Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

On Automatic Information Extraction from Large Web Sites

Valter Crescenzi[1], Giansalvatore Mecca[2]

RT-DIA-76-2003 **Gennaio 2003**

(1) Università di Roma Tre,
Via della Vasca Navale, 79
I-00146 – Roma, Italy.
`crescenz@dia.uniroma3.it`

(2) Università della Basilicata,
C.da Macchia Romana,
I-85100 - Potenza, Italy.
`mecca@unibas.it`

Abstract

Information extraction from Web sites 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 Web sites, 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 – i.e., 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 paper stand in the definition of a class of regular languages, called the prefix mark-up languages, that nicely abstract the structures usually found in HTML pages, and in the definition of a polynomial-time unsupervised learning algorithm for this class. The paper 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 paper has been implemented in a working prototype. Experiments on known Web sites further demonstrate the practical applicability of the proposed approach.

Contents

1 Introduction

We may probably consider the Web as the largest "knowledge base" ever developed and made available to the public. However HTML sites are in some sense modern 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 Web sites were based on manual techniques [5, 18, 33, 9, 20]. 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 paper 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 [1, 14, 34, 24, 28, 12] have attacked the problem of information extraction from Web sites. 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 [23, 25] 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 – i.e., 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, due to some limitations of the traditional framework, essentially none of the recent approaches to Web information extraction reuse theories and techniques from the grammar inference community. As a consequence, 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 [15]) 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* [15]. 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 [3, 31], 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 mainly 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*, i.e., a set of samples of the target language with specific features: (*i*) it has to be a "finite telltale" of the language, i.e., 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 very unlikely that a wrapper can be correctly inferred by looking at a bunch of HTML pages that have been randomly sampled from a Web site.

In summary, the literature either suggests practical wrapper generation techniques for Web information extraction, which however have the limitation of being supervised, i.e., inherently semi-automatic, or fully unsupervised techniques from the grammar inference community, which are in practice of little applicability.

The RoadRunner project introduced in this paper 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 Web sites. To show this, the paper 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 Web sites.

2 Overview and Contributions

Target of this research are the so-called data-intensive Web sites, i.e., 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 – i.e., programs – from 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*, i.e. 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.

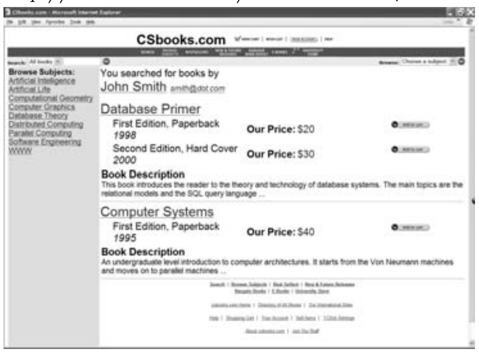
In the paper, we formalize such *schema finding problem*, and develop a fully automatic algorithm to solve it. A key contribution stands in the definition of a new class of regular languages, called the *prefix mark-up languages*, which nicely abstract the typical structures usually found in HTML pages. 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, i.e., that there exist unsupervised algorithms for their inference from positive examples only *a. Source Dataset*

Name	*Email*	*Books*				
		Title	*Descr.*	*Editions*		
				Details	*Year*	*Price*
John Smith	smith@dot.com	Database Primer	This book...	1st Ed., Paperback	1998	20$
				2nd Ed., Hard Cover	2000	30$
		Computer Systems	An undergraduate...	1st Ed., Paperback	1995	40$
Paul Jones	*null*	XML at Work	A comprehensive...	1st Ed., Paperback	1999	30$
		HTML and Scripts	A useful HTML...	*null*	1993	30$
				2nd Ed., Hard Cover	1999	45$
		JavaScript	A must in...	*null*	2000	50$
...	...	...	...	...	...	...

b. HTML Pages

*http://www.csbooks.com/*__author?John+Smith__



CSbooks.com

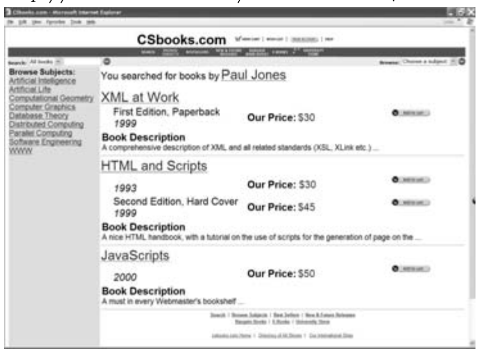
Browse Subjects:
Artificial Intelligence
Artificial Life
Computational Geometry
Computer Graphics
Database Theory
Distributed Computing
Parallel Computing
Software Engineering
WWW

You searched for books by
John Smith smith@dot.com

Database Primer
First Edition, Paperback
1998
Our Price: $20
Second Edition, Hard Cover
2000
Our Price: $30
Book Description
This book introduces the reader to the theory and technology of database systems. The main topics are the relational models and the SQL query language ...

Computer Systems
First Edition, Paperback
1995
Our Price: $40
Book Description
An undergraduate level introduction to computer architectures. It starts from the Von Neumann machines and moves on to parallel machines ...

*http://www.csbooks.com/*__author?Paul+Jones__



CSbooks.com

Browse Subjects:
Artificial Intelligence
Artificial Life
Computational Geometry
Computer Graphics
Database Theory
Distributed Computing
Parallel Computing
Software Engineering
WWW

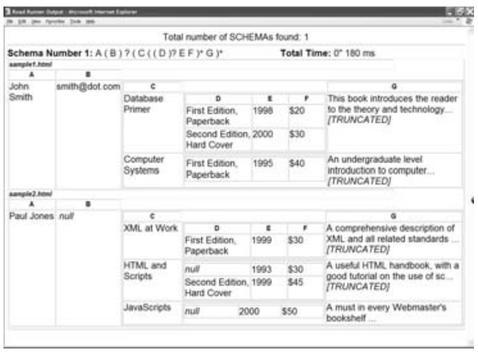
You searched for books by Paul Jones

XML at Work
First Edition, Paperback
1999
Our Price: $30
Book Description
A comprehensive description of XML and all related standards (XSL, XLink etc.) ...

HTML and Scripts
1993
Our Price: $30
Second Edition, Hard Cover
1999
Our Price: $45
Book Description
A nice HTML handbook, with a tutorial on the use of scripts for the generation of page on the ...

JavaScripts
2000
Our Price: $50
Book Description
A must in every Webmaster's bookshelf ...

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

sample1.html

A	B	C	D	E	F	G
John Smith	smith@dot.com	Database Primer	First Edition, Paperback	1998	$20	This book introduces the reader to the theory and technology... *[TRUNCATED]*
			Second Edition, Hard Cover	2000	$30	
		Computer Systems	First Edition, Paperback	1995	$40	An undergraduate level introduction to computer... *[TRUNCATED]*

sample2.html

A	B	C	D	E	F	G
Paul Jones	*null*	XML at Work	First Edition, Paperback	1999	$30	A comprehensive description of XML and all related standards ... *[TRUNCATED]*
		HTML and Scripts	*null*	1993	$30	A useful HTML handbook, with a good tutorial on the use of sc... *[TRUNCATED]*
			Second Edition, Hard Cover	1999	$45	
		JavaScripts	*null*	2000	$50	A must in every Webmaster's bookshelf ...

Figure 1: Examples of HTML Code Generation

- we show that prefix mark-up languages, differently from other classes previously know 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 Web sites
- finally, we discuss how a system based on this framework has been implemented in a working prototype and used to conduct several experiments on Web sites; these experiments

confirm the practical applicability of the techniques discussed in the paper

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"). However, this is a separate research problem; for space and simplicity reasons we don't deal with it in this work.

The paper 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 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. Finally, implementation and experiments are discussed in Section 9, and related works in Section 10.

3 Preliminary Definitions

This section introduces a number of preliminary definitions used throughout the paper

Abstract Data Types and Instances The abstract data we consider are nested relations [21], 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, i.e., (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 non-atomic *types* (and their respective domains) can be 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 $dom([T_1, \ldots, T_n]) = \{[a_1, \ldots, a_n] \mid a_i \in dom(T_i)\}$ (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 paper, we shall use as a running example a simplified version of the books Web site discussed in Section 1. 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 [21] 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: so, for example, any list-instance node, i.e., any node that represents an instance of a list node, is called $list_{inst}$.

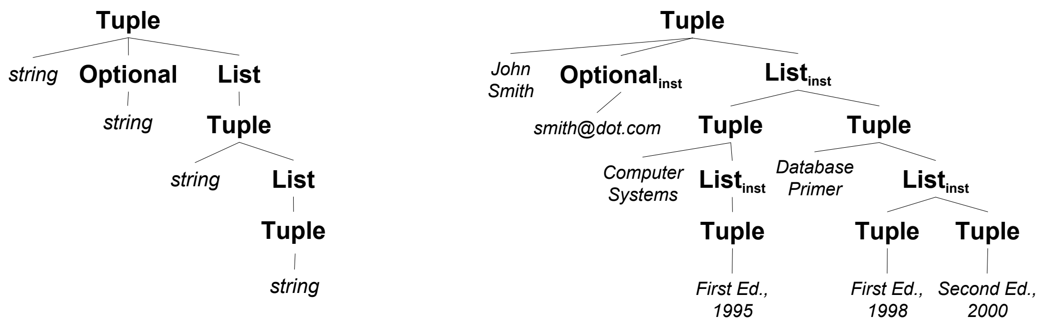


Figure 2: Running Example

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 types and *union–free regular expressions*. Given a special symbol #PCDATA, and an alphabet of symbols Σ not containing #PCDATA, a *union-free regular expression (UFRE)* over Σ is a string over alphabet $\Sigma \cup \{\texttt{\#PCDATA}, \cdot, +, ?, (,)\}$ defined as follows. First, the empty string, ϵ and all elements of $\Sigma \cup \{\texttt{\#PCDATA}\}$ 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.

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 $r_1 \cdot \ldots \cdot r_n$ corresponds to a tree rooted at an *And* node with n subtrees, one for each r_i.[1] Figure 3 also shows the AST associated with the regular grammar. Throughout the paper 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.

[1]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 $Plus(And(a, Hook(And(b)), c))$.

```
<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>
```

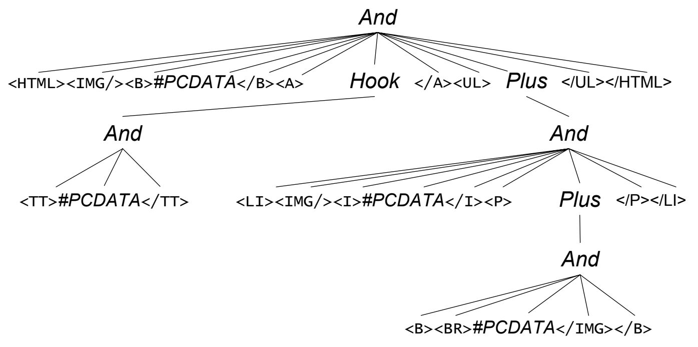


Figure 3: Regular Grammar for the Running Example

4 Problem Formalization

In this Section, we formalize our problem by setting a theoretical framework[2] 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*, i.e., partially defined types that nicely generalize types and their instances. Prefix mark-up languages will be then defined as those languages based on encodings of templates.

