Scalable Data Extraction via Program Synthesis

Adi Omari

Scalable Data Extraction via Program Synthesis

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Adi Omari

Submitted to the Senate
of the Technion — Israel Institute of Technology
Nissan 5778 Haifa March 2018

This research was carried out under the supervision of Prof. Eran Yahav and Dr. Sharon Shoham, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period, the most up-to-date versions of which being:

Adi Omari, David Carmel, Oleg Rokhlenko, and Idan Szpektor. Novelty based ranking of human answers for community questions. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 215–224. ACM, 2016.

Adi Omari, Benny Kimelfeld, Eran Yahav, and Sharon Shoham. Lossless separation of web pages into layout code and data. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1805–1814. ACM, 2016.

Adi Omari, Sharon Shoham, and Eran Yahav. Cross-supervised synthesis of web-crawlers. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 368–379. IEEE, 2016.

Adi Omari, Sharon Shoham, and Eran Yahav. Synthesis of forgiving data extractors. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 385–394. ACM, 2017.

ACKNOWLEDGEMENTS

The Technion's funding of this research is hereby acknowledged.

Contents

List of Figures

Abstract

Web extraction is an important research topic that has been studied extensively, receiving a lot of attention and focus. Large amounts of data are produced and consumed online in a continuous and growing rate. The ability to collect and analyze these data has become essential for enabling a wide range of applications and improving the effectiveness of modern businesses. Web extraction methods facilitate the collection and analysis of these data by transforming the human friendly data available online into structured information that can be automatically manipulated and analyzed.

In this work we address the data extraction problem from a software synthesis perspective. Our goal is not only to extract data from web-sites, but also to synthesize programs that extract the data from these sites. The growing popularity of data extraction query languages and their increasing use in a wide variety of applications make them a natural target for automatic synthesis methods. Another motivation for using synthesis for web extraction related applications is the fact that web applications are often generated dynamically using template code. Reverse engineering web pages at the page and site level may facilitate - among other applications - unsupervised web extraction.

First, we focus on the problem of automatic synthesis of web-crawlers for a family of web-sites that contain the same kind of information but differ on layout and formatting. We propose a method that uses the data shared among sites from the same category in order to decrease or eliminate the manual tagging needed for generating extraction schemes for these sites. We use the data on one site to identify data on another site. The identified data is then used to learn the website structure and synthesize an appropriate extraction scheme. This process iterates, as synthesized extraction schemes result in additional data to be used for re-learning the website structure.

In our second work, we address the problem of synthesizing robust data extractors from a family of web-sites that contain the same kind of information. Robust data extractors are more likely to withstand structural changes in the data-source, and can therefore reduce the data extractor's maintenance cost. We introduce and implement the idea of forgiving extractors that dynamically adjust their precision to handle structural changes, without the need to sacrifice precision upfront.

Finally, to address the unsupervised data extraction problem, we propose a solution to the more general problem of separation of web-pages into template-code and data. Web pages are often served by running layout code on data, producing an HTML document that formats the data into a human readable and elegant presentation. We considered the opposite task:

separating a given web page into a data component and a layout program. This separation has various important applications including unsupervised data extraction, traffic compression, data migration and template-code simplification. In our last work, we generalized our separation approach to address the problem of site-level separation.

Chapter 1

Introduction

In this thesis, we address the problem of scalable data extraction via program synthesis. The work is composed of four chapters:

1. Cross-Supervised Synthesis of Web-Crawlers (Chapter 2) addresses the problem of automatically synthesizing web-crawlers for a family of websites that contain the same kind of information but may significantly differ on layout and formatting.

2. Synthesis of Forgiving Data Extractors (Chapter 3) addresses the problem of synthesizing a robust data-extractor from a family of websites that contain the same kind of information.

3. Lossless Separation of Web Pages into Layout Code and Data (Chapter 4) addresses the issue of separating a given web page into a data component and a layout program.

4. Separation of Web Sites into Layout Code and Data (Chapter 5) addresses the separation of a website into a small number of layout programs and structured data sources

Program synthesis for data-extraction: In this work we explore different ways in which program synthesis can facilitate data-extraction. One direction is to use program-synthesis techniques to automatically synthesize web-extraction programs. We explore this direction in Chapter 2 where we propose a method for automatic synthesis of data-extraction oriented web crawlers for a group of sites, and in Chapter 3 we propose a method for automatic synthesis of robust web-extraction queries. Another direction that we explore in Chapters 4 and 5 is to separate web-pages into template code and data. Our separation approach outperforms state of the art methods for unsupervised record-level data extraction.

Many different techniques has been developed to deal with the data-extraction problem, leveraging methods coming from various disciplines including Machine Learning and Natural Language Processing. Ferrera et. al. [FDMFB14] review some of these data-extraction approaches, and classify them into categories. All of the method proposed in this thesis can be classified (According to [FDMFB14]) as tree-based techniques. We represent web-pages as labeled ordered rooted trees, where labels represent the HTML tags, and the tree hierarchy represents the nesting structure of the web-page elements. Our extractor synthesis methods (detailed in Chapters 2 and 3) use XPath to address the document elements, while our separation based methods (detailed in Chapters 4 and 5) use a variance of tree-alignment algorithm to perform the separation. We compare the input, output and type of our different methods in Table 1.1.

Work	Type	Input	Output
Cross-Supervised Synthesis of Web-Crawlers (Chapter 2)	Extractor synthesis	A set of websites containing the same type of data, and a web-crawler for one of these sites.	Set of web-crawlers, one for each site in the input set.
Synthesis of Forgiving Data Extractors (Chapter 3)	Extractor synthesis	A set of tagged documents from different sites	A single data extractor that works on all the different input sites
Lossless Separation of Web Pages into Layout Code and Data (Chapter 4)	Separation	an HTML page	A layout-code file and a data file
Separation of Web Sites into Layout Code and Data (Chapter 5)	Separation	A sample set of web pages from a single web site	A set of layout code pages and their respective data files.

Table 1.1: A summary of the techniques presented in this dissertation.

In the rest of this section, we introduce each of our works separately.

1.1 Cross-Supervised Synthesis of Web-Crawlers

1.1.1 Motivation

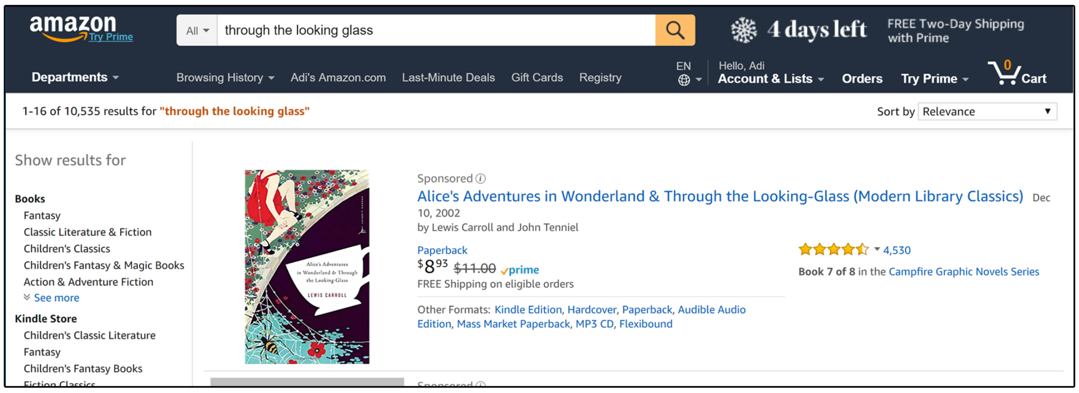
The world wide web is a rich source with growing amounts of unstructured and semi-structured of valuable data. Lately, a lot of efforts were put in order to enable web crawlers to identify and extract these unstructured data. Due to the different formats of websites, the crawling scheme for different sites can differ dramatically. Manually customizing a crawler for each specific site is time consuming and error-prone. Furthermore, because sites periodically change their format and presentation, crawling schemes have to be manually updated and adjusted.

While the structure of documents is significantly different between different sites, we found that sites from the same category often have shared data instances. For example, Fig. 1.1 shows two different bookseller web sites. Documents of these web sites have significantly different structure and style. However, these bookseller web sites have many shared book titles and authors names. In this work, we use this fact to enable automatic synthesis of web crawlers for a group of web sites based on a web crawler for a single site from the same category.

1.1.2 Problem Definition

Chapter 2 considers websites whose data-containing webpages share the following logical structure: each webpage describes one main relation, denoted *data*. As such, data items are tuples of the *data* relation. Further, the set *Att* of attributes consists of the columns of the *data* relation.

We define the problem of *crawler synthesis* w.r.t. a set *Att* of attributes as follows: Given a



1-16 of 10,535 results for "through the looking glass"

Sponsored

Alice's Adventures in Wonderland & Through the Looking-Glass (Modern Library Classics) Dec 10, 2002

by Lewis Carroll and John Tenniel

Paperback

$8.93 ~~$11.00~~ prime

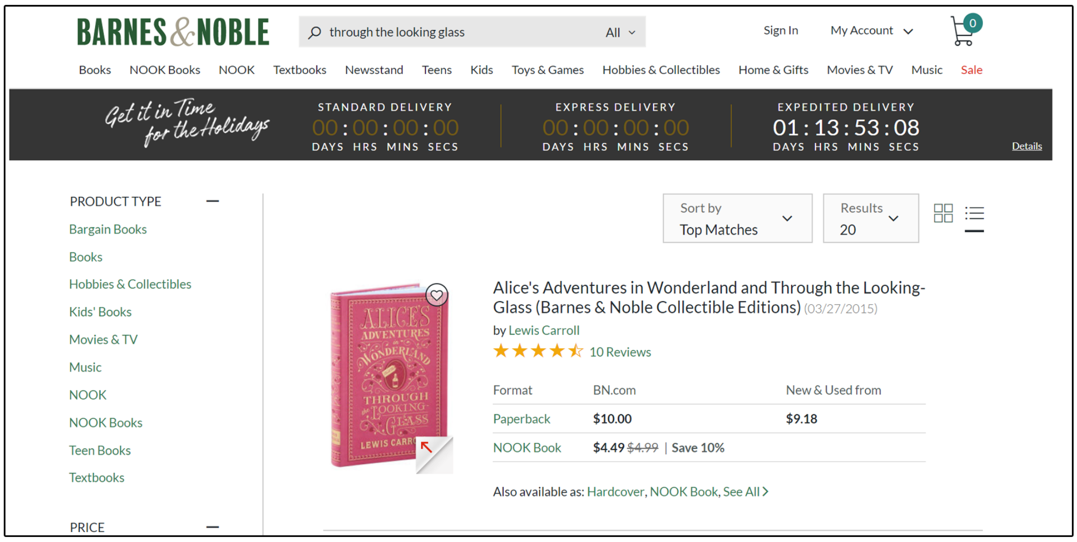
FREE Shipping on eligible orders

4,530

Book 7 of 8 in the Campfire Graphic Novels Series

Other Formats: Kindle Edition, Hardcover, Paperback, Audible Audio Edition, Mass Market Paperback, MP3 CD, Flexibound

Amazon US



Alice's Adventures in Wonderland and Through the Looking-Glass (Barnes & Noble Collectible Editions) (03/27/2015)

by Lewis Carroll

10 Reviews

Format	BN.com	New & Used from
Paperback	$10.00	$9.18
NOOK Book	$4.49 ~~$4.99~~ | Save 10%	

Also available as: Hardcover, NOOK Book, See All

Barnes & Noble

Figure 1.1: "Through the looking glass" book in Amazon.com and Barnes & Noble

set S of websites, where each website $s \in S$ is associated with a set of webpages, denoted $P(s)$, and with a data extractor, denoted $E(s)$, which might be partial or even empty. The goal is to synthesize a complete data extractor for every $s \in S$. A data extractor is a mapping between the logical structure of a webpage - defined by the set *Att* of attributes each data item conatins - and its concrete layout.

1.1.3 Existing Techniques

One of the related problems to crawler synthesis is the problem of *wrapper induction* [KWD]. The goal of wrapper induction is to automatically generate extraction rules for a website based on the regularity of pages inside the site. In contrast to supervised wrapper-induction techniques [KWD, LG14, LPH00, CH04], which require labeled examples, and unsupervised wrapper-induction techniques [AGM03, CL, CMM01, MK07, MTH$^+$09, RGSL04, ZL05] that frequently require manual annotation of the extracted data, our approach uses *cross supervision*, where the learned extraction rules of one site are used to produce labeled examples for learning the extraction rules in another site.

Our crawler synthesis technique is also related to corpus-based schema matching techniques [MBDH05, HCH04] and distant-supervision relation extraction techniques [HZL$^+$11, MBSJ09]. In contrast to corpus-based schema matching methods that match two structured (or annotated) data sources, our technique matches an unstructured data-source with a target data-scheme and synthesizes extraction queries to extract the data into a target structured data-source. Distant-supervision relation extraction techniques use weak-supervision to solve the different problem of automatically learning entity-relation extractors (e.g. learn a regular expression to extract company-founder information from text).

Our main idea is to try and leverage a similar regularity across multiple sites. However, because different sites may significantly differ on their layout, we have to capture this regularity at a more semantic level. Towards that end, we use the shared *data instances* among these different websites, which allows us to identify commonality even in the face of different formatting details.

The technique described in chapter 2 generates XPath [CD$^+$99] expressions, a widely used web documents query language along with regular expressions. This makes our resulting extraction schemes human readable and easy to modify when needed. XPath *robustness* to site changes [DBS09a, LSRT14, LWLY12] is another related problem to our work. While robustness is a desirable property, our ability to efficiently synthesize a crawler circumvents this challenge as a crawler can be regenerated whenever a site changes.

1.1.4 Key Aspects of the Approach

Despite the significant differences in the concrete layout of websites, the pages of websites that present the same product category often share the same *logical structure*; they present similar attributes for each product. The main idea is to exploit this data overlap across sites in order to learn the concrete structure of a new website s based on other websites.

Iterative synthesis of crawling schemes Our approach (illustrated in Fig. 1.2) considers a set of websites, and a set of attributes to extract. To bootstrap the synthesis process, the user is required to provide the set of websites for which crawler synthesis is desired, as well as a crawling scheme for at least one of these sites. Our iterative algorithm consists of three main steps:

Extraction*:* For each site (denoted s_{done}) that have a new web-crawler (that we have as input or from the previous iteration) run the web-crawler on s_{done} to extract data instances for each data-attribute and add them to our attribute instances-database.

Identification *of data-attribute instances:* For each site s that is still without a web-crawler. We use the concrete data extracted from other sites (instances database) to identify attribute occurrences in s and as such learn the structure in which this data is represented in s. We do not require a precise match of data across sites, as our technique also handles noisy data. (For example, prices do not have to be identical; any number can be a match.)

Crawler* synthesis *using the identified occurrences as examples: We then use multiple examples of the structure in which the data appears in s in order to generalize and get an extraction

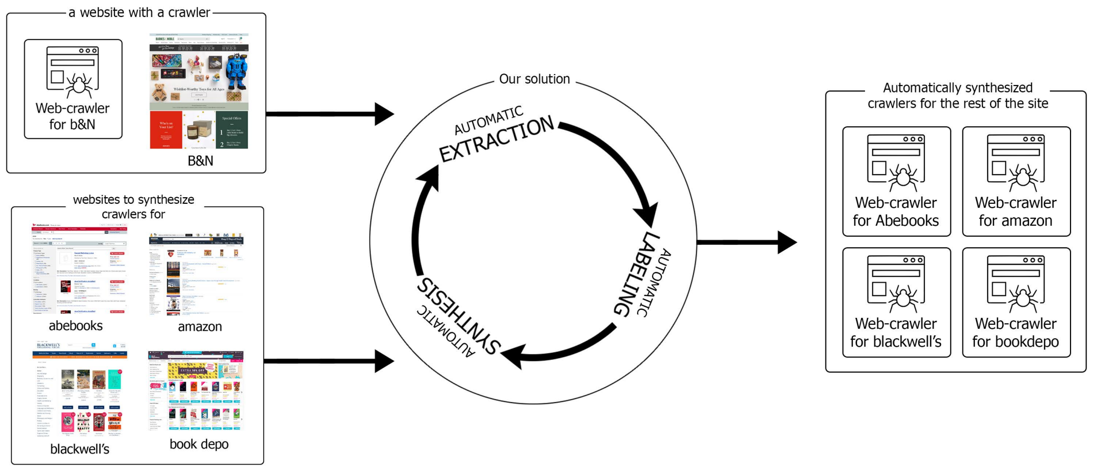


Figure 1.2: Automatic synthesis of web-crawlers for a group of site given a web-crawler for a site from the same category.

query for s. This enables our algorithm to extrapolate a crawler for s.

Crawling schemes Given a set of attributes, a crawling scheme consists of the following two components: (i) A data extraction query that defines how to obtain values of the attributes for each item listed on the site. (ii) A starting point URL and a URL filtering pattern which let the crawler locate "relevant" pages and filter out irrelevant pages without downloading and processing them.

Two-level data extraction schemes We assume that the data extraction query has two levels: The first level query describes the item container. Intuitively, a container is a sub-tree that contains all the attribute values we would like to extract. The second level queries contain an extraction XPath for values of each individual attribute. These XPaths are relative to the root of the container.

1.1.5 Main Contributions

The contributions in Chapter 2 are:

- A framework for automatic synthesis of web-crawlers. The main idea is to use hand-crafted crawlers for a number of websites as the basis for crawling other sites that contain the same kind of information.
- A new cross-supervised crawler synthesis algorithm that extrapolates crawling schemes from one web-site to another. The algorithm handles pages with multiple items and synthesizes crawlers using only positive examples.
- An implementation and evaluation of our approach, automatically synthesizing 30 crawlers for websites from nine different categories: books, TVs, conferences, universities, cameras, phones, movies, songs and hotels. The crawlers that we synthesize are real crawlers that were used to crawl more than $12,000$ webpages over all categories.

1.2 Synthesis of Forgiving Data Extractors

1.2.1 Motivation

Web sites often change their formatting and structure, even when their semantic content remains the same. This change in structure causes extractors to break repeatedly, which results in high maintenance costs for these extractors. A robust extractor [DBS09b, LSRT14, LWLY12] can withstand modifications to the target site. A non-robust extractor would have to be adjusted (typically manually) every time the formatting of a site changes.

When constructing a data extractor, there is a natural tradeoff between accuracy and robustness. Constructing a precise extractor may prevent it from being robust to future changes. Constructing a loose extractor makes it more robust, but would yield results of poor accuracy. *Forgiving extractors* can dynamically adjust their precision to handle structural changes, without sacrificing precision when it is unnecessary.

1.2.2 Problem Definition

Given a set of example annotated web pages from multiple sites in a family, our goal is to synthesize a robust data extractor that performs well on all sites in the family (not only on the provided example pages).

1.2.3 Existing Techniques

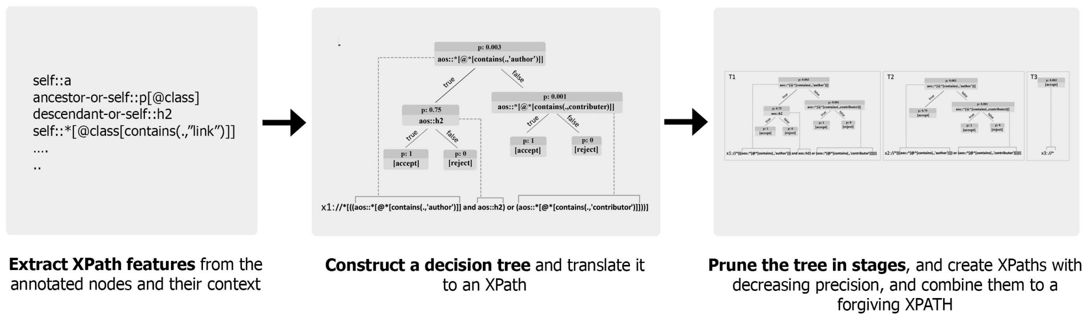
Manually writing a data extractor is extremely challenging. This motivated techniques for automatic generation of extraction queries from examples [KWD]. Automated techniques reduce the burden of writing extractors, but still require manual effort (e.g., providing tagged samples for each site).

There has been a lot of work on pattern based techniques, using alignment of XPaths (e.g., [RGSL04, NDMBDT14, NBdT16]). These techniques learn paths to the annotated data items and generalize them to generate an extraction query. When provided with items that have significantly different paths (e.g., originate from different sites) these techniques may result in an overly relaxed query expression, and significant loss of precision. As a result, pattern based techniques are often limited to a single web site, and are sensitive to formatting changes.

Model based techniques [HCPZ11, FK04, SWL^{+}12, Kus00], use features of DOM elements to learn a model that classifies DOM items to *data* and *noise*. These methods have several drawbacks. First, they lack a standard execution mechanism for extraction (in contrast to XPath queries that are available for pattern based techniques). Second, the classifiers trained by these techniques are often hard to understand and modify. Last, but not least, the generalization in these models is performed *at training time* and thus presents the standard dilemma between precision and generality.

1.2.4 Key Aspects of the Approach

Our approach is based on the assumption that, despite the structural differences between sites from the same semantic family, the data locations in these sites still share *some* local structural features (e.g., a book title may appear under some heading class, or its parent may be of some



self::a
ancestor-or-self::p[@class]
descendant-or-self::h2
self::*[@class[contains(.,"link")]]
Extract XPath features from the annotated nodes and their context
p: 0.003
aos::*[@*[contains(.,'author')]]
true
false
p: 0.75
aos::h2
p: 0.001
aos::*[@*[contains(.,contributer)]]
p: 1
[accept]
p: 0
[reject]
x1://*[((aos::*[@*[contains(.,'author')]] and aos::h2) or (aos::*[@*[contains(.,'contributor')]]))]
Construct a decision tree and translate it to an XPath
T1
T2
T3
Prune the tree in stages, and create XPaths with decreasing precision, and combine them to a forgiving XPATH

Figure 1.3: Forgiving XPath synthesis process.

particular class). Training on few sites from the family will cover most of these shared local structural patterns. Our forgiving XPath generation process (illustrated in Fig. 1.3) consists of three steps:

XPath features extraction: Given a set of tagged HTML documents, our method extracts a large set of XPath features from these documents. The extracted features are expressible by valid XPath queries. These features are used as the building blocks for generating the XPaths. Each feature is defined by an XPath predicate, such that the value of the feature in a document node is 1 iff the value of the XPath predicate on the same node is true.

Precise extractor learning using decision-trees given a set of annotated pages from different sites, we use decision tree learning to synthesize a *common XPath query* that *precisely* extracts the data items from these pages. Intuitively, the decision tree is used as a generalization mechanism that picks the important features for precise extraction. The overall set of features consists of XPath predicates that capture local syntactic structural properties. The decision tree is used to synthesize a query with high robustness, while maintaining its precision on the training set.

Forgiving XPath synthesis The XPath query constructed in the previous step is precise for pages in the training set, but may be overly restrictive for extraction from other pages. To generalize to other pages, we introduce the novel notion of *forgiving XPaths*—a query that *dynamically relaxes its requirements*, trading off precision to make sure that target items are extracted. Given the decision tree we learnt earlier, we create a sequence of decision trees with a monotonically decreasing precision and a monotonically increasing recall by pruning the resulting decision tree in steps with decreasing minimal precision threshold. This sequence of monotonically pruned decision trees is used to generate a sequence of XPaths with decreasing precision and increasing recall. The forgiving XPath is then generated by concatenating these XPath into a single XPath in a way that guarantees that highest precision XPath is invoked first, and only if it returns no results the following XPath with next lower precision is invoked.

The query generated by our approach has the benefits of both pattern based techniques and model based techniques. On the one hand, it is a standard XPath query, which has a wide support from web-browsers, programming languages and DOM parsers. This also makes it human readable, easy to review and modify by a programmer. On the other hand, the query has the flexibility and generalization ability of the techniques based on learning models. Table 1.2 compares our method to other popular extraction methods.

	XPath alignment	Classifier based	Forgiving XPath
Readable/Editable	Yes	No	Yes
Standard XPath	Yes	No	Yes
Generalization	No	Yes	Yes
Robustness	No	Yes	Yes
Forgiveness/ Dynamic precision	No	No	Yes

Table 1.2: Comparison between our forgiving XPaths and the different popular data-extraction approaches

1.2.5 Main Contributions

Main Contributions The contributions in Chapter 3 are:

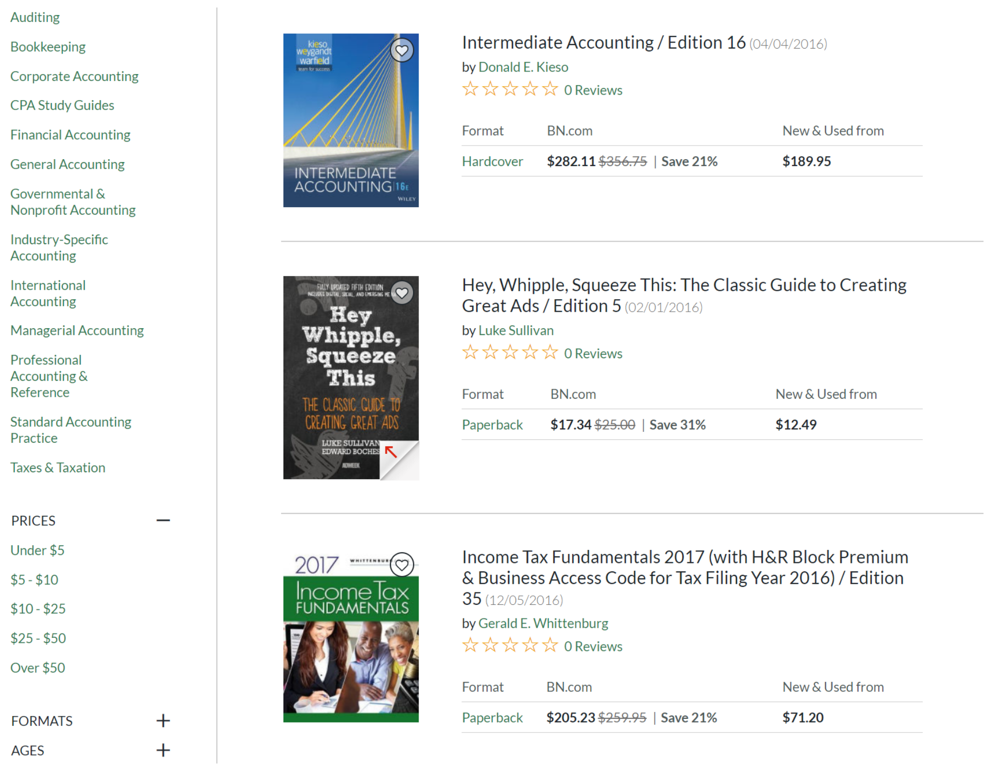
- A novel framework for synthesis of robust data extractors for a family of sites from examples (annotated web pages).
- A new technique for generalization of XPath queries using decision trees and "forgiving XPaths", which adjust precision dynamically.
- An implementation of our technique in a tool called TRACY and an experimental evaluation of the robustness and generality of the forgiving extractors. We evaluate precision and recall on: (i) different pages from sites in the training set (ii) pages from different versions of sites in the training set (iii) pages from different (unseen) sites. Our evaluation shows that TRACY is able to synthesize robust extractors with high precision and recall based on a small number of examples. Further, comparison to existing pattern based and model based techniques shows that TRACY provides a significant improvement.

1.3 Lossless Separation of Web Pages into Layout Code and Data

1.3.1 Motivation

A modern web page is often served by running layout code on data, producing an HTML document that enhances the data with front/back matters and layout/style operations. While templates are useful for the human user, improving readability and providing uniformity among the different pages of a web site, they are considered a problem for web crawlers and data extractors. Templates often contain irrelevant information that makes crawling less efficient and can affect the indexing accuracy. Fig. 1.4 shows a bookseller web page snippet (from barnesandnoble.com). The snippet contains three book items, each contains the book title, the author name and the book price among other info. These items share the same template, that is used to format the book information into a human readable listing.

In this work we consider the web-page separation task: separating a given web page into a data component and a layout program. We use the shared template among different page items as a guide for separating these items back into template and data items. This separation has various important applications: template removal, unsupervised data extraction, and page



Auditing
Bookkeeping
Corporate Accounting
CPA Study Guides
Financial Accounting
General Accounting
Governmental & Nonprofit Accounting
Industry-Specific Accounting
International Accounting
Managerial Accounting
Professional Accounting & Reference
Standard Accounting Practice
Taxes & Taxation

PRICES −
Under $5
$5 - $10
$10 - $25
$25 - $50
Over $50

FORMATS +
AGES +

Intermediate Accounting / Edition 16 (04/04/2016)
by Donald E. Kieso
☆☆☆☆☆ 0 Reviews

Format	BN.com	New & Used from
Hardcover	$282.11 ~~$356.75~~ | Save 21%	$189.95

Hey, Whipple, Squeeze This: The Classic Guide to Creating Great Ads / Edition 5 (02/01/2016)
by Luke Sullivan
☆☆☆☆☆ 0 Reviews

Format	BN.com	New & Used from
Paperback	$17.34 ~~$25.00~~ | Save 31%	$12.49

Income Tax Fundamentals 2017 (with H&R Block Premium & Business Access Code for Tax Filing Year 2016) / Edition 35 (12/05/2016)
by Gerald E. Whittenburg
☆☆☆☆☆ 0 Reviews

Format	BN.com	New & Used from
Paperback	$205.23 ~~$259.95~~ | Save 21%	$71.20

Figure 1.4: A list of books from Barnes & Noble

encoding may be significantly more compact (reducing traffic). Many of these applications are not limited to static web pages but can also be applied to dynamically generated pages (e.g., by using a headless browser to obtain a static HTML page).

