
end
end
$annotate(o_i, k)$
end while (some change occurs in str)
return str
end

Function $insert(str_a, i, str_b)$
input: str_a a string $s = a_1 \cdot \ldots \cdot a_n$ over an alphabet Σ,
i an integer $0 < i < n$,
str_b a string $s = b_1 \cdot \ldots \cdot b_m$ over an alphabet Σ'
output: a string over an alphabet $\Sigma \cup \Sigma'$
begin
return $a_1 \cdot \ldots a_{i-i} \cdot b_1 \ldots \cdot b_m \cdot a_i \cdot \ldots \cdot a_n$
end

FIGURE 6 Function segment.

(iii) introduces virtual occurrences of symbols in $\{< v >, < /v >\}$ to mark each segment.[3] The extent of each segment is determined by means of the occurrences of the input set U: namely, each segment is bounded by a pair of consecutive occurrences in U. The function guarantees that virtual nodes are inserted correctly, that is, that the output string is still well-formed.

Figure 7 shows the results of the segmentation produced on a well-formed string. Symbols in boldface correspond to the set of occurrences U on which the segmentation is based. The dashed lines indicate consecutive occurrences that enclose the same subset of U.

The segmentation, as it is defined above, can repair only missing wrapping delimiters that occur in the root tuple. However, we will see that based on the effects that can be obtained by means of the disambiguation technique, the segmentation can operate to fix also missing delimiters occurring in nested tuples.

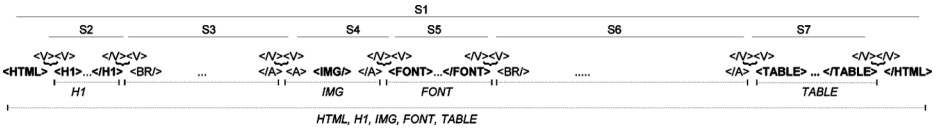

FIGURE 7 Segmenting well-formed strings.

The Disambiguation Technique

Violations of the point-of-choice delimiters condition arise when symbols in the delimiters of optional and list nodes occur inside the delimiters of their child node. We say that the occurrences that violate the condition are ambiguous because they do not allow the delimiters of parent and child constructs to be distinguished. In the example shown in Figure 3, the TR occurrences of the list and tuple delimiters are ambiguous.

Our technique for repairing missing point-of-choice delimiters is based on the observation that ambiguous occurrences can be distinguished by considering the set of symbols that occurs in the well-formed string that they enclose.

Consider again the encoding in Figure 3 and its serialization in Figure 4: the well-formed string that is surrounded by the TR occurrence of the outer list delimiter is <TR><TD><HR/></TD></TR> and includes an occurrence of the *HR* symbol; differently, the well-formed string that is surrounded by the TR occurrence of the tuple delimiter encloses a set of symbols, which includes, for example, *I* and *UL*.[4] If we annotate each ambiguous occurrence with one of the symbols taken from the well-formed string they surround, the point-of-choice delimiters condition is satisfied. For example, we can distinguish the TR occurrences of our example by annotating the list's one with *HR*, and the tuple's one with *I*. In essence, we create a partition of the ambiguous occurrences, by means of a partitioning set of symbols. This strategy is adopted by $tranf_{int}$ to disambiguate occurrences on the encoding.

The same approach can be applied also over pages: at the extensional level, symbols that participate in a violation of the point-of-choice delimiter on the encoding have multiple occurrences. Then, $trans f_{ext}^{\mathcal{P}}$ considers ambiguous every occurrence that occurs more than once in $\mathcal{P}$. If the ambiguous occurrences can be partitioned on the encoding based on the symbols they enclose, the same criterion can be applied at the page level. Then, also $trans f_{ext}^{\mathcal{P}}$ distinguishes ambiguous occurrences by means of the symbols that occur within the well-formed string they bound.

The correspondence between the intensional and the extensional transformation can be preserved only if the encoding is repaired by distinguishing ambiguous occurrences by means of symbols that do not occur under an optional node. This means that at the extensional level, we can correctly distinguish ambiguous occurrences if and only if the underlying encoding can be repaired by distinguishing occurrences that violate the point-of-choice delimiter using symbols that do not appear under an optional node.

To understand this point, consider the symbols that in the encoding occur under an optional node: in the enclosed instances, their presence

depends on the way the optional is instantiated. If $trans\,f_{ext}^{\mathcal{P}}$ used these symbols to disambiguate the occurrences, it would distinguish the ambiguous occurrences because of the presence/absence of the optional data in the encoded instance. For example, in the encoded instances of Figure 4, the first TR could be considered different from the second one, because the former encloses a well-formed string including the symbol *A*, while the latter does not, as the optional data is null-instantiated (as reported in Figure 2). This distinction would corrupt the pages, as instances of the same data type would be delimited by different delimiters in the encoded instances.

Our definition of $trans\,f_{ext}^{\mathcal{P}}$ prevents the creation of partitions based on symbols coming from the instantiation of optional nodes. In fact, $trans\,f_{ext}^{\mathcal{P}}$ creates the partition by associating with each ambiguous occurrence exactly one symbol from the well-formed string they bound. This choice guarantees that symbols coming from instances of optional nodes cannot participate in determining the partition. Let us illustrate this point by means of our example. Figure 8 reports the TR occurrences[5] of pages in Figure 4; the solid thin lines represent the associations with the set of symbols of the well-formed string each occurrence encloses. Observe that according to the above restriction, it is not possible to create a partition based on tags related to the instantiation of an optional data type. If we associate *A* with the second occurrence of TR, we cannot associate any symbol to the third TR, as any choice would lead the inclusion of the second TR as well. Conversely, we can create a partition of the TR occurrences if we assign exactly one tag with every element by considering the tags *HR* and *B*. The thicker lines represent a correct solution: it produces a partition of the ambiguous elements with a set of symbols that respect the above restriction, and it is easy to verify that it actually solves the point-of-choice violation.

It is important to observe that the approach fails if two (or more) optionals are instantiated with mutually disjunctive behaviors. In this case, ambiguous occurrences could be partitioned according to the optional that is actually instantiated under each occurrence. Clearly, such a partition would be a wrong solution. This reason imposes some restriction on the

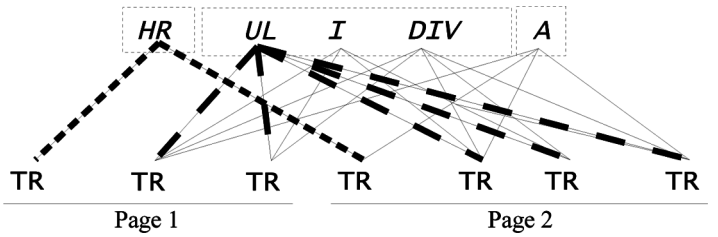

FIGURE 8 Partitioning ambiguous occurrences on pages.

characteristic sample for the class of languages we can repair; namely, it is required that optionals cannot be instantiated with mutually disjunctive behaviors.