[2]The problem formalized in this paper is an extension of that defined in [16].

4.1 Templates

Templates are built by mixing types – lists, tuples, optionals and basic – and instances – i.e., 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', i.e., a constant string, and the second one any list of monadic tuples. Another example is $T' = [U, \langle_i[a, U], [a, U], [a, U]\rangle]$, where $\langle_i...\rangle$ 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 $I = [c, \langle_i[a]\rangle]$ 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.

Definition 1 (Templates)

- *Every element $u \in dom(U)$ is a non nullable* constant template*; null is a null-template; it is nullable*
- *U, the basic type, is a* basic template*; it is not nullable*
- *if $T_1, \ldots, T_n$ are non-tuple templates, amongst which at least one is non nullable, then $[T_1, \ldots, T_n]$ 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 $T_1, \ldots, T_k$ are tuple templates, then $T = \langle_i T_1, \ldots, T_k\rangle$ 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 $(_iT)?$ 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 sub-templates; (ii) every instance of a type is itself a template, made of tuple, list-instance, optional-instance, constant and *null* sub-templates. 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 2 (Subsumption)

- *Every constant and null template subsumes itself.*
- *the basic template U subsumes every constant template*
- *a tuple template, $[T_1, \ldots, T_n]$ subsumes any tuple template in $\{[t_1, \ldots, t_n] \mid t_i \preceq T_i)\}$*

- *a list template, $\langle T\rangle$, subsumes any list template $\langle S\rangle$ such that $S \preceq T$, and any list-instance template $\langle_i T_1, \ldots, T_m\rangle$ such that $T_j \preceq T, j = 1, \ldots m$*
- *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 $T = \langle_i T_1, \ldots, T_n\rangle$ subsumes any template in $\{\langle_i t_1, \ldots, t_n\rangle \mid t_j \preceq T_j, j = 1 \ldots n\}$*
- *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.

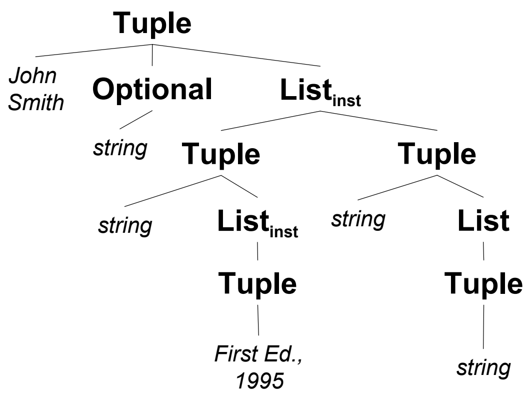


Figure 4: Example of a Template

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 root is labeled by the string *root*. If a list node or an optional node is labeled α, then its child is labeled $\alpha.0$; and if a tuple node with n children is labeled α, then its children are labeled $\alpha.1, \ldots, \alpha.n$. Instances are labeled similarly, with the children of a list, list-instance or optional-instance node labeled α, all labeled $\alpha.0$. 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.

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 non-optional or omitted). The concept of *rich* collection of instances defines this notion formally.

Definition 3 (Rich Collection of Instances) *A collection of instances I is* rich *with respect to a common type σ if it satisfies the following three properties: (i)* list richness*: for each label α of a list node, there are lists of distinct cardinalities labeled α, (ii)* basic richness*: for each leaf node labeled α, there are distinct objects labeled α, and (iii)* optional richness*: for each label α 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, i.e., 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 the set of all abstract instances to strings over those alphabets.

We introduce the *well-formed mark-up encodings*, which nicely abstract mark-up based languages like HTML or XML – which are used in practice to encode information on many data-intensive web sites. 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 database objects into strings in which the constants are encoded as strings over a data alphabet, Δ, and the structure is encoded by a tag alphabet, Σ. The tag alphabet will contain a closing tag `</a>` for each opening tag `<a>`.

Definition 4 (Alphabets) *We fix a* data alphabet Δ *and a* schema alphabet*, $\Sigma \cup \overline{\Sigma}$, with $\overline{\Sigma} = \{$`</a>`$|$`<a>`$\in \Sigma\}$,*

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*, δ, that is a 1-1 mapping from $dom(U)$ to Δ^+ according to which constants are encoded as words in Δ^+. However, in the following, for the sake of simplicity, we will assume that $dom(U)$ is a set of strings over Δ, i.e., $dom(U) \subseteq \Delta^+$, and that δ is the identity function; we will therefore omit to mention δ explicitly.

Second, we need to describe how the schema structure is encoded using tags. For this, we introduce a *tagging function*, *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*, i.e., tags are properly nested and balanced so that every occurrence of a symbol `<a>` in Σ is "closed" by a correspondent

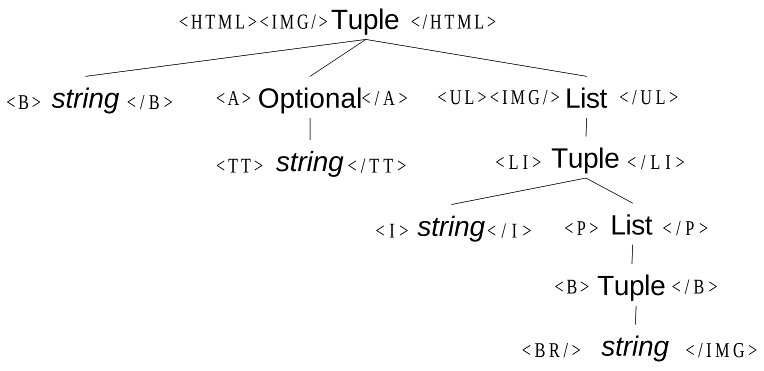


Figure 5: A Sample Mark-Up Encoding

occurrence of a symbol `</a>` in $\overline{\Sigma}$. For instance if $\Sigma = \{$`<a>,<b>,<c>`$\}$ and $\Delta = \{0, 1\}$, these are well-formed strings: `<a><b>11</b></a>`, `<a><b></b>100<c></c></a>`, while these are not: `<a><b>0</a></b>`, `<a><b></c>0</b></a>`, `<a><b><c></c>01</a>`. We formalize this concept by saying that well-formed strings need to belong to the language defined by a context–free grammar G_{tag}. Let $\triangle$ denote a place holder for data encodings and S be the starting non-terminal symbol. The productions of the grammar are as follows:

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

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

Definition 5 (Well-Formed Tagging Function) *Given a type σ, and the corresponding labeled tree T_σ, let $\mathcal{L}$ denote the set of its labels. A* well formed tagging function *for the type σ is defined as follows:*

$$\begin{array}{llll} tag: & \mathcal{L} & \rightarrow & (\Sigma \cup \overline{\Sigma})^+ \times (\Sigma \cup \overline{\Sigma})^+ \\ & \alpha & \mapsto & (start(\alpha), end(\alpha)) \ \ \textit{such that} \ \ start(\alpha) \cdot \triangle \cdot end(\alpha) \in L(G_{tag}) \end{array}$$

Definition 6 (Well-Formed Mark-Up Encodings) *Given a type σ, a* well-formed mark-up encoding *enc based on a structure (Σ, Δ, tag) — where Σ and Δ are two disjoint finite alphabets, and tag a well-formed tagging function for σ — is a function recursively defined on the tree of any template T subsumed by σ, as follows:*

- *for a constant leaf node $a \in dom(U)$ with label α, $enc(a) = start(\alpha) \cdot a \cdot end(\alpha)$;*[3]
- *for a null template with label α, $enc(null) = start(\alpha)end(\alpha)$*
- *for a list-instance node $\langle_i\, a_1, \ldots, a_n\rangle$ with label α, $enc(\langle_i\, a_1, \ldots, a_n\rangle) = start(\alpha) \cdot enc(a_1) \cdot \ldots \cdot enc(a_n) \cdot end(\alpha)$*
- *for an optional-instance node $(_ia)?$ with label α, $enc((_ia)?) = start(\alpha) \cdot enc(a) \cdot end(\alpha)$*

[3]Recall that we assume that $dom(U) \subseteq \Delta^+$, and therefore that leafs are encoded as themselves.

- *for a basic leaf node U labeled 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, $enc([T_1, \ldots, T_n]) = start(\alpha) \cdot enc(T_1) \cdot \ldots \cdot enc(T_n) \cdot end(\alpha)$;*
- *for a list node $\langle T \rangle$ with label α, $enc(\langle T \rangle) = start(\alpha) \cdot (enc(T))^+ \cdot end(\alpha)$.*

For example, the encoding shown in Figure 5 for the type σ 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 σ 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 σ. The following Proposition summarizes a close correspondence between the theoretical framework based on data and encoding functions, and the theory of regular languages.

Proposition 1 *Given a mark-up encoding enc based on a structure (Σ, Δ, tag), then, $enc(\sigma)$ is a regular language and for each instance I of type σ, $enc(I) \in L(enc(\sigma))$.*

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

Definition 7 (Well-Formed Mark-Up Languages) *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 of a type σ, 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 8 (Schema Finding Problem for Mark-up Encodings)
Input*: Σ, Δ as defined above, and a finite collection W of strings of $(\Sigma \cup \overline{\Sigma} \cup \Delta)^*$ which are encoding of a rich set of instances of a type σ.*
Output*: The type σ, an encoding function enc and a finite collection C of instances of type σ, 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 σ: $I_1 = \langle_i [0], [1] \rangle$ and $I_2 = \langle_i [0], [1], [2], [3] \rangle$. Suppose we fix the following encoding:

$enc(\sigma)$ = `<z> (<a></a><b></b>`Δ^+`<b></b><a></a>)`$^+$ `</z>`

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

$enc(\sigma')$ `= <z> (<a></a><b></b>`Δ^+`<b></b><a></a><a></a><b></b>`Δ^+`<b></b><a></a>)`$^+$ `</z>`

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

$enc(I_1) = enc'(J_1)$ `=<z> <a></a><b></b>0<b></b><a></a><a></a><b></b>1<b></b><a></a> </z>`
$enc(I_2) = enc'(J_2)$ `=<z> <a></a><b></b>0<b></b><a></a><a></a><b></b>1<b></b><a></a>`
`<a></a><b></b>2<b></b><a></a><a></a><b></b>3<b></b><a></a> </z>`

This shows that the problem stated above may in general admit multiple solutions. Intuitively, this due to the fact 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 derive. We define a special subclass of mark-up encodings called *prefix mark-up encodings* which force delimiters of list and optional node to be somehow identifiable.