1.3.2 Problem Definition

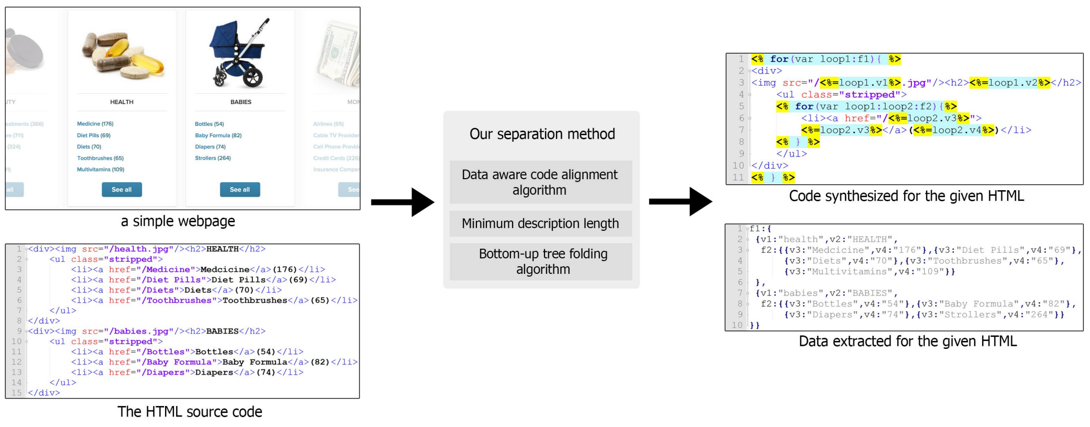
We define the problem of *lossless separation of web-pages* as follows: Given an HTML page, our goal is to *separate* it into a *layout code* component and a data component such that: (i) the separation is *lossless*, running the extracted layout code on the extracted data reproduces the original page, and (ii) the separation is *efficient* such that common elements become part of the layout code, and varying values are represented as data. .

A *separation* of a web page w is a pair $(\pi, \mathcal{E})$, where π is a layout tree and $\mathcal{E}$ is a data-component (which is a mapping from the variables in π to values), such that $\pi(\mathcal{E}) = w$. *Separating* w is the process of constructing a separation $(\pi, \mathcal{E})$ of w. Note that a web page may have infinitely many separations.

1.3.3 Existing Techniques

The separation problem is related to the problems of page-level data extraction [AGM03, KC10, SC13, SC$^+$14, WLF15, CKGS06, FWB$^+$11b] and record-level extraction [TW13, WL03, CL01, ZL05, LMM06, SL05, LMM10, CYWM03, LGZ03a, FWB$^+$11a]. A lot of related work has dealt with these two problems. However, the focus has been on the unsupervised data-extraction aspect only. More importantly, the resulting separation into templates and records is lossy; that is, we cannot recover the original HTML document from the output records and template.

Template identification and removal is another related problem. Templates are considered harmful for many automated tasks like semantic-clustering, classification and indexing by search engines. A lot of past work tackled the challenges of template identification [RGSL04, LGZ03a, ZL05, BYR02, KFN10] and template-extraction [KS11, VdSP+06, CKP07, GPT05, GF14, DC, GM13]. Typically, the goal of these works is to identify or extract the template so it can be ignored/discarded, and the data could be passed to further processing.



```
<div><img src="/health.jpg"/><h2>HEALTH</h2>
    <ul class="stripped">
        <li><a href="/Medicine">Medcicine</a>(176)</li>
        <li><a href="/Diet Pills">Diet Pills</a>(69)</li>
        <li><a href="/Diets">Diets</a>(70)</li>
        <li><a href="/Toothbrushes">Toothbrushes</a>(65)</li>
    </ul>
</div>
<div><img src="/babies.jpg"/><h2>BABIES</h2>
    <ul class="stripped">
        <li><a href="/Bottles">Bottles</a>(54)</li>
        <li><a href="/Baby Formula">Baby Formula</a>(82)</li>
        <li><a href="/Diapers">Diapers</a>(74)</li>
    </ul>
</div>
```

```
<% for(var loop1:f1){ %>
<div>
<img src="/<%=loop1.v1%>.jpg"/><h2><%=loop1.v2%></h2>
    <ul class="stripped">
    <% for(var loop1:loop2:f2){%>
        <li><a href="/<%=loop2.v3%>">
        <%=loop2.v3%></a>(<%=loop2.v4%>)</li>
    <% } %>
    </ul>
</div>
<% } %>
```

```
f1:{
 {v1:"health",v2:"HEALTH",
  f2:{{v3:"Medcicine",v4:"176"},{v3:"Diet Pills",v4:"69"},
      {v3:"Diets",v4:"70"},{v3:"Toothbrushes",v4:"65"},
      {v3:"Multivitamins",v4:"109"}}
 },
 {v1:"babies",v2:"BABIES",
  f2:{{v3:"Bottles",v4:"54"},{v3:"Baby Formula",v4:"82"},
      {v3:"Diapers",v4:"74"},{v3:"Strollers",v4:"264"}}
}}
```

Figure 1.5: Lossless separation of an HTML page to code and data.

1.3.4 Key Aspects of the Approach

Lossless separation: Instead of focusing solely on the data-extraction problem, we solve the web-page lossless separation problem (illustrated in Fig. 1.5). Separation combines, and generalizes, two aspects of the extraction problem that are typically considered separately-record extraction, and template extraction-and seeks to balance them. Rather than treating the template as noise when extracting data, or eliminating data when extracting a template, separation seeks to extract both at the same time.

Minimum description length: There are many possible ways to separate a web-page into layout code and data. We guide our choice of separation by attempting to minimize the joint representation size of the page, according to the minimum description length (MDL) [Ris78] principle. Therefore, the separation computed by our solution is tailored to minimizing the description cost of the result.

Bottom-up separation process: An HTML page can also be viewed as a Tree (called DOM tree). Our separation approach works by folding adjacent subtrees of this DOM tree in a bottom-up manner producing a *layout tree*, which represents the resulting template code. Initially, the layout tree is simply the DOM tree representing the web page. As subtrees are being folded (starting from leaf nodes up to the root), we synthesize unified code that represents their common structure, and create separate data elements to represent their different content values.

The synthesized layout tree may include loops to generalize repetition of layout across items and conditionals to allow these loops to format elements that exhibited a *loosely* similar structure.

Data-aware tree alignment algorithm: We present a novel tree-alignment algorithm that is tailored to handling layout trees, enabling it to handle loops, conditions and variables in the template. This tree-alignment algorithm enables us to represent (and calculate the cost of representing) two layout-subtrees as a single one. The tree alignment algorithm performs data extraction and modifies the data representation, to match the changes in the layout trees, so that invoking the layout trees on the data will result in the original HTML.

Extraction of hierarchical data: Our solution supports nested repetitions (e.g., a list of categories, each containing a list of products) by allowing nesting of loops in data trees alongside hierarchical structures of environments.

1.3.5 Main Contributions

The contributions in Chapter 4 are:

- We present a framework for defining the web-page separation task, and explain its importance for various application.
- We devise an algorithm for synthesizing layout code from a web page while distilling its data in a *lossless* manner.
- We have implemented our approach and conducted a thorough experimental study of its effectiveness. Our experiments show that our approach features state of the art (and higher) performance in both representation compression and record extraction.

1.4 Separation of Web Sites into Layout Code and Data

1.4.1 Motivation

Many of the popular data-extraction methods are applicable on a small set of user selected pages, or a single page. While these methods may be useful for record-level (from a single page) data extraction, they are often unapplicable for large scale modern websites. Modern websites often have millions of template generated pages. These pages are generated by applying a small set of template files to a structured data source, producing the set of HTML pages that are then presented to the user.

In this work we address the problem of scalable data-extraction by introducing the concept of site-level separation. Given a website, our technique separates it to a small number of layout programs and structured data sources.

Separation of website pages has many important applications such as data extraction, webpage clustering, template removal, and web traffic optimization. For example, the data files generated by our approach can be used for unsupervised data extraction (facilitating wrapping and retrieval). The resulting separation can also be used for traffic reduction and automatic conversion of web applications to Ajax applications. These applications are not limited to server-side generated sites but can also be applied to client-side (Angular or Ajax in general) generated sites (e.g., by using a headless browser to obtain a static HTML page).

1.4.2 Problem Definition

Given a set of HTML pages, our goal is to *separate* them into a small set of *layout programs* and a set of structured data sources such that: (i) each layout program represents a subset of HTML pages sharing the same (or a lot of) the formatting elements (ii) running the corresponding layout program of a page on its extracted data source reproduces the original page (the separation is *lossless*), and (iii) the separation is *efficient* such that the entire set of pages is represented using a small number of layout code files, where common elements among pages become part of their representative layout code file, and their varying values are represented as data.

1.4.3 Existing Techniques

There has been a lot of work on data extraction from web pages [AGM03, KC10, SC13, SC$^+$14, TW13, WL03, CL01, ZL05, SL05, LMM10, CYWM03, LGZ03a, LWLY12, DBS09b, HCPZ11, MTH$^+$09]. Record-level extraction handles the case where a single page has a list of records, where there is repetition within the same page. Our approach deals with the complementary problem where the repetition is *not only* within the same page, but also across different pages. Previous work [OKYS16], applied the idea of *separation* to separate single page and perform record-level extraction. In this paper, we address the challenging problem of *separating a set of web pages*.

1.4.4 Key Aspects of the Approach

Given a sample set of pages from a website, our solution generates a more efficient representation using a smaller set of template pages and structured data-sources.

To generate the website separation we need to identify pages in the sample that have similar formatting and can thus be formatted using the same layout code, and only differ on the data. We also need to select a representative set that efficiently covers the variety of different templates that are used in a website.

Unification-guided similarity between pages: We define a page similarity function that captures the effort required to unify two pages. This similarity function is an important aspect of our solution as ability to assess structural similarity of pages is essential to our approach.

To assess the effort requited for unifying two given pages, our similarity function measures their similarity at the layout-code level. To that end, it applies a preprocessing step of *page-level separation* in which it applies a tree folding algorithm [OKYS16] on each page individually in order to fold item lists and unify their representation among the different pages. Then it measures the similarity between the two resulting layout-codes by using a tree-alignment algorithm to calculate their shared representation cost.

Representative set selection: We propose a novelty-based summarization algorithm that finds a small set of representative pages that cover the different page templates in the input set. These representatives are used by our clustering algorithm to compute clusters of pages that share a common template.

Our novelty-based algorithm uses a greedy iterative process, which measures the novelty of each unselected page, as well as the coverage it provides. At each iteration the algorithm selects as an additional representative the page that contributes the highest total coverage of novel pages, and penalizes their novelty accordingly.

Clustering: After a representative set is selected, each page in the input set is associated with the representative that is most structurally similar to it, forming clusters of structurally similar webpages. We use the same unification-guided similarity function we use in the representative selection process.

Cluster level separation: We align together pages from each resulting cluster. The alignment process starts with the layout-code of the representative page, obtained by page-level separation, as the current layout-code. In each iteration, the layout code is updated based on a new page whose layout-code is aligned with the current layout-code. Further, a new data component is generated for the newly aligned page.

1.4.5 Main Contributions

Main Contributions The contributions in Chapter 5 are:

- We present a novel algorithm for efficient *separation* of a set of webpages into layout code files and structured data sources.
- We propose a representative selection algorithm that identifies a *guide set* of webpages that represent all the templates used in a larger set of webpages. The main idea of our algorithm is to use novelty-based summarization based on a structure-aware similarity function for picking the representative pages efficiently, to enable coverage of the different templates via a typically small number of representatives.
- We implement and evaluate our approach. Our evaluation shows that our approach is effective in capturing the different templates present in a site, and in extracting the structured data from pages with different templates.

Chapter 2

Cross-Supervised Synthesis of Web-Crawlers

Abstract

A web-crawler is a program that automatically and systematically tracks the links of a website and extracts information from its pages. Due to the different formats of websites, the crawling scheme for different sites can differ dramatically. Manually customizing a crawler for each specific site is time consuming and error-prone. Furthermore, because sites periodically change their format and presentation, crawling schemes have to be manually updated and adjusted. In this paper, we present a technique for automatic synthesis of web-crawlers from examples. The main idea is to use hand-crafted (possibly partial) crawlers for some websites as the basis for crawling other sites that contain the same kind of information. Technically, we use the data on one site to identify data on another site. We then use the identified data to learn the website structure and synthesize an appropriate extraction scheme. We iterate this process, as synthesized extraction schemes result in additional data to be used for re-learning the website structure. We implemented our approach and automatically synthesized 30 crawlers for websites from nine different categories: books, TVs, conferences, universities, cameras, phones, movies, songs, and hotels.

2.1 Introduction

A web-crawler is a program that automatically and systematically tracks the links of a website and extracts information from its pages. One of the challenges of modern crawlers is to extract *complex structured information* from different websites, where the information on each site may be represented and rendered in a different manner and where each data item may have multiple attributes.

For example, price comparison sites use custom crawlers for gathering information about products and their prices across the web. These crawlers have to extract the structured information describing products and their prices from sites with different formats and representations. The differences between sites often force a programmer to create a customized crawler for each site, a task that is time consuming and error-prone. Furthermore, websites may eventually change their format and presentation, therefore the crawling schemes have to be manually maintained and adjusted.

Goal The goal of this work is to automatically synthesize web-crawlers for a family of websites that contain the same kind of information but may significantly differ on layout and formatting.

We assume that the programmer provides one or more hand-crafted web-crawlers for some of the sites in the family, and would like to automatically generate crawlers for other sites in the family. For example, given a family of four websites of online book stores (each containing tens of thousands of books), and a hand-crafted crawler for one of them, we automatically generate crawlers for the other three. Note that our goal is not only to extract data from web-sites, but to synthesize the programs that extract the data.

Existing Techniques Our work is related to *wrapper induction* [KWD]. The goal of wrapper induction is to automatically generate extraction rules for a website based on the regularity of pages inside the site. Our main idea is to try and leverage a similar regularity across multiple sites. However, because different sites may significantly differ on their layout, we have to capture this regularity at a more abstract level. Towards that end, we define an abstract *logical representation* of a website that allows us to identify commonality even in the face of different formatting details.

In contrast to supervised techniques [KWD, LG14, LPH00, CH04], which require labeled examples, and unsupervised techniques [AGM03, CL, CMM01, MK07, MTH$^+$09, RGSL04, ZL05] that frequently require manual annotation of the extracted data, our approach uses *cross supervision*, where the learned extraction rules of one site are used to produce labeled examples for learning the extraction rules in another site.

Our technique uses XPath [CD$^+$99], a widely used web documents query language along with regular expressions. This makes our resulting extraction schemes human readable and easy to modify when needed. There has been some work on the problem of XPath *robustness* to site changes [DBS09a, LSRT14, LWLY12], trying to pick the most robust XPath query for extracting a particular piece of information. While robustness is a desirable property, our ability to efficiently synthesize a crawler circumvents this challenge as a crawler can be regenerated whenever a site changes.

Our Approach: Cross-Supervised Learning of Crawling Schemes We present a technique for automatically synthesizing data-extracting crawlers. Our technique is based on two observations: (i) sites with similar content have data overlaps, and (ii) in a given site, information with similar semantics is usually located in nodes with a similar location in the document tree.

Using these observations, we synthesize data-extracting crawlers for a group of sites sharing the same type of information. Starting from one or more hand-crafted crawlers which provide a relatively small initial set of crawled data, we use an iterative approach to discover data instances in new websites and extrapolate data extraction schemes which are in turn used to extract new data. We refer to this process as *cross-supervised learning*, as data from one web-site is repeatedly used to guide synthesis in other sites.

Our crawlers extract data describing different attributes of items. We introduce the notion of a *container* to maintain relationships between different attributes that refer to the same item. We use containers, which are automatically selected without any prior knowledge of the structure of the website, to handle pages with multiple items, and to filter out irrelevant data. This allows us to synthesize extraction schemes from positive examples only.

Our approach is scalable and practical: we used cross-supervision to synthesize crawlers for several product review websites, e.g., `tvexp.com`, `weppir.com`, `camexp.com` and `phonesum.com`.

Main Contributions The contributions of this paper are:

- A framework for automatic synthesis of web-crawlers. The main idea is to use hand-crafted crawlers for a number of websites as the basis for crawling other sites that contain the same kind of information.
- A new cross-supervised crawler synthesis algorithm that extrapolates crawling schemes from one web-site to another. The algorithm handles pages with multiple items and synthesizes crawlers using only positive examples.
- An implementation and evaluation of our approach, automatically synthesizing 30 crawlers for websites from nine different categories: books, TVs, conferences, universities, cameras, phones, movies, songs and hotels. The crawlers that we synthesize are real crawlers that were used to crawl more than $12,000$ webpages over all categories.

2.2 Overview

2.2.1 Motivating Example

Consider a price comparison service for books, which crawls book seller websites and provides a list of sellers and corresponding prices for each book. Examples of such book seller sites include `barnesandnoble.com` (B&N), `blackwell.co.uk` (BLACKWELL) and `abebooks.com` (ABE). Each of these sites lists a wide collection of books, typically presented in template generated webpages feeding from a database. Since these pages are template generated, they present *structured information* for each book in a format that is repeated across books. By recognizing this repetitive structure for a given site, one can synthesize a data extraction query and use it to automatically extract the entire book collection.

While the format within a single site is typically stable, the formats *between sites* differ considerably. Fig. 3.1 shows a small and simplified fragment of the page structure on B&N and BLACKWELL in HTML. Fig. 2.3 shows part of the tree representation of the corresponding sites (as well as of ABE), where $D_1, \ldots, D_4$ denote different pages. Due to the differences in structure, the data extraction query can differ dramatically. For example, in BLACKWELL and ABE, each of the pages (D_2, D_3, D_4) presents a single book, whereas in B&N the page D_1 shows a list of several books.

The goal of this work is to automatically synthesize crawlers for new sites based on some existing hand-crafted crawler(s). For example, given a crawler for the BLACKWELL site, our technique synthesizes a crawler for B&N website. The synthesized crawler is depicted in Fig. 2.2. We show that this can be done despite the significant differences between the sites BLACKWELL, and B&N, in terms of HTML structure. We note that the examples that we present in this section are abbreviated and simplified. For example, the real DOM tree for the B&N page we show here contains around $1,000$ nodes. The structure of the full trees, and the XPaths required for processing them are more involved than what is shown here.

```
barnesandnoble.com
<ol class="result-set box">
   <li class="result box">..
   <div class="details below-axis" >
    <a href="..." data-bntrack="Title_9781628718980"
      class="title" >
    THROUGH THE LOOKING GLASS</a>
    <a href=".."
    data-bntrack="Contributor_9781628718980"
      class="contributor" >
    David Winston Busch</a>
    ...
    <div class="price-format">
        <a href="..." data-bntrack="Paperback_Format">
                <span class="format">Paperback</span>
                <span class="price">$9.91</span>
        </a>
   </div>
   </div>
   ...</li>
   <li class="result box">..
   <div class="details below-axis" >
    <a href="..." data-bntrack="Title_9780071633604"
      class="title">
    Alice's Adventures in Wonderland</a>
    <a href=".." data-bntrack="Contributor_9780071633604"
    class="contributor" >Lewis Carroll</a>
    ...
    <div class="price-format">
      <a href="..." data-bntrack="Paperback_Format">
            <span class="format">Paperback</span>
            <span class="price">$6.49</span>
          </a>
   </div>
   </div>
   ...</li>
   </ol>
```

```
blackwell.com
<div id="product-biblio">
  <h1>Through the looking glass</h1>
  <a class="link_type1" href="/jsp/a/Lewis_Carroll">
    David Winston Busch
  </a>
  <div class="price-info" align="center">
    <span class="price">
    £8.99</span>
  </div>
</div>
```

Figure 2.1: Fragments of webpages with the similar attribute values for a book on two different book shopping sites.

2.2.2 Cross-Supervised Learning of Crawling Schemes

Our main observation is that despite the significant differences in the concrete layout of websites, the pages of websites that exhibit the same product category often share the same *logical structure*; they present similar attributes for each product. For example, each of the pages of Fig. 3.1 presents the same important attributes about the book, including its *title*, *author name* and *price*. Moreover, there is a large number of shared products between these websites. The book *"Through the looking glass"* is one such example for B&N and BLACKWELL.

```
class MySpider(CrawlSpider):
 name = "barnesandnoble"
 allowed_domains = ["www.barnesandnoble.com"]
 start_urls = [
    ((*@\textbf{"http://www.barnesandnoble.com/s/java-programming}\\
                 \textbf{?store=allproducts\&keyword=java+programming"}@*))
    ]

 rules = (
  Rule(LinkExtractor(
       allow=((*@\textbf{"/s/.*"}@*)),callback="parse_item", follow=True
  ),
 )

 def parse_item(self, response):
  sel = Selector(response)
  rows = sel.XPath((*@\textbf{'//body/div/.../div[@class="details below-axis"]'}@*))
  for r in rows:
   item = BooksItem()
   item['title'] = r.XPath(
    (*@ \textbf{'//a[@class="title"]'} @*)
   ).extract()
   item['author'] = r.XPath(
    (*@ \textbf{'//a[@class="contributor"]'} @*)
   ).extract()
   item['price'] = r.XPath(
    (*@ \textbf{'//div[@class="price-format"]/a/span[@class="price"]'} @*)
   ).extract()
   yield item
```

Figure 2.2: Crawler for java books from Barnes&Noble.

Our technique exploits data overlaps across sites in order to learn the concrete structure of a new website s based on other websites. Specifically, we identify in s concrete data extracted from other sites and as such learn the structure in which this data is represented in s. We then use multiple examples of the structure in which the data appears in s in order to generalize and get an extraction query for s. This enables our algorithm to extrapolate a crawler for s.

We do not require a precise match of data across sites, as our technique also handles noisy data. (For example, prices do not have to be identical; any number can be a match.)

Crawling schemes A crawler, such as the one of Fig. 2.2, contains some boilerplate code defining the crawler class and its operations. However, the essence of the crawler is its *crawling scheme*. For example, in Fig. 2.2 the crawling scheme is highlighted in boldface.

A crawling scheme is defined with respect to a set of semantic groups, called *attributes*, which define the types of data to be extracted. In the books example, the attributes are: *book title*, *author* and *price*.

Given a set of attributes, a crawling scheme consists of the following two components: (i) A data extraction query that defines how to obtain values of the attributes for each item listed on the site. (ii) A starting point URL and a URL filtering pattern which let the crawler locate "relevant" pages and filter out irrelevant pages without downloading and processing them.

Our crawlers use XPath as a query language for data extraction. XPath is a query language for selecting nodes from an XML document which is based on the tree representation of the XML document, and provides the ability to navigate around the tree, selecting nodes by describing their path from the document tree root node. For example, Fig. 2.4 and Fig. 2.5 show the crawling schemes for crawling books from BLACKWELL and B&N respectively, where the data extraction query is expressed using XPaths.

Two-level data extraction schemes We assume that the data extraction query has two levels: The

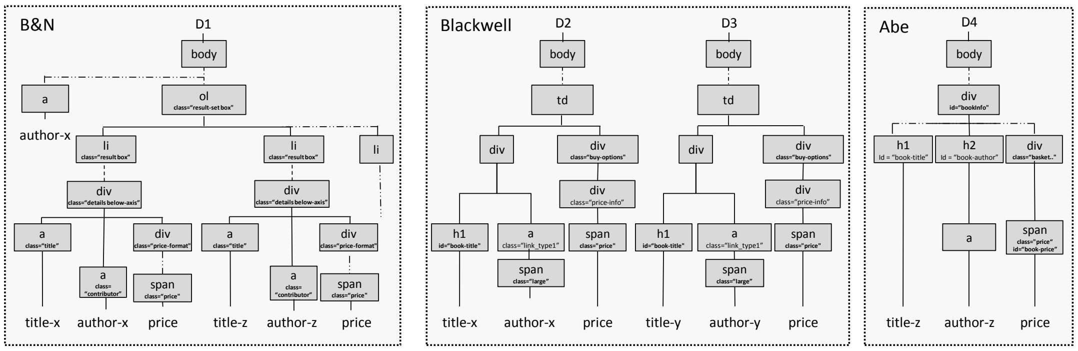


Figure 2.3: Example DOM trees

Container:	`//body/div[@class="content␣maincore--shop"]`
	`/table[@class="main-page"]/tr/`
	`td[@class="two-col-right"]/table/tr/td`
Title:	`//div/h1/`
Author:	`//div/a[@class="link␣type1"]`
Price:	`//div[@id="buy-options"]/div/span`
URL Pattern:	`.*jsp/id/.*`

Figure 2.4: Crawling scheme for BLACKWELL.

Container:	`//body/div/div/section/div/ol["result-set box"]`
	`/li[@class="result box"]/div`
	`/div[@class="details below-axis"]`
Title:	`//a[@class="title"]`
Author:	`//a[@class="contributor"]`
Price:	`//div[@class="price-format"]`
	`/a/span[@class="price"]`
URL Pattern:	`/s/.*`

Figure 2.5: Crawling scheme for B&N.

first level query is an XPath describing an item container. Intuitively, a container is a sub-tree that contains all the attribute values we would like to extract (defined more formally in Sec. 2.5.) For example, in Fig. 2.4, the XPath `//body/div[@class="content␣maincore--shop"]...` describes a container of book attributes on BLACKWELL pages.

The second level queries contain an extraction XPath for values of each individual attribute. These XPaths are relative to the root of the container. For example, `//div/h1/` in Fig. 2.4 is used to pick the node that has type `h1` (heading 1), containing the book title.

Iterative synthesis of crawling schemes Our approach considers a set of websites, and a set of attributes to extract. To bootstrap the synthesis process, the user is required to provide the set of websites for which crawler synthesis is desired, as well as a crawling scheme for at least one of these sites. Alternatively, the user can provide multiple partial crawling schemes for different sites, that together cover all the different item attributes.

The synthesis process starts by invoking the provided extraction scheme(s) on the corresponding sites to obtain an initial set of values for each one of the attributes. These values are then used to locate nodes that contain attribute values in the document trees of webpages of new sites. The nodes that contain attribute values reveal the structure of pages of the corresponding websites. In particular, smallest subtrees that exhibit all the attributes amount to containers. This allows for synthesis of data extraction schemes for new websites. The newly learned extraction schemes are used to extract more values and add them to the set of values of each attribute, possibly allowing for additional websites to be handled. This process is repeated until complete extraction schemes are obtained for all websites, or until no additional values are extracted.

In our example, the algorithm starts with the data extraction scheme for BLACKWELL (see Fig. 2.4), provided by a user. It extracts from D_2 `author-x`, `title-x`, and `price` as values of the book title, author, and price attributes, respectively (see Fig. 2.3). These values are identified in D_1 (B&N) within the subtree of the left most node represented by

```
  //body/.../ol["result-set box"]
    /li[@class="result box"]/...
    /div[@class="details below-axis"],
```

which then points to the latter node as a possible container. Additional values taken from D_3 and other pages in BLACKWELL identify additional nodes in the B&N tree as attribute and container nodes. Note that `author-x` is also found in another subtree in D_1. However, there are no instances of the remaining attributes in that subtree; Therefore, the subtree is not considered a container and the corresponding node is treated as noise.

By identifying the commonality between the identified containers and between nodes of the same attribute, a data extraction scheme for B&N is synthesized (see below). In the next iteration, the new data scheme is used to extract from B&N the values `author-z`, `title-z` and `price` as additional values for book title, author, and price respectively (that did not exist in BLACKWELL). The new values are located in ABE (see D_4 in Fig. 2.3), allowing to learn an extraction scheme for ABE as well.

XPath synthesis for two-level queries Our approach synthesizes a two level extraction scheme for each website from a set of attribute nodes and candidate containers identified in its webpages. The two-level query structure is reflected also in the synthesis process of the extraction scheme. Technically, we use a two-phase approach to synthesize the extraction scheme. In each site, we first generate an XPath query for the containers. We then filter the attribute nodes keeping only those reachable from containers that agree with the container XPath, and generate XPaths for their extraction relatively to the container nodes.

To generate an XPath query for a set of nodes (e.g., for the set of containers), we consider the concrete XPath of each node—this is an XPath that extracts exactly this node. We unify these concrete XPaths by a greedy algorithm that aims to find the most concrete (most strict) XPath query that agrees with a majority of the concrete XPaths. Keeping the unified XPath as concrete as possible prevents the addition of noise to the extraction scheme.

The generated XPaths for B&N are depicted in Fig. 2.5. In this example, unification is trivial since the XPaths are identical. However, if for example each of the container nodes labeled `div` in D_1 had different `id`'s, the `id` feature would have been removed during unification. Note that even if the subtree that contains the noisy instance of `author-x` in D_1 had been identified as a candidate container (e.g., if it had contained values of the other attributes), it would have been discarded during the unification.

URL pattern synthesis In order to synthesize a URL pattern for the crawling scheme of a new site, we extend the iterative technique used for synthesis of data extraction schemes; In each iteration of the algorithm, for each website we identify a set of pages of interest as pages that contain attribute data. We filter these pages in accordance with the filtering of container and attribute nodes. We then unify the URLs of remaining pages similarly to XPath unification.

Fig. 2.5 depicts the URL pattern generated by our approach for B&N. This pattern identifies webpages in B&N that present a list of books—these are the pages whose structure conforms with the synthesized extraction scheme. Note that B&N also presents the same books in a separate page each, but such pages require a different crawling scheme.

2.3 Preliminaries

In this section we define some terms that will later be used to describe our approach.

2.3.1 Logical Structure of Webpages

Each webpage implements some *logical structure*. Following [HAF+10], we use *relations* as a logical description of data which is independent of its concrete representation. A relational specification is a set of relations, where each relation is defined by a set of column names and a set of values for the columns. A *tuple* $t = \langle c_1 : d_1, c_2 : d_2, \ldots\rangle$ maps a set of columns $\{c_1, c_2, \ldots\}$ to values. A *relation* r is a set of tuples $\{t_1, t_2, \ldots\}$ such that the columns of every $t, t' \in r$ are the same.

For example, B&N, BLACKWELL and ABE described in Section 2.2 implement a relational description of a list of books, where each book has a title, an author and a price. Then "book title", "author" and "price" are columns, and the set of books is modeled as a relation with these columns, where each tuple is a book item.

Data items, attributes and instances We refer to each tuple of a relation r as a *data item*. The columns of a relation r are called *attributes*, denoted *Att*. Each attribute defines a class of data sharing semantic similarities, such as meaning and/or extraction scheme. The value of attribute $a \in \mathit{Att}$ in some tuple of r is also called an *instance* of a. The set of all values of all attributes is denoted V. Each attribute a is associated with an *equivalence relation* $\equiv_a$ that determines if two values are equivalent or not as instances of a. (The notion of "equivalence" may differ between different attributes.) By default (if not specified by the user) we use the bag of words representation of each value d, denoted $W(d)$, and use Jaccard similarity function [Jac], $J(d_1, d_2)$, with a threshold of 0.5 as an equivalence indicator between values d_1 and d_2:

$$d_1 \equiv_a d_2 \text{ iff } J(d_1, d_2) > 0.5 \text{ where } J(d_1, d_2) = \frac{|W(d_1) \cap W(d_2)|}{|W(d_1) \cup W(d_2)|}.$$

2.3.2 Concrete Layout of Webpages

Technically, webpages are documents with structured data, such as XML or HTML documents. The concrete layout of the webpage implements its logical structure, where attribute instances are presented as nodes in the DOM tree.