The following definitions formally describe the above concepts. We first formalize the concept of partitioning set, which is the set of symbols that determines a partition over a set of occurrences with the restriction discussed above; then we introduce the function *partition*, which is used by $trans\,f^{\mathcal{P}}_{ext}$ and by $tranf_{int}$ to disambiguate ambiguous occurrences.

Definition 2 (Partitioning set). *Given a set O of occurrences over the alphabet $\Sigma \cup \overline{\Sigma}$, a binary relation* dom *over the same alphabet and its occurrences, and a symbol h of the alphabet, consider the set O_h of k occurrences of that symbol $O_h = \{o_1, \ldots, o_k\}$, where* $\texttt{symbol}(o_i) = h\ \forall i = 1, \ldots, k$, *and the sets* $\mathcal{L}_i = \{l \in \Sigma \cup \overline{\Sigma} |\ \texttt{dom}(o_i, l)\}$ *of all alphabet symbols with which each occurrence is in relation according to* dom.

A partitioning set *of the occurrences of h in O is defined as a set of schema symbols $\mathcal{L}^*$ such that*

$$|\mathcal{L}^* \cap \mathcal{L}_i| = 1, \quad i = 1, \ldots, k.$$

A partitioning set $\mathcal{L}^* = \{l_1, \ldots, l_n\}$ of n alphabet symbols, determines a partition $\pi^{\mathcal{L}^*}$ of O_h in n subsets

$$\pi^{\mathcal{L}^*} = \{O_h^{l_j}\}, O_h^{l_j} = \{o | o \in O_h \texttt{ and } \texttt{dom}(o, l_j)\}, \quad j = 1, \ldots, n.$$

In Figure 8 the dom relation among the TR occurrences and symbols *HR*, *UL*, *I*, *DIV*, *A* is represented by the solid thin edges connecting them. According to such a relation, there are several partitioning sets for the TR occurrences (namely, $\{HR, I\}, \{HR, DIV\}, \{HR, UL\}$): they all define the same partition that separates in two distinct subsets the first TR occurrences of each pages from the remaining ones.

The definition of the partitioning set is based on a generic binary relation dom over an alphabet $\Sigma \cup \overline{\Sigma}$ and a set of occurrences O over $\Sigma \cup \overline{\Sigma}$, $\texttt{dom} \subset O \times (\Sigma \cup \overline{\Sigma})$. We now introduce the binary relations $\texttt{dom}^{enc}_{int}$ and $\texttt{dom}^{\mathcal{P}}_{ext}$, which are specific versions of dom for occurrences at the intensional and extensional level, respectively.

Definition 3 (Binary relation $\texttt{dom}^{enc}_{int}$ over an encoding). *Given an encoding enc defined over a type σ, and an occurrence $o \in O^{enc}$ $\cdot\ \texttt{dom}^{enc}_{int} = \{(o, \texttt{symbol}(o')) \mid o'$ occurs exactly once in $subStr_{(o)}(enc(\sigma))$ is outside of any optional or list substring$\}$.*

In the above definition, O^{enc} denotes the set of the schema symbol occurrences associated with an encoding *enc* over a type σ: $O^{enc} = \{o | o\ \in$

$\texttt{occur}(enc(\sigma))$ *and* $\texttt{symbol}(o) \in \Sigma \cup \overline{\Sigma}\}$. Similarly, $O^{enc(I)}$ denotes the set of occurrences associated with the encoded instance $enc(I)$, and $O^{\mathcal{P}}$ the set of occurrences associated with a set of pages: $O^{enc(I)} = \{o \in \texttt{occur}(enc(I)) | \texttt{symbol}(o) \in \Sigma \cup \overline{\Sigma}\}$, and $O^{\mathcal{P}} = \cup_{i=1,\ldots,n} O^{enc(I_i)}$, with $\mathcal{P} = \{enc(I_1), \ldots, enc(I_n)\}$.

For example, for the serialized encoding in Figure 4 $\texttt{dom}^{enc}$ maps the first and the second TR occurrences to $\{HR, TD\}$ and $\{I, UL, DIV, TD, IMG, BR\}$, respectively.

A similar relation, $\texttt{dom}^{enc(I)}$, holds for occurrences and symbols of an encoded instance $enc(I)$.

Definition 4 (Binary relation $\texttt{dom}^{enc(I)}$ over an encoded instance). *Given an encoding enc defined over a type σ, an instance I of that type, and an occurrence $o \in O^{enc(I)}$:* $\texttt{dom}^{enc(I)} = \{(o, \texttt{symbol}(o')) | o'$ *occurs exactly once in* $subStr_{(o)}(enc(I))\}$.

Then $\texttt{dom}^{enc(I)}(o, s)$ holds if s is a symbol associated to an occurrence that belongs to the substring of $enc(\sigma)$ enclosed between o and its corresponding balancing occurrence.

For example, for the (left) page in Figure 4 $\texttt{dom}^{enc(I)}$ maps the first and the second TR occurrences to $\{HR, TD\}$ and $\{I, UL\ DIV, A\}$, respectively.

Observe that the definition of the two relations slightly differs: in the $\texttt{dom}^{enc}$ symbols that occur under an optional node do not participate in the relation, while they cannot be excluded in the relation $\texttt{dom}^{enc(I)}$ defined over encoded instances.

Finally, we introduce $\texttt{dom}^{\mathcal{P}}_{ext}$, which maps the occurrences of symbols in a set of pages to the symbol alphabet.

Definition 5 (Binary relation $\texttt{dom}^{\mathcal{P}}_{ext}$ over a set of pages). *Given an encoding enc defined over a type σ, a set of instances $\mathcal{I} = \{I_1, \ldots, I_n\}$ over that type:* $\texttt{dom}^{\mathcal{P}}_{ext} = \cup_{i=1,\ldots,n} \texttt{dom}^{enc(I_i)}$.