Definition 9 (Prefix Mark-Up Encodings) *A* prefix mark-up encoding *is a* well-formed mark-up encoding *based on a structure* (Σ, Δ, tag) *where the tagging function tag satisfy the following additional conditions:*

- wrapping delimiters*: all delimiters of non-leaf 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 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 10 (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 11 (Schema Finding Problem - SFP)
Input*:* Σ, Δ *as defined above, and a finite collection* W *of strings of* $(\Sigma \cup \overline{\Sigma} \cup \Delta)^*$ *which are prefix mark-up encodings of a rich set of instances of a type* σ*.*
Output*: The type* σ*, an encoding function enc and a finite collection* C *of instances of type* σ*, 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 2 *Given a nested tuple type σ and a prefix mark-up encoding enc, it is possible to derive σ from enc(σ) 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, for example [30]). Gold gave a simple and widely accepted model of inductive learning also called "learning from text" which goes on as follows [15]: (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, abcbcbc ...`; (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 [15]). To recall the theorem, let us fix an alphabet Σ. We call a *superfinite class of languages* over Σ any class of languages that contains all finite languages over Σ 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 [2] based on the fundamental notion of *characteristic sample.* According to this theorem, a language class $\mathcal{L}$ is inferrable if and only if any 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 12 (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, i.e., L is the minimal language from $\mathcal{L}$ containing $\chi(L)$.*

Based on her work, subsequent works introduced several classes; prominent examples are the class of reversible grammars [3] and the class of terminal distinguishable languages [31], which were proven to be identifiable in the limit, and for which unsupervised algorithms were developed and formally proven to be correct.

Recently, Fernau [13] has introduced the notion of *f–distinguishable language* to generalize Angluin's work [3]. 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 13 (Distinguishing Function [13, Definition 1]) *Let F be a finite set. A mapping $f : \Sigma^* \to F$ is is called a* distinguishing *function if $f(w) = f(z)$ implies $f(wu) = f(zu)$ for all $u, w, z \in \Sigma^*$*

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

Definition 14 (f–Distinguishable Automaton [13, Definition 3]) *Let $A = (Q, \Sigma, \delta, q_0, Q_F)$*[4] *be a finite automaton, and $f : \Sigma^* \to F$ a distinguishing function. A is called f–*distinguishable *if*

1. *A is deterministic*
2. *For all states $q \in Q$ and all $x, y \in \Sigma^*$ with $\delta^*(q_0, x) = \delta^*(q_0, y)$, we have $f(x) = f(y)$. (In other words, for $q \in Q$, $f(q) := f(x)$ for some x with $\delta^*(q_0, x)$ 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 $q_3 \in Q$ 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 [13, 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. This result is based on the fact that prefix mark-up languages are a subset of a specific class of f–distinguishable languages, and therefore are identifiable in the limit.

Let us consider the following set of reductions of string over $\Sigma \cup \overline{\Sigma} \cup \Delta$:

$$\begin{array}{l} a\triangle\overline{a} \to \epsilon \text{ for } a \in \Sigma \\ d \to \triangle \text{ for } d \in \Delta^+ \\ \triangle\triangle \to \triangle \end{array}$$

By applying these reductions, a string over $\Sigma \cup \overline{\Sigma} \cup \Delta$ is reduced to a string over $\Sigma \cup \overline{\Sigma} \cup \{\triangle\}$. Intuitively, the reductions essentially remove from a string any well-formed substring. A word is said *reduced* or *irreducible* if it cannot be further reduced. Let $\rho(w)$ denote the unique irreducible word obtained from w and observe that $\rho(enc(I)) = \epsilon$ for any well-formed mark-up encoding enc and instance I. We now show that by appropriately choosing a distinguishing function f_π, it results that the $f_\pi - DL$ is a proper superset of the class of prefix mark-up languages.

[4]As usual: Q is a set of *states*, Σ is a set of *terminal symbols*, $\delta : Q \times \Sigma \to Q$ is the *transition function*, q_0 is the *initial state* and Q_F is the set of *final states*

Definition 15 (f_π) *Let w be any string over $\Sigma \cup \overline{\Sigma} \cup \Delta$. Function f_π is defined as follows:*

$$\begin{array}{rrcll} f_\pi: & (\Sigma \cup \overline{\Sigma} \cup \Delta)^* & \to & (\Sigma \cup \overline{\Sigma} \cup \{\triangle\})^* & \\ & \epsilon & \mapsto & \epsilon & \\ & wa & \mapsto & \rho(w) \cdot a, & a \in \Sigma \\ & wa & \mapsto & \rho(w \cdot a), & a \in \Delta \end{array}$$

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

$$\begin{array}{lll} f(a) \;= a & f(ab01) \;= ab\triangle & f(ab01\overline{b}b1) \;= ab\triangle \\ f(ab) \;= ab & f(ab01\overline{b}) \;= ab\triangle\overline{b} & f(ab01\overline{b}b1\overline{b}) \;= ab\triangle\overline{b} \\ f(ab0) = ab\triangle & f(ab01\overline{b}b) = ab & f(ab01\overline{b}b1\overline{b}b) = ab \end{array}$$

Lemma 1 *f_π is a distinguishing function*

PROOF To show that f_π is a distinguishing function, we need to show that for all strings w, u, z, $f_\pi(w) = f_\pi(u)$ implies that $f_\pi(wz) = f_\pi(uz)$. 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 ρ 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. □

Theorem 1 (Identifiability in the Limit) *The class of prefix mark-up languages is identifiable in the limit.*

PROOF: To prove identifiability in the limit we shall first show that prefix mark-up languages are f_π–distinguishable by showing that the corresponding automata are f_π–distinguishable.

Consider Figure 6.A-C which shows the "building blocks" needed to represent type nodes:

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 $start(\alpha)(start(\beta)enc(T)end(\beta))?end(\alpha)$.

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. This satisfies condition 1 in the definition of *f–distinguishable* automaton above. Condition 2 immediately follows from the fact that $\rho(enc(I)) = \epsilon$ for any prefix mark-up encoding *enc* of an instance I. Condition 3 must be checked on the states marked with q_1, q_2, and q_3 in Figure 6. It has to result $f_\pi(q_1) \neq f_\pi(q_2)$. In fact, by definition of f_π, in the cases of Figure 6.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

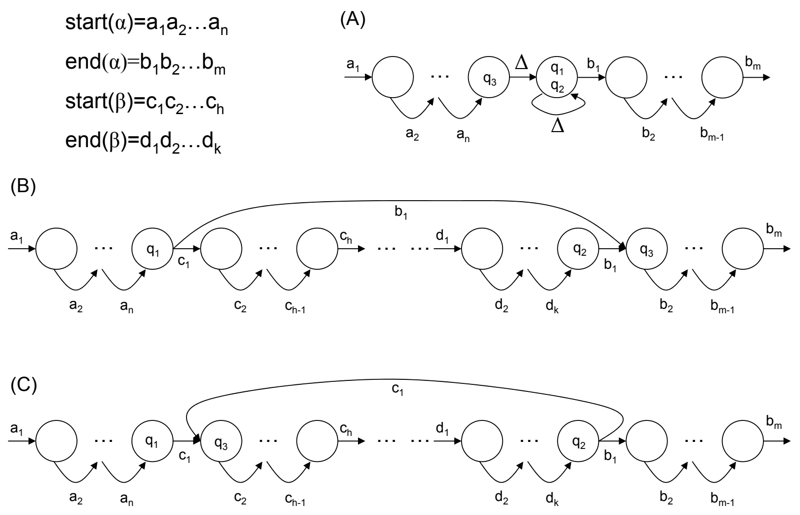


Figure 6: Building blocks for f_π–distinguishable automaton of prefix mark-up languages

is sufficient to prove that $f_\pi(q_1) \neq f_\pi(q_2)$. In the case of Figure 6.A, $f_\pi(q_2)$ ends with a symbol of Σ, while $f_\pi(q_1)$ ends with $\triangle$.

According to the definition of distinguishing function, f_π must have a finite range over all possible strings. In our setting this is not true, because f_π can assume as many values as the strings in $(\Sigma)^+$ that are not well-formed. Still theorem [13, Theorem 8] holds, because f_π assumes a finite set of values on any language from the class considered, which is the only requirement really needed during the demonstration[5]. $\square$

From Theorem 1 it also descends that prefix mark-up languages have a characteristic sample.

Corollary 1 *The class of prefix mark-up languages have characteristic samples.*

5.2 Limitations of Grammar Inference Algorithms 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, 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

[5]Actually in [13] the overall approach is slightly different: it consists in choosing a distinguishable function whose range is finite. Then the attention is restricted to the class of distinguishable languages according to that function. Assuming F finite is sufficient but not necessary to guarantee that f assumes a finite set of values on any language from the class.

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 [3] is related to the fact 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, i.e., 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.

Formally speaking, given a k-reversible language whose automaton $A = \{Q, \Sigma, \delta, q_0, Q_F\}$, a characteristic sample for the language is any set of strings of the form:

$$\begin{array}{rcl}\chi(A) & = & \{u(q)v(q) | q \in Q\} \\ & \cup & \{u(q)av(q) | q \in Q, a \in \Sigma\}\end{array}$$

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 7. The sample contains four strings: Sample a is a representative of the strings of minimal-length in the language (no email, 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 (email, one book with one edition); Sample c derives Sample a and exercises the portion of the automaton related with the external plus (no email, two books each with one edition); finally, Sample d derives from Sample a and exercises the internal plus (no email, 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 7 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 a database-theoretic notion, which essentially requires that the samples used in the inference make full use of the underlying type.

Sample a

```
<HTML>
<IMG/> <B>Wally Wood</B>
<A></A>
<UL>
  <LI> <IMG/> <I>Linux Programming</I>
       <P>
         <B><BR/>First Ed., 1998<IMG/></B>
       </P>
  </LI>
</UL>
</HTML>
```

Sample b

```
<HTML>
<IMG> <B>Jack Kirby</B>
<A><TT>kirby@dot.edu</TT></A>
<UL>
  <LI> <IMG/> <I>J2EE 1.4</I>
       <P>
         <B><BR/>First Ed., 2002<IMG/></B>
       </P>
  </LI>
</UL>
</HTML>
```

Sample c

```
<HTML>
<IMG> <B>Stan Lee</B>
<A></A>
<UL>
  <LI> <IMG/> <I>Superpowers</I>
       <P>
         <B><BR/>First Ed., 2001<IMG/></B>
       </P>
  </LI>
  <LI> <IMG/> <I>Microsoft .NET</I>
       <P>
         <B><BR/>First Ed., 2002<IMG/></B>
       </P>
  </LI>
</UL>
</HTML>
```

Sample d

```
<HTML>
<IMG> <B>John Romita</B>
<A></A>
<UL>
  <LI> <IMG/> <I>Security</I>
       <P>
         <B><BR/>First Ed., 2001<IMG/></B>
       </P>
       <P>
         <B><BR/>Second Ed., 2002<IMG/></B>
       </P>
  </LI>
</UL>
</HTML>
```