XML documents as DOM trees A well formed XML document, describing a webpage of some website, can be represented by a DOM tree. A DOM tree is a labeled ordered tree with a set of nodes N and a labeling function that labels each node with a set of node features (not to be confused with data attributes), where some of the features might be unspecified. Common node features include `tag`, `class` and `id`.

For example, Fig. 2.3 depicts part of the tree representation of pages of B&N, BLACKWELL and ABE. A node labeled by `a`, `class=title` is a node whose `tag` is `a`, `class` is `title`, and `id` is unspecified.

Node descriptors A *node descriptor* is an expression x in some language defining a set of nodes in the DOM tree. We use *Expr* to denote the set of node descriptors. For a node descriptor $x \in$ *Expr* and a webpage p, we define $[\![x]\!]_p$ to be the set of nodes described by x from p. When p is clear from the context, we omit it from the notation. A node descriptor is *concrete* if it represents exactly one node. We sometimes also refer to node descriptors as *extraction schemes*. In this work, we use XPath as a specification language for node descriptors.

2.3.3 XPath as a Data Extraction Language

XPath [CD+99] is a query language for traversing XML documents. XPath expressions (XPaths in short) are used to select nodes from the DOM tree representation of an XML document. An XPath expression is a sequence of instructions, $x = x_1 \dots x_k$. Each instruction x_i defines how to obtain the next set of nodes given the set of nodes selected by the prefix $x_1 \dots x_{i-1}$, where the empty sequence selects the root node only. Roughly speaking, each instruction x_i consists of (i) *axis* defining where to look relatively to the current nodes: at children ("`/`"), descendants ("`//`"), parent, siblings, (ii) *node filters* describing which `tag` to look for (these can be "all", "text", "comment", etc.), and (iii) *predicates* that can restrict the selected nodes further, for example by referring to values of additional node features (e.g. `class`) that should be matched.

For example, the XPath `//div/*/a[@class="link_type1"]` selects all nodes that follow a sequence of nodes that can start anywhere in the DOM tree, and has to consist of a node with `tag=div` followed by some node whose features are unspecified and is followed by a node with `tag=a` and `class=link_type1`.

2.4 The Crawler Synthesis Problem

In this section we formulate the crawler synthesis problem. A crawler for a website can be divided into two parts: a *page crawler*, and a *data extractor*. The page crawler is responsible for grabbing the pages of the site that contain relevant information. The data extractor is responsible for extracting data of interest from each page.

Logical structure of interest Our work considers websites whose data-containing webpages share the following logical structure: each webpage describes one main relation, denoted *data*. As such, data items are tuples of the *data* relation. Further, the set *Att* of attributes consists of the columns of the *data* relation.

Note that different concrete layouts can implement this simple logical structure. For example, if we consider a webpage that exhibits a list of books, then the concrete layout can first group books by author, and for each author list the books, or it can avoid the partition based on authors. Further, some websites will present each book in a separate webpage, whereas others will list several books in the same page. Even for websites that are structured similarly by the former parameters, the mapping of attribute instances to nodes in the DOM tree can vary significantly.

Page crawlers A page crawler for a website s is given by a URL pattern, denoted $U(s)$, which identifies the set of webpages of interest. These are the webpages of the website that contain data of the relevant kind. We denote by $P(s)$ the set of webpages whose URL matches $U(s)$.

Data Extractors Recall that we consider webpages where instances of different attributes are grouped into tuples of some relation, denoted *data*. We are guided by the observation that data in such webpages is typically stored in subtrees, where each subtree contains instances of all attributes for some data item (i.e., tuple of the *data* relation). We refer to the roots of such subtrees as *containers*:

Containers: A node in the DOM tree whose subtree contains all the entries of a single data item (i.e., a single tuple of *data*) is called a *container*. Note that any ancestor of a container is also a container. We therefore also define the notion of a *best* container to be a container such that none of its predecessors is a container. Depending on the concrete layout of the webpage, a best container might correspond to an element in a list or in another data structure. It might also be the root of a webpage, if each webpage presents only one data item.

For example, in the tree D_1 depicted in Fig. 2.3, both of the nodes selected by `//body/.../div[@class="details below-axis"]` are containers, and as such so are their ancestors, including the root. However, the latter are not best containers since they include strict subtrees that are also containers.

Attribute nodes: A node in the DOM tree that holds an instance of an attribute $a \in \mathit{Att}$ is called an a*-attribute node*, or simply an *attribute node* when a is clear from the context or does not matter.

Data extractors: A data extractor for the relation *data* over columns *Att* in some website s can be described by a pair $(container, f)$ where $container \in \mathit{Expr}$ is a node descriptor representing containers, and $f : \mathit{Att} \hookrightarrow \mathit{Expr}$ is a possibly partial function that maps each attribute name to a node descriptor, with the meaning that this descriptor represents the attribute nodes relatively to the container node, i.e., the attribute descriptor considers the container node as the root. The data extractor is *partial* if f is partial. If $container$ is empty, it is interpreted as a node descriptor that extracts the root of the page. If $container$ is empty and f is undefined for every attribute, we say that the data extractor is *empty*.

Examples of data extractors appear in Fig. 2.5 and Fig. 2.4.

Crawler synthesis The *crawler synthesis problem* w.r.t. a set *Att* of attributes is defined as follows. Its input is a set S of websites, where each website $s \in S$ is associated with a data extractor, denoted $E(s)$, over *Att*. $E(s)$ might be partial or even empty. The desired output is a page crawler, along with a complete data extractor for every $s \in S$.

2.5 Data Extractor Synthesis

In this section we focus on synthesizing data extractors, as a first step towards synthesizing crawlers. We temporarily assume that the page crawler is given, i.e., for each website we have the set of webpages of interest, and present our approach for synthesizing data extractors. We will remove this assumption later, and also address synthesis of the page crawler, using similar techniques.

The input to the data extractor synthesis is therefore a set S of websites, where each website $s \in S$ is associated with a set of webpages, denoted $P(s)$, and with a data extractor, denoted $E(s)$, which might be partial or even empty. The goal is to synthesize a complete data extractor for every $s \in S$. The main challenge in synthesizing a data extractor is identifying the mapping between the logical structure of a webpage, and its concrete layout as a DOM tree. The key to understanding this mapping amounts to identifying the container nodes in the DOM tree that contain all the attributes of a single data item (tuple). Once this mapping is learnt, the next step is to capture it by synthesizing extraction schemes in the form of XPaths.

The data extractor synthesis algorithm is first described using the generic notion of node descriptors. In Section 2.5.2 we then instantiate it for the case where node descriptors are provided by XPaths.

Before we describe our algorithm, we review its main ingredients. In the following, we use $N(p)$ to denote the set of nodes in the DOM tree of a webpage $p \in P(s)$.

Knowledge base of data across websites Our synthesizer maintains a knowledge base O : *Att* $\to 2^V$ which consists of a set of observed instances for each attribute $a \in$ *Att*. These are instances collected across different websites from S. They enable the synthesizer to locate potential a-attribute nodes in webpages for which the data extractor of a is unspecified.

Data to node mapping per website In addition to the global knowledge base, for each website $s \in S$ our synthesizer maintains: (i) a set $N^{cont}(p) \subseteq N(p)$ of (candidate) container nodes for each webpage $p \in P(s)$, and (ii) a set $N^a(p) \subseteq N(p)$ of (candidate) attribute nodes for each webpage $p \in P(s)$ and attribute $a \in$ *Att*.

Deriving extraction schemes per website The synthesis algorithm iteratively updates the container and attribute node sets for each webpage in $P(s)$, and attempts to generate a data extractor $E(s)$: *Expr* $\times$ (*Att* $\hookrightarrow$ *Expr*) for s by generating node descriptors for the set of containers, and for each of the attributes. The extraction scheme is shared by all webpages of the website. The updates of the sets and the attempts to generate node descriptors from the sets are interleaved, as one can affect the other; on the one hand node descriptors are generated in an attempt to represent the sets; on the other hand, once descriptors are generated, elements of the sets that do not conform to them are removed.

While attribute instances are used to identify attribute nodes across different websites, the synthesis of node descriptors is performed for each website separately and independently of others (while considering all of the webpages associated with the website).

2.5.1 Algorithm

Algorithm 1 presents our data extractor synthesis algorithm. The algorithm is iterative, where each iteration consists of two phases:

Phase 1: Data extraction for knowledge base extension. Initially, the sets $O(a)$ of instances of all attributes $a \in \mathit{Att}$ are empty. In each iteration, we use yet un-crawled extraction schemes to extract attribute nodes in all webpages of all websites and extend the sets $O(a)$ for every attribute a based on the content of the extracted nodes. At the first iteration, input extraction schemes are used. In later iterations, we use newly learnt extraction schemes, generated in phase 2 of the previous iteration.

Phase 2: Synthesis of data extractors. For every website $s \in S$ for which the extraction scheme is not yet satisfactory, we attempt to generate an extraction scheme by performing the following steps:

(1) *Locating attribute nodes per page:* We traverse all webpages $p \in P(s)$ and for each attribute a we use the instances $O(a)$ collected in phase 1 (from this iteration and previous ones) to identify potential a-attribute nodes in p. Technically, for every $p \in P(s)$ we iterate on all $n \in N(p)$ and use the (default or user-specified) equivalence relation $\equiv_a$ to decide whether n contains data that matches the attribute instances in $O(a)$. If so, n is added to $N^a(p)$.

(2) *Locating container nodes per page:* In every webpage $p \in P(s)$ we locate potential container nodes, and collect them in $N^{cont}(p)$. A container is expected to contain instances of all attributes *Att*. However, since our knowledge of the attribute instances is incomplete, we need to also consider subsets of *Att*. In each webpage, we define the "best" set of attributes to be the set of all attributes whose instances appear in it. Potential containers are nodes whose subtree contains attribute nodes of the "best" set of attributes, and no strict subtree contains nodes of the same set of attributes. The latter ensures that the container is best. Technically, for every node $n \in N(p)$ we compute the set of reachable attributes $a \in \mathit{Att}$ such that an a-attribute node in $N^a(p)$ is reachable from n. Nodes n whose set is best and no other node reachable from n has the same set of reachable attributes are collected in $N^{cont}(p)$. For each container node $n_c \in N^{cont}(p)$ we also maintain its *support* - the number of attribute nodes reachable from it.

(3) *Generating container descriptor:* We consider the concrete node descriptor of every container node $n_c \in N^{cont}(p)$ in every webpage $p \in P(s)$. We unify the concrete node descriptors across all webpages into a single node descriptor, and use it to update $E(s)$, relying on the observation that containers are typically elements of some data structure and are therefore accessed similarly.

(4) *Filtering attribute nodes based on container descriptor:* We filter the sets $N^{cont}(p)$ of containers in all webpages to keep only containers that match the unified node descriptor, and accordingly filter the sets $N^a(p)$ of attribute nodes in all webpages to contain only nodes that are reachable from the filtered sets of containers. This step enables us to automatically distinguish the nodes we are interested in from others that accidentally contain attribute instances, without any a-priori knowledge.

(5) *Generating attribute descriptors:* For each attribute $a \in \mathit{Att}$, we consider the concrete

Algorithm 1: Data Extractor Synthesizer

```
Input: set of attributes $Att$
Input: set of websites $S$
Input: a map $E : S \to (Expr \times (Att \hookrightarrow Expr))$ mapping a website $s$ to a data extractor $E(s)$
       which consists of a (possibly empty) container descriptor as well as a (possibly partial)
       mapping of attributes to node descriptors
$O = []$;
while there is change in $O$ or $E$ do
    /* Data extraction phase */
    foreach $s \in S$ s.t. $E(s)$ is uncrawled do
        $O = O \cup$ ExtractInstances $(Att, P(s), E(s), O)$ ;
    /* Synthesis phase */
    foreach $s \in S$ s.t. $E(s)$ is incomplete do
        /* Locate attribute nodes */
        foreach $p \in P(s)$ do
            foreach $a \in Att$ do
                $N^a(p) =$ FindAttNodes $(N(p), a, O(a))$ ;
        /* Locate container nodes */
        foreach $p \in P(s)$ do
            $bestAttSet = \{a \in Att \mid N^a(p) \neq \emptyset\}$ ;
            foreach $n \in N(p)$ do
                $reachAtt[p][n] = \{a \in Att \mid \exists n' \in reach(n) : n' \in N^a(p)\}$ ;
                $support[p][n] = \#\{n' \in reach(n) \mid \exists a \in Att : n' \in N^a(p)\}$ ;
            $N^{cont}(p) = candidates = \{n \in N(p) \mid reachAtt[n] = bestAttSet\}$ ;
            foreach $n \in candidates$ do
                foreach $n' \in children(n)$ do
                    if $n' \in candidates$ then
                        $N^{cont}(p) = N^{cont}(p) \setminus \{n\}$ ;
                        break ;
        /* Generate container descriptor */
        $Exprs = \{(relativeExpr(p, emptyExpr, n), support[p][n]) \mid p \in P(s), n \in$
          $N^{cont}(p)\}$ ;
        $containerExpr =$ UnifyExpr $(Exprs)$ ;
        FilterAttributeNodes() ;
        /* Generate attribute descriptors */
        foreach $a \in Att$ do
            $Exprs = \{(relativeExpr(p, containerExpr, n), 1) \mid p \in P(s), n \in N^a(p)\}$ ;
            $attExpr[a] =$ UnifyExpr $(Exprs)$ ;
        $E(s) = (containerExpr, attExpr)$ ;
return $E$ ;
```

node descriptors of all the nodes in the filtered sets $N^a(p)$ of all webpages $p \in P(s)$, where the concrete node descriptor of n is computed relatively to the container node whose subtree contains n. For each attribute a, we find a unified node descriptor for these concrete node descriptors, and use it to update $E(s)$. Again, we use the observation that containers are structured similarly and therefore attribute data within them is accessed similarly.

Remark. For a successful application of our algorithm, at least one extraction scheme should be provided for every attribute. Our approach is also applicable if a user provides a set of annotated webpages instead of a set of initial extraction expressions. □

Section 2.2 describes a running example of our algorithm.

Node descriptor unification Node descriptors for the container and attributes are generated by unifying concrete node descriptors of the nodes in $N^{cont}(p)$ and $N^a(p)$ respectively. Roughly speaking, the purpose of the unification is to derive a node descriptor that is general enough to describe as many of the concrete node descriptors as possible, but also as concrete as possible in order to introduce as little noise as possible. "Concreteness" of a node descriptor x is measured by an *abstraction score*, denoted abs(x). The node descriptor unification algorithm is parametric in the abstraction score. In Section 2.5.2, we provide a definition of this score when the node descriptors are given by XPaths.

Definition 2.5.1. For a set X of concrete node descriptors and a weight function *support* that associates each $x \in X$ with its support, the *unification problem* aims to find a node descriptor x_g, s.t.:

1. $\frac{support(\{x \in X | [\![x]\!] \subseteq [\![x_g]\!]\})}{support(X)} > \delta$, i.e., x_g captures at least δ of the total support of the node descriptors in X.
2. abs(x_g) is minimal.

In container descriptor unification (step 3), the given node descriptors represent container nodes. The support of each descriptor represents the number of attribute nodes reachable from the container. In attribute descriptors unification (step 5), the given descriptors represent attribute nodes for some attribute, all of which are reachable from a set of containers of interest. The attribute node descriptors are relative to the container nodes.

2.5.2 Implementation using sequential XPaths

In order to complete the description of our data extractor synthesizer, we describe how the ingredients of Algorithm 1 are implemented when node descriptors are given by XPaths. Specifically, our approach uses sequential XPaths:

Sequential XPaths A *path* π in the DOM tree is a sequence of nodes $n_1, \ldots, n_k$, where for every $1 \leq i < k$, there is an edge from n_i to n_{i+1}. Such a path can naturally be encoded using an XPath $\texttt{XS}(\pi) = x_1 \ldots x_k$ where each x_i starts with "/". x_1 may start with "//" rather than "/" if π does not necessarily start at the root of the tree. Further, each x_i uses node filters and predicates to describe the features of n_i. Therefore, x_i can be described via equalities $f_1 = v_1, \ldots, f_m = v_m$, such that $f_j \in F$, where F is the set of node features used. We consider $F = \{\texttt{tag}, \texttt{class}, \texttt{id}\}$ for simplicity, but our approach is not limited to these features. A feature might be unspecified for n_i, in which case no corresponding equality will be included in x_i.

For example, let π be the left most path in D_2 (Fig. 2.3). Then $\texttt{XS}(\pi) = \texttt{//body/}\ldots\texttt{/td/div/h1}$. $\texttt{XS}(\pi)$ can also be described as a sequence $\langle\texttt{tag=body}\rangle \ldots \langle\texttt{tag=td}\rangle\langle\texttt{tag=div}\rangle\langle\texttt{tag=h1}\rangle$.

We refer to XPaths of the above form as *sequential*. The XPaths that our approach generates as node descriptors are all sequential.

Concrete XPaths Each node n in the DOM tree can be uniquely described by the unique path, denoted π_n, leading from the root to n. The XPath $\texttt{XS}(\pi_n)$ is a sequential XPath such that $[\![\texttt{XS}(\pi_n)]\!] \supseteq \{n\}$, and $[\![\texttt{XS}(\pi_n)]\!]$ is minimal (i.e., every other sequential XPath that also describes

n, describes a superset of $[\![\texttt{XS}(\pi_n)]\!]$). We therefore refer to $\texttt{XS}(\pi_n)$ as the *concrete* XPath of n, denoted $\texttt{XS}(n)$ with abuse of notation. (If we include in F the position of a node among its siblings as an additional node feature, and encode it by an XPath instruction using sibling predicates then we will have $[\![\texttt{XS}(\pi_n)]\!] = \{n\}$).

Agreement of sequential XPaths We observe that for sequential XPaths, checking if a node n matches a node descriptor x_g (i.e. $n \in [\![x_g]\!]$) can be done by checking if the concrete XPath $\texttt{XS}(n)$ *agrees* with the XPath x_g, where agreement is defined as follows.

Definition 2.5.2. Let $x = x_1 \dots x_k$ and $x_g = x_1^g \dots x_m^g$ be sequential XPaths. The instruction x_i *agrees* with instruction x_i^g if whenever some feature is specified in x_i, it either has the same value in x_i^g or it is unspecified in x_i^g. The XPath x *agrees* with the XPath x_g if $m \leq k$, and for every $i \leq m$, x_i agrees with x_i^g.

For example, `//body/.../td/div[id=name1]/h1` agrees with both `//body/.../td/div/h1`, and `//body/.../td/div`.

Node descriptor unification via XPath unification We now describe our solution to the node descriptor unification problem in the setting of sequential XPaths. We first define the abstraction score:

Abstraction score For a sequential XPath instruction x_i we define *spec*(x_i) to be the subset of features whose value is specified in x_i, and *unspec*$(x_i) = F \setminus$ *spec*(x_i) is the set of unspecified features in x_i. We define the abstraction score of x_i to be the number of features in *unspec*(x_i), that is, abs$(x_i) = |$*unspec*$(x_i)|$.

For a sequential XPath $x = x_1 \dots x_k$, we define abs(x) to be the sum of abs(x_i).

Greedy algorithm for unification Algorithm 2 presents our unification algorithm. We use the observation that for sequential XPaths, the condition $[\![x]\!] \subseteq [\![x_g]\!]$ that appears in item 1 of the unification problem (see Definition 2.5.1) can be reduced to checking if the XPath x *agrees* with the XPath x_g.

Let X be a weighted set of sequential XPaths, with a weight function *support* that associates each XPath in X with its support. Let *TS* = *support*(X) denote the total support of XPaths in X. The unification algorithm selects k to be the length of the longest XPath in X. It then constructs a unified XPath $x_g = x_1^g, \dots, x_m^g$ top down, from $i = 1$ to k (possibly stopping at $i = m < k$). Intuitively, in each step the algorithm tries to select the most "concrete" instruction whose support is high enough. Note that there is a tradeoff between the high-support requirement and the high-concreteness requirement. We use the threshold as a way to balance these measures.

At iteration i of the algorithm, X^{i-1} is the restriction of X to the XPaths whose prefix agrees with the prefix $x_1^g, \dots, x_{i-1}^g$ of x_g computed so far (Initially, $X^0 = X$). We inspect the i'th instructions of all XPaths in X^{i-1}. The corresponding set of instructions is denoted by $I^i = \{x_i \mid x \in X^{i-1}\}$. The support of an instruction x_B w.r.t. I^i is *support*$(\{x \in X^{i-1} \mid x_i$ agrees with $x_B\})$.

To select the most "concrete" instruction whose support is high enough, we consider a predefined order on sets of feature-value pairs, where sets that are considered more "concrete" (i.e., more "specified") precede sets considered more "abstract". Technically, we consider only feature-value sets where each feature has a unique value. The order on such sets used in the algorithm is defined such that if $|B_1| > |B_2|$ then B_1 precedes B_2. In particular, we make sure that sets where all features are specified are first in that order.

For every set B of feature-value pairs, ordered by the predefined order, we consider the instruction x_B that is specified exactly on the features in B, as defined by B. If its support exceeds δ, we set x_i^g to x_B and X^i to $\{x \in X^{i-1} \mid x_i \text{ agrees with } x_B\}$. Otherwise, x_B is not yet satisfactory and the search continues with the next B. There is always a B for which the support of the x_B exceeds the threshold, for instance, the last set B is always the empty set with $x_B = \texttt{/*}$, which agrees with all the concrete XPaths in X^{i-1} .

If at some iteration $I^i = \emptyset$, i.e. the XPaths in X^{i-1} are all of length $< i$ and therefore there is no "next" instruction to discover, the algorithm terminates. Otherwise, it terminates when $i = k$.

Example 1. Given the following concrete XPaths as an input:

$$cx_1 = \texttt{/div[class=``title'']/span/a[id=``t1'']}$$
$$cx_2 = \texttt{/div[class=``title'']/span/a[id=``t2'']}$$
$$cx_3 = \texttt{/div[class=``note'']/span/a[id=``n1'']}$$

The unification starts with $X^0 = \{cx_1, cx_2, cx_3\}$, and $i = 1$. To select x_1^g, recall that the algorithm first considers the most specific feature-value sets (in order to find the most specific instruction). In our example it starts from $B_1 = \{\texttt{tag=div}, \texttt{class=note}\}$ for which $x_{B_1} =$ `/div[class=``note'']`. However, $cx_3$ is the only XPath in $X^0$ which agrees with $x_{B_1}$. Therefore it has support of $1/3$. We use a threshold of $\delta = 1/2$. Thus, the support of $x_{B_1}$ is insufficient. The algorithm skips to the next option, obtaining $x_{B_2} =$ `/div[class=``title'']`. This instruction is as specific as $x_{B_1}$ and has a sufficient support of 2/3 (it agrees with $cx_1$ and $cx_2$). Therefore, for $i = 1$, the algorithm selects $x_1^g = x_{B_2}$ and $X^1 = \{cx_1, cx_2\}$. For $i = 2$, the algorithm selects $x_2^g =$ `/span` as the most specific instruction, which also has support of 2/2 (both $cx_1$ and $cx_3$ from $X^1$ agree with it). For $i = 3$, the algorithm selects $x_3^g =$ `/a` as none of the more specific instructions (`/a[id=``t1'']` or `/a[id=``t2'']`) has a support greater than $\delta = 1/2$. The resulting unified XPath is $x =$ `/div[class="title"]/span/a`.

2.6 Crawler Synthesis

In this section we complete the description of our crawler synthesizer. To do so, we describe the synthesis of a page crawler for each website s. Recall that a page crawler corresponds to a URL pattern $U(s)$ which defines the webpages of interest. The synthesis of a page crawler is intertwined with the data extractor synthesis, and uses similar unification techniques to generate the URL pattern.

Initialization We assume that each website $s \in S$ is given by a "main" webpage $p_{main}(s)$. Initially, the set $P(s)$ of webpages of s is the set of all webpages obtained by following links in $p_{main}(s)$ and recursively following links in the resulting pages, where the traversed links are

Algorithm 2: Top-Down XPath Unification

```
Input: set $X$ of sequential XPaths
Input: support function $support : X \to \mathbb{N}$
Input: threshold $\delta$
$TS = support(X)$
$k = \max_{x \in X} |x|$
$X^0 = X$
foreach $i = 1, \dots, k$ do
    $I^i = \{x_i \mid x \in X^{i-1}\}$
    if $I^i = \emptyset$ then
        $i = i - 1$
        break
    foreach $B \subseteq F$ in decreasing order of $|B|$ do
        $support_B = \text{FindSupport}(x_B, X^{i-1}, i, support)$
        if $support_B > \delta \cdot TS$ then
            $x_i^g = x_B$
            $X^i = \{x \in X^{i-1} \mid x_i \text{ agrees with } x_B\}$
            break
return $x_1^g, \dots, x_i^g$
```

selected based on some heuristic function which determines which links are more likely to lead to relevant pages.

Iterations We apply the data extractor synthesis algorithm of Section 2.5 using the sets $P(s)$. At the end of phase 2 of each iteration, we update $U(s)$ using the steps described below. At the beginning of phase 1 of the subsequent iteration we then update $P(s)$ to the set of webpages whose URLs conform with $U(s)$.

(6) *Filtering webpage sets:* Based on the observation that relevant webpages of a website s have a similar structure, we keep in $P(s)$ only webpages that contain container and attribute nodes that match the generated $E(s)$ and are reachable from $p_{main}(s)$ via such webpages.

(7) *Generating URL patterns:* For each webpage $p \in P(s)$ we consider its URL. We unify the URLs into $U(s)$ by a variation of Algorithm 2 which views a URL as a sequence of instructions, similarly to a sequential XPath.

2.7 Evaluation

In this section we evaluate the effectiveness of our approach. We used it to synthesize data extracting web-crawlers for real-world websites containing structured data of different categories. Our experiments focus on two different aspects: (i) the ability to successfully synthesize web-crawlers, and (ii) the performance of the resulting web crawlers.

2.7.1 Experimental Settings

We have implemented our tool in C#. All experiments ran on a machine with a quad core CPU and 32GB memory. Our experiments were run on 30 different websites, related to nine different categories: books, TVs, conferences, universities, cameras, phones, movies, songs and hotels. For each category we selected a group of 3-4 known sites, which appear in the first page of Google search results.

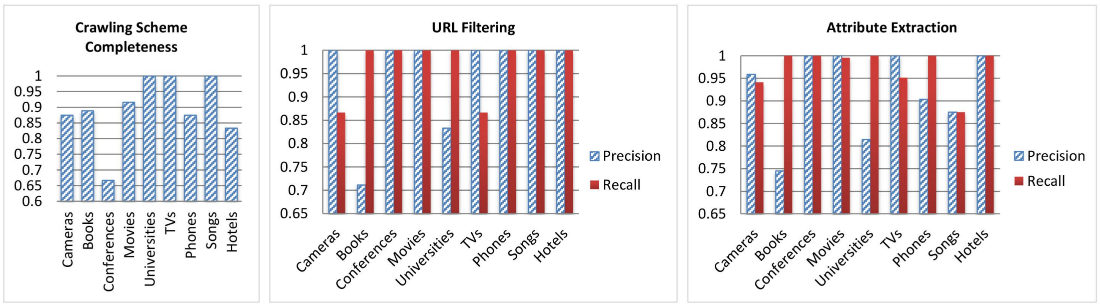


Figure 2.6: Results: Crawling scheme completeness (left), URL filtering (middle) and Attribute extraction (right) for each category.

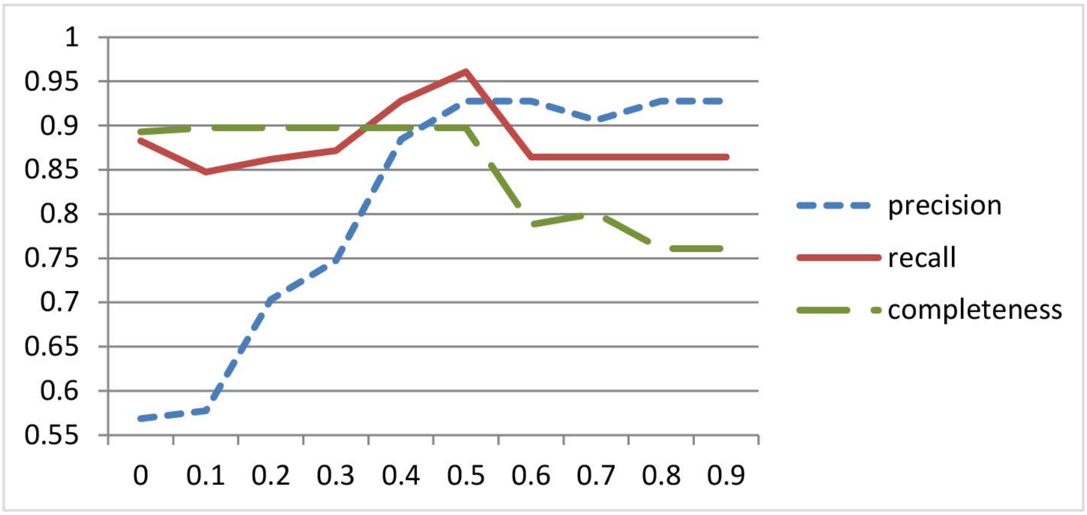


Figure 2.7: Attribute extraction precision and recall, and crawling scheme completeness, as a function of the threshold of Jaccard similarity used to define equivalence between instances.

The sites in each category have a different structure, but they share at least some of their instances, which makes our approach applicable. The complexity of the data extracted from different categories is also different. For instance a movie has four attributes: *title*, *genre*, *director* and list of *actors*. For a book, the set of attributes consists of *title*, *author* and *price*, while the attribute set of a camera consists of the *name* and *price* only. In each category we used one manually written crawler and automatically synthesized the others (for the books category we also experimented with 3 partial extraction schemes, one for each attribute). To synthesize the web crawlers, our tool processed over $12,000$ webpages from the 30 different sites.

To evaluate the effectiveness of our tool we consider 4 aspects of synthesized crawlers: (i) Crawling scheme completeness, (ii) URL filtering, (iii) Container extraction, and (iv) Attributes extraction.

2.7.2 Experiments and Results

Crawling Scheme Completeness A complete crawling scheme defines extraction queries for all of the data attributes. The completeness of the synthesized crawling schemes is an indicator for the success of our approach in synthesizing crawlers. To measure completeness, we calculated for each category the average number of attributes covered by the schemes, divided by the number of attributes of the category. The results are reported in Fig. 2.6 (left). The results show that the resulting extraction schemes are mostly complete, with a few missing attribute extraction queries.