Based on the above definitions, $tranf_{int}$ and $tranf^{\mathcal{P}}_{ext}$ can compute partitions over the set of their occurrences. This is done by means of the function partition, reported in Figure 9, that takes as input a set of occurrences and a dom relation, and returns as output a set of partitions over the input set. Since partition is parametric with respect to the dom relation, it can be applied by both $tranf_{int}$ and $tranf^{\mathcal{P}}_{ext}$.

Segmentation and Disambiguation in Action

Apparently, the aforementioned techniques work autonomously: the segmentation introduces wrapping delimiters, the disambiguation solves

Function partition
  input: O a set of occurrences of (possibly annotated) symbols over $\Sigma \cup \overline{\Sigma}$,
    dom a *binary relation*, dom $\subset O \times (\Sigma \cup \overline{\Sigma})$
  output: a set of partitions of O
begin
  return $\{ \pi^{\mathcal{L}^*} \mid \mathcal{L}^*$ is a partitioning sets on O according to dom$\}$
end

FIGURE 9 Function partition.

violations of the point of choice delimiters condition. Nevertheless, they can actually support each other to achieve the goal of bringing pages into a prefix mark-up language.

We have seen that each segment corresponds to a (possibly empty) set of attributes of the root tuple. Therefore, if the disambiguation is run over segments rather than over pages, the ambiguous occurrences are conveniently taken in a more focused scope, enhancing the effectiveness of the approach. Suppose, for example, that a pair of list attributes occurs in the root tuple. If the the tuples of these lists have the same delimiters, their occurrences would be processed by the same disambiguation step. On the contrary, if the segmentation puts the two lists in different segments, the occurrences of the delimiters of their tuples will be elaborated by two separate disambiguation processes.

The disambiguation allows the approach to apply the segmentation also over deeply nested tuples. In fact, as a result of the disambiguation, all the well-formed strings bounded by a distinguished occurrence correspond to the same substring of the encoding. Therefore, the whole process of segmentation and disambiguation can be recursively run over sets composed of well-formed strings surrounded by the same distinguished occurrence in the partition computed by the disambiguation.

To give an example consider again the encoding of Figure 3: suppose that also the optional in the first nested tuple (the tuple delimited by <TR>) misses the wrapping delimiter condition (for example, assume that <DIV> and <TD> do not occur). As this option does not occur in the root tuple, the segmentation cannot affect it. However, we have seen that when the disambiguation runs over the set of TR occurrences it creates a partition composed of two subsets, one including the TR occurrences that delimit the tuple, and the other including those that delimit the list. Hence, we can run the segmentation again over the set of well-formed strings surrounded by every tag occurrence in a subset of the partition. In our example, the segmentation would be triggered first against the set of well-formed strings bounded by the TR occurrences of the list; then against the set of well-formed strings bounded by the TR occurrences of the tuple: in this case

the presence of the occurrences of $</\texttt{IMG}>$, and $<\texttt{BR}>$, which are unique in the context of the inner tuple, allows the segmentation to introduce virtual delimiters, solving the missing wrapping delimiter in the nested tuple as well.

THE *trans* $f^{\mathcal{P}}_{ext}$ PREPROCESSING TRANSFORMATION

We now present the complete definition of our preprocessing transformation *trans* $f^{\mathcal{P}}_{ext}$, which builds on the techniques discussed in the previous section, and operates on pages to bring them into the class of prefix mark-up languages. To prove its correctness we also present its intensional counterpart, *trans* f_{int}: based on it we define the conditions that must hold in the originating encoding to guarantee that the transformation actually operated on pages by *trans* $f^{\mathcal{P}}_{ext}$ succeeds in bringing pages into the class of prefix mark-up languages.

Overall, our approach corresponds to define a new class of languages that we call distinguishable mark-up languages: we prove the new class of languages is still inferrable in the limit, and we formally define its associated characteristic sample.

We first introduce the concept of distinguishable rich set of instances, which is a restriction of the concept of rich set of instances given in Definition 1 for prefix mark-up languages.

Definition 6 (Distinguishable Rich Set of Instances). *A set of instances* $\mathcal{I} = \{I_1, \ldots, I_n\}$ *of a schema* σ*, is a distinguishable rich set of instances if* $\mathcal{I}$ *is a rich set of instances and additionally*: (i) *optionals do not exhibit mutually disjunctive behaviors; that is, for every pair of optionals, and for every tuple that either directly or indirectly contains both optionals, there is at least one such tuple in which they are both instantiated at least once or they are both null*; (ii) *optionals do not exhibit segmenting behaviors; that is, for every optional and for every tuple that either directly or indirectly contains that optional, there is at least one such tuple in which it is not instantiated or it is instantiated more than once.*

Similarly to prefix mark-up languages, also for distinguishable mark-up language the notion of characteristic sample is based on distinguishable rich set of instances.

The Intensional and Extensional Transformations

Figure 10 reports the algorithms corresponding to the intensional (left) and extensional (right) transformations. Due to the strict correspondence between an encoding and the corresponding encoded instances, the

Algorithm $transf_{int}$
input: a Mark-up encoding enc over a data type σ

output: a Mark-up encoding enc over σ
begin
Let $\mathcal{T}$ be a set of occurrences in O^{enc}
add the first occurrence of $enc(\sigma)$ to $\mathcal{T}$

while ($\mathcal{T}$ is not empty) begin
Let n be one occurrence removed from $\mathcal{T}$
do begin
$enc = \mathsf{segment}_{int}(enc, n)$
Let H be the set of all the occurrences of
(possibly annotated) schema symbols in $enc(\sigma)$
Let $E_h = \{h_k \in H | \mathsf{symbol}(h_k) = h\}$
for each E_h begin
choice $\pi^{\mathcal{L}^*}$ in partition$(E_h, \mathsf{dom}^{enc}_{int})$
for each $O^{l_j}_h, j = 1, \ldots, n$
for each $o \in O^{l_j}_h$
$annotate(o, l_i)$
add to $\mathcal{T}$every occurrence
that has been distinguished in p