Figure 7: A Characteristic Sample for the Running Example

6 The Matching Technique

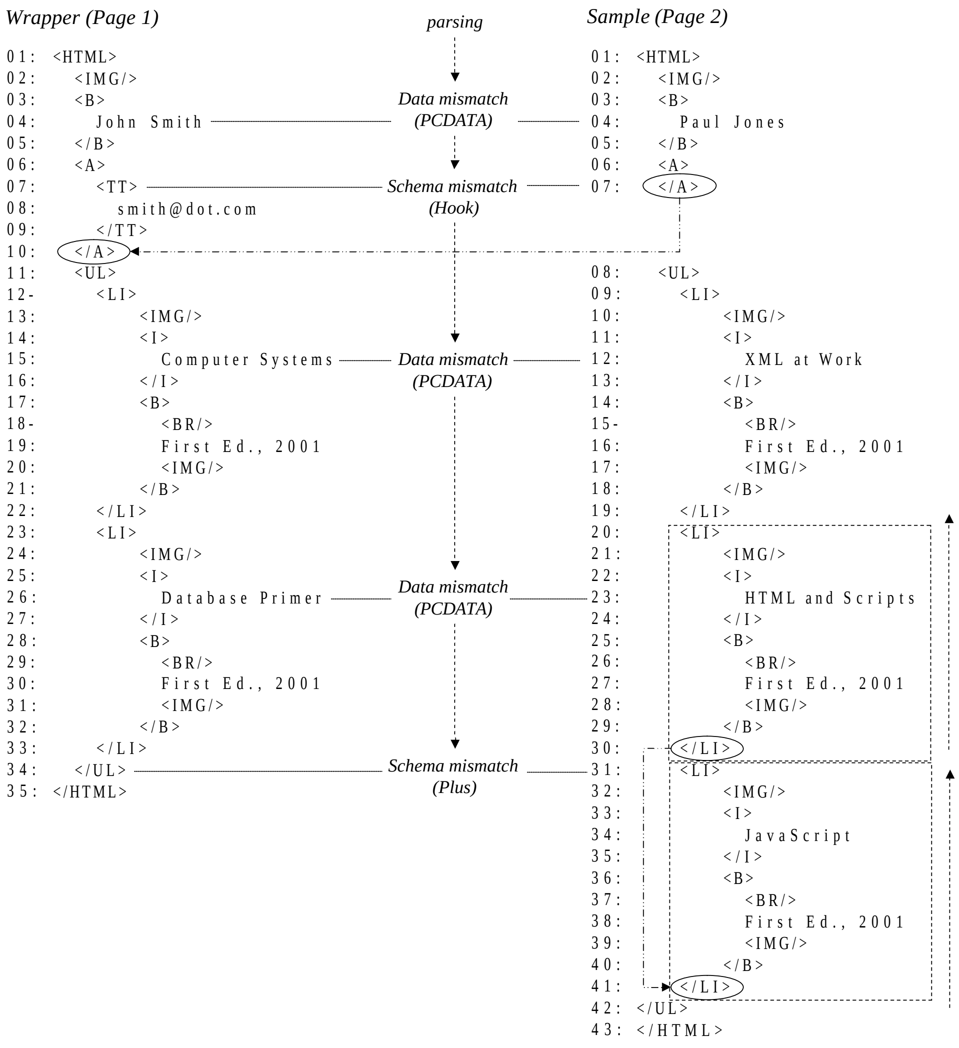
This Section is devoted to the informal presentation of algorithm *match* [10] for solving SFP for prefix mark-up encodings. We assume that HTML sources are preprocessed to transform them into lists of tokens. Each token is either an HTML tag or a string. Tags will be represented by symbols of the schema alphabet Σ, strings will be represented by symbols of the data alphabet Δ. Figure 8 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 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*, i.e., a list of tokens corresponding to one of the sample pages, and (*ii*) a *wrapper*, i.e., 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



```
<HTML>
  <IMG/>
  <B>
    John Smith
  </B>
  <A>
    <TT>
      smith@dot.com
    </TT>
  </A>
  <UL>
    <LI>
        <IMG/>
        <I>
          Computer Systems
        </I>
        <B>
          <BR/>
          First Ed., 2001
          <IMG/>
        </B>
    </LI>
    <LI>
        <IMG/>
        <I>
          Database Primer
        </I>
        <B>
          <BR/>
          First Ed., 2001
          <IMG/>
        </B>
    </LI>
  </UL>
</HTML>
```

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

Figure 8: One Simple Matching

context, is the one of *mismatch*: a mismatch happens when some token in the sample does not comply to 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*, i.e., 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. 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 8. 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 8 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*, i.e., 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, i.e., the application of operator *addHook*. Lists will be discussed in the following section.

Consider again Figure 8. The first schema mismatch happens at token 7 due to the presence of the email in the wrapper and not in the sample, i.e., 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 conclude 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 one pattern of the form `(<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 8; 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, i.e, about its *start delimiter*; however, we don't know exactly where the list with the higher cardinality is located, i.e., 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, i.e., 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 8.

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 8, 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.

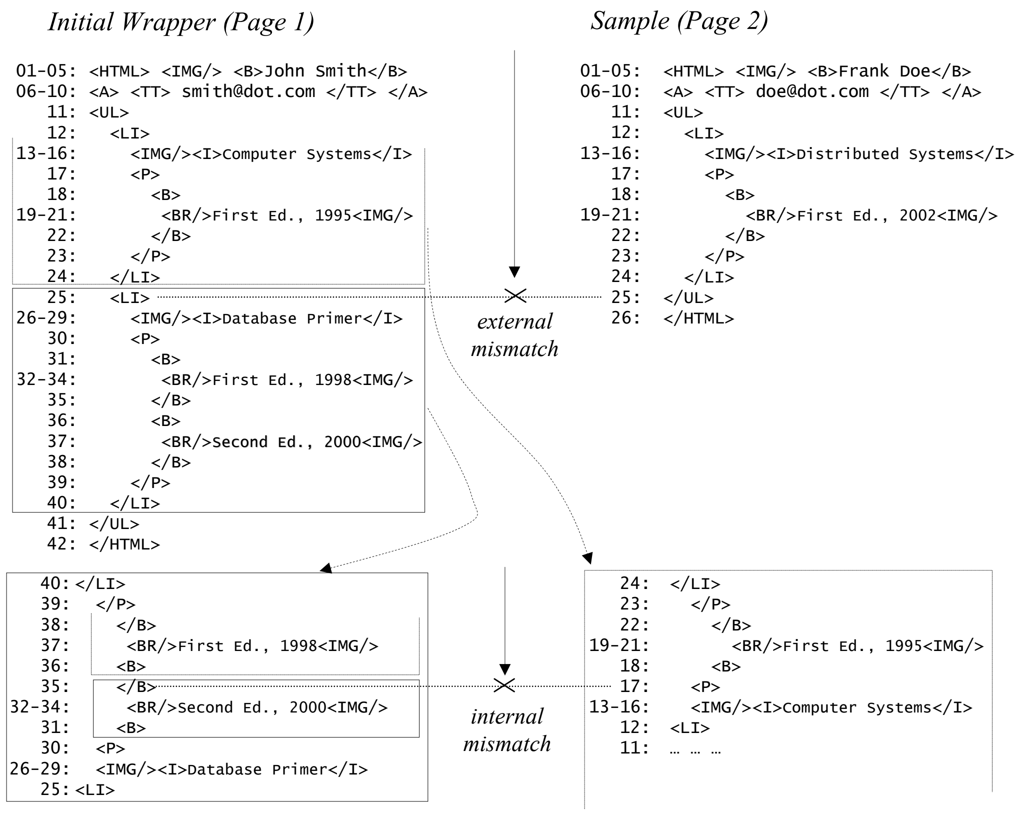
To see this, consider Figure 9, which shows the process of matching two pages of our running example with the list of editions nested inside the list of books. The wrapper (page 1) is matched against the sample (page 2). After solving a couple of data mismatches, the parsing stops at token 25, where 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 25–40) 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 25–40 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, i.e., starting from comparing the two occurrences of the end delimiter (tokens 40 and 24), then move to tokens 39 and 23 and so on.

This comparison has been emphasized in Figure 9 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 discover the list of editions inside the list of books.

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, i.e. either wrapper or sample.[6]

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

[6]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 to match two regular expressions, instead of one expression and one sample. We will discuss how to solve this problem in Section 7.



```
01-05: <HTML> <IMG/> <B>John Smith</B>
06-10: <A> <TT> smith@dot.com </TT> </A>
   11: <UL>
   12:   <LI>
13-16:     <IMG/><I>Computer Systems</I>
   17:     <P>
   18:       <B>
19-21:         <BR/>First Ed., 1995<IMG/>
   22:       </B>
   23:     </P>
   24:   </LI>
   25:   <LI>
26-29:     <IMG/><I>Database Primer</I>
   30:     <P>
   31:       <B>
32-34:         <BR/>First Ed., 1998<IMG/>
   35:       </B>
   36:       <B>
   37:         <BR/>Second Ed., 2000<IMG/>
   38:       </B>
   39:     </P>
   40:   </LI>
   41: </UL>
   42: </HTML>
```

```
01-05: <HTML> <IMG/> <B>Frank Doe</B>
06-10: <A> <TT> doe@dot.com </TT> </A>
   11: <UL>
   12:   <LI>
13-16:     <IMG/><I>Distributed Systems</I>
   17:     <P>
   18:       <B>
19-21:         <BR/>First Ed., 2002<IMG/>
   22:       </B>
   23:     </P>
   24:   </LI>
   25: </UL>
   26: </HTML>
```

```
   40: </LI>
   39:   </P>
   38:     </B>
   37:       <BR/>First Ed., 1998<IMG/>
   36:     <B>
   35:     </B>
32-34:       <BR/>Second Ed., 2000<IMG/>
   31:     <B>
   30:   <P>
26-29:   <IMG/><I>Database Primer</I>
   25: <LI>
```

```
   24: </LI>
   23:   </P>
   22:     </B>
19-21:       <BR/>First Ed., 1995<IMG/>
   18:     <B>
   17:   <P>
13-16:   <IMG/><I>Computer Systems</I>
   12: <LI>
   11: … … …
```

Figure 9: A More Complex Matching

new space may in turn trigger other instances of the same search problem. These ideas are summarized in Figure 10, 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 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 them 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. Such nice property makes the complexity of the algorithm a polynomial one.

7 The Algorithm

This section formalizes the matching algorithm informally described above.