URL Filtering The ability to locate pages containing data is an important aspect of a crawler's performance. To evaluate the URL filtering performance of the synthesized crawlers, we measure the *recall* and *precision* of the synthesized URL pattern for each site:

$$recall = \frac{|Rel \cap Sol|}{|Rel|} \quad precision = \frac{|Rel \cap Sol|}{|Sol|} \tag{2.1}$$

To do so, we have manually generated two sets of URLs for each site: one containing URLs for pages that contain relevant data, comprising the Rel set (ground truth), and another, denoted Irr, contains a mixture of irrelevant URLs from the same site. Sol contains the URLs from $Rel \cup Irr$ that match the synthesized URL pattern for the site (i.e., the URLs accepted by the synthesised URL pattern). A good performing URL filtering pattern should match all the URLs from Rel and should not match any from Irr. The average recall and precision scores of the sites of each category are calculated and reported in Fig. 2.6 (middle).

Container Extraction To check the correctness of the synthesized container extraction query, we have manually reviewed the resulting container XPaths against the HTML sources of the relevant webpages for each site, to verify that each extracted container contains exactly one data item. We found that the containers always contained no more than one item. However, in a few cameras and songs websites, the container query was too specific and did not extract some of the containers (this happened in tables containing class="odd" in some rows and class="even" in others), which affected the recall scores of attribute extraction.

Attributes Extraction We calculate the recall and precision (see equation (2.1)) of the extraction query for each attribute. Technically, for each category of sites, we have manually written extraction queries for each attribute in every one of the category related sites. For each attribute a, we used these extraction queries to extract the instances of a from a set of sample pages from each site. The extracted instances are collected in Rel. We have also applied the synthesized extraction queries (as a concatenation of the container XPath and attribute XPath) to extract instances of a from the same pages into Sol. For each site, the precision and recall are calculated according to equation (2.1). The average (over sites of the same category) recall and precision scores of all attributes of each category are reported in Fig. 2.6 (right).

Equivalence Relation To evaluate the effect of the threshold used in the equivalence relation, $\equiv_a$, on the synthesized crawlers, we have measured the average completeness, as well as the average recall and precision scores of attribute extraction as a function of the threshold. The results appear in Fig. 2.7.

Remark. The reported attribute extraction recalls in Fig. 2.6 and Fig. 2.7 are computed based on queries for which synthesis succeeded (missing queries affect only completeness, and not recall). $\square$

2.7.3 Discussion

The completeness of the synthesized extraction schemes is highly dependent upon the ability to identify instances in pages of some site by comparison to instances gathered from other sites. For most categories, completeness is high. For the conferences category, however, completeness is low. This is due to the use of acronyms in conference names (e.g., ICSE) in some sites vs. full names (e.g., International Conference on Software Engineering) in others, which makes it hard for our syntax-based equivalence relation to identify matches. This could be improved by using semantic equivalence relations (such as ESA [GM07] or W2V [MCCD13]).

As for the quality of the resulting extraction schemes and URL filtering patterns, most of the categories have perfect recall (Fig. 2.6). However, some have a slightly lower recall due to our attempt to keep the synthesized XPaths (or regular expressions, for URL filtering) as concrete as possible while having majority agreement. This design choice makes our method tolerant to small noises in the identified data instances, and prevents such noises from causing drifting, without negative examples. Yet, in some cases, the resulting XPaths are too specific and result in a sub-optimal recall.

For precision, most categories have good scores, while a few have lower scores. Loss of precision can be attributed to the majority-based unification and the lack of negative examples. For the books category, for instance, the synthesized extraction XPath of price for some sites is too general, since they list multiple price instances (original price, discount amount, and new price). All are listed in the same "parent container" with the author and book title, and are therefore not filtered by the container, hence affecting XPath unification. This could be improved with user guidance.

The results in Fig. 2.7 reflect the tradeoff between precision and crawling scheme completeness. A more strict equivalence relation (with higher threshold) leads to a better precision but has negative effect on the scheme completeness, whereas the use of a forgiving equivalence relation (with lower threshold) severely affects the precision. We use a threshold of 0.5 as a balanced threshold value. According to our findings, the attribute queries suffer from a low recall for both low and high threshold values. In low threshold, it is due to wrong queries, that extract wrong nodes (e.g., menu nodes), without including attribute nodes. For higher threshold values, the tool identified less instances of attribute nodes (sometimes only one), leading to a lower quality generalization.

Real-World Use Case We used our crawler synthesis process as a basis for data extraction for several product reviews websites. For instance, `tvexp.com`, `weppir.com`, `camexp.com` and `phonesum.com` extract product names and specifications (specs) using our approach. We manually added another layer of specs scoring, and created comparison sites for product specs. These websites have a continually updated database with over 20,000 products.

2.8 Related Work

In this section, we briefly survey closely-related work. While there has been a lot of past work on various aspects of mining and data extraction, our technique has the following unique combination of features: (i) works across multiple websites, (ii) synthesizes both the extraction XPath queries, and the URL pattern, (iii) is automatic and does not require user interaction, (iv) works with only positive examples, (v) does not require an external database, and (vi) synthesizes a working crawler.

Data Mining and Wrapper Induction Our work is related to data mining and wrapper induction. In contrast to supervised techniques (e.g., [KWD, LG14, LPH00, CH04, GMM$^+$11]), our approach only requires an initial crawler (or partial crawling scheme) and requires no tagged examples. FlashExtract [LG14] allows end-users to give examples via an interaction model to extract various fields and to link them using special constructs. It then applies an inductive synthesis algorithm to synthesize the intended data extraction program from the given examples. In contrast, our starting point is a crawler for one (or more) sites, which we then extrapolate from. Further, our technique only requires positive examples (obtained by bootstrapping our knowledge base by crawling other sites).

Unsupervised extraction techniques [AGM03, CL, CMM01, MK07, MTH$^+$09, RGSL04, ZL05, SC13] have been proposed. Several works [CL, MTH$^+$09, AGM03, DKS11, TW14, LGZ03b, TW13] propose methods that use repeated pattern mining to discover data records, while [RGSL04, ZL05] use tree-edit distance as the basis for record recognition and extraction in a single given page. These methods require manual annotation of the extracted data or rely on knowledge bases [GZAC13, Hon11]. Roadrunner [CMM01] uses similarities and differences between webpages to discover data extraction pattern. Similarities are used to cluster similar pages together and dissimilarities between pages in the same cluster are used to identify relevant structures. Other information extraction techniques rely on textual, or use visual features of the document [ZNW$^+$06, LMM10] for data extraction. ClustVX [Gri13] renders the webpage in contemporary web browser, for processing all visual styling information. Visual and structural features are then used as similarity metric to cluster webpage elements. Tag paths of the clustered webpages are then used to derive extraction rules. In contrast, our approach does not use visual styling, but relies on similar content between the different sites.

HAO et al. [HCPZ11] present a method for data extraction from a group of sites. Their method is based on a classifier that is trained on a seed site using a set of predefined feature types. The classifier is then used as a base for identification and extraction of attribute instances in unseen sites. In contrast, our goal is to synthesize XPaths that are human-readable, editable, and efficient. Further, with the lack of an attribute grouping mechanism (such as our notion of container), the method cannot handle pages with multiple data items.

Program Synthesis Several works on automatic synthesis of programs [JGS$^+$10, GRB$^+$14, LG14, GJTV11] were recently proposed, aiming for automating repetitive programming tasks. Programming by example, for instance, is a technique used in [JGS$^+$10, LG14] for synthesizing a program by asking the user to demonstrate actions on concrete examples. Inspired by these works, our approach automatically synthesizes data extracting web crawlers. However, we require no user interaction.

Semantic Annotation Many works in this area attempt to automatically annotate webpages with semantic meta-data.

Seeker [DEG+03] is a platform for large-scale text analysis, and an application written on the platform called SemTag that performs automated semantic tagging of large corpora. Ciravegna et al. [CCDW04] propose a methodology based on adaptive information extraction and implement it in a tool called Armadillo [CDC04]. The learning process is seeded by a user defined lexicon or an external data source. In contrast to these works, our approach does not require external knowledge base and works by bootstrapping its knowledge base.

Other Aspects of Web Crawling There are a lot of works dealing with different aspects of web crawlers. Jiang et al. [JWF+10] and Jung et al. [AGWC07] deal with deep-web related issues, like the problem of discovering webpages that cannot be reached by traditional web crawlers mostly because they are results of a query submitted to a dynamic form and they are not reachable via direct links from other pages. Some other works like [RM+99, VS05] address the problem of efficient navigation of website pages to reach pages of specific type by training a decision model and using it do decide which links to follow in each step. Our paper focuses on the different problem of data extraction, and is complementary to these techniques.

2.9 Conclusion

We presented an automatic synthesis of data extracting web crawlers by extrapolating existing crawlers for the same category of data from other websites. Our technique relies only on data overlaps between the websites and not on their concrete representation. As such we manage to handle significantly different websites. Technically, we automatically label data in one site based on others and synthesize a crawler from the labeled data. Unlike techniques that synthesize crawlers from user provided annotated data, we cannot assume that all annotations are correct (hence some of the examples might be false positives), and we cannot assume that unannotated data is noise (hence we have no negative examples). We overcome these difficulties by a notion of containers that filters the labeling.

We have implemented our approach and used it to automatically synthesize 30 crawlers for websites in nine different product categories. We used the synthesized crawlers to crawl more than $12,000$ webpages over all categories. In addition, we used our method to build crawlers for real product reviews websites.

Chapter 3

Synthesis of Forgiving Data Extractors

Abstract

We address the problem of synthesizing a robust data-extractor from a family of websites that contain the same kind of information. This problem is common when trying to aggregate information from many web sites, for example, when extracting information for a price-comparison site.

Given a set of example annotated web pages from multiple sites in a family, our goal is to synthesize a robust data extractor that performs well on all sites in the family (not only on the provided example pages). The main challenge is the need to trade off precision for generality and robustness. Our key contribution is the introduction of *forgiving extractors* that dynamically adjust their precision to handle structural changes, without sacrificing precision on the training set. Our approach uses decision tree learning to create a generalized extractor and converts it into a forgiving extractor, in the form of an XPath query. The forgiving extractor captures a series of pruned decision trees with monotonically decreasing precision, and monotonically increasing recall, and dynamically adjusts precision to guarantee sufficient recall.

We have implemented our approach in a tool called TRACY and applied it to synthesize extractors for real-world large scale web sites. We evaluate the robustness and generality of the forgiving extractors by evaluating their precision and recall on: (i) different pages from sites in the training set (ii) pages from different versions of sites in the training set (iii) pages from different (unseen) sites. We compare the results of our synthesized extractor to those of classifier-based extractors, and pattern-based extractors, and show that TRACY significantly improves extraction accuracy.

3.1 Introduction

We address the problem of synthesizing a robust data extractor based on a set of annotated web pages. Web sites often change their formatting and structure, even when their semantic content remains the same. A robust extractor [DBS09b, LSRT14, LWLY12] can withstand modifications to the target site. A non-robust extractor would have to be adjusted (typically manually) every time the formatting of a site changes.

Our idea is to construct a robust extractor by training it on a family of sites that have content that is semantically similar. We conjecture that the ability of a single extractor to handle multiple different sites means that the extractor is likely robust, as it is able to overcome differences between sites, and thus possibly also differences due to future changes of the same site.

The notion of a *family of sites* is somewhat vague, and we assume that it is user-defined. The intent is that sites of the same family contain the same kind of information, even if they differ on the way it is presented. Sites of the same family could be, for example, a family of book-selling sites, hotel reservation sites, etc. We conjecture that despite the fact that sites in a family may differ, they have a similar set of underlying semantic concepts, that a generalized extractor will be able to capture. For example, for book-selling sites, the notions of *author*, *title*, and *price* are likely to be presented in some way on all sites in the family.

Goal Given a set of training pages (HTML documents) within a family, where each page is annotated by tagging data items of interest, our goal is to synthesize a *robust extractor* that maximizes accuracy over the training set.

When constructing the generalized extractor for a family, there is a natural tradeoff between accuracy and generality. Constructing a precise extractor may prevent it from being robust to future changes. Constructing a loose extractor makes it more robust, but would yield results of poor accuracy. Both of these options are undesirable. Instead, our key contribution is the construction of *forgiving extractors* that adjust their precision dynamically and do not commit to a specific generalization tradeoff upfront.

Existing Techniques Manually writing a data extractor is extremely challenging. This motivated techniques for automatic "wrapper induction", learning extraction queries from examples [KWD]. Automated techniques reduce the burden of writing extractors, but still require manual effort (e.g., providing tagged samples for each site).

There has been a lot of work on pattern based techniques, using alignment of XPaths (e.g., [RGSL04, NDMBDT14, NBdT16]). These techniques learn paths to the annotated data items and generalize them to generate an extraction query. When provided with items that have significantly different paths (e.g., originate from different sites) these techniques may result in an overly relaxed query expression, and significant loss of precision. As a result, pattern based techniques are often limited to a single web site, and are sensitive to formatting changes.

Model based techniques [HCPZ11, FK04, SWL$^+$12, Kus00], use features of DOM elements to learn a model that classifies DOM items to *data* and *noise*. These methods have several drawbacks. First, they lack a standard execution mechanism for extraction (in contrast to XPath queries that are available for pattern based techniques). Second, the classifiers trained by these techniques are often hard to understand and modify. Last, but not least, the generalization in these models is performed *at training time* and thus presents the standard dilemma between precision and generality.

Our Approach Our approach is based on the assumption that, despite differences in presentation, *some* sites in the family do share *some* local syntactic structural similarity (e.g., a book title may appear under some heading class, or its parent may be of some particular class). Our claim (which we support empirically) is that training on a few sites from the family will cover most of the local structural patterns. Following this insight, our approach tackles the problem in two steps:

(1) Precise generalization for the training set: given a set of annotated pages from different sites, we use decision tree learning to synthesize a *common XPath query* that *precisely* extracts the data items from all sites. Intuitively, the decision tree is used as a generalization mechanism that picks the important features for precise extraction. The overall set of features consists of XPath predicates that capture local syntactic structural properties. The synthesized query increases robustness, while maintaining precision on the training set.

(2) Dynamic generalization: the XPath query constructed in the previous step is precise for pages in the training set, but may be overly restrictive for extraction from other pages. To generalize to other pages, we introduce the novel notion of *forgiving XPaths*—a query that *dynamically relaxes its requirements*, trading off precision to make sure that target items are extracted.

The query generated by our approach has the benefits of both pattern based techniques and model based techniques. On the one hand, it is a standard XPath query, which has a wide support from web-browsers, programming languages and DOM parsers. This also makes it human readable, easy to review and modify by a programmer. On the other hand, the query has the flexibility and generalization ability of the techniques based on learning models.

Main Contributions The contributions of this paper are:

- A novel framework for synthesis of robust data extractors for a family of sites from examples (annotated web pages).
- A new technique for generalization of XPath queries using decision trees and "forgiving XPaths", which adjust precision dynamically.
- An implementation of our technique in a tool called TRACY and an experimental evaluation of the robustness and generality of the forgiving extractors. We evaluate precision and recall on: (i) different pages from sites in the training set (ii) pages from different versions of sites in the training set (iii) pages from different (unseen) sites. Our evaluation shows that TRACY is able to synthesize robust extractors with high precision and recall based on a small number of examples. Further, comparison to existing pattern based and model based techniques shows that TRACY provides a significant improvement.

3.2 Overview

In this section, we provide an informal overview of our approach. We elaborate on the formal details in later sections.

3.2.1 Motivating example

Consider the problem of synthesizing an extractor of book information based on examples from three book-seller sites: `abebooks.com` (ABE), `alibris.com` (ALIBRIS) and `barnesandnoble.com` (B&N). To simplify presentation, we focus on an extractor for the *author* attribute.

Fig. 3.1 shows simplified fragments of HTML documents presenting book information from the three different sites. We annotated these documents by tagging the HTML nodes that contain instances of the *author* attribute with a special HTML-attribute, *userselected*.

```
(*@\underline{\emph{Abe}}@*)
<div id="bookInfo">
 <h1 id="book-title">Rachael Ray 30-Minute Meals 2</h1>
 <h2>
   <a userselected href="/servlet/Se.._-author">
     Ray, Rachael</a>
 </h2>...
</div>

(*@\underline{\emph{B\&N}}@*)
<section id="prodSummary">
<h1 itemprop="name">History of Interior Design</h1>
 <span>
  by <a userselected href="/s/.._contributor..">
       Jeannie Ireland</a>
 </span>...
</section>

(*@\underline{\emph{Alibris}}@*)
<div class="product-title">
<h1 itemprop="name">La Baba del Caracol</h1>
 <h2>by
  <a userselected itemprop="author" itemscope href="..">
       C. Maillard</a>
 </h2>
 </div>
```

Figure 3.1: Fragments of webpages with the author attribute values for a book on three different book seller sites.

Note that the three different sites have different structure, and that the attribute of interest (*author*) appears differently in each site. There are many ways to write an XPath query that extracts the author from each site. For example:

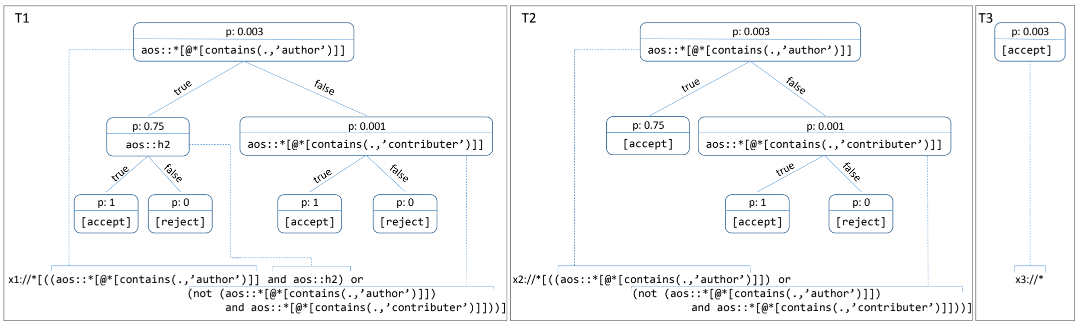
- `//div[@id="bookInfo"//a` for ABE
- `//section/span/a` for B&N
- `//a[@itemprop="author"]` for ALIBRIS

Alternatively, other example queries could be:

- `//a[@*[contains(.,"author")]]` for ABE and ALIBRIS
- `//*[contains(text(),"by")/a` for B&N (also ALIBRIS).

Some of these queries are more robust than others, and some would generalize better across site versions. For example, the query `//*[contains(text(),"by")/a` for extracting the author name in B&N is more general than the query `//section/span/a`, which relies on a specific nesting structure of `section`, `span`, and `anchor`. Because there are many possible queries that extract the same information, it may be possible to use the different sites to pick queries that would be more robust. However, it is not clear how to do this.

For example, using an alignment based XPath generation technique on our three training



```
x1://*[((aos::*[@*[contains(.,'author')]] and aos::h2) or
                    (not (aos::*[@*[contains(.,'author')]])
                         and aos::*[@*[contains(.,'contributer')]]))]
```

```
x2://*[((aos::*[@*[contains(.,'author')]]) or
                    (not (aos::*[@*[contains(.,'author')]])
                         and aos::*[@*[contains(.,'contributer')]]))]
```

```
x3://*
```

Figure 3.2: Example decision trees over XPath predicates and their respective XPath translations.

pages would result in an XPath like `//*/*/a`, which is too general, and would lead to significant loss of precision (as shown in Section 5.6). This is because alignment based techniques (and pattern based techniques in general) assume that documents in the training set have a shared template. They therefore fail to handle significant structural differences between documents in the training set, as occur when the training set contains documents from multiple sites. Pattern-based generalization techniques also produce queries that are sensitive to changes, and are therefore not robust.

3.2.2 Our Approach

Our extractor synthesis process attempts to automatically pick the queries that would be more robust and general. The synthesis starts by a precise query generalization for the training set, which is obtained by learning a decision tree for identification of the tagged nodes. This is followed by a dynamic generalization, which is obtained by creating XPath queries which we call *forgiving*.

Precise Generalization for the Training Set Synthesis starts by using the annotated documents to learn a decision tree which correctly identifies all the tagged HTML nodes. To do so, our method first extracts features that describe the tagged HTML nodes.

Extracting features: Feature extraction is done by traversing the paths from tagged nodes to the root and collecting features from nodes on the path and from their surrounding context. The extracted features are all valid XPath predicates (explained in Section 3.3.1). These features are used as tests in the inner nodes of the decision tree, and are the building blocks of our XPath queries.

Consider the *author* node in the ALIBRIS website (Fig. 3.1). Our feature extraction step starts from the `a` node having the *userselected* attribute, and traverses the path to the HTML root node though the nodes `h2` and `div`, while collecting features both from the nodes on the path and from their surrounding context. Among the extracted features are the following:

- `ancestor-or-self::h2`, which matches nodes of type `h2` or an ancestor of type `h2` (later, e.g., in Fig. 3.2, we write `aos` as a shorthand for `ancestor-or-self`),
- `@itemprop="author"`, which matches nodes with the attribute `itemprop` having the value `author`, and
- `@*[contains(.,"author")]`, which matches nodes that *contain* the value `author`.

The feature `@*[contains(.,"author")]` is a generalization of the feature `@itemprop="author"`. Our feature extraction step generates such generalized features, in addition to the precise ones, in order to later enable the use of generalized structural features in the decision tree when possible (when the generalization does not result in loss of precision).

Learning a Decision Tree: Once the feature extraction step is complete, our method uses a recursive procedure to learn the decision tree, based on the extracted features. Our algorithm is a variant of the ID3 algorithm [Qui]. One of the differences is that our algorithm prioritizes generalized features when selecting the feature to use as a test in inner nodes. Technically, this is done by assigning costs to features, with generalized features having lower costs. Additional details and differences are described in Section 3.3.2.

Fig. 3.2 (T1) presents the decision tree generated by our algorithm to identify *author* nodes in ABE, B&N and ALIBRIS. The root node of T1 has `@*[contains(.,"author")]` as a test. This is because this feature has the highest improvement-to-cost ratio: it results in the highest information gain, when prioritizing low-cost (generalized) features. The *true*-branch of the root leads to a node with test `aos::h2`, while the *false*-branch leads to `@*[contains(., "contributer")]` as a test.

A decision tree has a natural recursive translation into a valid XPath query that extracts the nodes identified by the decision tree. The XPath query x1 presented at the bottom of Fig. 3.2 is the extraction query generated from T1 for the *author* attribute.

Dynamic Generalization using Forgiving XPaths The decision tree learned from the annotated HTML documents, and the corresponding XPath query, identify the annotated nodes as precisely as possible (depending on the extracted features). However, in order to improve the ability to extract semantically similar nodes in other versions of the web pages, some generalization is desired. A natural generalization is by pruning the decision tree, turning some inner nodes to accepting nodes. Such generalization trades off precision on the training set for a potentially higher recall on other pages. However, the question remains where to prune the decision tree, and how much precision to sacrifice.

As an example, consider a modified version of one of the sites where books also contain an author attribute. Fig. 3.3 shows such an example HTML fragment. Fig. 3.2 shows three trees T1, T2 and T3 and their respective XPath translations x1,x2 and x3. As explained above, T1 is the tree learned by our algorithm when applied on ABE, B&N and ALIBRIS. T2 is a pruned version of T1, with a slightly lower precision, while T3 is a fully-pruned tree accepting every node. Running x1 on the modified site returns no results. Using x2 will have better performance on the modified site, however, it will have a lower precision on sites from the training set.

Forgiving XPaths: Our goal is to find the highest level of precision that obtains sufficient recall. As the books example demonstrates, this level might be different for different sites. An important aspect of our approach is the introduction of *forgiving XPaths*, which allow us to postpone the decision of how much precision to sacrifice to the evaluation time of the XPath, and hence adjust the generalization to the web page at hand.

Technically, a forgiving XPath query is an XPath query obtained as a union of several

```
<p id="product_subtitle">.. in Simple Words</p>
<p id="product_author">By
 <a class="blue_link" href="...">Randall Munroe</a></p>
```

Figure 3.3: Fragment of a modified book information page.

queries (using the "|" XPath operator). Each query exhibits a different precision level on the training set, and is conditioned by the more precise queries not extracting any nodes. This means that at run time, when invoked on a HTML document, the forgiving XPath will evaluate to the query with the maximal precision level that extracts a non-empty set of nodes. This property allows the forgiving XPath to perform well on both seen and unseen documents, avoiding the trade-off between precision on seen data and recall on unseen data. It is important to note that our forgiving XPath query is a standard XPath query that may be used in any XPath interpreter without the need for modifications of any kind.

We construct the forgiving XPaths from a series of XPaths with monotonically decreasing precision, such as x1, x2 and x3 above. In the books example, we derive the following forgiving XPath:

$$fx = \texttt{/*/x1|/*[not(x1)]/x2.}$$

fx uses the root node (which never contains data), to enable returning the result of invoking x2 only when x1 returns no results.

3.3 Decision Tree Learning

In this section we present the first step of our approach for synthesizing forgiving XPaths, which constructs a decision tree based on a given set of annotated web pages.

Notation Given a set of web documents $D = \{d_1, \ldots, d_n\}$, where each d_i is represented as a DOM tree with a set states_i of nodes, and given a target set $N_i \subseteq \mathit{states}_i$ of nodes of interest for each $d_i \in D$, we denote by $\mathit{states} = \bigcup_{i=1}^{n} \mathit{states}_i$ the global set of DOM nodes, and by $N = \bigcup_{i=1}^{n} N_i$ the global target set of nodes of interest. We also denote the remainder of the nodes by $\overline{N} = \mathit{states} \setminus N$.

Our approach starts with a feature extraction phase, where we generate a set F of features based on the DOM nodes in N. We then construct a decision tree that uses the features in F to classify *states* into N and $\overline{N}$. The decision tree will serve as the basis for the creation of a forgiving XPath, as described in Section 3.4.

3.3.1 Feature Extraction

For every individual document d_i in the training set D, there are many correct XPaths for the extraction of N_i. Each such XPath might use different structural features of the document. To improve the ability to generate a concise *common* XPath for all of the documents in D, we first generate a large set of structural features.

In the feature extraction phase, our method extracts features of N that are expressible by valid XPath queries. Each feature f is defined by an XPath predicate, with the meaning that the value of f in a DOM node is 1 iff the value of the XPath predicate on the same node is *true*.

The extracted features of a node in N are divided to *node features*, and *context features*. The extent of the surrounding context that we consider is configurable. In our implementation, context features consist of *children features* and *ancestor features*.

Node features consist of:

- **Node Name** the node name is used to create a node-test feature (for instance, `self::a`).
- **Text Content** the text content of the text children of a node is used to define text-equality features and text-containment features. For instance, the predicate `text()="Sale price:"` is a text-equality feature, which holds iff the current node has a text child node whose value equals "Sale price:". On the other hand, the predicate `contains(text(),"price")` is a text-containment feature, that requires the current node to have a text child node containing the string "price".
- **Node Attributes** the node attributes define attribute features. Each attribute consists of a name and value pair (for instance, `id = "product_title"`). Attribute features are of three different types:
  1. Attribute name features (e.g. `@class` requires having an attribute with name "class")
  2. Attribute value features with or without conditions on the attribute name (e.g. `@*="title"` requires having an attribute with value "title", while `@id="title"` requires having an attribute named "id" and value "title")
  3. Text-containment features on attribute values, with or without conditions on the attribute name. For instance, `@*[contains(.,"title")]` requires that some attribute has a value that contains the token "title".

Context features are divided to:

- **Children Features** node features as defined above are extracted also for the children of the node (e.g., `child::div` is a predicate that evaluates to *true* on nodes that have children with node name 'div').
- **Ancestor Features** node features and children features as defined above are extracted also for ancestors of the node (e.g., `ancestor-or-self::*[child::span]` evaluates to *true* on nodes with ancestors that have children named 'span').

Technically, the feature extraction process is performed on each document $d_i \in D$ separately. For each document d_i, our method starts from the nodes in N_i. For each node it computes its node and children features and ascends up the path to the DOM tree root, while gathering ancestor features. To generate text-containment features we tokenize the relevant text and use the resulting tokens to define containment conditions `contains(text(),token)`.

Features as XPath Queries Each of the extracted features f is an XPath predicate. It induces an XPath query `//*[f]` that, when invoked on the root of a DOM tree, extracts the set of all nodes in the DOM tree that satisfy f. With abuse of notation we sometimes refer to the feature f itself as the induced XPath query.

3.3.2 Decision Tree Learning

Given the extracted set of features F, we use a variant of ID3 to learn a decision tree that classifies A into N and $\overline{N}$.

Decision tree A decision tree T is a complete binary labeled directed tree, in which each inner node (including the root) is labeled by a feature $f \in F$, and has exactly two children: a 0-child, and a 1-child (each child corresponds to a value of f). We also refer to the children as *branches*. The leaves in the tree are labeled by 0 or 1, where each value corresponds to a possible class. The elements in A whose feature vectors match paths in T from the root to a leaf labeled by 1 form the 1-class induced by T. The 0-class is dual.

Decision Tree Learning Given the partitioning of A into N and $\overline{N}$, and given a set of features F, ID3 constructs a decision tree T whose 1-class is N and whose 0-class is $\overline{N}$. (Note that a correct classifier exists only if whenever two elements $\mathrm{elem}_1, \mathrm{elem}_2 \in A$ share a feature vector, they also share a class.)

The construction of the decision tree is recursive, starting from the root. Each node t in the tree represents the subset of the samples, denoted $S(t)$, whose feature vector matches the path from the root to t. If all samples in $S(t)$ have the same classification, then t becomes a leaf with the corresponding label. Otherwise, ID3 may choose to partition $S(t)$ based on the value of some feature $f \in F$. The node t is then labeled by the feature f, and the construction continues recursively with its children, where its 1-child represents the subset of $S(t)$, denoted $S(t)_f$, in which f has value 1, and its 0-child represents the set $S(t)_{\overline{f}} = S(t) \setminus S(t)_f$ where f has value 0. In our setting these sets are computed by running the feature $f \in F$ on every $d \in D$ and computing $S(t)_f = (\bigcup_{d \in D} [\![f]\!]_d) \cap S(t)$, and $S(t)_{\overline{f}} = S(t) \setminus S(t)_f$.