end
end
while (some change occurs in $enc(\sigma)$)
end
return enc
end

Algorithm $transf^{\mathcal{P}}_{ext}$
input: a set of sample pages $\mathcal{P} = \{p_i = enc(I_i)\}$
generated by an encoding enc defined over a data type σ
and applied over a set of instances $\{I_i\}$ of σ
output: a set of sample pages $\mathcal{P}$
begin
Let $\mathcal{T}$ be a set of set of occurrences in $O^{\mathcal{P}}$
add $\mathcal{P}$ to $\mathcal{T}$
while ($\mathcal{T}$ is not empty) begin
Let $\mathcal{F}$ be one set of occurrences removed from $\mathcal{T}$
do begin
$\mathcal{P} = \mathsf{segment}_{ext}(\mathcal{P}, \mathcal{F})$
Let H be the set of all the occurrences of
(possibly annotated) schema symbols in $\mathcal{P}$
Let $E_h = \{h_k \in H | \mathsf{symbol}(h_k) = h\}$
for each E_h begin
choice $\pi^{\mathcal{L}^*}$ in partition$(E_h, \mathsf{dom}^{\mathcal{P}}_{ext})$
for each $O^{l_j}_h, j = 1, \ldots, n$
for each $o \in O^{l_j}_h$
$annotate(o, l_i)$
add to $\mathcal{T}$every set of occurrences
that has been distinguished in p
end
end
while (some change occurs in $\mathcal{P}$)
end
return $\mathcal{P}$
end

FIGURE 10 Intensional and extensional transformations.

structure of the algorithms is the same: the former works on an encoding, the latter on a set of sample pages, i.e., encoded instances. Both transformations iteratively transform their input by performing segmentation and disambiguation steps.

The segmentation operates on pages and on the serialized representation of the encoding: it introduces virtual delimiters and annotates occurrences with symbols that identify the segment each occurrence belongs to.

The disambiguation is performed by computing a partition over a set of ambiguous occurrences, that is, occurrences that are associated with the same symbol of the schema alphabet. The partition is computed by applying the proper binary relation: $\mathtt{dom}^{\mathcal{P}}_{ext}$ on the pages, $\mathtt{dom}^{enc}_{int}$ on the encoding.

As several partitions may be returned, there is a `choice` construct to nondeterministically pick one of the computed partitions. The partitioning set associated to the chosen partition is then used to annotate, that is, distinguish, the ambiguous occurrences. The distinguished occurrences individuate at the intensional level a substring, at the extensional level a set of page fragments, i.e., substrings of encoded instances. These are added to

the set $\mathcal{T}$, which stores the elements on which the whole process is executed until some change occurs on the input.

We say that $trans\,f_{int}$ deterministically distinguishes the input encoding whenever all its invocations of the function `partition` return at most one partition, and all the returned partitions are distinguishing partitions.

Definition 7 (Distinguishing Partitioning Set). *A partitioning set $\mathcal{L}^*$ of a set of occurrences O is called a distinguishing partitioning set if $|\mathcal{L}^*| = |O|$.*

In other words, a distinguishing partitioning set of occurrences identifies a partition made of singletons. The following results establish the class of languages for which $trans\,f^{\mathcal{P}}_{ext}$ is correct, that is, the class of languages that corresponds to pages that can be repaired by means of our transformation.

Definition 8 (Distinguishable Mark-Up Encoding, Distinguishable Mark-Up Languages). *The class of distinguishable mark-up languages is defined as the class of languages obtained by applying to the instances of an abstract data type σ a distinguishable mark-up encoding, that is, a mark-up encoding function enc for which $trans\,f_{int}$ deterministically distinguishes enc and $trans\,f_{int}(enc)$ is a prefix mark-up encoding.*

Theorem 1 (Correctness). *Given a set of encoded strings, called $\mathcal{P} = \{enc(I_1), enc(I_2), \ldots, enc(I_n)\}$, of a distinguishable rich set of instances of a type σ according to a distinguishable mark-up encoding enc, then*

$$trans\,f^{\mathcal{P}}_{ext}(enc(I_i)) = trans\,f_{int}(enc) \cdot (I_i), \quad i = 1, \ldots, n.$$

Since the proof of the theorem requires the introduction of a number of technical notions, for the sake of readability we have moved it to Appendix A.

From Theorem 1, the corollary follows.

Corollary 1 (Characteristic Sample of Distinguishable Mark-Up Languages). *Any set of pages $\mathcal{P} = \{enc(I_1), \ldots, enc(I_n)\}$, obtained as the encodings of a distinguishable rich set of instances of a schema σ, according to a distinguishable mark-up encoding enc is a characteristic sample for $enc(\sigma)$.*

The hypothesis of Theorem 1, in practice can easily be satisfied because of the richness of the HTML of modern websites; the segmentation tends to isolate instances of distinct subtypes, augmenting the possibility that the disambiguation problem triggered over each segment admits a unique solution. On the other hand, disambiguated occurrences allow finer segmentations on instances of deeply nested types.

The results of our empirical experimentation, as reported in the next section, confirm the effectiveness of the approach in real-life web pages.

IMPLEMENTATION AND EXPERIMENTS

To evaluate the overall impact of our approach, we have developed a Java prototype that implements the transformation $trans f_{ext}^{\mathcal{P}}$ and we have run it to preprocess pages from several real-life websites.

The implementation of the segmentation and disambiguation techniques is rather simple, but it is worth reporting here some technical details about the development of the module that performs the disambiguation, as the problem of computing a partitioning set is an instance of the set partitioning problem, a well-known and widely studied NP-complete problem (Balas and Padberg 1976; Hoffman and Padberg 1993). To solve it, we have developed a depth-first search algorithm in a state space. The algorithm takes as input a set of elements $E = \{e_1, e_2 \ldots, e_k\}$, each element e_i associated with a set of symbols $\mathcal{L}_i$, and produces as output a partition of the elements such that each class of the partition is associated with exactly one symbol. As a very first step, before starting the search procedure, the algorithm discards the symbols, if any, that are associated with all the elements of the input set: these symbols would lead to the trivial solution made of a unique subset (containing all the elements). In the search procedure, the algorithm makes a decision of one symbol assignment in each step. If the constraints of the problem are violated the algorithm backtracks.

Because of the backtracking, the algorithm is subject to exponential behaviors. In the worst case, the search is of exponential size with respect to the number of symbols. However, we observe that the number of symbols that can really contribute to the solution is usually small. Also, the problem is strongly constrained, in the sense that early choices immediately prune a large number of successive searches. We claim that the number of symbols is small, as several symbols are associated to the same set of elements and therefore they are equivalent in the sense that they would produce the same partition. In Figure 8, the dashed boxes indicate clusters of equivalent symbols. For example, *UL* and *I* are equivalent, and $\{HR, I\}$ and $\{HR, UL\}$ are equivalent partitioning sets that produce the same partition. As the number of clusters is much smaller than the number of symbols, we compute the partition using clusters of equivalent symbols instead of symbols. This choice produces great improvements in terms of performances, as it significantly reduces the search space. The second aspect that limits the processing time is related to the nature of the problem, which is strongly constrained, and then simple heuristics can contribute to minimize the branching factor of the search tree. For example, during the search we

follow a most constrained strategy by considering first the symbols that are associated with the smallest set of elements; if this choice leads to a failure, we prune the search tree earlier.