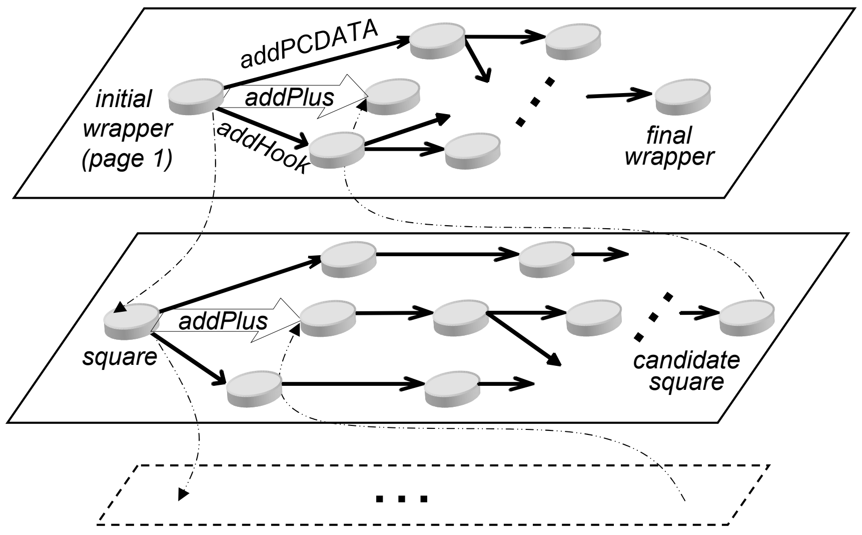


Figure 10: Matching as a search problem in a state space

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 τ a (possibly empty) list of contiguous subtrees of τ. We will denote a region in a tree τ 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 $[n, \ldots)$. A region including all subtrees rooted at siblings preceding a node n (included) will be denoted $(\ldots, n]$. Similarly for $(n, \ldots)$ and $(\ldots, n)$. So, for example, with respect to the tree in Figure 3, $(\ldots, \mathit{Hook})$ 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*, $\mathsf{subst}(\tau, r, \tau')$, replaces a region r in a tree τ by a new subtree τ'. Then, we have two *insert operators* that insert new subtrees in an existing tree. $\mathsf{insBefore}(\tau, n, \tau')$ inserts subtree τ' in τ immediately before the subtree rooted at node n. $\mathsf{insAfter}(\tau, n, \tau')$ inserts subtree τ' in τ immediately after the subtree rooted at node n. For example, if τ is the abstract tree of Figure 3 and $\tau' = \mathit{Hook}(\mathit{And}($`<HR>`, `<TT>`, PCDATA, `</TT>`$))$ then:

$\mathsf{subst}(\tau, [Hook, \texttt{</UL>}], \tau') = And(\texttt{<HTML>}, \texttt{<B>}, PCDATA, \texttt{</B>}, \texttt{<HR>}, \texttt{<TT>}, PCDATA, \texttt{</TT>}, \texttt{</HTML>})$

We also define two *search operators* on trees, that are used to search subtrees in an abstract syntax both forward and backward. If τ and τ' are trees, and n is a node in τ, $\overrightarrow{\mathsf{search}}_k(\tau, n, \tau')$ returns the k-th occurrence in tree τ of the subtree τ' following node n (included); $\overleftarrow{\mathsf{search}}_k(\tau, n, t)$ returns the k-th occurrence in tree τ of subtree τ' 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, i.e., a prefix mark-up language represented through its abstract syntax tree, s is a sample, i.e., 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:

```
AST addPCDATA(mismatch $m = (w, s, (n, t))$, $d \in \{\rightarrow, \leftarrow\}$)
begin
  if ($m$ is a data mismatch) return subst(w,w[n,n],$PCDATA$);
  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 11. The Figure contains both the case in which the optional is located on the wrapper ($\overrightarrow{addHook}^w$) and on the sample ($\overrightarrow{addHook}^s$).

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

Predicate checkSquare is needed to avoid the mistake of applying an *addHook* operator whenever *addPlus* would be the right choice. In fact, whenever *addPlus* is applicable, also *addHook* is applicable. Such situations can be sketched as follows:

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

```
AST $\overrightarrow{addHook}^w$(mismatch $m = (w, s, (n, t))$)
begin
    if ($m$ is a data mismatch) return $\perp$;
    Let $csquare$ be $\overrightarrow{\text{findHookSquare}}(w,n,t)$;
    if ($csquare$=$\perp$) return $\perp$;
    if (not $\overrightarrow{\text{checkSquare}}(m,csquare)$ return $\perp$;
    return subst($w$,$csquare$,$Hook(And(csquare))$);
end
```

```
AST $\overrightarrow{addHook}^s$(mismatch $m = (w, s, (n, t))$)
begin
    if ($m$ is a data mismatch) return $\perp$;
    Let $csquare$ be $\overrightarrow{\text{findHookSquare}}(s,t,n)$;
    if ($csquare$=$\perp$) return $\perp$;
    if (not $\overrightarrow{\text{checkSquare}}(m,csquare)$ return $\perp$;
    return insBefore($w$,$n$,$Hook(And(csquare))$);
end
```

```
AST $\overrightarrow{\text{findHookSquare}}(tree,from,wanted)$
begin
    Let $k$ be 0;
    do  $k = k+1$;
        Let $occ$ be $\overrightarrow{\text{search}}_k(tree,from,wanted)$;
        if ($occ = \perp$) return $\perp$;
        Let $csquare$ be $tree[from, occ)$;
    while (not isWFMarkUp($csquare$));
    return $csquare$;
end
```

```
boolean $\overrightarrow{\text{checkSquare}}(m = (w, s, (n, t)),square)$
begin
    return (last token of square equals
    token immediately before $t$)
end
```

```
boolean $\overleftarrow{\text{checkSquare}}(m = (w, s, (n, t)),square)$
begin
    return (first token of square equals
    token immediately after $t$)
end
```

Figure 11: *addHook* Operator

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:

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

if, on the contrary, *addHook*^{w} is applied, it would produce:

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

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 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 '→' but the same description symmetrically holds in the opposite direction.[7] We briefly comment

[7]In the following, whenever the discussion holds for both directions, we will omit to specify the direction.

the code in Figure 12. The Figure contains both the case in which the candidate square is located on the wrapper ($\overrightarrow{addPlus}^w$) and 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 $\mathsf{lastDelim}$. Function $\overrightarrow{\mathsf{findPlusSquare}}$ 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 $enc(I_1), enc(I_2), \ldots,$ $enc(I_n)$, 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:

$$\begin{array}{lcl} w_1 & = & enc(I_1) \\ w_2 & = & \mathsf{match}(w_1, enc(I_2)) \\ \ldots & & \\ w_n & = & \mathsf{match}(w_{n-1}, enc(I_n)) \end{array}$$

We define $\mathsf{match}(enc(I_1), enc(I_2), \ldots, enc(I_n)) = w_n$. It can be seen that all of the complexity of the matching stands in the binary algorithm $\mathsf{match}(w, s)$ 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

```
AST $\overrightarrow{addPlus}^w$(mismatch $m$=$(w, s, (n, t))$)
begin
  if ($m$ is a data mismatch) return $\perp$;
  //Square Location by Delimiter Search
  Let $ldel$ be $\overrightarrow{\text{lastDelim}}$(m);
  if ($ldel$=$\perp$) return $\perp$;
  Let $csquare^w$=$w[n, l_1]$ be
    $\overrightarrow{\text{findPlusSquare}}$($w$,$n$,$ldel$);
  if ($csquare^w$=$\perp$) return $\perp$;
  //Candidate Square Evaluation
  if ($\overleftarrow{\text{match}}$($w(\ldots n)$,$csquare^w$)) begin
    Let $i$ be 0;
    Let $square_i$ be $\overleftarrow{\text{match}}$.getResult();
    Let $f_i$ be $\overleftarrow{\text{match}}$.getLastMatchingNode();
    //Candidate Square Matching on the Left
    while ($\overleftarrow{\text{match}}$($square_i$,$w(\ldots f_i)$)) do
      $i$ = $i$ - 1;
      Let $square_i$ be $\overleftarrow{\text{match}}$.getResult();
      Let $f_i$ be $\overleftarrow{\text{match}}$.getLastMatchingNode();
    end
    //Candidate Square Matching on the Right
    Let $f$ be $f_i$;
    Let $square_1$ be $square_i$;
    Let $j$ be 1;
    while ($\overrightarrow{\text{match}}$($square_j$,$w(l_j \ldots)$)) do
      $j$ = $j$ + 1;
      Let $square_j$ be $\overrightarrow{\text{match}}$.getResult();
      Let $l_j$ be $\overrightarrow{\text{match}}$.getLastMatchingNode();
    end
    Let $l$ be $l_j$;
    Let $square$ be $square_j$;
    Let $squareExt$ be $w[f, l]$;
    //Wrapper Generalization
    return subst($w$,$squareExt$,$Plus(square)$);
  end
  return $\perp$;
end
```

```
AST $\overrightarrow{addPlus}^s$(mismatch $m$=$(w, s, (n, t))$)
begin
  if ($m$ is a data mismatch) return $\perp$;
  //Square Location by Delimiter Search
  Let $ldel$ be $\overrightarrow{\text{lastDelim}}$(m);
  if ($ldel$=$\perp$) return $\perp$;
  Let $csquare^s$=$s[t, l_1]$ be
    $\overrightarrow{\text{findPlusSquare}}$($s$,$t$,$ldel$);
  if ($csquare^s$=$\perp$) return $\perp$;
  //Candidate Square Evaluation
  if ($\overleftarrow{\text{match}}$($w(\ldots n)$,$csquare^s$)) begin
    Let $i$ be 0;
    Let $square_i$ be $\overleftarrow{\text{match}}$.getResult();
    Let $f_i$ be $\overleftarrow{\text{match}}$.getLastMatchingNode();
    //Candidate Square Matching on the Left
    while ($\overleftarrow{\text{match}}$($square_i$,$w(\ldots f_i)$)) do
      $i$ = $i$ - 1;
      Let $square_i$ be $\overleftarrow{\text{match}}$.getResult();
      Let $f_i$ be $\overleftarrow{\text{match}}$.getLastMatchingNode();
    end
    //Candidate Square Matching on the Right
    Let $f$ be $f_i$;
    Let $square_1$ be $square_i$;
    Let $j$ be 1;
    while ($\overrightarrow{\text{match}}$($square_j$,$s(l_j \ldots)$)) do
      $j$ = $j$ + 1;
      Let $square_j$ be $\overrightarrow{\text{match}}$.getResult();
      Let $l_j$ be $\overrightarrow{\text{match}}$.getLastMatchingToken();
    end

    Let $square$ be $square_j$;
    Let $squareExt$ be $w[f, n]$;
    //Wrapper Generalization
    return subst($w$,$squareExt$,$Plus(square)$);
  end
  return $\perp$;
end
```

```
AST region $\overrightarrow{\text{lastDelim}}$(mismatch $(w, s, (n, t))$)
begin
  Let $d$ be the last data token preceding $t$;
  if ($d$ does not exist) return $\perp$;
  Let $lct$ be the last closing tag of region $s[d, t)$
      for which there is no opening tag;
  if ($lct$ does not exist) return $\perp$;
  if (not isWellFormed(s(lct,t))) return $\perp$;
  return $s[lct, t)$;
end
```

```
boolean isWellFormed(a region $s$)
  {check well-formedness of $s$ }
```

```
AST $\overrightarrow{\text{findPlusSquare}}$($tree$,$from$,$wanted$)
begin
  Let $k$ be 0;
  do  $k$ = $k$+1;
      Let $occ$ be $\overrightarrow{\text{search}}_k$($tree$,$from$,$wanted$);
      if ($occ$ = $\perp$) return $\perp$;
      Let $csquare$ be  $tree[from, occ]$;
  while (not isWFMarkUp($csquare$));
  return $csquare$;
end
```

Figure 12: *addPlus*

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 the encoding of an instance), and returns a new abstract syntax tree that generalizes the inputs. Figure 13 shows the pseudo-code of algorithm match.