ID3 selects the feature $f \in F$ to split on as the feature with the highest *information gain* for $S(t)$, where the information gain [Mit97], denoted $IG(S(t), f)$, of $S(t)$ based on f measures the reduction in uncertainty after splitting $S(t)$ into $S(t)_f$ and $S(t)_{\overline{f}}$. Our learning algorithm modifies ID3 in two ways: (i) features have costs that are taken into account in the computation of the information gain. (ii) the choice of a feature f to split on is restricted in a way that ensures a correlation between an element having value 1 for f in the feature vector and the element being classified into the 1-class. We elaborate on these modifications below.

Feature Costs We consider features with different costs. Intuitively, the more specific the feature is, the higher its cost. We denote the cost of $f \in F$ by $Cost(f)$. In order to prioritize generalized features over more specific ones, we define the information-gain-to-cost ratio and use it instead of the information gain:

Definition 3.3.1. The *information-gain-to-cost ratio* of a feature f for a set $S \subseteq A$, denoted $IGR(S, f)$, is:

$$IGR(S, f) = IG(S, f)/Cost(f)$$

Feature-Class Correlation Guided by the intuition that, in many cases, existence of features is important for being classified as part of the 1-class, while their nonexistence is incidental, we create a decision tree that correlates feature values with the classification: specifically, value 1 for a feature is correlated with class 1.

The feature-class correlation is obtained by restricting the set of features to split $S(t)$ on only to the subset $F_t \subseteq F$ for which $S(t)_f$ (which will form the 1-branch if the split by f takes place) has more instances from N than $S(t)_{\overline{f}}$ (which will form the 0-branch).

Among all the features in F_t, the feature f with the highest $IGR(S(t), f)$ is selected as the feature to split t.

3.4 Forgiving XPath Synthesis

Building a decision tree is an intermediate step towards synthesizing a forgiving XPath query. In this section we define the notion of forgiving XPaths and describe their synthesis.

Forgiving XPaths A forgiving XPath is constructed based on a series of XPaths $x_0, x_1, \ldots, x_k$ s.t. for every $0 \leq j < k$ and $d \in D$, $[\![x_j]\!]_d \subseteq [\![x_{j+1}]\!]_d$, where $[\![x]\!]_d$ is the set of nodes extracted by x from d. Note that if x_0 is precise for every seen document, then the precision of the XPaths in the sequence is monotonically decreasing, but the recall on unseen documents is potentially increasing.

Definition 3.4.1. Given a series of XPaths $x_0, x_1, \ldots, x_k$ as defined above, a *forgiving XPath* is defined by

$$\mathit{fx} = / * /x_0 \mid / * [not(x_0)]/x_1 \mid \ldots \mid / * [not(x_{k-1})]/x_k$$

We say that x_0 is the *base* of *fx*.

fx uses the union operator ("|"), which means that it extracts the union of the nodes extracted by the individual queries. However, the queries are constructed in such a way that for every document d, seen or unseen, $[\![\mathit{fx}]\!]_d = [\![x_j]\!]_d$ for the least j such that $[\![x_j]\!]_d \neq \emptyset$. This is ensured since the XPath x_j is applied on $/ * [not(x_{j-1})]$, which will extract an empty set of nodes if $[\![x_{j-1}]\!]_d \neq \emptyset$, and will extract the root node of d otherwise. Therefore, at run time, for every document, *fx* evaluates to the most precise query in the series that actually extracts some nodes. In particular, if x_0 has perfect precision and recall for the training set, so will *fx*.

In the following, we present the construction of a forgiving XPath from a sequence of XPaths that are monotonically decreasing in their precision. We obtain such a sequence by first obtaining a sequence of decision trees with monotonically decreasing precision.

3.4.1 Decision Trees with Varying Precision

We measure the precision and recall of a decision tree T by the precision and recall of its 1-class w.r.t. the "real" class, N. We use pruning to define a sequence of decision trees with monotonically decreasing precision scores. Our motivation for doing so is to gradually trade off precision (on the training set) for a better recall on new documents. Pruning is based on the nodes precision:

Definition 3.4.2. The *precision* of a node $t \in T$ is defined as $precision(T, t) = \frac{|S(t) \cap N|}{|S(t)|}$.

In Fig. 3.2, the "p" labels of the inner nodes denote their precision. For instance, the node `aso::h2` in T1 has a precision of 0.75, which means that 75% of the elements in the training set that reach it when classified by T1, are from N.

Tree Pruning: Limiting Precision In the construction of the decision tree, a node with precision 1 becomes a 1-labeled leaf. We prune a tree by limiting the maximal precision of its nodes; Formally, given a precision threshold α, the method $prune(T, \alpha)$ prunes every subtree of T whose root t has $precision(T, t) \geq \alpha$, and turns t to a leaf labeled by 1. Thus, the induced 1-class is increased.

For example, T2 in Fig. 3.2 results from pruning T1 by limiting the precision of its nodes to 0.75.

Clearly, the tree T' obtained after pruning has a lower precision score than T (since the 1-class defined by T' is a superset of the 1-class defined by T). However, the recall score of T' can only increase compared to T: The recall score of T on the training set is already maximal, hence it is not improved by pruning. However, when considering new documents, the recall score of the pruned tree, T', can be strictly higher than that of T.

Layered Decision Tree Sequence Given the decision tree T learnt in Section 3.3.2, we create a sequence of decision trees with a monotonically decreasing precision and a monotonically increasing recall as follows:

Definition 3.4.3. Let $\alpha_0 > \alpha_1 > \ldots > \alpha_k$ be the sequence of precision scores of nodes in T, in decreasing order, i.e., $\{\alpha_0, \ldots, \alpha_k\} = \{precision(T, t) \mid t \in T\}$. The *layered decision tree sequence* based on T is the sequence $T_0, T_1, \ldots, T_k$ of decision trees, such that $T_0 = T$, and for every $0 \leq i < k$, $T_{i+1} = prune(T_i, \alpha_{i+1})$.

Note that it is possible that as a result of pruning, a node with precision score α_{i+1} no longer exists in T_i. In this case, the definition implies that $T_{i+1} = T_i$. However, we will simply omit T_{i+1} from the sequence. Therefore, each layer in the layered sequence reflects a "one step" reduction in the precision of the decision tree (and hence, potentially, an increase in the recall).

For example, the decision trees T1, T2 and T3 depicted in Fig. 3.2 form a layered decision tree sequence based on T1.

3.4.2 Translation of Decision Trees to Forgiving XPaths

The features used in the decision trees are all XPath predicates. This allows for a natural translation of a decision tree to an XPath query that extracts the set of nodes classified as belonging to the 1-class by the decision tree. In the following we describe this translation, which we use to generate a series of XPaths from the layered decision tree sequence generated from T (see Definition 3.4.3). We use the series to construct a forgiving XPath as defined in Definition 3.4.1

Decision Tree to XPath The translation of a decision tree $\tilde{T}$ to an XPath is performed by a recursive procedure, *GetXPath*(t), which is invoked on the root t of $\tilde{T}$ and returns an XPath predicate.

The procedure *GetXPath*(t) works as follows:

1. if t is a leaf node, return *true*() if it labeled 1, and *false*() if it is labeled 0.
2. if t is an inner node, labeled by feature f, then call *GetXPath* on its 0-child, t_0, and on its 1-child, t_1. Let $p_0 = \textit{GetXPath}(t_0)$, and $p_1 = \textit{GetXPath}(t_1)$. Then return the XPath predicate:

$$\texttt{(f and } p_1\texttt{) or (not(f) and } p_0\texttt{)} \tag{3.1}$$

Finally, from the XPath predicate p returned by the invocation of *GetXPath* on the root of $\tilde{T}$, an XPath query `//*[p]` is constructed. When invoked on the root of a DOM tree, `//*[p]` extracts the set of all nodes in the DOM tree that satisfy p. We denote the XPath query `//*[p]` by $\textit{XPath}(\tilde{T})$. The translation ensures that a DOM node is in the 1-class induced by $\tilde{T}$ iff it is in $[\![\textit{XPath}(\tilde{T})]\!]$.

Fig. 3.2 shows the XPath queries x1, x2, x3 generated from T1, T2, T3, respectively.

3.5 Evaluation

In this section we evaluate the effectiveness of our approach. Our experiments focus on three different aspects: (i) the accuracy of the synthesized queries on different pages from sites that have some pages in the training set (seen sites), (ii) their accuracy on different versions (newer and older versions) of the same pages, and (iii) their accuracy on pages from sites that have no pages in the training set (unseen sites).

3.5.1 Implementation

We implemented our approach in a tool called TRACY, and used it to synthesize extraction queries for multiple websites. For comparison, we also implemented four other extractors:

- To represent model-based approaches we trained two classifiers, C4.5 [Qui14] (J48 implementation) and naive bayes (NB) [JL95] (using Weka [HFH$^+$09]). In each experiment we train the classifiers on the same training set based on the same features used to construct our extractor.
- To represent other XPath-based approaches, we have implemented an alignment-based XPath (XA) generation technique proposed in [NDMBDT14].
- In addition to forgiving XPath (FX) queries, we used our approach to generate non-forgiving XPath queries (NFX) by directly generating an XPath from the obtained decision trees.

3.5.2 Experimental Settings

Datasets We constructed three datasets (available at: goo.gl/a16tiG) to evaluate our approach and the baselines. The first dataset, denoted DS1, contains pages from 30 real-life largescale websites divided to four different categories: books, shopping, hotels and movies. For each category, a set of (typically 2 or 3) common attributes were selected as the target data to be extracted. DS1 contains 166 manually annotated pages. To construct the second dataset, denoted DS2, we used archive.org to obtain five different versions (current version and four older ones) of the same page from four sites (with two or three different attributes to extract). DS2 contains 55 manually tagged webpages. The third dataset, denoted DS3, contains 5025 product pages

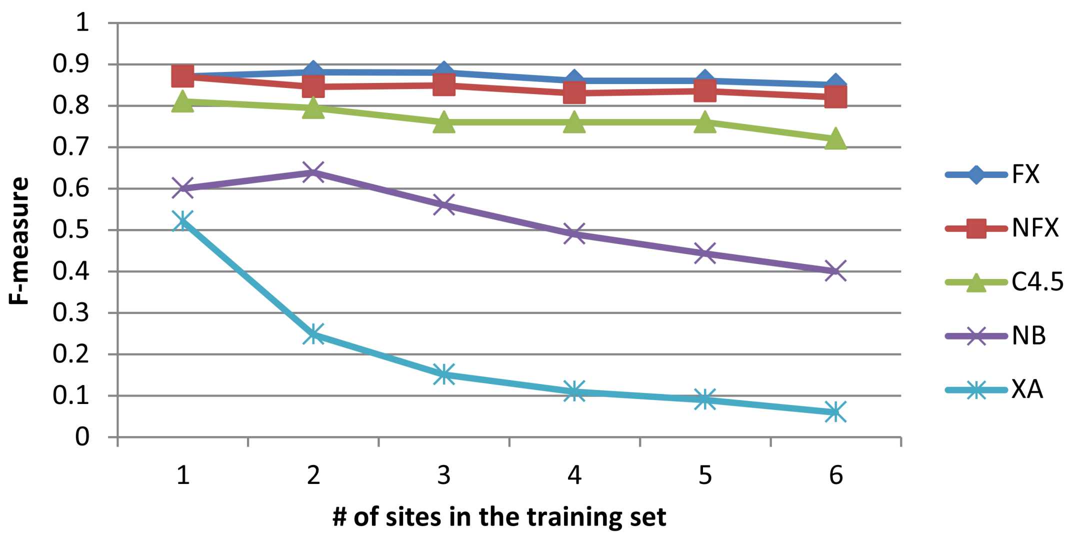


Figure 3.4: The F-measure values of different approaches for seen sites, as a function of number of sites in training set.

from five widely known websites. The pages were annotated (with two attributes, product name and price) using manually written XPaths.

In our experiments we used different subsets of the dataset as training sets and evaluation sets.

Performance Metrics Given a page d from one of the datasets and a target attribute for extraction $attr$, we denote by $N_{attr}(d)$ the set of tagged nodes containing instances of $attr$ in d. For a data extractor x extracting instances of $attr$, we measure the precision, recall and F-measure of x on d by considering $N_{attr}(d)$ as a target set. The precision, recall and F-measure of x (for $attr$) are defined as the averages over all pages d in the evaluation set containing instances of $attr$. The reported precision (P), recall (R) and F-measure (F1) for each approach are the averages over all the data extractors x that it synthesizes (for all categories and all attributes).

	$s = 2$	$s = 3$	$s = 4$	$s = 5$	$s = 6$
FX	0.6%	0.8%	-0.8%	-1.5%	-2.6%
NFX	-3.9%	-3.0%	-4.2%	-5.0%	-5.8%
C4.5	-1.8%	-5.6%	-6.5%	-7.7%	-10.6%
NB	6.8%	-7.3%	-17.9%	-26.0%	-34.4%
XA	-51.3%	-70.1%	-77.8%	-83.3%	-89.2%

Table 3.1: Performance (F-measure) decrease on seen sites as a function of the size of the training set (s). Lower decreases (negative values with smaller absolute value) indicate higher robustness to structural differences in the training set.

3.5.3 Evaluating the Different Performance Aspects

Accuracy on Seen Sites We evaluated the effectiveness of our approach (and the baseline approaches) for data extraction by evaluating the accuracy of the queries it generates on the different sites in the dataset. This is the standard use case, where we generate a query and evaluate it on each site separately.

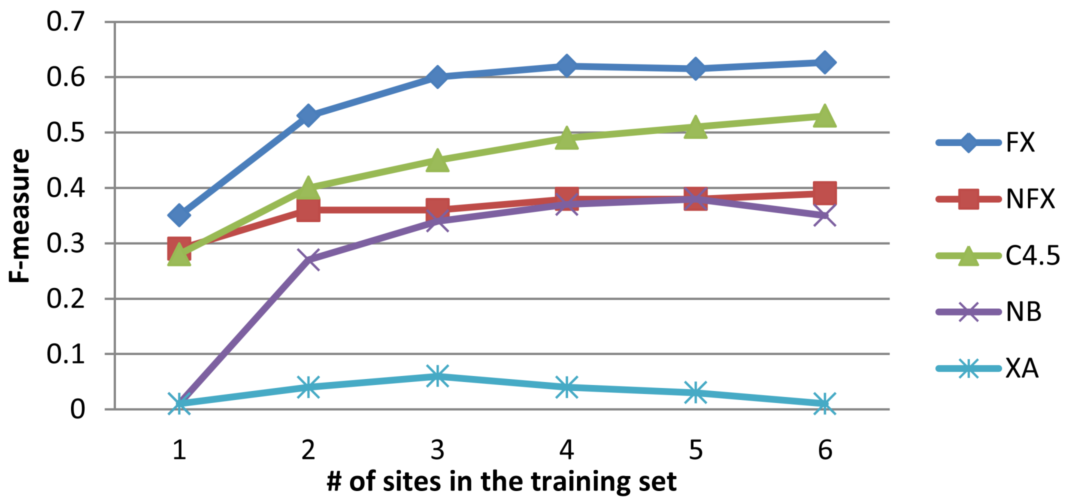


Figure 3.5: The F-measure values of different approaches for unseen sites, as a function of number of sites in training set.

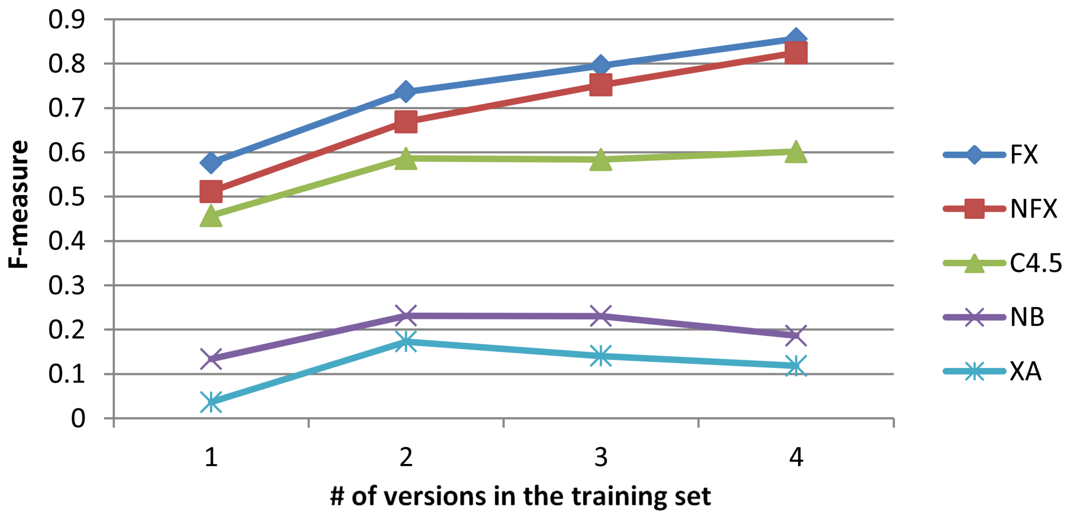


Figure 3.6: The F-measure values when evaluated on different page versions, as a function of number of versions in training set.

	$s = 1$			$s = 2$			$s = 3$			$s = 4$			$s = 5$			$s = 6$		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Performance on Documents from Seen Sites																		
FX	0.86	0.93	0.87	0.86	0.94	0.87	0.87	0.93	0.88	0.86	0.91	0.86	0.85	0.90	0.86	0.85	0.89	0.85
NFX	0.86	0.90	0.87	0.83	0.87	0.84	0.84	0.87	0.85	0.84	0.85	0.83	0.83	0.85	0.83	0.84	0.84	0.82
C4.5	0.80	0.87	0.81	0.72	0.93	0.79	0.69	0.91	0.76	0.68	0.91	0.76	0.67	0.92	0.75	0.65	0.91	0.72
NB	0.55	0.92	0.60	0.55	0.93	0.64	0.45	0.93	0.56	0.39	0.92	0.49	0.34	0.92	0.45	0.29	0.92	0.40
XA	0.53	0.54	0.52	0.24	0.38	0.25	0.14	0.33	0.15	0.10	0.29	0.11	0.07	0.28	0.09	0.04	0.26	0.06
Performance on Documents from Unseen Sites																		
FX	0.31	0.66	0.35	0.48	0.75	0.53	0.56	0.79	0.60	0.60	0.79	0.62	0.60	0.77	0.61	0.63	0.78	0.62
NFX	0.30	0.31	0.29	0.36	0.38	0.36	0.39	0.38	0.36	0.41	0.40	0.38	0.42	0.40	0.38	0.43	0.41	0.39
C4.5	0.28	0.32	0.28	0.38	0.48	0.40	0.43	0.56	0.45	0.46	0.60	0.49	0.48	0.63	0.51	0.51	0.66	0.53
NB	0.01	0.01	0.01	0.26	0.35	0.27	0.31	0.49	0.34	0.34	0.56	0.37	0.34	0.62	0.38	0.32	0.65	0.35
XA	0.01	0.01	0.01	0.04	0.07	0.04	0.05	0.12	0.06	0.04	0.14	0.04	0.03	0.14	0.03	0.01	0.11	0.01

Table 3.2: The average precision (P), recall (R) and F-measure (F1) of different approaches on seen and unseen sites, for different numbers of sites (s) in the dataset

	$s = 1$	3 $s = 2$	$s = 3$	$s = 4$	$s = 5$	$s = 6$
author	0.18	0.25	0.52	0.68	0.78	0.82
price	0.36	0.62	0.66	0.66	0.62	0.55
title	0.94	0.92	0.93	0.93	0.96	0.98

Table 3.3: The average F-measure (F1) for forgiving XPaths (FX) on different book attributes in unseen sites.

	$s = 1$			$s = 2$			$s = 3$			$s = 4$		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
FX	0.53	0.88	0.58	0.72	0.88	0.74	0.81	0.88	0.80	0.87	0.93	0.86
NFX	0.52	0.52	0.51	0.69	0.67	0.67	0.78	0.75	0.75	0.85	0.84	0.82
C4.5	0.46	0.48	0.46	0.53	0.75	0.59	0.52	0.80	0.58	0.55	0.84	0.60
NB	0.11	0.27	0.13	0.18	0.59	0.23	0.18	0.71	0.23	0.15	0.76	0.19
XA	0.04	0.04	0.04	0.17	0.23	0.17	0.14	0.20	0.14	0.13	0.21	0.12

Table 3.4: The average precision (P), recall (R) and F-measure (F1) of different approaches trained on a set of size s of page versions and tested on the rest.

	FX						XA					
	Name			Price			Name			Price		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
currys	1.00	1.00	1.00	0.92	1.00	0.96	1.00	1.00	1.00	1.00	0.65	0.79
pricespy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.26	0.42
bestbuy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.99
pricerunner	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.67	0.80	1.00	0.67	0.80
ebuyer	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.26	0.42	1.00	1.00	1.00
overall	1.00	1.00	1.00	0.98	1.00	0.99	1.00	0.79	0.84	1.00	0.71	0.80

Table 3.5: Performance of forgiving-XPath (FX) and Alignment based approach (XA) on the XPath generated data set (DS3)

In addition, we used our approach (and the baselines) for synthesizing a *single* generalized query for *multiple* sites. In particular, this allowed us to evaluate the robustness of our approach to structural differences in the training set. Such robustness is important to allow training the tool on a larger training set with a variety of websites without harming accuracy.

Accuracy on Different Page Versions To evaluate the robustness of queries generated by our method (and other baselines) to structural changes in the pages, we synthesized the queries using some versions of the page and evaluated them on different versions. Robustness to future changes on the page structure is important, as a more robust query will break less frequently.

Accuracy on Unseen Sites One major advantage of our technique compared to other XPath-based extraction techniques (in addition to its ability to synthesize a single query for multiple sites) is its generalization ability and the flexibility of its resulting queries. Other XPath based techniques work on pages from a single site, and generate XPath expressions that are strictly related to the structure of these pages.

To evaluate the generalization ability of our approach, we evaluated the extractors on unseen sites. Note that our goal is only to compare the generality of queries synthesized by our method to the other baselines; we do not expect the accuracy on unseen sites to be as good as for seen sites, as this is a much more challenging task.

3.5.4 Methodology

We evaluated our approach on seen and unseen sites by applying it on sets of sites from the DS1 dataset of different sizes (i.e., with different number of sites in the training set). We denote the number of sites in the training set in each experiment by s.

For every category among books, shopping, hotels and movies, for which the dataset contains documents from 7 to 8 sites (we denote by n the number of sites in the category), we considered all the possible $\binom{n}{s}$ subsets of size s of sites in the dataset, where $s \in [1, 6]$ (while s can be as large as 8, we limit it to 6 to be consistent with the unseen case). For each subset of s sites, we created the subset of the dataset restricted to the pages of the selected sites. We denote the corresponding subset of the dataset by D. For evaluating the performance on seen sites, we split D to two disjoint subsets D_t (for training) and D_e (for evaluation). Each of the two subsets D_t and D_e contains document pages from each website. Hence the training set and the evaluation set refer to exactly the same sites (allowing us to evaluate the performance on seen sites), but they have no documents in common. We used our approach to synthesize an extraction query with D_t as a training set, and evaluated the precision, recall and F-measure of the resulting query on D_e.

In addition, we used DS3 to compare the performance of our approach to that of the alignment based approach [NDMBDT14]. For each site and attribute (among product name and price) DS3 contains 1005 pages, five of which are used for training and the rest are used for testing.

For evaluating the performance on unseen sites, we used our approach to synthesize an extraction query with D as a training set and evaluated the precision, recall and F-measure of the resulting query on $\overline{D}$. The set $\overline{D}$ contains documents in the dataset from the rest of the sites from the same category (these sites are considered *unseen* – as their documents are not in D).

We did the same for all the other baseline approaches we have implemented. For every value of $s \in [1, 6]$ and for every category, we calculated the average precision, recall and F-measure over the different subsets D of the dataset obtained with s sites for both seen and unseen cases. Finally, for each value of $s \in [1, 6]$, we calculated the average over all categories. The results obtained for the two variants of our approach, as well as for the baselines, are reported in Table 3.2, Fig. 3.4 and Fig. 3.5 (as a function of s).

To evaluate our approach (and the baselines) on different page versions, we used a similar process as before, this time on pages from the DS2 dataset. In these experiments, s refers to the number of versions considered in the training set, D consists of a subset of page versions and $\overline{D}$ consists of the remaining versions. Results are reported in Table 3.4 and Fig. 3.6.

3.5.5 Results

Results for Seen Sites The results for performance on seen sites are reported in Fig. 3.4 and Table 3.2. The results show that the XPath queries generated by our approach (FX and NFX) have the best accuracy (F-measure) among the different approaches for all different values of s, well ahead of the closest competing approach C4.5. Both classifier based methods (C4.5 and NB) suffer from loss in precision, especially for higher s values. The alignment-based

XPath generation (XA), which has the lowest precision and recall among all the approaches, suffers from significant decrease in precision for $s > 1$. According to Table 3.1, which reports the performance loss on seen sites as a function of the number of sites in the training set, our forgiving XPath method (FX) is the most robust approach to diversity in the training set. The results for performance of the alignment based XPath generation (XA) and our forgiving-XPath (FX) on DS3 are reported in Table 3.5. The XPath queries generated by our approach have better precision and recall than those generated by the alignment based approach. However, both approaches perform better on DS3 than on DS1 (our approach has perfect recall and close to perfect precision). This is because pages in DS3 were annotated using XPaths while pages in DS1 were manually annotated. It is therefore hard even for a human programmer to write an XPath that accurately extracts the data from them.

Results for Different Page Versions The results of invoking the data extractors on different (older and newer) versions of pages are reported in Fig. 3.6 and Table 3.4. The results show that when trained with a large enough set of different archived versions of a page, our forgiving-XPath (FX) has an F-measure of 0.86 on newer (different) yet unseen version of the page (0.93 recall and 0.87 precision, which is close to its performance on seen pages). This shows the usefulness of our approach for generating a robust extractor when used on a set containing a variety of previous versions of a target page. That is, it is more tolerant to structural changes in future versions of the page. Classifier-based methods show poor precision for higher s values, making them unsuitable for such an application (of generating robust extractor). Alignment-based XPath generation (XA) has the lowest scores (for all values of s).

Results for Unseen Sites Fig. 3.5 shows that our forgiving XPath approach (FX) has the best accuracy (F-measure) for every number of sites in the training set. Table 3.2 shows that our forgiving XPath technique significantly outperforms the other approaches in terms of both precision and recall.

While the non-forgiving XPath (NFX) performs well (very close to the forgiving XPath), and outperforms C4.5 on seen sites, it is outperformed by C4.5 in the case of unseen sites. This is because we use a pruned C4.5 tree, while we keep our non-forgiving XPath strict. This is also the reason that C4.5 is the closest competing approach to NFX. This highlights the trade-off between accuracy on seen and unseen sites when pruning is used (or not, as in NFX).

Table 3.3 shows an interesting aspect of the evaluation. The accuracy results that we report are averaged over all categories and all attributes extracted from a site. However, it is common for some attributes to be more robust and generalizable than others. For example, in sites listing books, the accuracy of extracting the book author and the book title is very high, and improves as more sites are added to the training set. In contrast, the accuracy of extracting the price, might deteriorate with more sites, as the sites may really have very little in common in the way that a price is represented (sites may even differ on the number of prices that they present, e.g., special discounts, shipping, etc.).

3.5.6 Discussion

The evaluation results show that forgiving XPath queries, synthesized by our approach, beat the accuracy of other approaches for all the three different performance aspects.

While some approaches (like NB and C4.5) have a trade-off between precision and generality (performance on seen and unseen sites), the use of forgiving XPaths enables our approach to have good performance on unseen sites without noticeable loss of precision on seen sites. In addition, the robustness of our approach to the structural differences of documents in the training set enables using datasets with high structural variety (containing different versions of the page), which improves the robustness of the synthesized XPath to future structural changes.

3.6 Related Work

There has been a lot of work on data extraction from web pages. In the following we survey closely-related work.

XPath Data Extractors The adaptation of XPath based data extractors has been studied widely. Several works on automatic generation of XPath based data extractors have been proposed [RGSL04, NDMBDT14, NBdT16, ZSWG09, Ant05]. Nielandt et al. [NDMBDT14] propose a method for generating XPath wrappers, which revolves around aligning the steps within the XPaths. Given a set of XPath samples describing the data nodes of interest in a DOM tree, the method uses an alignment algorithm based on a modification of the Levenshtein edit distance to align the sample XPaths and merge them to a single generalized XPath. The XPaths generated by this method are not robust to structural changes. Our evaluation show that our technique outperforms this method both on seen and unseen sites. Their latest work [NBdT16] builds on [NDMBDT14] and enriches the resulting generalised XPaths with predicates, based on context and structure of the data sources, to improve the precision of the resulting XPath on the training data with minimal recall decrease. While this method generates richer XPaths compared to [NDMBDT14], and yields more precise queries in some cases, the resulting XPaths are still too specific and suffer from the same robustness issue. This results from using an alignment based algorithm which finds the most specific generalized XPath, keeping unnecessary features as long as they do not affect its recall. Zheng et al. [ZSWG09] propose a record-level wrapper system. The generated wrappers include the complete tag-path from the root to record boundary nodes, making them sensitive to tag modifications. The major weakness of these techniques is related to the lack of flexibility of their generated XPath queries. These techniques work on pages from a single site, and generate XPath expressions that are strictly related to the structure of the pages from the site on top of which they are defined. Omari et al.[OSY16] propose a method for generating XPath extraction queries for a family of websites that contain the same type of information. They learn the logical representation of a website by utilizing the shared data instances among these sites, which allows them to identify commonality even when the structure is different.

Robustness There has been some work on the problem of XPath *robustness* to site changes [DBS09b, LSRT14, LWLY12], trying to pick the most robust XPath query for extracting a particular piece of information. The generalization applied by these techniques is based on alignment of XPaths, and picks a single query with a fixed generalization/precision tradeoff. In contrast, our approach uses decision-trees and forgiving XPaths to adjust precision dynamically.

Cohen et al. [CDB15] propose an extension of XPath called XTPath. XTPath stores additional information from the DOM tree and uses recursive tree matching to fix XPath wrappers automatically when shifts in the tree structure happen. Instead of constructing a robust XPath, the method uses the XTPath whenever an XPath fails.