Experiments

The developed prototype has been used to run a number of experiments on real HTML sites. All experiments have been conducted on a machine equipped with an Intel Pentium Mobile processor working at 2 GHz, with 1 GB of RAM, running Linux (kernel 2.6) and Sun JRE 1.5.

We have conducted two sets of experiments. The first set of experiments involves the Wien test-bed, which was selected by Kushmerick in his seminal work on wrapper induction (2000). It is worth saying that this test-bed is rather old; however, experimenting our technique against this data-set has a two-fold significance for our approach. First, since the same test-bed has been used in our previous works (Crescenzi et al. 2001, 2004; Crescenzi and Mecca 2004) it allows us to progressively compare the results of our developments. Second, as the HTML code of Wien's pages is quite poor, applying our preprocessing technique over these pages is challenging, as the poorness of HTML is a handicap for our approach, which on the contrary leverages on the richness of HTML of modern websites.

The second set of experiments involves pages taken from well-known websites. To evaluate the ability of our preprocessing technique, we have chosen pages with different levels of nesting of the underlying data types.

The experiments were conducted as follows. First, we have preprocessed the pages with the techniques discussed in the paper;[6] then, the output pages were given as input to our prototype that implements the MATCH algorithm. The generated wrappers were finally used to extract data from the pages. For each experiment we have used approximately 10 sample pages.

It is worth saying that MATCH is able to also run on pages that do not comply to prefix mark-up languages; but in this case, as discussed (Crescenzi and Mecca 2004), there is no guarantee that a solution can be found, or that it can produce useful solutions. Moreover, our prototype implementation of MATCH includes techniques that enhance the formal framework with methods that identify and extract irregular regions (Crescenzi et al. 2004). Essentially, when it is not able to infer a wrapper, it computes the minimal DOM subtree that causes the failure and considers it as an attribute to be extracted. The drawback of this approach is that the extracted subtree might be rather large (in the worst case it can be rooted even at the document root), and it might include several data fields.

The presence of ambiguous occurrences (in the sense discussed in this article) in the HTML code of real-life web pages is one the most frequent causes that forces MATCH to extract large subtrees. Therefore, in this context the preprocessing technique proposed in the article assumes a relevant role, and its effectiveness can be evaluated by observing the number of values that can be correctly extracted with the wrappers generated by MATCH with or without the preprocessing step.

To this end, for each dataset we have manually developed an ideal wrapper, which has then been used to extract the correct values.[7] Then we have run MATCH against the original pages, and against the pages treated by means of our preprocessing technique. The number of values extracted by the wrappers generated by MATCH in the two stages can then be compared with the number of correct values.

We distinguish extracted and partially extracted values. For a given basic type attribute A, let S_M denote the set of values extracted by a wrapper generated by MATCH, and S_i the set of values extracted by the ideal wrapper. We say a value for A is correctly extracted if $S_M = S_i$. It is partially extracted if each element of S_i occurs as a part of a value of a basic attribute B in the dataset extracted by MATCH. Usually this happens because several attributes are grouped within a subtree.

As the preprocessing is an additional step for the inference process, we also report computing times; namely, the computing times include both the cost of the transformation and of the MATCH application.

Figure 11 illustrates the results of our experiments on the Wien testbed. We list the results obtained running MATCH directly on the sample pages, and those obtained after preprocessing the input pages with the transformation *trans* f_{ext} presented in the article.

Relevant improvements are marked in gray (sources 4, 7, 9, 18, 29, 30). Several values that previously were extracted in a subtree are now extracted correctly. This is a consequence of the preprocessing, which has disambiguated the HTML code, letting MATCH to infer a more accurate wrapper. It is worth saying also that the results obtained from samples 3, 13, 26, and 27 could be considered optimal with respect to our current system, as the expected values of the Wien dataset were computed with a wrapper based on a schema alphabet richer than ours: we are currently using only tags, whereas the alphabet of the ideal wrappers includes punctuation symbols as well.

Let us now comment on the results produced on the second dataset, which are reported in Figure 12. In this case we had net improvements in the majority of sources (1, 3, 7, 10–17); this is mainly due to the more involved and richer HTML code of these pages with respect to the samples in the Wien dataset. However, we have also observed one case (sample 5) in which the effects of the preprocessing technique have been negative.

		without Preprocessing		Time (ms.)	with Preprocessing		Time (ms.)
Sample	#values	Extracted	Partially Extracted		Extracted	Partially Extracted	
1	1612	1612		4588	1612		16420
2	1010*	1010		2124	1010		4233
3	1044	522	522	2455	522	522	9003
4	400		400	787	267	133	1602
5	144	144		2022	144		27278
6	100*	100		753	100		2529
7	1688		1688	372	359	1329	988
8	654	654		2415	654		16163
9	572*	80	492	2845	310	182	27409
10	400	295	42	2296	295	42	4979
11	400*		400	2083	10	390	10719
12	888	862		1904	862		2078
13	400	290	108	1154	290	108	3154
14	2910	2900	10	1904	2900	10	6178
15	708	708		2637	708		1929
16	100*	100		302	100		8198
17	1891	1891		1065	1891		11058
18	2436	389	2047	1383	2436	389	5121
19	1000	794	200	939	794	200	2305
20	1962	1308		946	1308		13261
21		no wrapper			no wrapper		
22	3000	3000		2462	3000		17044
23	1635	1635		2269	1635		5634
24	1550*	1550		5468	1550		18567
25	654	654		456	654		1373
26	386*	10	376	970	10	376	43942
27	60	30	30	1510	30	30	38632
28		no wrapper			no wrapper		
29	425*		425	735	425		13383
30	240	30	210	131	120	120	835

**not labeled in the wien dataset, estimated by the authors of the present paper*

FIGURE 11 Experimental results: the effects of the preprocessing on the Wien data set.

Analyzing the logs of the experiment, we have observed that the failure is related to the segmenting behavior of an optional within a list. According to the formal framework of this article, we can conclude that the set of pages used to infer the wrapper is not a characteristic sample for the correct language.