```
boolean match([regions of] AST $w$, $w'$, 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([regions of] AST $w$,$s$, 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
```

Figure 13: The Algorithm match

Initially, the second template encoding is represented by a characteristic sample of the corresponding prefix mark-up language (Definition 12). 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 10. 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 14). 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 *bi-directional* in the sense that it can be performed in both directions with the help of simple abstractions called *enumerators*, which enumerate the element of a given collection. Enumerators accept only two kind of message: $hasNext()$, which returns a boolean indicating if all elements of the underlying collection has already been already enumerated; *next*(), which returns the next element according to the enumeration or an error if it does not exist.[8]

```
boolean parse([a region of] a wrapper $w$, a sample $s$, $d \in \{\rightarrow, \leftarrow\}$)
begin
  Let $m$ be the current mismatch;
  Let $wEn$ be an Enumerator over $w$'s nodes in the direction $d$;
  Let $sEn$ be an Enumerator over $s$ in the direction $d$;
  return (align($wEn$,$sEn$) and not $wEn$.hasNext());
end
```

```
boolean visit($n$:$Token$, Enumerator $en$)
begin
  Let $t$ = $en$.next();
  if ($n$ matches $t$) return true;
  else begin
    save ($w$,$s$,($n$,$t$)) as mismatch $m$;
    return false;
  end
end
```

```
boolean visit($n$:$PCDATA$, Enumerator $en$)
begin
  Let $t$ = $en$.next();
  if ($t$ is a data token) return true;
  else begin
    save ($w$,$s$,($n$,$t$)) as mismatch $m$;
    return false;
  end
end
```

```
boolean visit($n$:$Hook$, Enumerator $en$)
begin
  visitChildren($m$,$en$);
  return true;
end
```

```
boolean visit($n$:$Plus$, Enumerator $en$)
begin
  Let $result$ be false;
  while (visitChildren($n$,$en$)) do
    $result$ = true;
    discard current mismatch $m$;
  end
end
```

```
getLastMatchingToken() {returns the last matching token of the sample}
getLastMatchingNode() {returns the last matching node of the wrapper}
getMismatch() {returns mismatch $m$ }
```

```
boolean visit($And(t_1, \ldots, t_h)$, Enumerator $sEn$)
begin
  Let $oldIndex$ be the position of $sEn$ over $s$;
  Let $wEn$ be an Enumerator over $t_1, \ldots, t_h$
      in the direction $d$;
  if (not align($wEn$,$sEn$)) begin
    roll back $sEn$ to $oldIndex$;
    return false;
  end
  return (not $wEn$.hasNext());
end
```