Pattern based techniques Several works [CL01, AGM03, DKS11, LGZ03b, TW13] propose methods that use repeated pattern mining to discover and extract data records. Kayed et al. [KC10] propose a technique that works on a set of pages to automatically detect their shared schema and extract the data. Their method (called FivaTech) uses techniques such as *alignment* and *pattern mining* to construct a "fixed/variant pattern tree," which can be used to construct an extraction regular expression. DEPTA [ZL05] uses partial tree alignment to align generalized nodes and extract their data. DeLa [WL03] automatically generate regular expression wrappers based on the page HTML-tag structures to extract data objects from the result pages. IEPAD [CL01] discovers repeated patterns in a document by coding it into a binary sequence and mining maximal repeated patterns. These patterns are then used for data extraction. Omari et al. [OKYS16] propose a method for separating webpages into layout code and data. Their method uses a tree alignment based algorithm to calculate the cost of the shared representation of two given sub-trees, then it uses the calculated shared representation cost to decide whether it should fold them or not in order to minimize the representation cost of the given DOM tree. The result of the folding process is a separation of the page to a layout-code component and a data component. Chang el al. [CCCD16] propose a system for page-level schema induction, and uses wrapper verification to extract data. Given a large amount of webpages, they use a subset of these pages to learn the schema in unsupervised manner, and then use scheme verification procedure to extract data from the rest of the pages.

Model based techniques Several approaches relying on Machine Learning algorithms were presented for data extraction [HCPZ11, FK04, SWL$^+$12, Kus00]. HAO et al. [HCPZ11] present a method for data extraction from a group of sites. Their method is based on a classifier that is trained on a seed site using a set of predefined feature types. The classifier is then used as a base for identification and extraction of attribute instances in unseen sites. [FK04] foucses on detecting the boundaries of interesting entities in the text and treats information extraction as a classification problem. It uses an SVM classifier to identify the start and end tags of the data. Song et al. [SWL$^+$12] propose a dynamic learning framework to extract structured data of various verticals from web pages without human effort. The technique is based on the observation that there is adequate information with similar layout structures among different web sites that can be learned for structured data extraction. In contrast to model based techniques, our method synthesizes a (standard) XPath query, that is human readable and easy to understand and modify, while maintaining the generalization ability and flexibility of these techniques. In addition, the use of forgiving XPath queries enables our technique to avoid the tradeoff that classifier based techniques have, between precision on seen sites and precision on unseen sites.

3.7 Conclusion

We have presented, and implemented, a novel approach for synthesizing a robust cross-site data extractor for a family of sites.

The cross-site extractor that we synthesize combines the benefits of generalization based on decision tree learning, with the efficient realization of a query as a forgiving XPath that can be directly and efficiently executed by browsers (and extraction tools).

As our experiments show, the synthesized extractor manages to remain precise for the training set, despite its generality. In addition, it can extract information from unseen pages and sites with a relatively high accuracy due to its novel forgiveness property, which allows it to *dynamically adjust its precision* for the web page at hand. Interestingly, our extractors outperform not only other pattern-based extractors, but also classifier-based extractors which are typically more suited for handling unseen sites. This is achieved while keeping the many benefits of XPath queries such as readability and efficient execution.

Chapter 4

Lossless Separation of Web Pages into Layout Code and Data

Abstract A modern web page is often served by running layout code on data, producing an HTML document that enhances the data with front/back matters and layout/style operations. In this paper, we consider the opposite task: separating a given web page into a data component and a layout program. This separation has various important applications: page encoding may be significantly more compact (reducing web traffic), data representation is normalized across web designs (facilitating wrapping, retrieval and extraction), and repetitions are diminished (expediting updates and redesign).

We present a framework for defining the separation task, and devise an algorithm for synthesizing layout code from a web page while distilling its data in a *lossless* manner. The main idea is to synthesize layout code hierarchically for parts of the page, and use a combined program-data representation cost to decide whether to align intermediate programs. When intermediate programs are aligned, they are transformed into a single program, possibly with loops and conditionals. At the same time, differences between the aligned programs are captured by the data component such that executing the layout code on the data results in the original page.

We have implemented our approach and conducted a thorough experimental study of its effectiveness. Our experiments show that our approach features state of the art (and higher) performance in both size compression and record extraction.

4.1 Introduction

Many modern webpages are served by applying *layout code* to structured data, producing an HTML page that is presented to the user. The resulting HTML page contains both formatting elements inserted by the layout code, and values that are obtained from the structured data. The page is therefore a blend of formatting elements, and actual data.

Goal Our goal is to *separate* a given HTML page into a *layout code* component and a data component such that: (i) the separation is *lossless*, running the extracted layout code on the extracted data reproduces the original page, and (ii) the separation is *efficient* such that common elements become part of the layout code, and varying values are represented as data.

This separation has various important applications: page encoding may be significantly more compact (reducing traffic), data representation is normalized across different Web designs (facilitating wrapping, retrieval and extraction), and repetitions are diminished (expediting updates and redesign). Many of these applications are not limited to static web pages but can also be applied to dynamically generated pages (e.g., by using a headless browser to obtain a static HTML page).

Existing Techniques

There has been a lot of past work on data extraction from web pages [AGM03, KC10, SC13, SC$^+$14, TW13, WL03, CL01, ZL05, LMM06, SL05, LMM10, CYWM03, LGZ03a, LWLY12, DBS09b, HCPZ11, MTH$^+$09]. Techniques have also been presented for *wrapper induction*, using the template structure of a page to produce a wrapper that extracts particular data elements [KWD, FFT05, CMM01]. While the use of templates is essential for improving uniformity, readability, and maintainability of web-pages, templates are considered harmful for many automated tasks like semantic-clustering, classification and indexing by search engines. Therefore, a lot of past work tackled the challenges of template identification [RGSL04, LGZ03a, ZL05, BYR02, KFN10] and template-extraction [KS11, VdSP$^+$06, CKP07, GPT05, GF14, DC, GM13]. Typically, the goal of these works is to identify or extract the template so it can be ignored/discarded, and the data could be passed to further processing.

Separation combines, and generalizes, two aspects of the extraction problem that are typically considered separately—record extraction, and template extraction—and seeks to balance them. Rather than treating the template as noise when extracting data, or eliminating data when extracting a template, separation seeks to extract both at the same time. The separation algorithm we present extracts *layout code* and not a static template. Furthermore, it attempts to maintain a balance between the quality of extracted layout code, and the structure of the extracted data.

Our Approach We present, implement, and evaluate a method for automatically separating a static template-generated HTML page into a template layout-code and data. The main idea of the separation algorithm is to synthesize layout code hierarchically for parts of the page, and use a combined program-data representation cost to decide whether to align intermediate programs. When intermediate programs are aligned, they are transformed into a single program, possibly with loops and conditionals. At the same time, differences between the aligned programs are captured by the data component.

In contrast to previous work on template extraction, which identifies template-chunks to remove or extract as features, we synthesize fully working template-code which when invoked on the extracted data reproduces the original static HTML page. There are many possible ways to represent a page as a layout-code and data. We guide our choice of separation by attempting to minimize the joint representation size of the page, according to the MDL [Ris78] principle. Our approach could be applied to any form of tree-structured data, and can be used for applications such as tree retrieval [LLYZ08].

We have implemented our approach in a tool called SYNTHIA, and conducted a thorough experimental study of its effectiveness. Our experiments show that SYNTHIA features state of the art (and higher) performance in both size compression and record extraction.

4.2 Related Work

There has been a considerable amount of work on page-level data extraction (e.g., [AGM03, KC10, SC13, SC$^+$14, WLF15, CKGS06, FWB$^+$11b]) and record-level extraction (e.g., [TW13, WL03, CL01, ZL05, LMM06, SL05, LMM10, CYWM03, LGZ03a, FWB$^+$11a]). In the following, we focus on closely related work.

A lot of related work has dealt with the problem of *template extraction* or *wrapper induction*, for *record extraction*. As such, the focus has been on templates that are based on regular expressions, and more importantly, the resulting separation into templates and records is lossy; that is, we cannot recover the original HTML document from the output records and template. We focus on lossless separations into data and code, where the code involves (nested) loops and conditions. The notion of *wrappers* and *patterns* is different from our notion of *code*, since the former describes how to access the DOM tree to extract data, whereas the latter states how data is processed to generate the DOM tree. Moreover, we aim at finding separations of a short description; traditionally there has not been much focus on the complexity of the template, but rather mainly on its ability to perform high-quality record extraction. Another distinction between our solution and many existing ones is that those require multiple different pages (of the same template) as input, whereas our solution already works on a single page. Next, we describe some of these systems.

FiVaTech [KC10] works on a set of pages to automatically detect their shared schema and extract the data. Their solution applies several techniques such as *alignment* and *pattern mining* to construct a structure called "fixed/variant pattern tree," which can be used to identify the template and detect the schema. TEX [SC13] extracts data, but does not extract the template or the schema of the data. Trinity [SC$^+$14] builds on TEX and improves it by using the template tokens to partition the document into prefixes, separators and suffixes. They recursively analyze the results to discover patterns and build a "Trinity tree," which is later transformed into a regular expression for data extraction. RoadRunner [CMM01] uses a matching algorithm to identify differences between the input documents and build a common regular expression. It starts by considering one page as a wrapper, and matches it with another page. It then refines it using generalization rules to compensate for mismatches. EXALG [AGM03] uses the concept of equivalence classes and "differentiating roles" to discover a template, which is a regular expression. TPC [MTH$^+$09] considers a web document as a string of HTML tag paths. It detects the repeated patterns of tags paths called "visual signals" within a page, clusters them based on a similarity measure that captures how closely the visual signals appear in the document. For each one of the clusters the method uses the paths of its visual signal to extract records from the page. RSP [TW13] takes as an input a web page and a sample subject string which is used to help identify subject nodes. The method uses the repetitive pattern of subject items in a page to identify the boundary of data records. It aligns data records to find a generalized pattern, which is used to generate a wrapper. The method generates a template wrapper that describes the location of data records and can be used to extract them.

MDR [LGZ03a] and DEPTA [ZL05] use tag strings representation of DOM nodes to

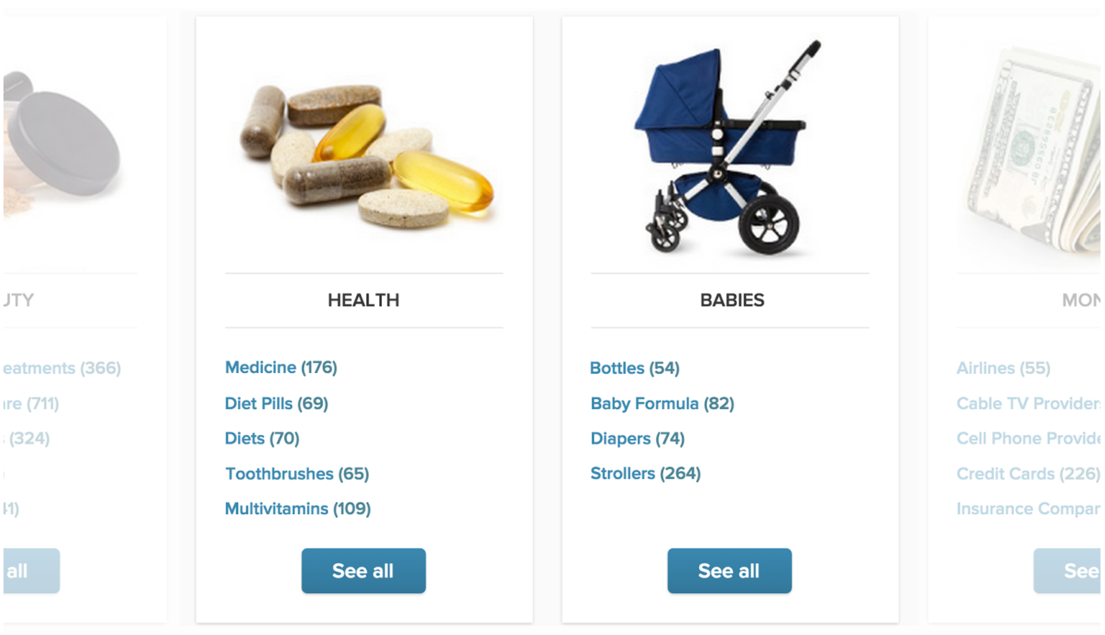


Figure 4.1: A simple webpage.

compare individual nodes and combinations of adjacent nodes. Similar individual nodes or node combinations are considered as generalized nodes, and sequences of generalized nodes are considered as a data region. DEPTA [ZL05] uses partial tree alignment to align generalized nodes and extract their data. DeLa [WL03] automatically generate regular expression wrappers based on the page HTML-tag structures to extract data objects from the result pages. IEPAD [CL01] discovers repeated patterns in a document by coding it into a binary sequence and mining maximal repeated patterns. These patterns are then used for data extraction.

In contrast to alignment algorithms used in other works (e.g., [KC10, MTH$^+$09, ZL05, TW13]), our tree-alignment algorithm operates on layout code trees with their data, and updates both the code and the data components. In addition, as opposed to classical alignment algorithms, which define a fixed cost per alignment operation, the costs in our algorithm are context dependent.

To the best of our knowledge, none of the related works are able to produce runnable layout code and provide *lossless separation* into layout code and data.

4.3 Overview: Problem and Solution

In this section, we give an informal overview of the problem we formulate in this paper, and of our solution SYNTHIA. We provide a formal treatment in the following sections.

4.3.1 Motivating Example

Fig. 4.1 shows part of a navigation web page taken from `www.viewpoints.com/explore` (we focus on a part of a page for illustrative purposes; the solution of this paper works on full pages). Given this page, our goal is to separate it to a *layout-code* component, and a *data* component. Technically, a *layout tree* is a tree representation of a program that formats data into a web page. We formally define layout trees in Section 4.4.

```
<div><img src="/health.jpg"/><h2>HEALTH</h2>
<ul class="stripped">
<li><a href="/Medicine">Medcicine</a>(176)</li>
<li><a href="/Diet␣Pills">Diet Pills</a>(69)</li>
<li><a href="/Diets">Diets</a>(70)</li>
<li><a href="/Toothbrushes">Toothbrushes</a>(65)</li>
<li><a href="/Multivitamins">Multivitamins</a>(109)</li>
</ul>
</div>
<div><img src="/babies.jpg"/><h2>BABIES</h2>
<ul class="stripped">
<li><a href="/Bottles">Bottles</a>(54)</li>
<li><a href="/Baby␣Formula">Baby Formula</a>(82)</li>
<li><a href="/Diapers">Diapers</a>(74)</li>
<li><a href="/Strollers">Strollers</a>(264)</li>
</ul>
</div>
```

Figure 4.2: A sample static html snippet that we would like to separate into code and data.

Fig. 4.2 shows the HTML document of Fig. 4.1. This HTML contains repeated formatting elements for the listed items. For example, the items `Medicine`, `Diet Pills`, `Diets`, `Toothbrushes`, and `Multivitamins` are formatted in a similar HTML structure.

The HTML document can be viewed as a DOM tree [W$^+$98]. Fig. 5.5(a) shows the DOM tree for the HTML document of Fig. 4.2. In this tree, the subtrees of the `div` elements share a similar structure, and so do the subtrees under the `ul` elements. SYNTHIA is able to detect these common structures, synthesize the corresponding layout trees, and extract hierarchical data that captures the different contents that are laid in the common structures.

Synthesized layout tree Fig. 5.2 shows the code synthesized by our technique for the page of Fig. 4.2. The code can also be viewed in tree form as the layout tree shown in Fig. 5.5(b).

This code uses two iteration (`for`) instructions to create a nested loop structure that is used to format the data. Our layout tree uses standard control constructs common in any layout language, and uses a syntax similar to JSP. The tree refers to *variables*, such as `f1` and `v1`, that are assigned actual values in the extracted data.

Extracted data Fig. 5.3 shows the data extracted by our technique. Data is extracted as a hierarchical structure, where data elements are labeled by their corresponding loop or variable. For example, the data elements under the label `f1` are the elements that are iterated over by the `for` operation in line 1 of the extracted code. The data elements under the label `f2` (in lines 3 and 7 of Fig. 5.3) are the elements that are iterated over by the `for` operation in line 4 of the code. As our data is viewed as an assignment of values to variables that are used in the layout tree, we refer to a data instance as an *environment*. An important feature of our approach is the fact that the separation is lossless—executing the synthesized layout tree on the extracted environment reproduces an exact copy of the original HTML document. This should be contrasted with common lossy techniques for wrapper induction and record extraction.

4.3.2 Our Approach

From a high-level perspective, SYNTHIA works by *folding adjacent subtrees* of a layout tree. Initially, the layout tree is simply the DOM tree representing the web page. As subtrees are being folded, we synthesize unified code that represents their common structure, and create separate data elements to represent their different values. There are two trivial solutions to this

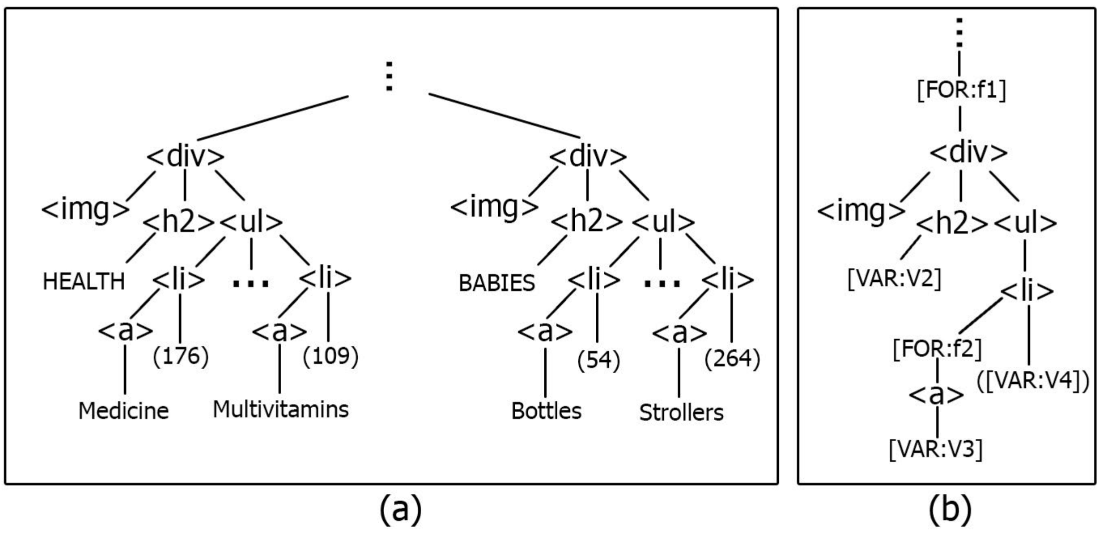


Figure 4.3: (a) The DOM tree of the original HTML document, and (b) the layout tree produced by our approach from this DOM tree.

```
<% for(var loop1:f1){ %>
<div>
<img src="/<%=loop1.v1%>.jpg"/><h2><%=loop1.v2%></h2>
<ul class="stripped">
<% for(var loop2:f2){
<li><a href="/<%=loop2.v3%>">
<%=loop2.v3%></a>(<%=loop2.v4%>)</li>
<% } %>
 </ul>
 </div>
<% } %>
```

Figure 4.4: Code synthesized for the given static HTML.

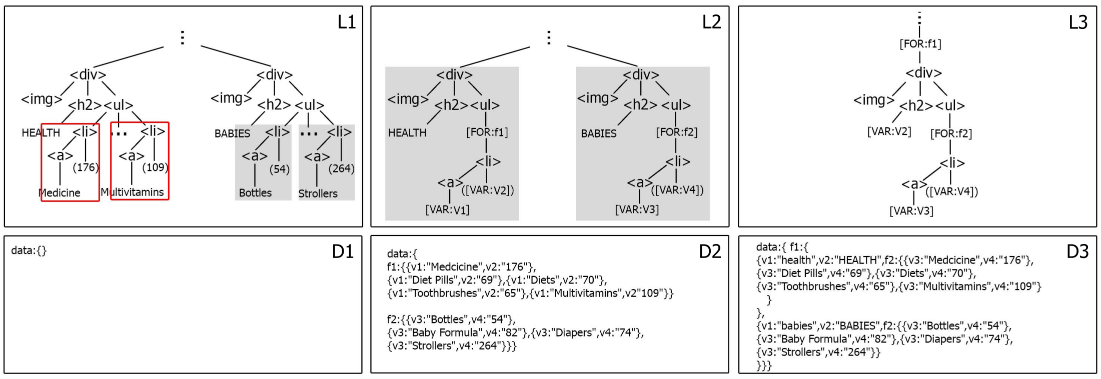
```
f1:{
 {v1:"health",v2:"HEALTH",
  f2:{{v3:"Medcicine",v4:"176"},{v3:"Diet_Pills",v4:"69"},
      {v3:"Diets",v4:"70"},{v3:"Toothbrushes",v4:"65"},
      {v3:"Multivitamins",v4:"109"}}
 },
 {v1:"babies",v2:"BABIES",
  f2:{{v3:"Bottles",v4:"54"},{v3:"Baby_Formula",v4:"82"},
      {v3:"Diapers",v4:"74"},{v3:"Strollers",v4:"264"}}
}}
```

Figure 4.5: The data extracted for the given static HTML.

problem. The first is where all subtrees are folded, forcing a single layout tree and effectively pushing all differences into the data. The other trivial solution applies no folding at all, and then the layout tree effectively dumps the entire web page. Naturally, we are not interested in the trivial solutions, but rather in a solution that *minimizes the representation cost*. Intuitively, this means that we should only fold subtrees when they share a sufficiently common structure.

To find a folding that minimizes the representation cost, we have to answer two technical questions:

- *When* should we fold given layout subtrees?



```
data:{}
```

```
data:{
f1:{{v1:"Medcicine",v2:"176"},
{v1:"Diet Pills",v2:"69"},{v1:"Diets",v2:"70"},
{v1:"Toothbrushes",v2:"65"},{v1:"Multivitamins",v2"109"}}

f2:{{v3:"Bottles",v4:"54"},
{v3:"Baby Formula",v4:"82"},{v3:"Diapers",v4:"74"},
{v3:"Strollers",v4:"264"}}}
```

```
data:{ f1:{
{v1:"health",v2:"HEALTH",f2:{{v3:"Medcicine",v4:"176"},
{v3:"Diet Pills",v4:"69"},{v3:"Diets",v4:"70"},
{v3:"Toothbrushes",v4:"65"},{v3:"Multivitamins",v4:"109"}
  }
},
{v1:"babies",v2:"BABIES",f2:{{v3:"Bottles",v4:"54"},
{v3:"Baby Formula",v4:"82"},{v3:"Diapers",v4:"74"},
{v3:"Strollers",v4:"264"}}
}}}
```

Figure 4.6: Example steps of the separation algorithm.

- *How* should we fold such subtrees to produce the desired separation of code and data?

We address both of these questions using a novel *alignment* algorithm. We use the alignment algorithm as a building block for deciding when to fold subtrees, and also for computing the separation into layout tree and data when subtrees are folded.

Subtree folding In a bottom-up manner, we analyze adjacent layout subtrees by evaluating their structural similarity. This is done by calculating the *representation cost* for representing the subtrees using a shared single layout tree and two data components. That is, we estimate the benefit of forcing the subtrees into using the same layout tree with separate data.

To that end, we use our alignment algorithm to compute an alignment that attempts to minimize the resulting representation cost, and also returns the cost. We fold together adjacent subtrees which are found to be similar (based on the calculated shared representation cost). Folding is done by (i) applying the calculated alignment. The result may include conditional instructions and variable references in text nodes and attributes to overcome differences; (ii) introducing a `for` instruction whose body is the resulting shared layout tree while verifying losslessness (i.e., when invoked on the data, the result is the same as that of the sequence of folded subtrees).

Alignment We present a novel tree-alignment algorithm, that is tailored to handling layout trees, enabling it to handle loops, conditions and variables in the template. In addition, we enable it to perform data extraction and modifications to the data representation, in order to fit changes in the layout trees, so that invoking the layout trees on the data will result in the original HTML.

Example 4.3.1. Fig. 4.6 shows a few steps of our algorithm applied to the (partial) tree of Fig. 5.5. Initially, the layout tree is the original DOM tree (L1), with no folding, and no extracted data (D1). The algorithm works in a bottom-up manner, looking for folding opportunities. The algorithm detects that the subtrees rooted at list items (`<li>`) for `Medicine`,`Diet Pills`,`Diets`,`Toothbrushes`, and `Multivitamins` could be folded with common structure and extracting the varying data. The algorithm folds the subtrees corresponding to the following items:

```
<li><a href="/Medicine">Medcicine</a>(176)</li>
```

```
<li><a href="/Diet␣Pills">Diet Pills</a>(69)</li>
<li><a href="/Diets">Diets</a>(70)</li>
<li><a href="/Toothbrushes">Toothbrushes</a>(65)</li>
<li><a href="/Multivitamins">Multivitamins</a>(109)</li>
```

By introducing new layout trees and extracted data. The layout tree is as follows:

```
<% for(var loop:f1){
  <li>
  <a href="/<%=loop.v1%>"><%=loop.v1%></a>
    (<%=loop.v2%>)
  </li>
<% } %>
```

This synthesized layout tree uses variables `v1` and `v2` to refer to data elements in the extracted data. The synthesized code is shown in Fig. 4.6 (L2) as the subtree rooted at `FOR:f1`. The extracted data is shown at the bottom part of the figure (D2). The data is structured and is labeled by names corresponding to the variables in the layout tree. For example, the data maps the variable `f1`, used at `FOR:f1`, to a sequence of four possible values, each providing the data for one invocation of the loop body, resulting in one of the four aligned subtrees. The inner data values provide the interpretation of variables `v1` and `v2`.

This folding reduces the original combined description length of the code and data, as the template part that repeats in all four items is described only once, in the code, and only the differentiating details are described for each item (in the data).

The subtrees rooted at list items for `Bottles`,. . .,`Strollers` are folded in a similar manner, resulting with the layout subtree rooted at `FOR:f2` (this is also depicted in L2 in Fig. 4.6).

After creating the layout trees rooted at `FOR:f1` and `FOR:f2`, the algorithm proceeds by identifying that these subtrees could be folded together. This folding renames variables of the two subtrees to match each other (e.g., `f1` is renamed to `f2`, unifying it with the existing `f2` variable of the subtree on the right). Folding also introduces new variables, `v1` and `v2`, to account for differences (note that these are fresh variables, as the previously used `v1` and `v2` were renamed). Finally, folding introduces an additional external `for` loop with variable `f1` (recall that the previous `f1` was renamed). The produced layout tree is shown in Fig. 5.2. A graphical representation is shown in Fig. 4.6 (L3). The corresponding extracted data is shown in (D3). Note that folding also adds another layer to the data, corresponding to the nested loop structure.

In this simple example folding does not introduce conditional constructs. However, if, for example, all items in the first list had an additional attribute, the folding of the two `for` subtrees depicted in L2 would introduce a conditional construct guarded by a boolean variable, with “true” in the first loop and ”false” in the second.

4.3.3 Key Aspects

The example of the previous section highlights a few key aspects of our approach:

- **Lossless separation:** In contrast to other extraction schemes, the separation to layout tree and data performed by SYNTHIA is lossless. That is, applying the synthesized layout tree on the extracted environment reproduces the original web page.
- **Minimization of representation cost:** The separation computed by SYNTHIA is tailored to minimizing the description cost of the result.
- **Synthesis of loops and conditionals:** SYNTHIA synthesizes layout trees that may include loops to generalize repetition of layout across items. When some of the layout differs between items that could otherwise be formatted using looping code, SYNTHIA is able to insert conditional formatting. With that, SYNTHIA allows for a compact (and lossless) looping structure for formatting elements that exhibit a *loosely* similar structure.
- **Extraction of hierarchical data:** SYNTHIA supports nested repetitions (e.g., a list of categories, each containing a list of products) by allowing nesting of loops in data trees alongside hierarchical structures of environments.

4.4 Preliminaries and Model

In this section we formally define the notions of a webpage, data and template code which is used to generate webpages by invoking it on a given data.

DOM Trees We model an HTML document as a *DOM tree*, which is a tree of elements and textual values. Formally, a *DOM tree* is a rooted and ordered tree with two types of nodes. An *element node* has a *name*, and an *attribute set*, which is a mapping from a finite set of attribute names to values (strings). The children of an element node form an (ordered) sequence of nodes. A *text node* is associated with a textual value. We require all text nodes to be leaves (i.e., childless).

Environment As we explain later, we model the construction of DOM trees by executing programs over data. We model data by means of an *environment*, which is a hierarchy of assignments to variables. Formally, we assume an infinite set $\mathbf{Var}$ of *variables*. An *environment* is inductively defined as follows. It is a mapping from a finite set of variables to values, where a *value* is either (i) a text item, or (ii) a list of environments.

Layout Trees We now define our model of a program, namely the *layout tree*, that executes over an environment to produce a DOM tree. This model is very simple, and is straightforward to translate into common languages that embed code with HTML/XML (e.g., server side like ASP and JSP, or client side like Javascript, AngularJS and XSLT).

Recall that a DOM tree has two types of nodes: element and text nodes. A layout tree is similar to a DOM tree, except that it has a third type of nodes, namely *instruction nodes*. An instruction node v is associated with a *type* and a variable. The type of an instruction node can be one of three: *condition*, *iteration*, and *reference*. The variable of an instruction node is a member of $\mathbf{Var}$. We refer to an instruction node with the variable x and the type condition, iteration and reference as $\mathsf{if}(x)$, $\mathsf{for}(x)$ and $\mathsf{ref}(x)$, respectively. The root of a layout tree is either an element node or a text node.[1]

Semantics of a Layout Tree The result of executing a layout program π over an environment $\mathcal{E}$ is a DOM tree that we denote by $\pi(\mathcal{E})$. To define $\pi(\mathcal{E})$ formally, we need some notation.

A *DOM hedge* is a sequence of DOM trees. Similarly, a *layout hedge* is a sequence of layout trees, except that we allow each layout tree to be rooted at an instruction node. For hedges $h = t_1, \dots, t_k$ and $g = u_1, \dots, u_m$, we denote by $h \cdot g$ the hedge that is obtained by concatenating g to h (i.e., $t_1, \dots, t_k, u_1, \dots, u_m$). If v is a node and h is a hedge, then we denote by $v[h]$ the tree that is obtained by adding v to h as the root (with the roots of h being the children of v). If t is a tree with the root v, then we denote by t^{-v} the hedge that is obtained from t by removing v.

To define $\pi(\mathcal{E})$, we give a more general (inductive) definition of the semantics of executing a layout hedge Π over $\mathcal{E}$, again denoted by $\Pi(\mathcal{E})$, and is generally a DOM hedge.