Finally, we observe that computing times are usually greater when the preprocessing is enabled. However, it is interesting to observe that there are some exceptions (sources 1, 4, 5, 8, 9): in these cases, the repairing technique significantly simplifies the inference process, and improves the overall efficiency.

Sample	site	#values	Nesting Level	without Preprocessing		Time (ms.)	with Preprocessing		Time (ms.)
				Extracted	Partially Extracted		Extracted	Partially Extracted	
1	http://www.fifaworldcup.com (players)	319	1		319	25202	264	55	13639
2	http://catalogue.berkenstockcentral.com	188	1	129	59	3866	129	59	4472
3	http://hotjobs.yahoo.com	1203	1		1203	20280	1203		22042
4	http://www.zappos.com	1104	1	1094	10	41213	1104		40886
5	http://finance.yahoo.com	249	1	249		7027	170	79	6216
6	http://www.watchzone.com (watches)	90	1	90		3825	90		5023
7	http://www.zap2it.com	184	1	136	40	2028	176	8	2126
8	http://www.polo.com	44	1	44		5953	44		4816
9	http://www.fifaworldcup.com (history)	140	0	140		10996	140		9250
10	http://www.fifaworldcup.com (coaches)	892	0	32	160	9979	892		10752
11	http://www.google.it (searches)	117	1	39	39	1335	78		8543
12	http://www.google.it (scholar)	132	1	33	99	1840	93	33	17384
13	http://movies.yahoo.com (awards)	504	2	68	436	7683	395	109	26487
14	http://www.sportline.com (cbs tracks)	2317	1	1801	36	26314	1837		35923
15	http://half.ebay.com (books)	189	2	0	168	4528	153	15	16018
16	http://www.ncbi.nlm.nih.gov (searches)	400	1	80	320	7744	320	80	149410
17	http://www.informatik.uni-trier.de (authors)	533	2	61	470	5468	228	303	22766

FIGURE 12 Experimental results: the effects preprocessing on modern websites.

RELATED WORK

The issue of semi-automatically generating wrappers for extracting data from fairly structured HTML pages is a well-studied problem, and several approaches have been proposed in the literature (Freitag 1998; Soderland 1999; Muslea 1999; Embley et al. 1999b; Baumgartner et al. 2001; Laender et al. 2002b). One of the main limitations of these approaches is that they need a training phase, in which the system is fed with a number of labelled examples. This task involves a manual phase as pages need to be labelled by a human expert that marks the relevant pieces of information. Also, most of these proposals assume *a-priori* knowledge about the organization of data in the target pages (e.g., pages must contain a list of flat records). Modern tools ease the burden of the labelling activity; however, since pages can change frequently, wrappers are brittle and their maintenance is costly. Thus developing techniques that automate the wrapper generation task can help web data extraction systems to reduce the costs of the wrapper maintenance.

Other examples of wrapper generating systems base the data-extraction process on the use of domain-specific ontologies (Embley et al. 1999a; Davulcu, Mukherjee, and Ramakrishnan 2002). In this approach, an ontology provides concise descriptions of the conceptual model of data in the page and also allows for recognizing attribute occurrences in the text. An interesting contribution of these researches is that an 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. On the other hand, the approach strongly depends on the domain and assumes that the extracted data must be organized as a flat table.

More recently, several researchers have tackled the issue of automatically inferring a wrapper given an input set of sample pages. Lerman et al. (2004) have developed a system for the automatic extraction and segmentation of records from web tables. Their approach relies on a specific pattern that occurs in many websites for presenting lists of items: an index page containing a list of short summaries, one for each item, which includes a link leading to a page about details of the specific item. The proposed technique aims at segmenting the index page by leveraging the redundancy of information that this pattern produces: first information from detail pages is used to segment the index page into records. The main limitations of this approach is that it is based on a quite specific pattern.

Automatic data extraction from pages containing flat lists of tables is the subject of several works. ViNTs (Zhao et al. 2005) proposes a technique that uses both visual and DOM-related features. The former are related to features depending on the visual presentation of one page when displayed on a browser; the underlying idea is that the iterative structure of a list of records is reflected both in the spatial organization of data in the browser and in the structure of the DOM tree. Visual information for segmenting web pages is also proposed by Zhai and Liu (2005) and Liu and Zhai (2005); their approach complements visual and tree alignment of the fields of a repeated item. Compared to our approach, these proposals do not need multiple pages to infer a wrapper. On the other side, they can infer a wrapper only for a flat list of tuples.

Arasu and Garcia-Molina (2003) have proposed an algorithm, called EXALG, for extracting structured data from a collection of web pages generated by encoding data from a database into a common template. To discover the underlying template that generated the pages, EXALG uses so-called large and frequently occurring equivalent classes (LFEQ), i.e., sets of words that have similar occurrence patterns in the input pages. EXALG has some points in common with our approach. First, also in EXALG, pages are seen as the result of an encoding process that serializes complex objects into strings. Also, LFEQs can be considered as a generalization of the clusters of labels used in the disambiguation algorithm.

The main limitation of traditional grammar inference techniques, such as those developed by Angluin (1982) and by Radhakrishnan and Nagaraja (1987), when applied to modern information extraction problems 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 because of the unrealistic assumptions on the characteristic samples that need to be presented to the inference algorithm.

Some proposals use grammar inference techniques for information extraction, in the spirit of this article. For example, Fernau concentrates on XML documents, and studies the issue of using regular language-learning

algorithms to infer the productions of the (contextual-free) grammar associated with the DTD (Fernau 2000). Fernau's work has strongly influenced the definition of the formal framework of ROADRUNNER (Crescenzi and Mecca 2004). Ideally, his approach could be applied over XHTML documents; however, the DTD associated to the XHTML of a web unlikely reflects the structure of the encoded data, due to large amounts of XHTML code used only for presentation purposes. Moreover, the strategy proposed by Fernau requires a bias on the learning algorithm. Other contributions try to apply grammar inference techniques to information extraction from HTML codes. For example, Chidlovskii (2000) defines a wrapper generation algorithm based on the inference of *k*-reversible grammars; however, the approach is not fully automatic, and suffers from some of the limitations of traditional grammar inference techniques discussed earlier in this article. In Hong and Clark (2001), the authors 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 article, tree automata are used to infer tree languages for HTML pages; also in this case a preliminary annotation phase is required.

CONCLUSIONS AND FUTURE WORK

Grammar inference provides an elegant and sound formal framework for studying thorough techniques for the automatic generation of web wrappers. Developing robust systems that implement these techniques for extracting data from real-world web pages is a challenging issue.