```
boolean align(Enumerators $wEn$, $sEn$)
begin
  while ($wEn$.hasNext() and $sEn$.hasNext()) do
    if (not visit($wEn$.next(),$sEn$)) return false;
  end
  return true;
end
```

Figure 14: 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 2 (Correctness) *Given a set of strings* $\{enc(I_1), enc(I_2), \ldots enc(I_n)\}$ *of a rich set of instances of a type* σ *according to a prefix mark-up encoding enc, then:*

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

[8]Popular examples of enumerators are `java.util.Iterator` and `java.util.Enumeration`.

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 A. From Theorem 2 it immediately follows that:

Corollary 2 *Any encoding of a rich set of instances of a type σ according to a prefix mark-up encoding enc is a characteristic sample for enc(σ).*

Theorem 2 and Corollary 2 represent the main contributions of this paper. 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 to 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 re-build the wrapper.

On the other side, the new notion of characteristic sample – i.e., 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. To see this, consider this simple argument (for simplicity, we focus on lists only): consider random instances with a probability p that, for a given label α, two instances have all lists with label α of equal cardinality. Then, the probability of finding two different cardinalities among all lists labeled α in a collection of n instances is $(1 - p^{n-1})$. 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%.

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

Theorem 3 (Complexity) *Given two templates T and S subsumed by a common type σ, 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 A, and is given along with the proof of Theorem 2.

9 Implementation and Experiments

To validate 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. In order to clean HTML sources, fix errors and make the code compliant with XHTML, and also to build DOM trees, it uses *JTidy*, a Java port of *HTML Tidy* (`http://www.w3.org/People/Raggett/tidy/`), a library for HTML cleaning.

To perform our experiments, we have selected several classes of pages from real-life Web sites, with the requirement that these pages obey to a prefix mark-up grammar. All experiments have

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

site			schema			comparative results			
n.	name (URL)	#s	nest	opt	ord	RoadRunner		Wien	Stalker
7	Okra (discontinued)	20	1	no	no	√	0’0”700ms	√	√
8	BigBook (bigbook.com)	20	1	no	no	√	0’0”781ms	√	√
9	La Weekly (laweekly.com)	28	1	yes	no	√	0’0”391ms	no	√
10	Address Finder (iaf.net)	10	1	yes	yes	no		no	√

Figure 15: Experimental Results

been conducted on a machine equipped with an Intel Pentium III processor working at 450MHz, with 128 MBytes of RAM, running Linux (kernel 2.2) and Sun Java Development Kit 1.4.

We report in Figure 15 a list of results relative to several well known data-intensive Web sites. In each site, we have selected a number of classes of fairly regular pages; for each class we have downloaded a number of samples (usually between 10 and 20). Figure 15 actually contains two tables; while Table A refers to experiments we have conducted independently, in Table B we compare for 5 page classes our results with those of other data extraction systems for which experimental results are available in the literature, namely Wien [24, 22] and Stalker [28, 27], two wrapper generation systems based on a machine learning approach.

Table A in Figure 15 contains the following elements: (*i*) *class:* a short description of each class, and the number of samples considered for that 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 examples shown above, 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).

To compare our results with those of other wrapper induction systems, namely Wien [22] and Stalker [28, 27], Table B in Figure 15 reports a number of elements with respect to 5 page classes for which experimental results were known in the literature; the original test samples for classes 7 to 10 have been downloaded from RISE (`http://www.isi.edu/~muslea/RISE`), a repository of information sources from data extraction projects. Table B contains the following elements: (*i*) *site* from which the pages were taken, and number of samples; (*ii*) description of the target *schema*, i.e., 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.[9]

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 [22, 28, 27], it appears that inferring the grammar takes considerably less than training Wien and Stalker. (*iii*) Differently from RoadRunner and Stalker, Wien is unable to handle optional fields, and therefore fails on samples 9 and 10. (*iv*) Stalker has more considerable 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 Related Works

Early approaches to structuring and wrapping Web sites were based on manual techniques. Different approaches have been pursued to this end: they range from the adoption of specialized procedural languages [5], to the definition of declarative specification languages that work on the text [18, 17], or on GUI-based tools [33, 26, 6] that help the user to rapid-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 [9, 20].

The first attempts to automate the wrapper generation process heavily relied on the use of heuristics. For example, in [4], the authors 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 [14, 34] available on the Web, others on fairly structured HTML pages.

One example in this category is Wien [24, 22]. 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 – i.e., 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 paper 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–

[9]Since for some of the pages Wien and Stalker consider only a portion of the HTML code, to have comparable results when needed we have restricted our analysis to those portions only.

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, [22] surveys 30 sites from different domains, 70% of which can be wrapped using the formalisms discussed in the paper.

A similar approach is pursued in Stalker [28, 27] and SoftMealy [19]. These approaches are strictly more expressive than [24, 22], since they also allow for disjunctions, missing attributes, and can handle various permutations in the order of attributes. 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*, i.e., 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 [12]. 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 try 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 paper to show the effectiveness of this approach.

One final approach to wrapper generation from labeled examples is represented by NoDoSe [1]. 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 [34, 19, 12], 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 paper 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 [32].

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 paper.

References

[1] Adelberg, B. NoDoSE – a tool for semi-automatically extracting structured and semistructured data from text documents. In *ACM SIGMOD International Conf. on Management of Data (SIGMOD'98), Seattle, Washington* (1998).

[2] Angluin, D. Inductive inference of formal languages from positive data. *Information and Control*, 45 (1980), 117–135.

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

[4] Ashish, N., and Knoblock, C. Wrapper generation for semistructured Internet sources. In *Proceedings of the Workshop on the Management of Semistructured Data (in conjunction with ACM SIGMOD 1997)* (1997).

[5] Atzeni, P., and Mecca, G. Cut and Paste. In *Sixteenth ACM SIGMOD Intern. Symposium on Principles of Database Systems (PODS'97), Tucson, Arizona* (1997), pp. 144–153.

[6] Baumgartner, R., Flesca, S., and Gottlob, G. Visual web information extraction with lixto. In *Int. Conf. on Very Large Data Bases (VLDB'2001), Roma, Italy, September 11-14* (2001), pp. 119–128.

[7] Bruggemann-Klein, A., and Wood, D. One-unambiguous regular languages. *Information and Computation 142*, 2 (May 1998), 182–206.

[8] Crescenzi, V. *On Automatic Information Extraction from Large Web Sites.* PhD thesis, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Rome (Italy), 2002.

[9] Crescenzi, V., and Mecca, G. Grammars have exceptions. *Information Systems 23*, 8 (1998), 539–565. Special Issue on Semistructured Data.

[10] Crescenzi, V., Mecca, G., and Merialdo, P. roadRunner: Towards automatic data extraction from large Web sites. In *International Conf. on Very Large Data Bases (VLDB'2001), Rome, Italy, September 11-14* (2001), pp. 109–119.

[11] Crescenzi, V., Mecca, G., and Merialdo, P. Roadrunner: Automatic data extraction from data-intensive web sites. In *ACM SIGMOD International Conf. on Management of Data (SIGMOD'2002), Madison, Wisconsin* (2002).

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

[13] Fernau, H. Identification of function distinguishable languages. *Theoretical Computer Science* (2002). To Appear.

[14] Freitag, D. Information extraction from html: Application of a general learning approach. In *Proceedings of the Fifteenth Conference on Artificial Intelligence AAAI-98* (1998), pp. 517–523.

[15] Gold, E. M. Language identification in the limit. *Information and Control 10*, 5 (1967), 447–474.

[16] Grumbach, S., and Mecca, G. In search of the lost schema. In *Seventh International Conference on Data Base Theory, (ICDT'99), Jerusalem (Israel), Lecture Notes in Computer Science, Springer-Verlag* (1999), pp. 314–331.

[17] Gupta, A., Harinarayan, V., and Rajaraman, A. Virtual database technology. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA* (1998), IEEE Computer Society, pp. 297–301.

[18] Hammer, J., Garcia-Molina, H., Cho, J., Aranha, R., and Crespo, A. Extracting semistructured information from the Web. In *Proceedings of the Workshop on the Management of Semistructured Data (in conjunction with ACM SIGMOD 1997)* (1997).

[19] Hsu, C., and Dung, M. Generating finite-state transducers for semistructured data extraction from the web. *Information Systems 23*, 8 (1998), 521–538.

[20] Huck, G., Frankhauser, P., Aberer, K., and Neuhold, E. J. Jedi: Extracting and synthesizing information from the web. In *Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS'98)* (1998), pp. 32–43.

[21] Hull, R. A survey of theoretical research on typed complex database objects. In *Databases*, J. Paredaens, Ed. Academic Press, 1988, pp. 193–256.

[22] Kushmerick, N. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence 118* (2000), 15–68.

[23] Kushmerick, N. Wrapper verification. *World Wide Web Journal 3*, 2 (2000), 79–94.

[24] Kushmerick, N., Weld, D. S., and Doorenbos, R. Wrapper induction for information extraction. In *International Joint Conference on Artificial Intelligence (IJCAI'97)* (1997).

[25] Lerman, K., and Minton, S. Learning the common structure of data. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)* (2000).

[26] Liu, L., Pu, C., and Han, W. Xwrap: An xml-enabled wrapper construction system for web information sources. In *Sixteenth IEEE International Conference on Data Engineering (ICDE'00), San Diego, California* (2000), pp. 611–621.

[27] Muslea, I., Minton, S., and Knoblock, C. Hierarchical wrapper induction for semistructured sources. *Journal of Autonomous Agents and Multi-Agent Systems 4* (2001), 93–114.

[28] Muslea, I., Minton, S., and Knoblock, C. A. A hierarchical approach to wrapper induction. In *Proceedings of the Third Annual Conference on Autonomous Agents* (1999), pp. 190–197.

[29] Papadimitriou, C. H. *Computational Complexity.* Addison-Wesley, 1994.

[30] Pitt, L. Inductive inference, DFAs and computational complexity. In *Analogical and Inductive Inference, Lecture Notes in AI 397*, K. P. Jantke, Ed. Springer-Verlag, Berlin, 1989, pp. 18–44.

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

[32] Ribeiro-Neto, B. A., Laender, A. H. F., and Soares da Silva, A. Extracting semistructured data through examples. In *Proceedings of the 1999 ACM International Conference on Information and Knowledge Management (CIKM'99)* (1999), pp. 94–101.

[33] Sahuguet, A., and Azavant, F. Web ecology: Recycling HTML pages as XML documents using W4F. In *Proceedings of the Second Workshop on the Web and Databases (WebDB'99) (in conjunction with SIGMOD'99)* (1999).

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

A Proof of the Correctness Theorems

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 homogenous 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, $\textsc{lub}(S)$ 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 $\mathcal{I} = \{I_1, \ldots, I_n\}$ of some type σ, σ is a common upper bound of $\mathcal{I}$ in the join-semilattice of templates homogeneous to σ, $\mathcal{H}(\sigma)$. More specifically, we have the following Lemma.

Lemma 2 *Given a set of instances $\mathcal{I} = \{I_1, \ldots, I_n\}$ of type σ, $\mathcal{I}$ is rich for σ iff $\sigma = \textsc{lub}(\mathcal{I})$.*

Proof: Let us prove the two directions separately.

$\mathcal{I}$ is rich for $\sigma \Rightarrow \sigma = \textsc{lub}(\mathcal{I})$. The proof is by contradiction. Assume there is some $\sigma' = \textsc{lub}(\mathcal{I})$. Assume that $\sigma' \neq \sigma$. Still, σ' belongs to $\mathcal{H}(\sigma)$, the class of templates homogeneous to σ. Therefore, it must be the case that $\sigma' \preceq \sigma$. Since σ is a fully instantiated type, σ' cannot be fully instantiated; thus, it must exist at least one instance node; assume the node is labeled α. 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 α in $\mathcal{I}$ all have the same cardinality k, so $\mathcal{I}$ cannot be rich wrt σ.

$\sigma = \text{LUB}(\mathcal{I}) \Rightarrow \mathcal{I}$ is rich for σ. Similarly, let us assume that $\mathcal{I}$ is not rich for σ. Then, it must exist a type node for which $\mathcal{I}$ is not rich. Let us assume that node is a list node labeled α, but the same discussion would hold for any other type node. In that case all list-instance nodes labeled α 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 α. We can build $\sigma' \neq \sigma$ such that σ' is obtained from σ by replacing the subtree rooted at node α 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 σ is $\text{LUB}(\mathcal{I})$. □

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

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

$T \preceq T' \Rightarrow enc(T) \subseteq enc(T')$: follows directly from Definitions 1, 2, and 6 respectively of template, subsumption relationship, and well-formed mark-up encodings. Proposition 1 is just a special case of this assertion.

$enc(T) \subseteq enc(T') \Rightarrow T \preceq T'$: We shall prove equivalently, $T \npreceq T' \Rightarrow enc(T) \nsubseteq enc(T')$.

First note that any regular expression $enc(R)$ such that $R \in \mathcal{H}(\sigma)$ is unambiguous [7]: every symbol occuring in a given string $enc(I) \in L(enc(R))$ (I is an instance of σ) can be unambiguously associated to the terminal symbol it unifies with. Therefore, we can unambiguously label every symbols $x \in \Sigma \cup \overline{\Sigma} \cup \Delta$ of $enc(I)$ with the corresponding type label α. Finally, let us distinguish different terminal symbol occurrences: if the node labeled α is not a leaf ($x \in \Sigma \cup \overline{\Sigma}$), we denote by x_i the i-th occurrence of x within $start(\alpha)end(\alpha)$; if it is a leaf ($x \in \Sigma \cup \overline{\Sigma} \cup \Delta$) 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 α.