- If Π consists of a single tree π with a non-instruction root v, then $\Pi(\mathcal{E})$ is the tree $v[\pi^{-v}(\mathcal{E})]$.
- If Π consists of a single tree π with the root $\mathsf{if}(x)$, then the following holds. If $\mathcal{E}(x)$ is defined and $\mathcal{E}(x) = 1$, then $\Pi(\mathcal{E})$ is the hedge $\pi^{-v}(\mathcal{E})$; otherwise, $\Pi(\mathcal{E})$ is the empty hedge.
- If Π consists of a single tree π with the root $\mathsf{for}(x)$, then the following holds. If $\mathcal{E}(x)$ is defined and $\mathcal{E}(x)$ is a list $(\mathcal{E}_1, \dots, \mathcal{E}_m)$, then $\Pi(\mathcal{E})$ is the hedge $\pi^{-v}(\mathcal{E}_1) \cdots \pi^{-v}(\mathcal{E}_m)$; otherwise, $\Pi(\mathcal{E})$ is the empty hedge.
- If Π consists of a single tree π with the root $\mathsf{ref}(x)$, then the following holds. If $\mathcal{E}(x)$ is defined and $\mathcal{E}(x)$ is a string, then $\Pi(\mathcal{E}) = \mathcal{E}(x)$; otherwise, $\Pi(\mathcal{E})$ is the empty hedge.
- If Π is a hedge $\pi_1, \dots, \pi_k$ where $k > 1$, then $\Pi(\mathcal{E})$ is the hedge $\pi_1(\mathcal{E}) \cdots \pi_k(\mathcal{E})$.

Finally, recall that a layout tree has a non-instruction root. Then the above definition implies that the result of executing a layout tree over an environment is always a single DOM tree.

4.5 Problem Definition

In this section we formally define the space of separation solutions, and the desirable separation solutions in that space.

4.5.1 Separation and Solution Space

Our goal is to describe a given DOM tree by a layout tree and an environment. Formally, a *separation* of a DOM tree t is a pair $(\pi, \mathcal{E})$, where π is a layout tree and $\mathcal{E}$ is an environment, such that $\pi(\mathcal{E}) = t$. *Separating* t is the process of constructing a separation $(\pi, \mathcal{E})$ of t. Note that a DOM tree may have many separations (in fact, infinitely many separations). We denote by $\mathbf{Sep}(t)$ the set of all separations of t.

$$\mathbf{Sep}(t) \overset{\text{def}}{=} \{(\pi, \mathcal{E}) \mid \pi(\mathcal{E}) = t\}$$

[1]Our approach creates layout trees by folding subtrees of a DOM tree. As the root is never folded, it remains a non-instruction node.

A special case of a separation in $\mathbf{Sep}(t)$ is the trivial one $(\pi, \mathcal{E})$ where π is identical to t and $\mathcal{E}$ is empty.

4.5.2 Separation Quality

Since there are many possible ways to separate a given DOM tree, it is important to define what makes one separation better than another. In this work, we define a quality metric that is inspired by the principle of *Minimal Description Length* (MDL) [Ris78]. According to MDL, one should favor the model that gives the shortest description of the observed data [HY01]. MDL is well-suited for dealing with model selection, estimation, and prediction problems in situations where the models under consideration can be arbitrarily complex, and overfitting the data is a serious concern [Grü07]. In particular, SYNTHIA aims at synthesizing a separation that minimizes the length of the representation of the separation. To define the *length* of the separation, we define a size measure for a given separation $(\pi, \mathcal{E})$. The description length of $(\pi, \mathcal{E})$ is defined based on the size in characters of the string representations of π and $\mathcal{E}$, denoted $\mathsf{sizeof}(\pi)$ and $\mathsf{sizeof}(\mathcal{E})$, respectively. Hence, we define the following: $\mathsf{cost}(\pi, \mathcal{E}) \stackrel{\text{def}}{=} \mathsf{sizeof}(\pi) + \mathsf{sizeof}(\mathcal{E})$.

Our algorithm (defined in the next section) does not guarantee a separation of minimal cost. Instead, it applies a heuristic approach that uses the above cost for guiding intermediate decisions along the way. We leave for future work the challenge of obtaining optimality and analyzing the associated computational complexity.

4.6 Our Approach

In this section we describe our algorithm for folding a DOM tree into a layout tree and an associated environment.

4.6.1 The General Separation Algorithm

Given a DOM tree t, our algorithm constructs the separation $(\pi, \mathcal{E})$ recursively, as we describe below. We denote the input DOM tree t as $v[t_1, \ldots, t_n]$ (that is, the root is v and it has n children, each is the root of a subtree t_i). The separation algorithm goes as follows.

1. Recursively separate each t_i into a separation $s_i = (\pi_i, \mathcal{E}_i)$.
2. *Split* $s_1, \ldots, s_n$ into m chunks $(s_1, \ldots, s_{j_1-1})$, $(s_{j_1}, \ldots, s_{j_2-1})$, …, $(s_{j_m}, \ldots, s_n)$ where, intuitively, each chunk consists of "similar" separations.
3. *Fold* each chunk $(s_{j_l}, \ldots, s_{j_{l+1}-1})$ into a single separation $(\pi'_l, \mathcal{E}'_l)$. Roughly speaking, π'_l will be rooted at a $\mathsf{for}(x)$ node, and $\mathcal{E}'_l$ will map its variable x to a list of environments (one for each of the folded trees) such that $\pi'_l(\mathcal{E}'_l) = \pi_{j_l}(\mathcal{E}_{j_l}) \cdots \pi_{j_{l+1}-1}(\mathcal{E}_{j_{l+1}-1})$. That is, executing π'_l on $\mathcal{E}'_l$ will result in the same hedge as the concatenation of the hedges obtained by executing the layouts of each s_i in the chunk on its environment.
4. Return the separation $(\pi, \mathcal{E})$ where $\pi = v[\pi'_1 \cdots \pi'_m]$, and $\mathcal{E} = \mathcal{E}'_1 \cup \cdots \cup \mathcal{E}'_m$.

Note that in step 4, the different $\mathcal{E}'_l$ use pairwise-disjoint sets of variables; hence, their union $\mathcal{E}$ is a legal environment.

We denote by $\mathsf{fold}(s_1, \ldots, s_k)$ the procedure used in step 3 for folding a sequence $s_1, \ldots, s_k$ of separations $s_i = (\pi_i, \mathcal{E}_i)$ into a new separation $s = (\pi, \mathcal{E})$. Next, we explain how splitting and folding are implemented (In practice, they are weaved together).

4.6.2 Splitting

To split a sequence $s_1, \ldots, s_n$ of separations into chunks, we define a pairwise *similarity function* σ that assigns a score to each pair of separations. We define a chunk to be a maximal continuous subsequence $s_{j_l}, \ldots, s_{j_l+q_l}$ of $s_1, \ldots, s_n$ where $\sigma(s_i, s_{i+1})$ is larger than some fixed threshold for every $i = j_l, \ldots, j_l + q_l - 1$. That is, the chunks are broken where similarity is below the threshold. The similarity function σ is based on the fold procedure, as follows. For two separations s and s', let $s_f = \mathsf{fold}(s, s')$. Recall the definition of $\mathsf{cost}(s)$ in . Then $\sigma(s, s')$ is the relative reduction of cost gained by replacing s and s' with s_f; that is,

$$\sigma(s, s') = \frac{\mathsf{cost}(s) + \mathsf{cost}(s') - \mathsf{cost}(s_f)}{\mathsf{cost}(s) + \mathsf{cost}(s')} .$$

4.6.3 Folding

In the rest of the section, we describe the procedure fold. Recall that the input is a sequence $s_1, \ldots, s_k$ of separations $s_i = (\pi_i, \mathcal{E}_i)$, and the output is a single separation $(\pi, \mathcal{E})$ with the property that $\pi(\mathcal{E})$ is the hedge $\pi_1(\mathcal{E}_1) \cdots \pi_k(\mathcal{E}_k)$.

$\mathsf{fold}(s_1, \ldots, s_k)$ is performed by introducing a new $\mathsf{for}(x)$ node with a single child π^c. The single child π^c captures the common layout of $\pi_1, \ldots, \pi_k$. The differences between them are captured by conditional and reference nodes in π^c, along with an environment, $\mathcal{E}_i^c$, that is generated for each π_i. The environment $\mathcal{E}_i^c$ is based on $\mathcal{E}_i$ (the environment that π_i was accompanied with), but also includes the values of the new conditional and reference variables that are introduced in π_i^c. Finally, the output environment $\mathcal{E}$ that accompanies π is constructed by

$$\mathcal{E} = \{x \mapsto \mathcal{E}_1^c(x) \cdot \ldots \cdot \mathcal{E}_k^c(x)\}.$$

Remark. Splitting is aimed at identifying separations that will be unified by fold into a new for root with a *single* child that generates all of them (with proper environments). If the number of chunks exceeds some threshold (above 30% of the number of children number), we consider folding into a for node with $d > 1$ children. To do so, SYNTHIA looks for chunks in which separations in distance d from each other are similar. Folding is adapted accordingly to collapse separations in distance d from each other to one child of the for node (rather than collapsing all separations in the chunk to a single child). This enables SYNTHIA to deal with data items that correspond to a sequence of adjacent nodes in the tree.

The crux of folding is the construction of π^c (the child of the for node), along with the environments $\mathcal{E}_1^c, \ldots, \mathcal{E}_k^c$. This construction is done by applying on the input trees $\pi_1, \ldots, \pi_k$ and their environments $\mathcal{E}_1, \ldots, \mathcal{E}_k$ an *alignment* algorithm, which we describe next. Alignment operations may introduce conditional and reference nodes, and may align trees (or hedges) with for nodes, but they never introduce new for nodes. for nodes are introduced by folding (the procedure fold).

4.6.4 Alignment

We consider alignment of two layout trees with their environments. To handle a larger number of layout trees, we apply alignment incrementally: we first align two layout trees (and their environments), then align the result with another and so on, until all are aligned.

Intuitively, when given two separations $(\pi_1, \mathcal{E}_1)$ and $(\pi_2, \mathcal{E}_2)$, alignment unifies their layout trees by establishing a common layout tree and updating the environments. The result is a triple $(\pi', \mathcal{E}_1', \mathcal{E}_2')$ such that $\pi'(\mathcal{E}_1') = \pi_1(\mathcal{E}_1)$ and $\pi'(E_2') = \pi_2(\mathcal{E}_2)$. In order to allow an incremental alignment (as needed for the alignment of more than two layout trees), where we apply alignment on the result of a previous alignment which consists of two environments, we work with *environment series* $\mathbf{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_k)$ instead of environments $\mathcal{E}$. Alignment is defined inductively, and for that another generalization is required. Namely, instead of two layout trees π we work with two layout hedges Π. The need for this generalization will later become apparent. We denote by $\Pi(\mathbf{E})$ the series $(\Pi(\mathcal{E}_1), \ldots, \Pi(\mathcal{E}_k))$ of DOM hedges.

Definition 4.6.1. Let Π_1 and Π_2 be layout hedges and $\mathbf{E}_1$ and $\mathbf{E}_2$ be two environment series. An *alignment* of $(\Pi_1, \mathbf{E}_1)$ and $(\Pi_2, \mathbf{E}_2)$ is a triple $(\Pi', \mathbf{E}_1', \mathbf{E}_2')$ such that $\Pi'(\mathbf{E}_1') = \Pi_1(\mathbf{E}_1)$ and $\Pi'(\mathbf{E}_2') = \Pi_2(\mathbf{E}_2)$.

The objective of our alignment is to minimize the combined description length of the unified layout tree and the corresponding environments. We therefore define an alignment cost, similarly to the notion of separation cost:

$$\mathsf{cost}(\Pi', \mathbf{E}_1', \mathbf{E}_2') = \sum_{\pi \in \Pi'} \mathsf{sizeof}(\pi) + \sum_{\mathcal{E} \in \mathbf{E}_1'} \mathsf{sizeof}(\mathcal{E}) + \sum_{\mathcal{E} \in \mathbf{E}_2'} \mathsf{sizeof}(\mathcal{E})$$

Scope Environments

The most tricky part of the alignment is the update of the environments. To explain this update we need the following definitions.

Scope. Given a layout tree π, each iteration node in π defines a scope. The scope of a node v in π is determined by its lowest ancestor v_s which is an iteration node $\mathsf{for}(x)$ (or by the root if no such ancestor exists). In the former case, we say that v_s is the *scope node* and x is the *scope variable* of v. To simplify matters and prevent ambiguity, we do not allow two iteration nodes to have the same variable. We define the scope node and variable of a hedge similarly by considering the lowest common ancestor.

Scope environments. Given a layout tree π and an environment $\mathcal{E}$, the *scope environments* of a node v in π, denoted $S(v)$, are defined inductively based on the scope of v. If the scope of v is the root, then $S(v) = \{\mathcal{E}\}$. Otherwise, let v_s and x be the scope node and scope variable of v, respectively (i.e., v resides in the subtree of $v_s = \mathsf{for}(x)$). Then $S(v) = \bigcup_{\mathcal{E}_s \in S(v_s)} \{\mathcal{E}_i \mid \mathcal{E}_s(x) = \mathcal{E}_1 \cdot \ldots \cdot \mathcal{E}_m\}$. That is, that scope environments of v are all the environments in the lists that x is mapped to.

Example 4.6.1. Consider the layout tree L3 in Fig. 4.6 and the environment depicted in D3. The node `<li>` resides in the subtree of the node `FOR:f2`. Therefore, its scope environments are the nine environments consisting of the five environments in the first list of environments associated with variable `f2`: $\mathcal{E}_{11} = \{\texttt{v3} : \text{“}Medcicine''\text{,} \texttt{v4} : \text{“176''}\}, \ldots, \mathcal{E}_{15} = \{\texttt{v3} :$ “$Multivitamins''$,
$\texttt{v4} : \text{“109''}\}$, along with the four additional environments in the second list, $\mathcal{E}_{21} = \{\texttt{v3} :$ “$Bottles'', \texttt{v4} : \text{“54''}\}, \ldots, \mathcal{E}_{24} = \{\texttt{v3} : \text{“}Strollers'', \texttt{v4} : \text{“264''}\}$.

Alignment Operations

Alignment of two hedges Π_1 and Π_2, (usually these are children hedges of two nodes that are being aligned) with environment series $\mathbf{E}_1$ and $\mathbf{E}_2$ respectively, is performed using a dynamic programming algorithm. The algorithm advances along the two given hedges simultaneously and aligns their trees.

The operations considered by our alignment algorithm are: $Align$, $Skip$ and $AlignFor$, which we describe next. $Align$ and $Skip$ are conventional operations in alignment algorithms (unlike traditional alignments, in our case special care is taken to ensure that the alignment is lossless). The $AlignFor$ operation enables for-rooted trees to be aligned with a hedge rather than a single tree.

$Align$. aligns $(\pi_1, \mathbf{E}_1)$ with $(\pi_2, \mathbf{E}_2)$ where π_1 and π_2 are two single trees with *matching* roots, which means that they have the same name and type. In this case, we introduce a new root node v which unifies the roots v_1 and v_2 of π_1 and π_2 (as demonstrated below). The children of the new root are the result of recursively aligning the children hedges Π_1 and Π_2 into a hedge Π. In particular, the recursive operation might update $\mathbf{E}_1$ and $\mathbf{E}_2$.

For example, if v_1 and v_2 are both text nodes (meaning that Π_1 and Π_2 are empty) and $text(v_1) = text(v_2)$, then the unified root v is identical to (both of) them and $\mathbf{E}_1$ and $\mathbf{E}_2$ remain unchanged. However, if $text(v_1) \neq text(v_2)$, then v is a reference node of the form $\mathsf{ref}(x)$, where x is a fresh variable. For $i = 1, 2$, we add to each scope environment of v_i in $\mathbf{E}_i$ the mapping $x \mapsto text(v_i)$.

If v_1 and v_2 are $\mathsf{for}(x_1)$ and $\mathsf{for}(x_2)$, then v is $\mathsf{for}(x)$, where x is a fresh variable, and we update all the scope environments of v_1 and v_2 in $\mathbf{E}_1$ and $\mathbf{E}_2$ respectively by renaming every occurrence of x_i with x.

$Skip$. aligns $(\Pi_1, \mathbf{E}_1)$ with $(\pi_2, \mathbf{E}_2)$ where Π_1 is an empty hedge and π_2 is a tree by introducing a conditional node v of the form $\mathsf{if}(x)$, where x is a fresh variable. The alignment result is then the tree $\pi' = v[\pi_2]$, with $\mathbf{E}_1$ updated by adding the mapping $x \mapsto 0$ to each scope environment of Π_1, and $\mathbf{E}_2$ updated by adding the mapping $x \mapsto 1$ to each scope environment of π_2. Technically, in this case, we also receive the scope node (and variable) of Π_1 (the empty hedge) as input (in other cases this input is not needed since it is uniquely defined given the tree or hedge).

$AlignFor$. aligns $(\pi_1, \mathbf{E}_1)$ and $(\pi_2, \mathbf{E}_2)$ where π_1 is a for-rooted tree (which is possibly the result of alignment with previous trees from Π_2). Intuitively, the result of the alignment will be a for-rooted tree that in addition to the trees captured by π_1 also generates π_2. Repeated applications of $AlignFor$ enable aligning a for tree with a hedge. This operation has some resemblance to fold, yet it utilizes an existing for node (from π_1), rather than introducing a new one. The tricky part in this operation is that it breaks up existing scopes in π_2 due to the import of the for-node from π_1. As a result, a new hierarchical level is also created in the environments in $\mathbf{E}_2$. Due to space constraints we omit the detailed description.

Alignment Algorithm

Given $(\Pi_1, \mathbf{E}_1)$ and $(\Pi_2, \mathbf{E}_2)$ our algorithm computes an alignment while trying to minimize its cost. It also calculates the resulting cost. It uses dynamic programming to find the sequence of alignment operations which minimizes the alignment cost.

The algorithm gradually fills a two dimensional matrix B of size $n \times m$, where $n = |\Pi_1|$ and $m = |\Pi_2|$. For each $1 \leq i \leq n$ and $1 \leq j \leq m$ $B[i,j].\mathsf{cost}$ contains the minimal alignment cost of the prefix hedge Π_1^i of Π_1 of length i, and the prefix hedge Π_2^j of Π_2 of length j. $B[i,j].op$ contains the operation for π_1^i and π_2^j that resulted in the minimal cost.

The algorithm calculates $B[i,j].\mathsf{cost}$ and $B[i,j].op$ by calculating the cost of all possible alignment operations for π_1^i and π_2^j and by using the costs calculated in B for $i'<i$ and $j'<j$. The algorithm picks the option with the minimal cost. Finally, $B[n,m].cost$ is the alignment cost of $(\Pi_1, \mathbf{E}_1)$ and $(\Pi_2, \mathbf{E}_2)$.

The cost of operations The cost of an operation reflects the change in both the layout tree and environment costs. As the following example demonstrates, the latter is not fixed per operation, but depends on $\mathbf{E}_1$ and $\mathbf{E}_2$.

Example 4.6.2. Consider the alignment of two subtrees, π_l and π_r, where π_l is a subtree residing under a loop node $\mathsf{for}(x)$ which has 10 scope environments and π_r is an element subtree with a single scope environment. A conditional subtree insertion during the alignment will introduce a new conditional value in these 11 environments (one of π_r and 10 of π_l). As such, its effect on the cost depends on the number of scope environments.

We demonstrate the cost calculation on some of the operations.

$Align$ (aligning single trees). The algorithm recursively calculates the minimal alignment cost for $(\pi_1^i, \mathbf{E}_1)$ and $(\pi_2^j, \mathbf{E}_2)$, where π_1^i is rooted at v_1 and π_2^j is rooted at v_2.

- *Two text nodes alignment.* If v_1 and v_2 are text nodes with different text, the cost of applying the *two text nodes* alignment operation is $B[i-1,j-1].\mathsf{cost} + \mathsf{sizeof}(text_1)|S(v_1)| + \mathsf{sizeof}(text_2)|S(v_2)| - \mathsf{sizeof}(text_1) - \mathsf{sizeof}(text_2)$, where $S(v_i)$ is the set of scope environments of v_i in $\mathbf{E}_i$.
- *Two element nodes alignment.* If v_1 and v_2 are element nodes, the cost of applying the *two element nodes* alignment is $B[i-1,j-1].\mathsf{cost}$ plus the cost of aligning their children, subtracting the cost of one of them.

$Skip$ (aligning an empty hedge with a tree (to left)). We calculate the code cost of wrapping π_1^i with a conditional node v_c, and the data cost of updating every scope environment in $\mathbf{E}_1$ and $\mathbf{E}_2$.

We denote the cost of adding the conditional subtree and updating the environments as $\mathsf{cost}_c^{\mathsf{left}}$. Then the cost of applying the *skip* alignment operation is $\mathsf{cost}_c^{\mathsf{left}} + B[i-1, j].\mathsf{cost}$.

Example 4.6.3. Consider the folding of the left-most sequence of `<li>` nodes in the layout tree L1 from Fig. 4.6. Each of these nodes is accompanied by an environment capturing the mapping of the variables in its subtree. In our case these environments are initially empty, as the `<li>` subtrees have no variables (yet). The fold operation first aligns these subtrees. Alignment recursively aligns the respective text nodes under the `<a>` nodes from different subtrees. These text nodes have different values (e.g., Medicine vs. Multi-vitamins). Therefore alignment introduces a reference node with variable name `v1` and updates the scope environments of the different subtrees to include a mapping of `v1` to the respective value. Similarly, `v2` is introduced. Therefore, each of the environments includes a mapping of both `v1` and `v2`. The fold operation then wraps the resulting aligned subtree with a for node `FOR:f1` (introduced in layout tree L2) and adds a mapping of the variable `f1` in the main environment to a list containing the updated environments (as reflected in the environment D2).

Remark. As a post-processing phase, SYNTHIA identifies variables that always have the same value whenever they appear together in the same environment. Such variables are renamed to the same variable to avoid duplications in the data.

4.7 Evaluation

In this section, we evaluate our approach across multiple dimensions. First, we show that our technique is good for data extraction by evaluating it on standard datasets, and comparing it to three other state of the art data extraction techniques. Then, we show that our technique is good for separation of code and data by computing the combined representation size (MDL).

4.7.1 Evaluation of Data Extraction

Methodology

To evaluate the effectiveness of our approach for data extraction, we have used the common testbeds TBDW [YCNH04] and RISE [RIS98]. We compare our approach to DEPTA [ZL05], a technique that works on single pages, as well as to techniques that handle multiple pages: MDR [LGZ03a], TPC [MTH+09], FivaTech, and Trinity (as reported by [SC+14]).

Testbed 1: TBDW The TBDW testbed contains 253 web pages from 51 sites. Each web page in the testbed is manually labeled with the correct number of records, and the content of the first record. We use TBDW to compare the performance of our algorithm with that of DEPTA [ZL05], MDR [LGZ03a], and TPC [MTH+09]. For DEPTA, where the code is available, we reproduce the results by running the DEPTA tool. For MDR and TPC, we compare our results to those reported in [LGZ03a, MTH+09].

Testbed 2: RISE The RISE testbed contains 663 pages from 5 different site. We use it to compare the performance of SYNTHIA to FivaTech and Trinity as reported by [SC+14]. RISE checks the performance of page-level record extractors. It contains pages with single records, something that SYNTHIA is not meant to handle, but is able to handle if pages are merged into a single page. To enable our tool to deal with single record pages, we put all the DOM trees of these different pages as children subtrees under a shared "root" node, and apply SYNTHIA on the resulting tree. DEPTA is excluded from the comparison on RISE, because it was not designed to handle multiple pages, and applying it to our single merged page produces very poor results (which we consider unfair comparison).

Experiment We run SYNTHIA on the 253 pages from TBDW and 663 pages from RISE, and collect the records extracted from each page. For each page, our approach extracts a hierarchical representation (json) of the data on the page. We consider the list of environments in the data corresponding to a "for" variable as a table of records. If the relevant data was separated to two or more different tables, we only consider the table containing the biggest number of relevant records as the table of records identified by SYNTHIA.

Ground Truth As suggested in TBDW, the first record on a page, together with the page itself, defines the ground truth of the set of data records of the page. The ground truth for each site is obtained by the union of all ground truth sets of its pages.

Comparing Results We ran both our algorithm and DEPTA [ZL05] on the 253 webpages from the 51 websites in the testbed. We compare the set of records extracted by each approach to the ground truth. For the comparison, we consider the ground truth over all websites, as well as the set of *true positives*, which consists of data records correctly extracted by the algorithms, and the set of *false positives*, which consists of items that are wrongfully identified as data records. We report the precision and recall of each approach:

$$Precision = \frac{|\text{true-positives}|}{|\text{true-positives}| + |\text{false-positives}|}$$

$$Recall = \frac{|\text{true-positives}|}{|\text{ground-truth}|}$$

In addition, to compare our results with TPC [MTH$^+$09] and MDR [LGZ03a], we use the same partial set of 43 websites containing 213 web pages from TBDW used in [MTH$^+$09]. In this case, the ground truth, true positives and false positives, are computed per website. The results are the average recall and precision aggregated for all sites.

To compare to FivaTech and Trinity, we ran our tool on RISE dataset and compared its results to those reported by Trinity in [SC$^+$14].

Results

The results of SYNTHIA when compared to DEPTA on the whole TBDW dataset are reported in Table 4.1. As seen from the table, SYNTHIA is favorable in both recall and precision. We found that in many cases DEPTA fails to find the boundaries of the data records, frequently merging several records into one.

Table 4.2 shows the results of our tool when compared to DEPTA, MDR and TPC on a partial set of 43 websites containing 213 web pages from TBDW (we denote it TBDW-P). The TBDW-P is suggested by TPC [MTH$^+$09] and it excludes pages from TBDW containing nested

Table 4.1: Accuracy comparison with DEPTA on the TBDW dataset

	DEPTA	SYNTHIA
Ground Truth	4620	
True Positives	2506	4445
False Positives	27	23
Precision	98.9%	99.5%
Recall	54.2%	96.2%
F-Score	70%	97.8%

Table 4.2: Accuracy comparison on TBDW-P dataset

Algorithm	Precision	Recall	F-score
DEPTA	97.6%	59.5%	73.9%
MDR	93.2%	61.8%	74.3%
TPC	96.2%	93.1%	94.6%
SYNTHIA	99.7%	95.6%	97.6%

structures, in order to provide a fair comparison with the MDR algorithm, which is designed for flat data records. The As can be seen from the results, MDR suffers from similar recall issues as DEPTA, while having a lower recall. Generally speaking, our algorithm has the best performance, both in precision and recall compared to the three other algorithms.

The TBDW dataset contains a few pages with a single result record. Our algorithm fails to extract such records since it works on a single page and not on a group of pages generated using a similar template. This contributes to the small loss of recall (95.6% and 96.2% on the partial and full sets respectively) of our algorithm.

The results for running our tool on RISE dataset are reported in Table 4.3. Our tool outperforms both Trinity and FivaTech in most of the sites of this dataset. Trinity is the closest among the two in terms of performance. Our tool has low recall when extracting the records from IAF. We reviewed the web pages, and found that different data records are not of the same length. While our tool is capable of dealing with data records of length¿1 (not wrapped by a single html tag, which is not so common), our tool does not deal with records of varying lengths.

Table 4.3: Record extraction performance comparison between SYNTHIA, Trinity and FivaTech on RISE dataset.

	SYNTHIA			Trinity			FivaTech		
	P	R	F1	P	R	F1	P	R	F1
BigBook	0.99	1	0.99	0.95	0.94	0.94	-	-	-
IAF	1	0.11	0.2	0.84	0.38	0.52	0.53	0.69	0.6
Okra	1	1	1	1	0.82	0.9	0.49	0.34	0.4
LA.W	1	0.75	0.86	0.97	0.92	0.94	0.83	0.57	0.68
Zagat	1	1	1	1	0.86	0.92	1	0.98	0.99

4.7.2 Evaluation of Code and Data Separation

Methodology

The TBDW dataset contains the search results generated by searchable databases, also called search result records (SRRs). These pages are always of a common format of list of results. In this dataset, our approach recognizes that the format does not vary between records, and that formatting is part of the template. To evaluate the quality of the resulting code/data on general websites, we consider benchmarks with more significant page structure. We created our own dataset(available at: https://goo.gl/PKY0VI) by collecting 200 pages from 40 popular websites in 8 categories, with 5 different pages from each site. In all of our experiments we also verify that the separation computed by SYNTHIA is indeed lossless by running the extracted code on the extracted data and comparing the result to the original page.

Quality of Separation Inspired by the MDL principle [DC], we consider the length of the combined code/data representation as an indicator for the quality of separation. On the one hand, considering similar code subtrees as separate subtrees will prevent potential reduction in representation size due to the code representation. On the other hand, folding together different subtrees and representing them using a single code tree will introduce many conditional constructs and dynamic references, resulting in a more complicated and bigger data. A good solution will know when to fold two subtrees and when to keep them separate, in a way that keeps the representation length minimal.

For each page we compute the size in bytes of the data and code representation produced by our approach, denoted $|data(page)|$ and $|code(page)|$ respectively. We use these to compute

$$\textit{Reduction-Ratio} = \frac{|code(page)| + |data(page)|}{|page|}$$

When the entire page is considered code (this is one of the trivial solutions for separation), the reduction-ratio is 1. The reduction in representation size results from deduplication of the shared template repeating in a code-generated page. The reduction is bigger in pages having more regularity. Previous work on template extraction [GPT05] reported that the template size is around 40%-50% of the page size. If this template is regular, we can expect to obtain significant savings in representation from this part of the page.

Comparing Results Since we are not aware of any other approach that separates a single page into data and code, we compute the reduction-ratio of our approach and compare it to 2 other simplified implementations with different features (referred to as *basic* and *w/nesting* in the table). The first implementation is inspired by RTDM [RGSL04, VdSP$^+$06] and is based on a traditional tree edit distance metric both for the decision which subtrees to fold and for folding them. The second algorithm is based on a bottom-up tree edit distance computation. The main difference between the two is that the latter can deal with nested data regions by first running on a node's children before trying to fold them. In contrast, our algorithm calculates the minimal shared representation size of two subtrees, and uses it as the basis for deciding which subtrees to fold. In addition, folding minimizes the shared representation of the folded subtrees.

Results

The results in Table 4.4 show that using a bottom-up algorithm, which enables dealing with nested data-regions, improves the reduction-ratio compared to the simpler approach based on tree edit distance. Furthermore, the results show that our algorithm significantly outperforms both of the algorithms that are based on tree edit distance in terms of representation size.