In this article, we have shown that it is possible to support the theoretical framework studied for the inference of prefix mark-up languages with an effective and efficient preprocessing phase. Our practical solution is the result of a study of the relationship between real pages and formal languages.

We are currently working in order to further improve our approach for the automatic extraction of data from the web. A first direction we are evaluating is that of developing methods to extend the schema alphabet to symbols that also occur in the data alphabet. The idea is that of computing the schema alphabet dynamically by means of a statistical analysis of the symbols that occur in the sample pages. Also, we have observed that another limitation of our approach is the lack of expressive power of our languages, which do not include the disjunction. Therefore, we are studying techniques for addressing the issues of introducing disjunctions in the inference process to find a better trade-off between expressivity and performances.

REFERENCES

Abiteboul, S. and C. Beeri. 1995. On the power of languages for the manipulation of complex objects. *VLDB Journal* 4(4):117–138.

Angluin, D. 1982. Inference of reversible languages. *Journal of the Association for Computing Machinery* 29(3):741–765.

Arasu, A. and H. Garcia-Molina. 2003. Extracting structured data from web pages. In: *ACM SIGMOD International Conf. Management of Data (SIGMOD'2003)*, San Diego, CA, pp. 337–348.

Balas, E. and M. W. Padberg. 1976. Set partitioning: A survey. *SIAM Review* 18:710–760.

Baumgartner, R., S. Flesca, and G. Gottlob. 2001. Visual web information extraction with lixto. In: *International Conf. Very Large Data Bases (VLDB 2001)*, Roma, Italy, September 11–14, pp. 119–128.

Chang, C.-H., M. Kayed, M. R. Girgiz, and K. F. Shaalan. 2006. A suvery of web information extraction systems. *Computer* 18(10):1411–1428.

Chidlovskii, B. 2000. Wrapper generation by *k*-reversible grammar induction. In: *Proc. Internat Workshop on Machine Learning and Information Extraction (ECAI'00)*, pp. 61–72.

Crescenzi, V. and G. Mecca. 2004. Automatic information extraction from large web sites. *Journal of the ACM* 51(5):731–773.

Crescenzi, V., G. Mecca, and P. Merialdo. 2001. ROADRUNNER: Towards automatic data extraction from large web sites. In: *International Conf. Very Large Data Bases (VLDB 2001)*, Roma, Italy, September 11–14, pp. 109–118.

Crescenzi, V., G. Mecca, and P. Merialdo. 2004. Handling irregularities in roadrunner. In: *ATEM-2004: The AAAI-04 Workshop on Adaptive Text Extraction and Mining*, San Jose, CA.

Davulcu, H., S. Mukherjee, and I. V. Ramakrishnan. 2002. Extraction techniques for mining services from web sources. In: *IEEE International Conference on Data Mining*, pp. 601–604.

Embley, D. W., M. D. Campbell, Y. S. Jiang, S. W. Liddle, Y. K. Ng, D. Quass, and R. D. Smith. 1999a. Conceptual-model-based data extraction from multiple-record web pages. *Data & Knowledge Engineering* 31(3):227–251.

Embley, D. W., Y. S. Jiang, and Y. Ng. 1999b. Record-boundary discovery in web documents. In: *ACM SIGMOD International Conf. Management of Data*, pp. 467–478.

Fernau, H. 2000. Learning XML grammars. In: *Proc. 2nd Machine Learning and Data Mining in Pattern Recognition MLDM'01*, vol. 2123, LNCS/LNAI. Leipzig, Germany: Springer, pp. 73–87.

Freitag, D. 1998. Information extraction from html: Application of a general learning approach. In: *Proc. 15th Conference on Artificial Intelligence AAAI-98*, pp. 517–523.

Hoffman, K. and M. Padberg. 1993. Solving airline crew scheduling problems by branch and cut. *Management Science* 39(6):657–682.

Hong, T. W. and K. L. Clark. 2001. Using grammatical inference to automate information extraction from the Web. In: *Principles of Data Mining and Knowledge Discovery, 5th European Conference, PKDD 2001*, Freiburg, Germany, September 3–5, Lecture Notes in Computer Science 2168, pp. 216–227.

Hull, R. 1988. A survey of theoretical research on typed complex database objects. In: *Databases*, ed. J. Paredaens. London: Academic Press, pp. 193–256.

Kosala, R., J. Van den Bussche, M. Bruynooghe, and H. Blockeel. 2002. Information extraction in structured documents using tree automata induction. In: *Proc. European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2002)*, pp. 299–310.

Kushmerick, N. 2000. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence* 118: 15–68.

Laender, A., B. Ribeiro-Neto, A. Da Silva, and Teixeira, J. 2002a. A brief survey of web data extraction tools. *ACM SIGMOD Record* 31(2):84–93.

Laender, A. H. F., B. A. Ribeiro-Neto, and A. S. da Silva. 2002b. Debye – data extraction by example. *Data Knowl. Eng.* 40(2):121–154.

Lerman, K., L. Getoor, S. Minton, and C. A. Knoblock. 2004. Using the structure of web sites for automatic segmentation of tables. In: *ACM SIGMOD International Conf. Management of Data (SIGMOD'2004)*, Paris, France.

Liu, B. and Y. Zhai. 2005. Net – a system for extracting web data from flat and nested data records. In: *Proc. Web Information Systems Engineering – WISE 2005, 6th International Conference on Web Information Systems Engineering*, New York, November 20–22, pp. 487–495.

Muslea, I., S. Minton, and C. A. Knoblock. 1999. A hierarchical approach to wrapper induction. In: *Proc. Third Annual Conference on Autonomous Agents*, pp. 190–197.

Radhakrishnan, V. and G. Nagaraja. 1987. Inference of regular grammars via skeletons. *IEEE Transactions on Systems, Man and Cybernetics* 17(6):982–992.

Soderland, S. 1999. Learning information extraction rules for semistructured and free text. *Machine Learning 34*(1–3):233–272.

Wang, J. and F. H. Lochovsky. 2002. Data-rich section extraction from html pages. In: *Proc. 3rd International Conference on Web Information Systems Engineering (WISE 2002)*, December 12–14, Singapore: IEEE Computer Society. Washington, DC, pp. 313–322.

Zhai, Y. and B. Liu. 2005. Web data extraction based on partial tree alignment. In: *Proc. 14th International Conference on World Wide Web, WWW 2005*, Chiba, Japan, May 10–14, pp. 76–85.