We show the thesis by structural induction on the common type σ.

- σ is a basic type. Trivial.
- $\sigma = (\sigma')?$ is an optional type and σ' is a non nullable type. If $T \npreceq T'$ there are four possibilities: (*i*) $T = (_iS)?$ and $T' = (S')?$ such that $S \npreceq S'$, the thesis follows directly from the inductive hypothesis because by Definition 6 of mark-up 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 $S \npreceq S'$, the thesis follow from the case (*i*) by observing that from $(_iS)? \preceq (S)?$ it follows that $L(enc((_iS)?)) \subseteq L(enc((S)?))$ as proved above; (*iii*) $T = (_iS)?$ and $T' = (_iS')?$ such that $S \npreceq S'$, 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 $T' = (_iS')?$, this case can be reduced to case (*iii*) exactly as done for case (*ii*) wrt case (*i*);

• $\sigma = [\sigma_1, \ldots, \sigma_n]$ is tuple type and $\sigma_1 \ldots \sigma_n$ are non tuple types. If $T \not\preceq T'$, it must exist a subtemplate $T_h \preceq \sigma_h$ of T, and a subtemplate $T'_h \preceq \sigma_h$ of T' such that $T_h \not\preceq T'_h$. Observe that any string of $L(enc(T))$ can be unambiguously decomposed in $start(root)w_1w_2 \ldots w_n end(root)$ such that $w_j \in L(enc(T_j))$, $j = 1 \ldots n$, and similarly any string of $L(enc(T'))$ can be unambiguously decomposed in $start(root)w'_1w'_2 \ldots w'_n end(root)$ such that $w'_j \in L(enc(T'_j))$, $j = 1 \ldots n$. 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 σ' is a tuple type. There are several possibilities to consider: (i) $T = \langle_i T_1, \ldots, T_k\rangle$ and $T' = \langle S'\rangle$ such that there exist a $T_j \not\preceq S'$. Consider that any string of $L(enc(T))$ can be unambiguously decomposed in $start(root)w_1w_2 \ldots w_k end(root)$ such that $w_j \in L(enc(T_j))$, $j = 1 \ldots k$. Similarly, any string of $L(enc(T'))$ can be written as $start(root)w'_1w'_2 \ldots w'_n end(root)$ where $n > 0$ and $w'_j \in L(enc(S'))$, $j = 1 \ldots n$. 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 $S \preceq S'$. Consider a new template $T'' = \langle_i S, \ldots, S\rangle$, that is, a list-instance of cardinality k such that $T'' \preceq T = \langle S\rangle$. Then, it follows that $L(enc(T'')) \subseteq L(enc(T'))$ and therefore this case can be reduced to case (i). (iii) $T = \langle_i T_1, \ldots, T_k\rangle$ and $T' = \langle_i T'_1, \ldots, T'_h\rangle$ 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 $L(enc(T)) \subseteq L(enc(T'))$. (iv) $T = \langle S\rangle$ and $T' = \langle_i T'_1, \ldots, T'_h\rangle$, this case can be reduced to case (iii), exactly as done for case (ii) wrt case (i). (v) $T = \langle_i T_1, \ldots, T_k\rangle$ and $T' = \langle_i T'_1, \ldots, T'_k\rangle$ such that for a given h, $T_h \not\preceq T'_h$. Observe that any string of $L(enc(T))$ can be unambiguously decomposed in $start(root)w_1w_2 \ldots w_k end(root)$ such that $w_j \in L(enc(T_j))$, $j = 1 \ldots k$, and similarly any string of $L(enc(T'))$ can be written as $start(root)w'_1w'_2 \ldots w'_k end(root)$ such that $w'_j \in L(enc(T'_j))$, $j = 1 \ldots k$. Since by inductive hypothesis $L(enc(T_h)) \not\subseteq L(enc(T'_h))$, it follows the thesis. □

Lemma 3 and Proposition 2 (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 σ, $\mathcal{E} = \{enc(I_1), \ldots, enc(I_n)\}$ according to some mark-up encoding enc, the strategy to solve the schema finding problem is: (i) to find the regular expression encoding σ as the least upper bound $e_\sigma = \textsc{lub}_\subseteq(\mathcal{E})$; (ii) from the regular expression, to construct σ; (iii) based on the grammar defined by e_σ, from each $enc(I_j)$ to derive a representation of an instance I_j of σ.

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:

$$\begin{cases} lub_1 & = & e_1 \\ lub_{i+1} & = & \textsc{lub}(lub_i, e_{i+1}), \quad \text{for } i = 2, \ldots, k \end{cases}$$

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.[10] This means that, in order to prove Theorem 2, 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 16 (Proper mismatch) *Let T be a template, I an instance of T, and enc a mark-up encoding. A mismatch $(enc(T), enc(I), (n,t))$ produced by $\overrightarrow{\mathsf{parse}}(enc(T), enc(I))$ [respectively $\overleftarrow{\mathsf{parse}}(enc(T), enc(I))$] is called* proper *with respect to a template node labeled α if:*
(i) it is a data mismatch, and (n,t) are encodings of constant templates nodes with the same label α as shown in Figure 16
(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 $\beta = \alpha.0$)

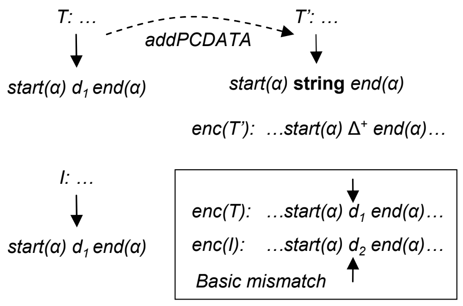


Figure 16: Proper Basic Mismatches

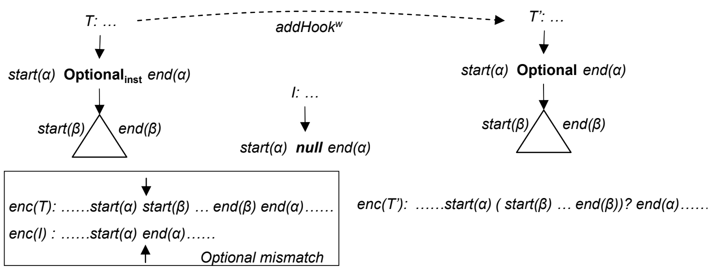
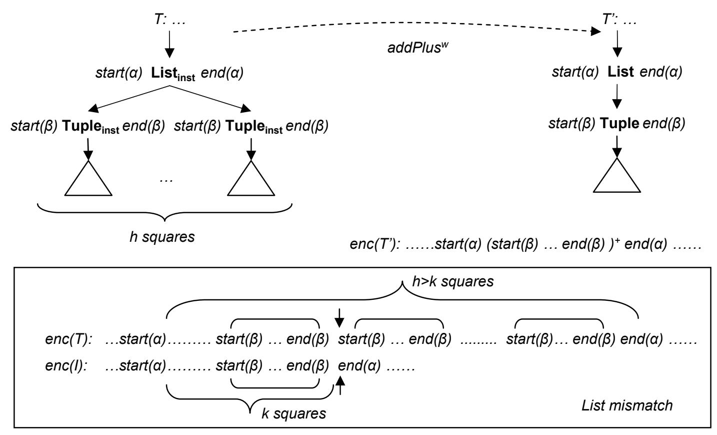

T: ...
addHook^w
T': ...
start(α) Optional_inst end(α)
start(β)
end(β)
I: ...
start(α) null end(α)
start(α) Optional end(α)
start(β)
end(β)
enc(T): ......start(α) start(β) ... end(β) end(α)......
enc(I) : ......start(α) end(α)......
Optional mismatch
enc(T'): ......start(α) (start(β) ... end(β))? end(α)......

Figure 17: Proper Optional Mismatches

Based on the type of the node labeled α 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 α is subsumed respectively by a basic, an optional or a list

[10]Note that computing upper bounds of regular expression implies testing containment. The containment problem for regular expression is complete for PSPACE [29]. 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.



T: ...
addPlus^w
T': ...
start(α) List_inst end(α)
start(α) List end(α)
start(β) Tuple_inst end(β) start(β) Tuple_inst end(β)
start(β) Tuple end(β)
h squares
enc(T'): ......start(α) (start(β) ... end(β))+ end(α) ......
h>k squares
enc(T): ...start(α)......... start(β) ... end(β) start(β) ... end(β) .... start(β)... end(β) end(α) ......
enc(I): ...start(α)......... start(β) ... end(β) end(α) ......
k squares
List mismatch

Figure 18: Proper List Mismatches

node. Proper mismatches are fundamental in our setting. In fact, it can be easily seen that, from Definitions 5 and 6 it follows that:

Proposition 3 *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 17 (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 α 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 α.*

A first result we can prove is the following.

Lemma 4 *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 α 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 11), and *addPlus* (Figure 12), 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 α with a basic template. This generalization solves m and is minimal, because by definition 2 of the $\preceq$ 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, $\overrightarrow{addPlus}$ 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 $\overrightarrow{addPlus}$ defined in Figure 12 would look for a last delimiter by calling $\overrightarrow{\mathsf{lastDelim}}(m)$. Then observe that by definition 9 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 $\overrightarrow{\mathsf{lastDelim}}$ 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 $\overrightarrow{addPlus}$ operators return $\perp$.

Next we show that $\overrightarrow{addHook}^{s}$ 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 starts 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 $\overrightarrow{addHook}^{w}$ is defined, and its invocation of $\overrightarrow{\mathsf{findHookSquare}}$ 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 1 of template: the node labeled β is subsumed by either a basic or a list node. In the former case then $start(\beta) \ldots end(\beta)$ 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 $\overrightarrow{addHook}^{w}$ computes the encoding of a template T' which solves the mismatch by generalizing the optional-instance node labeled α with an optional node. From the point of view of the encodings, consider the last return statement of $\overrightarrow{addHook}^{w}$'s code in Figure 11: it introduces an *Hook* over the candidate square just located. Observe that the generalization is minimal because by definition 2 of the $\preceq$ relationship, an optional-instance template and a null-template are both subsumed only by an optional template. □

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 2) and the complexity result (i.e., Theorem 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 wrt the length of the input encodings[11]. We have the following Lemma, from which both Theorem 2 and Theorem 3 immediately descend.

Lemma 5 *Given two homogeneous templates T and S, and a prefix mark-up encoding enc,* match$(enc(T), enc(S))$ *computes* $enc(\textsc{lub}(T, S))$ *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 4 for list mismatches.

[11]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.

Claim 1 *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 13.

Claim 2 *Let R be a template, I an instance homogeneous to R, and enc a prefix mark-up encoding:* solveMismatches$(enc(R), enc(I)) = enc(\textsc{lub}(R, I))$.

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

Claim 3 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 χ of $w' = enc(S)$, that is, a set of encodings of instances $\mathcal{I} = \{I_1, \ldots, I_n\}$ homogeneous to S (and therefore to T) and such that $\chi = \textsc{lub}_{\subseteq}(enc(I_1), \ldots, enc(I_n))$. By Lemma 3 it follows that $S = \textsc{lub}(I_1, \ldots, I_n)$.

Then we show that Claim 2 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 3 is proper, and by hypothesis is either a basic or an optional mismatch. By Lemma 4 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 $w = enc(\textsc{lub}(T, I)) = enc(T^k)$. This proves Claim 2 while Claim 1 trivially holds as consequence of Lemma 4 because every mismatch is either basic or optional by hypotheses. Finally observe that by the associative property of the LUB, match returns:

$$w = enc(\textsc{lub}(\ldots \textsc{lub}(\textsc{lub}(\textsc{lub}(T, I_1), I_2)), \ldots, I_n)) = enc(\textsc{lub}(T, \textsc{lub}(I_1, \ldots, I_n))) = enc(\textsc{lub}(T, S))$$

This proves the correctness for the basis case.

To discuss the complexity, note that charSample(S) produces $\mathcal{O}(n)$ instances whose encodings are $\mathcal{O}(n)$ long. The number of external mismatches solved by applyOperator for each instance is $\mathcal{O}(n)$. Overall, there are $\mathcal{O}(n^2)$ applyOperator invocations and operator applications. First, observe that from the discussion above it easy to show that undefined operators return $\perp$ only by invoking functions that run in PTIME by assumption. Finally, Claim 3 follows from the fact that by inductive hypothesis and by Claim 1, the only operators invoked are *addPCDATA* and *addHook*, which runs in PTIME by assumption.

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

We prove Claim 1 by exploiting the inductive hypothesis and so we complete Lemma 4 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 $addPlus^s$. Let us call $enc(T_j^w) = start(\beta)\dots end(\beta)$ the j-th square occurrence on the wrapper; for all j, these are all encodings of homogeneous templates whose root node is labeled β. We start by showing that in the case of list mismatches, the *addHook* operators are not defined. $addHook^s$ is not defined because it searches the first symbol of $start(\beta)$ that follows the mismatch on the sample: the candidate squares are not well-formed; in fact, they start with $end(\alpha)$. $addHook^w$ would select as candidate square the sequence of $h-k$ patterns of the form $start(\beta)\dots end(\beta)$ but then it fails since $\overrightarrow{\mathsf{checkSquare}}(m, csquare)$ would detect that the candidate square ends with the same symbol preceding the mismatch point, namely, the last symbol of $end(\beta)$. Finally observe that $addPlus^s$ is not defined because the candidate squares are not well formed: they start with $end(\alpha)$.

In order to complete the proof of Claim 1, we have to show that $addPlus^w$ is defined and solves the mismatch with a minimal generalization. By Definition 9 of prefix mark-up encodings, we know that $end(\beta)$, which marks a tuple node, includes a wrapping delimiter such that $\overrightarrow{\mathsf{lastDelim}}(m)$ and $csquare^w$ are defined, and that the latter corresponds to $enc(T_{h+1}^w)$ on the wrapper.

During the candidate square evaluation we know by inductive hypothesis that $square_0$ equals $enc(\textsc{lub}(T_k^w, T_{k+1}^w))$. Then, occurrences of the candidate square are consumed both on the left and on the right. When occurrences on the left are consumed – i.e., $square_{-1}, square_{-2}, \dots$ $square_{-k+1}$ – we have, by inductive hypothesis that:

$$square_{-1} = enc(\textsc{lub}(T_{k-1}^w, T_k^w, T_{k+1}^w)), \quad \dots \quad , square_{-k+1} = enc(\textsc{lub}(T_1^w, \dots, T_{k-1}^w, T_k^w, T_{k+1}^w)$$

The matching on the left is stopped when the call to $\overleftarrow{\mathsf{match}}(square_{-k+1}, w(\dots f_{-k+1}))$ 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 1 is still valid. In fact, whichever is the sample generated by $\mathsf{charSample}(w(\dots f_{-k+1}))$, a *non*-proper schema mismatch occurs between the last token of $end(\beta)$ and the last token of $start(\alpha)$; this mismatch is not proper because the two template sections are not homogeneous. $\overleftarrow{addPlus}$ is not defined because $\overleftarrow{\mathsf{lastDelim}}(m)$ looks for a last delimiter in an empty region. Finally, $\overleftarrow{addHook}$ is not defined since β marks a tuple node and $end(\beta)$ contains a wrapping delimiter. $\overleftarrow{addHook}^w$ 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 after $start(\beta)$ and ends with $end(\beta)$. Finally, the only candidate squares $\overleftarrow{addHook}^s$ can locate are not well-formed because they end with $start(\alpha)$.

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

$$square_2 = enc(\textsc{lub}(T_1^w, \dots, T_{k-1}^w, T_k^w, T_{k+1}^w, T_{k+2}^w)), \quad \dots \quad , square_{h-k} = enc(\textsc{lub}(T_1^w, \dots, T_h^w))$$

Again, the last recursive call on the right is $\overrightarrow{\mathsf{match}}(square_{h-k}, w(f_{h-k}\dots))$, 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 $enc(\textsc{lub}(T_1^w, \ldots, T_h^w))$ 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 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 $\textsc{lub}(T_1^w, \ldots, T_h^w)$ is the least general. This proves that list mismatches are correctly solved by applyOperator; as a consequence, Claim 1 holds, and Claim 2 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 $\overleftarrow{\textsf{match}}$ 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, *addPlus* operator makes $\mathcal{O}(n)$ recursive invocations of match that by inductive hypothesis runs in PTIME wrt n. This proves that *addPlus* runs in PTIME wrt n and Claim 3 follows exactly like for the basis case. □