Zhao, H., W. Meng, Z. Wu, V. Raghavan, and C. T. Yu. 2005. Fully automatic wrapper generation for search engines. In: *Proc. 14th International Conference on World Wide Web, WWW 2005*, Chiba, Japan, May 10–14, pp. 66–75.

NOTES

1. <IMG/> can be considered as a shortcut for <IMG></IMG>; similarly for
 and <HR/>.
2. Also the UFRE symbols that occur in the serialization of the encoding are balanced.
3. We assume $<v>, </v>$ are special symbols in the schema alphabet that cannot be used to construct the encoding.
4. For the sake of presentation, we will explain later how to obtain precisely these symbols even in the presence of occurrences that enclose the delimiters of inner nodes of the underlying type as in this case.
5. For the sake of readibility of the figure, we do not report all the involved symbols.
6. We also tidy the pages with `nekoHTML` (`http : //www.apache.org/~andyc/neko/doc/html/`), a tool to fix up fix errors and make the code compliant with XHTML.
7. For the Wien test-bed, when available, we have used the set of labels provided with the dataset.

APPENDIX A. PROOFS: CORRECTNESS, CHARACTERISTIC SAMPLES OF A DISTINGUISHABLE MARK-UP LANGUAGE

To prove the correctness theorem, we need a couple of preliminary results.

Proposition 1. *Given an encoding enc of a type σ over a schema alphabet $\Sigma \cup \overline{\Sigma}$, and a distinguishable rich set of instances $\mathcal{I} = I_1, \ldots, I_n$, let $\mathcal{P} = \{enc(I_i)\}$, $i = 1, \ldots, n$.*

A symbol h occurs only once in O^{enc} iff it occurs exactly one in every $O^{enc(I_i)}$, $i = 1, \ldots, n$

Observe that since the occurrences of alphabet symbols in an encoded instance are produced by "serializing" occurrences of alphabet symbols in the corresponding encoding, it holds a functional relationship between the two types of occurrences. We introduce a function, `generator`, which maps

every symbol occurrence in an encoded instance into the symbol occurrence in the encoding from which it has been generated during the encoding process.

Observe that given $o \in O^{enc(I)}$, then $\texttt{generator}(o) \in O^{enc}$ and $\texttt{symbol}(\texttt{generator}(o)) = \texttt{symbol}(o) \in \Sigma \cup \overline{\Sigma}$.

Given a set of occurrences O over an alphabet $\Sigma \cup \overline{\Sigma}$, dom is defined as a binary relation over the alphabet symbols and its occurrences: $\texttt{dom} \subset O \times (\Sigma \cup \overline{\Sigma})$.

The following lemma clarifies the relationship between intensional and extensional partitions of occurrences set.

Lemma 1. *Given an encoding enc of a type σ over a schema alphabet $\Sigma \cup \overline{\Sigma}$, a schema symbol h, a characteristic sample $\mathcal{P} = \{enc(I_i)\}$, $i = 1, \ldots, n$ of the markup language corresponding to $enc(\sigma)$, let $\pi^{\mathcal{L}^*}_{int}$ be the only intensional partition associated with the occurrence set O^{enc}_h. Then, $O^{\mathcal{P}}_h$ admits only one extensional partition $\pi^{\mathcal{L}^*}_{ext}$ associated with the same partitioning set $\mathcal{L}^* = \{l_1, \ldots, l_n\}$ and*

$$\pi^{\mathcal{L}}_{int}{}^* = \{O^{l_j}_h\}, O^{l_j}_h = \{o | o \in O^{enc}_h \text{ and } \texttt{dom}^{enc}_{int}(o, l_j)\}, j = 1 \ldots n;$$
$$\pi^{\mathcal{L}}_{ext}{}^* = \{N^{l_j}_h\}, N^{l_j}_h = \{o | o \in O^{\mathcal{P}}_h \text{ and } \texttt{dom}^{\mathcal{P}}_{ext}(o, l_j)\}, j = 1 \ldots n;$$
$$O^{l_j}_h = \{o' | o' = \texttt{generator}(o), o \in N^{l_j}_h\}$$

Proof. h can be written as $\texttt{symbol}(\texttt{generator}(o))$, where o is a symbol occurrence in one of the pages; then consider another generic symbol occurrence o' in a sample page and let α and β, respectively, be the labels of the two nodes whose delimiters contain $\texttt{generator}(o)$ and $\texttt{generator}(o')$. Consider that by definition of types and encodings over such types, and by definition of the binary relations $\texttt{dom}^{enc}_{int}$ and $\texttt{dom}^{\mathcal{P}}_{ext}$, every intensional partition trivially has an extensional counterpart. The other direction is slightly more complex since $(o', h) = (o', \texttt{symbol}(\texttt{generator}(o))) \in \texttt{dom}^{\mathcal{P}}_{ext}$ entails $(\texttt{generator}(o'), \texttt{symbol}(\texttt{generator}(o))) \in \texttt{dom}^{enc}_{int}$ only if in the tree representation of type σ an optional node does not occur between the nodes labelled α and β. The optional nodes can produce extensional partitions that do not correspond to any intensional partition, if and only if the optional instantiations exhibit mutually disjunctive behaviors which are excluded by hypothesis.

The correctness (Theorem 1) can now be proven.

Proof. Observe that $trans\,f_{int}$ and $trans\,f^{\mathcal{P}}_{ext}$ have the same structure; but whereas $trans\,f_{int}$ works on occurrences of schema symbols that delimit portions of the encodings, $trans\,f^{\mathcal{P}}_{ext}$ works on sets of occurrences of schema symbols that delimit fragments of pages, i.e., well-formed substrings of the encoded instances. The theorem is proven if we show that $trans\,f_{int}$ works on

one substring bounded by n if and only if $trans f_{ext}^{\mathcal{P}}$ works on one set of substring bounded by a set of occurrences O such that $\texttt{generator}(o) = n \; \forall \; o \; \in O$. The proof can be given by induction on the number of steps performed by one of the two algorithms by observing that initially $\mathcal{P} = \cup_{i=1,\ldots,n} enc(I_i)$, Proposition 1 guarantees the first segmentation step and Lemma 1 the inductive step.

Corollary 1, characteristic sample of distinguishable mark-up languages, follows directly from Theorem 1 and from the fact that that prefix mark-up languages are identifiable in the limit (Crescenzi and Mecca 2004).