Table 4.4: Ratio of reduced size to original size (reduction ratio). Lower numbers represent better reduction.

	basic	w/nesting	SYNTHIA
Movie	87.8%	66.4%	65.3%
Cars	81.6%	74.1%	59.1%
Real-estate	96.2%	86%	68.9%
Forums	92.7%	77.4%	61.3%
Sports	94.9%	80.4%	64.5%
Jobs	97%	90.5%	81.8%
E-commerce	71.9%	62.5%	43.5%
Photography	82.1%	78.3%	67.5%
Overall	**87.4%**	**75.5%**	**62.8%**
TBDW	**93.9%**	**89.9%**	**65.8%**

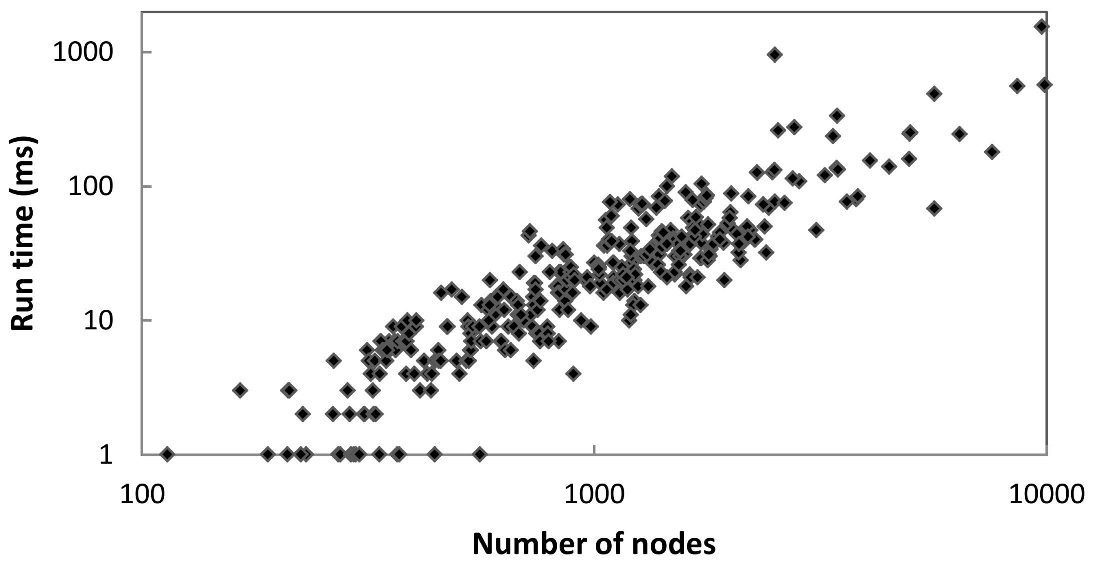


Figure 4.7: Running times as a function of the nodes count for the different documents in the two datasets

4.7.3 Running Time

We recorded the running time of our algorithm on the 453 different web pages from both TBDW and our dataset. All experiments were run in a single thread on a Macbook Pro with Core i7 CPU and 16GB memory. The running time is reported in Fig. 4.7. We found that our tool has an average run-time of $53ms$ on pages from the two datasets. It processes 95.9% of the pages in the two datasets in less than $150ms$ and 99.5% of the documents in less than $1sec$.

4.8 Conclusion and Future Work

We presented a technique for separating a webpage into *layout code* and *structured data*. Our technique computes a separation that is *lossless*, which means that running the extracted code on the extracted data reproduces the original page. Because there are many ways to separate a webpage into layout code and data, we make sure that our separation is efficient by aiming to minimize the joint description length of code and data. What this means intuitively, is that our technique attempts to find a separation such that common elements become part of the layout code, and varying values are represented as data. The ability to separate webpages has various important applications: page encoding may be significantly more compact (reducing Web traffic), data representation is normalized across different Web designs (facilitating wrapping, retrieval and extraction), and repetitions are diminished (expediting site updates and redesign). We show the effectiveness of our approach by evaluating its performance both for size compression and record extraction. Despite the fact that our approach is not specifically tailored to these applications, it outperforms state of the art data extraction tools, and achieves impressive compression ratios.

As code becomes increasingly important in producing the content of a page [WPC+15], we believe that layout code (more generally, page code) and the problem of *code extraction* should receive more attention. In future work, we plan to address the separation problem with primitives for data generalization.

Chapter 5

Separation of Web Sites into Layout Code and Data

Abstract We present a novel technique for separating a website into layout code and data. Given a website, our technique separates it to a small number of layout programs and structured data sources. The main idea is to define a distance measure between web pages such that the distance measures the cost of unifying the pages into a common template. We then use this distance to pick representative pages in a way that increases coverage of the different templates used in the site. Each representative forms a cluster, for which we generate a single unified layout code, accompanied with a data component for each of the pages in the cluster. The combination of separation to layout code and data with the concrete representatives for each template allows to benefit both from automatic manipulations of the extracted data and code, as well as from manual ones which require user input. For example, a user who is interested in data extraction of certain items may use the concrete representatives for tagging them. We have implemented our approach in a tool called SAPPORO and evaluated its effectiveness.

5.1 Introduction

Pages in many modern websites are template generated by applying a small set of *layout code* files to a structured data source, producing the set of HTML pages that are then presented to the user. These pages are therefore a blend of formatting elements inserted by the specific layout code file that is used to generate them, and actual values that are obtained from the structured data. It is common that many HTML pages in a website are produced using the same layout code (template) and therefore share a lot of the formatting elements.

Goal Given a set of HTML pages, our goal is to *separate* them into a small set of *layout programs* and a set of structured data sources such that: (i) each layout program represents a subset of HTML pages sharing the same (or a lot of) the formatting elements (ii) running the corresponding layout program of a page on its extracted data source reproduces the original page (the separation is *lossless*), and (iii) the separation is *efficient* such that the entire set of pages is represented using a small number of layout code files, where common elements among pages become part of their representative layout code file, and their varying values are represented as data.

Applications Separation of website pages has many important applications such as data extraction, webpage clustering, template removal, and web traffic optimization. For example, the data files generated by our approach can be used for unsupervised data extraction (facilitating wrapping and retrieval). The resulting separation can also be used for traffic reduction and automatic conversion of web applications to Ajax applications. These applications are not limited to server-side generated sites but can also be applied to client-side (Angular or Ajax in general) generated sites (e.g., by using a headless browser to obtain a static HTML page).

Challenges Separating a website into layout code and data requires that we identify pages that have similar formatting and can thus be formatted using the same layout code, and only differ on the data. It also requires that we efficiently cover the variety of different templates that are used in a website.

Existing Techniques There has been a lot of work on data extraction from web pages [AGM03, KC10, SC13, SC$^+$14, TW13, WL03, CL01, ZL05, SL05, LMM10, CYWM03, LGZ03a, LWLY12, DBS09b, HCPZ11, MTH$^+$09]. Record-level extraction handles the case where a single page has a list of records, where there is repetition within the same page. Our approach deals with the complementary problem where the repetition is *not only* within the same page, but also across different pages. Previous work [OKYS16], applied the idea of *separation* to separate single page and perform record-level extraction. In this paper, we address the challenging problem of *separating a set of web pages*.

While the use of templates is essential for improving uniformity, readability, and maintainability of web-pages, templates are considered harmful for many automated tasks like semantic-clustering, classification and indexing by search engines. Therefore, a lot of past work tackled the challenges of template identification [RGSL04, LGZ03a, ZL05, BYR02, KFN10] and template-extraction [KS11, CKP07, GF14, DC, GM13]. Typically, the goal of these works is to identify or extract the template so it can be ignored/discarded, and the data could be passed to further processing.

Our Approach Our approach is based on the assumption that, pages produced by the same HTML layout code share more of the formatting elements than they share with pages generated by different layout code files, or in other words they have higher structural similarity.

We therefore use a *unification-guided similarity between pages*—a similarity function that captures the effort required to unify two pages. Further, we use novelty-based summarization algorithm that finds representatives for different templates present in the input set of pages. We use these representatives together with the same similarity function to compute clusters of pages that share a common template. We then unify the pages in each cluster to produce the separation of pages in the cluster into common layout code and data.

Our algorithm has the additional advantage that it computes not only a separated representation of code and data, but also maintains a concrete representative page for each template class.

The fact that we maintain concrete representatives provides a convenient way to involve a user. For example: (i) it allows human supervision as part of a summarization process, where the user can label certain templates as important, and dismiss others. (ii) it allows a human to label data for extraction on concrete (representative) pages instead of working with more involved internal representations.

We implement and evaluate a method for automatically separating a set of static template-generated HTML pages into a set of layout code files and a set of data sources (each associated with a single code file). The main idea, is to use a novelty based summarization algorithm to select a set of representatives which are the cores of the layout code files, then align the rest of the HTML files each with the most structurally similar representative, producing a single layout code file and list of data sources from each representative and its associated HTML files.

Main Contributions The contributions of this paper are:

- We present a novel algorithm for efficient *separation* of a set of webpages into layout code files and structured data sources.
- We propose a representative selection algorithm that identifies a *guide set* of webpages that represent all the templates used in a larger set of webpages. The main idea of our algorithm is to use novelty-based summarization based on a structure-aware similarity function for picking the representative pages efficiently, to enable coverage of the different templates via a typically small number of representatives.
- We implement and evaluate our approach. Our evaluation shows that our approach is effective in capturing the different templates present in a site, and in extracting the structured data from pages with different templates.

5.2 Related Work

In this section, we briefly survey closely-related work on web structure mining and data extraction.

Data extraction a lot of research has been done on page-level data extraction (e.g., [AGM03, KC10, SC13, SC$^+$14, CMM01])

EXALG [AGM03] gets as an input a set of pages sharing the same unknown template and deduces the template and uses it for data extraction. Trinity [SC$^+$14] is an unsupervised web data extraction technique that learns extraction rules from a set of similar web documents. It uses the template tokens to partition each document into prefixes, separators and suffixes. It then recursively analyzes the results to discover patterns and build a “Trinity tree” which is later transformed into a regular expression for data extraction. Roadrunner [CMM01] uses similarities and differences between webpages to discover data extraction pattern. Similarities are used to cluster similar pages together, it then uses a matching algorithm to identify dissimilarities between pages in the same cluster and build a common regular expression for data extraction. FiVaTech [KC10] extracts data from a set of pages by automatically detect their shared schema and using it for data extraction. Their solution uses *pattern mining* and *alignment* to construct a structure called “fixed/variant pattern tree,” which can be used to identify the template and detect the schema. These techniques focus solely on data extraction without generating a layout code component. In addition, [SC$^+$14, KC10, AGM03] assume that pages in the input set are all generated using the same template and do not deal with cases when the provided documents have multiple templates. Our tool, in comparison, does not make such assumptions and is able to extract data and generate layout code also for page sets with multiple templates.

Record-level data extraction is another related problem. Given a single page containing multiple data records as an input, the goal is to to extract these records. Many techniques have been proposed dealing with this problem [ZL05, WL03, CL01, MTH$^+$09]. DEPTA [ZL05] use tag strings representation of DOM nodes and partial tree alignment to align generalized nodes and extract their data. IEPAD [CL01] discovers repeated patterns in a document by coding it into a binary sequence and mining maximal repeated patterns. These patterns are then used for data extraction. Compared to our method which can work on multiple pages with different templates, these methods are limited to a single page only and can not align records from different pages, something essential for making the data annotation process more efficient.

A lot of supervised data extraction techniques (e.g., [KWD, LG14, OSY17, DBS09b]) has been proposed in literature. FlashExtract [LG14] for instance, allows end-users to give examples via an interaction model to extract various fields and to link them using special constructs. It then applies an inductive synthesis algorithm to synthesize the intended data extraction program from the given examples. [NDMBDT14] propose a method for generating XPath wrappers. Given a user selected set of XPath samples describing the data nodes of interest in a DOM tree, the method uses an alignment algorithm based on a modification of the Levenshtein edit distance to align the sample XPaths and merge them to a single generalized XPath. In contrast to these techniques which require user annotation and interaction, our approach can work in an unsupervised manner allowing for later annotation of the extracted data.

Web structure mining is a family of web mining tasks that deal with the problem of discovery and analysis of the hyper-link structure between pages and sites in the web. Such analysis has a wide range of applications including web page importance scoring and ranking as proposed by Page Rank [PBMW99] and HITS [Kle99]. These techniques can be used along with graph mining techniques for web clustering. However, these techniques assume that page links are available, and that the link structure between pages can be discovered, assumptions that are not true in cases like web-mail clustering. Our approach, on the other hand, relies solely on the document structure without making such assumptions.

XML similarity A lot of XML similarity metrics [SM02, CMM$^+$02, But04, RMS06, DCWS06] has been proposed in literature. Information retrieval based techniques [SM02, CMM$^+$02] are tolerated more towards measuring content similarity. While edit distance (ED) and tag-based techniques [But04, RMS06, DCWS06] are more tolerated towards structural similarity. Compared to these methods our approach uses a similarity function that assesses its ability to efficiently represent the two input pages as a single layout-code (template) and data. We use [But04, RMS06] for clustering and compare the quality of the resulting clustering to that of ours. Our clustering show better clustering quality compared to these baselines.

Novelty based ranking Novelty based ranking is an important problem in the information retrieval field [CJL$^+$11, OCRS16, CKC$^+$08a]. It is used by search engines for ranking of search results [CJL$^+$11] in a way the promotes diversity, and in answer ranking as well [OCRS16].

These works are focused on semantic diversification, while we propose a novel method that promotes structural novelty of the selected representatives.

5.3 Overview: Problem and Solution

In this section, we illustrate our approach using an example.

5.3.1 Motivating example

Consider the problem of separating pages of pricespy.co.uk (PriceSpy) into layout code and data.

Goal Given a set of webpages from PriceSpy, our goal is to generate a set of layout programs (code files), representing the different templates that generate the provided pages. For each of the generated layout programs we also generate a list of data files (which we also call environments), such that invoking each layout program on its corresponding data files will generate the original input pages. We refer to the layout programs and the data files associated with them as a *separation of a set of webpages*.

When running our approach on 40 pages from PriceSpy, it generated four different layout-code files, each with 10 corresponding data files.

Fig. 5.1 shows two HTML snippets from two different product pages in PriceSpy. These two pages share a similar template (product page template) while presenting information of different products. Therefore, in the separation computed by our approach, these pages are generated by the same layout-code file. Fig. 5.2 shows the synthesized layout-code for these pages. As can be seen in Fig. 5.2, the generated layout code contains the shared formatting elements (template) while the differing data is replaced with variable references.

In order to generate the original pages in Fig. 5.1 using the code-template in Fig. 5.2, the layout-code needs to be invoked on a data source with value-assignments to every variable referenced in the layout code file. Fig. 5.3 shows snippets from the two data files generated by our method for the two HTML pages in Fig. 5.1. These data files contain assignments to the variables referenced in the generated layout-code in Fig. 5.2, and contain the data it needs in order to generate the two original HTML snippets.

Tree representation for convenience we often use tree representation of the HTML pages (DOM tree) and the layout code files (layout code tree). Fig. 5.5(a) shows the DOM tree representation of the HTML snippets from Fig. 5.1, and Fig. 5.5(b) shows the layout code tree representation of their synthesized layout code.

5.3.2 Our Approach

Our separation process attempts to represent a given set of pages of a website in an efficient and minimal way as layout programs and data files, exploiting the property that typically the pages are generated by a small number of layout programs, invoked on different data elements.

Fig. 5.4 illustrates our separation process for a set of pages. The separation process starts by selecting a set of representative pages to cover the different page structures present in the provided sample set. This is obtained using a novelty-based summarization algorithm, which aims to achieve coverage of the different templates present in the sample set with a small number

```
<div class="intro_body">
 <h1 class="intro_header">HTC One M10 32G</h1>
 <div class="product-brand-box">
  Compare price on all
  <a href="branda851.html">HTC Mobile Phones</a> (113)
 </div>
</div>
```

```
<div class="intro_body">
 <h1 class="intro_header">Apple iPhone 7 64GB</h1>
 <div class="product-brand-box">
  Compare price on all
  <a href="brandb9c2.html">Apple Mobile Phones</a> (40)
 </div>
</div>
```

Figure 5.1: Snippets of static HTML pages.

```
<div class="intro_body">
 <h1 class="intro_header">{var1}</h1>
 <div class="product-brand-box">
  Compare price on all
  <a href="brandb{var2}.html">
        {var3} Mobile Phones</a>({var4})
 </div>
</div>
```

Figure 5.2: Code synthesized for the given static HTML pages.

```
{"var1":"HTC One M10 32GB",
"var3":"HTC",
"var2":"a851","var4":"113"}
```

```
{"var1":"Apple iPhone 7 64GB",
"var3":"Apple",
"var2":"b9c2","var4":"40"}
```

Figure 5.3: Data synthesized for the given static HTML pages.

of representatives. Towards that end, we define a similarity function that measures structural similarity between two pages, and propose a novelty-based representative-selection algorithm.

A user can inspect the representatives and determine which of them are of interest, allowing the algorithm to focus on the pages for which separation is desired.

The representative selection step is followed by a classification step which is applied on all the pages in the sample set. The classification forms clusters of similar pages by associating each page in the provided sample with a single, most similar, representative among the selected representatives.

Finally, separation is performed within each cluster of interest (determined by the representatives), by iteratively aligning each page in the cluster with the representative. The alignment process results in a set of layout-code files, each associated with a list of data files.

Next, we provide additional details on the different steps.

Webpage similarity The ability to assess structural similarity of pages is essential to our approach. Therefore we define a page similarity function that captures structural similarity of the pages.

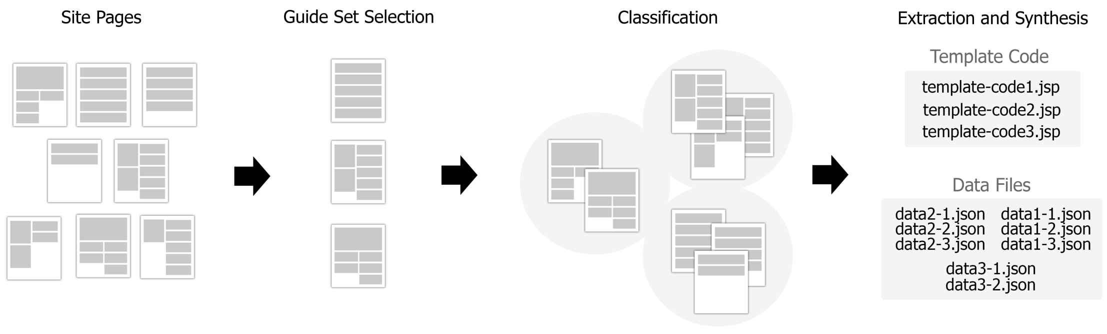


Figure 5.4: Main steps of the separation algorithm.

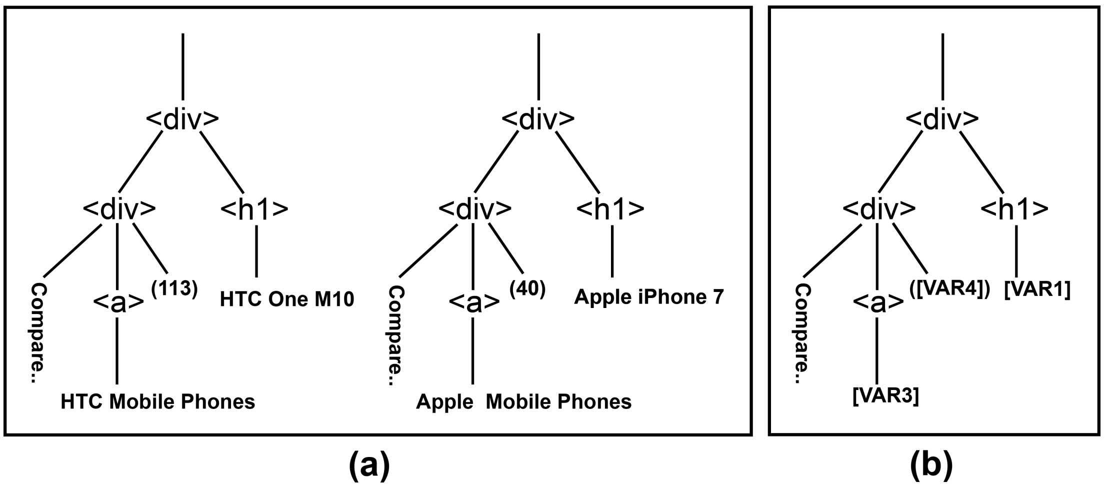


Figure 5.5: (a) Two DOM trees of the original HTML documents, and (b) the layout tree produced by our approach from these DOM trees.

An important aspect of our similarity function is that it strives to measure the similarity at the layout-code level. To do so, we apply a preprocessing step of *page-level separation* in which we apply a tree folding algorithm [OKYS16] on each page individually in order to fold item lists and unify their representation among the different pages. The tree folding algorithm works in a bottom-up manner: it analyzes adjacent layout subtrees and folds similar ones by aligning them and representing them using a single template subtree while introducing their differences as variable assignments in the data.

Guide set selection We present a novelty-based structural summarization algorithm and use it to generate a representative set, also called a *guide set*, for a given set of pages. The goal of the algorithm is to find a small set of representative pages that cover the different page templates in the input set.

The novelty-based algorithm uses a greedy iterative process, which measures the novelty of each unselected page, as well as the coverage it provides. Initially, all pages have the same novelty measure. At each iteration the algorithm selects as an additional representative the page that contributes the highest total coverage of novel pages, and updates the novelty accordingly. Technically, the selection happens in two steps:

1. **Representative selection:** the algorithm calculates a novelty aware coverage score (which

we call support) for each yet-unselected page in the provided set. The coverage that a page d_1 contributes to another page d_2 is their similarity-value multiplied by the novelty of page d_2. The new representative is selected as the one with the maximal support.

2. **Penalization:** the algorithm penalizes the novelty of the pages that are similar to the newly selected representative.

The representative selection process stops when a specific threshold of coverage is reached, or when we reach the number of representatives (budget) specified by the user.

Classification After a guide set is selected, each page in the input set is associated with the representative in the guide set that is most structurally similar to it, forming clusters of structurally similar webpages. We use the same structural similarity function we use in the representative selection process.

Template-code synthesis and data extraction Pages in each resulting cluster are sequentially aligned with their corresponding representative, where alignment identifies a common layout code and introduces variables and assignments to account for the differences. The alignment process starts with the layout-code of the representative page, obtained by page-level separation, as the current layout-code. In each iteration, the layout code (as well as the data of the previously aligned pages) is updated based on a new page whose layout-code is aligned with the current layout-code. Further, a new data component is generated for the newly aligned page.

5.4 Problem Definition

In this section we formally define the problem of separating a web site into layout code and data.

5.4.1 Separation and Solution Space

Our goal is to describe a website by a set of layout code trees, each associated with a set of environments that generate the original pages of the website.

Definition 5.4.1 (Separation)**.** Given a set of webpages $D = \{d_1, \dots, d_k\}$ of some website of interest, a *separation* of D is a set of pairs $\{(\pi_1, \mathbf{E}_1) \dots (\pi_N, \mathbf{E}_N)\}$, where

1. π_i is a layout tree,
2. $\mathbf{E}_i = \{\mathcal{E}_{i1} \dots \mathcal{E}_{im_i}\}$ is a set of environments,
3. for every $d \in D$ there exist (a unique) $1 \leq i \leq N$ and (a unique) $1 \leq j \leq m_i$ such that $\pi_i(\mathcal{E}_{ij}) = d$, and
4. $\bigcup_{i=1}^{N} \bigcup_{j=1}^{m_i} \pi_i(\mathcal{E}_{ij}) = D$.

Namely, invoking each layout tree on the set of environments associated with it results in the input set of pages, D. If $\pi_i(\mathcal{E}_{ij}) = d$, we say that d is represented by π_i. Separating D is the process of constructing a separation $\{(\pi_1, \mathbf{E}_1) \dots (\pi_N, \mathbf{E}_N)\}$ of D.

Note that a set of pages may have many possible separations. For example, $\{(d_1, \emptyset), \dots, (d_k, \emptyset)\}$ is a trivial separation of D where each page forms its own layout tree with an empty set of environments. We denote by $\mathbf{Sep}(D)$ the set of all separations of D.

5.4.2 Separation Quality

Since there are many possible ways to separate a given web site, it is important to define what makes one separation better than another. In this work, we focus on two quality aspects: (i) simplicity of the resulting layout code: in terms of the number and the structure of the resulting layout code files, and (ii) description length: the size of the resulting separation.

Inspired by the principle of *Minimal Description Length* (MDL) [Ris78], we define the *cost* of the separation based on its description length and aim to minimize it in our separation process. To do so, we consider the size in characters of the string representations of a layout tree π and an environment $\mathcal{E}$, denoted $\mathsf{sizeof}(\pi)$ and $\mathsf{sizeof}(\mathcal{E})$, respectively. We define

$$\mathsf{cost}(\{(\pi_1, \mathbf{E}_1) \dots (\pi_N, \mathbf{E}_N)\}) \overset{\text{def}}{=} \sum_{i=1}^{N} (\mathsf{sizeof}(\pi_i) + \sum_{\mathcal{E} \in \mathbf{E}_i} \mathsf{sizeof}(\mathcal{E})).$$

We use the cost function to assess the similarity of two pages by calculating the reduction in cost that can be gained by their alignment. In addition, it is used by our alignment algorithm to decide whether or not to align two sub-trees and to calculate the most efficient alignment.

5.5 Our Approach

Our approach for separating a set D of webpages consists of three main steps: (i) building a guide set which consists of a small number of representative pages from D, (ii) partitioning D into clusters based on the representatives in the guide set, and (iii) synthesising a layout code template and a set of environments for each cluster.

These steps use the page separation algorithm and the tree alignment algorithm developed in [OKYS16]. We therefore start with a short explanation of these algorithms.

Tree alignment A tree alignment algorithm aligns two (or more) DOM trees, or layout trees, into a single layout tree while storing the differences between them as variable assignments in corresponding environments. Formally, an alignment of trees, accompanied by environments, is defined as follows:

Definition 5.5.1. Let π_1 and π_2 be layout trees and let $\mathbf{E}_1 = (\mathcal{E}_{11}, \dots, \mathcal{E}_{1m_1})$ and $\mathbf{E}_2 = (\mathcal{E}_{21}, \dots, \mathcal{E}_{2m_2})$ be two series of environments. An *alignment* of $(\pi_1, \mathbf{E}_1)$ and $(\pi_2, \mathbf{E}_2)$ is a pair $(\pi', \mathbf{E}')$ where π' is the unified layout tree of π_1 and π_2 and $\mathbf{E}' = (\mathcal{E}'_{11}, \dots, \mathcal{E}'_{1m_1}, \mathcal{E}'_{21}, \dots, \mathcal{E}'_{2m_2})$ is the joint series of updated environments such that for every $j = 1, \dots, m_1$, $\pi'(\mathcal{E}'_{1j}) = \pi_1(\mathcal{E}'_{1j})$ and for every $j = 1, \dots, m_2$, $\pi'(\mathcal{E}'_{2j}) = \pi_2(\mathcal{E}'_{2j})$.

The objective of the tree alignment algorithm of [OKYS16] is to minimize the combined description length of the unified layout tree and the corresponding environments. To do so, it unifies the trees by establishing a common layout tree and updating the environments. In the following we consider alignments computed by the algorithm of [OKYS16].

Page-level separation The page separation algorithm of [OKYS16] separates a single page d into a layout tree π and data $\mathcal{E}$ such that $\pi(\mathcal{E}) = d$, while trying to minimize combined description length of π and $\mathcal{E}$. This is done in a bottom-up manner, where it analyzes adjacent subtrees (starting from the leaves) and folds together similar ones by aligning them. In the following we refer to $(\pi, \mathcal{E})$ obtained by the separation algorithm of [OKYS16] as the separation of d. We compute these separations for the given set of pages as a preprocessing step of our algorithm.

We are now ready to explain our website separation algorithm in detail. In the sequel, we fix a set D of webpages from a website of interest.

5.5.1 Building the Guide Set

A guide set G is a subset of the pages from the set D, each representing the set of pages created by the same layout code template. The pages in the guide set have to efficiently summarize the different templates of all the pages in D without representing the same template twice.

To construct a guide set, we develop a novelty based summarization algorithm that constructs the guide set while ensuring coverage and avoiding redundancy. The algorithm computes a novelty score for each page, as well as a coverage (support) score, which is based on a similarity measure between pages. Before we describe the algorithm, we first define the similarity function as well as the novelty and support score.

Page Similarity

The similarity of two pages attempts to estimate how efficiently they can be represented using a single layout code template. To calculate how similar two pages d_1 and d_2 are, we first separate them into $(\pi_1, \mathcal{E}_1)$ and $(\pi_2, \mathcal{E}_2)$ respectively, and then use the tree alignment algorithm to calculate their most efficient shared representation. Formally, let $(\pi_1, \mathcal{E}_1)$ be the separation of d_1, $(\pi_2, \mathcal{E}_2)$ the separation of d_2 and let $(\pi', (\mathcal{E}'_1, \mathcal{E}'_2))$ be the alignment of $(\pi_1, \mathcal{E}_1)$ and $(\pi_2, \mathcal{E}_2)$. We define $Sim(d_1, d_2) =$

$$\frac{\mathsf{cost}(\{(\pi_1, \{\mathcal{E}_1\}), (\pi_2, \{\mathcal{E}_2\})\}) - \mathsf{cost}(\{(\pi', \{\mathcal{E}'_1, \mathcal{E}'_2\})\})}{\frac{1}{2} * (\mathsf{cost}(\{(\pi_1, \{\mathcal{E}_1\}), (\pi_2, \{\mathcal{E}_2\})\}))}$$

Where $\mathsf{cost}(\{(\pi_1, \{\mathcal{E}_1\}), (\pi_2, \{\mathcal{E}_2\})\})$ denotes the accumulative representation cost of d_1 and d_2 when each of them is kept separately, while $\mathsf{cost}(\{(\pi', \{\mathcal{E}'_1, \mathcal{E}'_2\})\})$ denotes their *shared* representation cost.

Intuitively, the similarity of d_1 and d_2 is the reduction in representation cost when they are represented as a single layout code compared to the cost when they are represented as separate layout codes, normalized by their average initial cost.

Note that the similarity is calculated based on the separations of the pages, where similar subtrees within a single page are already folded together. This is important in order to account for, e.g., pages that contain list items with different numbers of items.

Support and Novelty of a Page

In the following we define the support score and the novelty score of a page used to select the pages in the guide set.

The novelty of a page d with respect to a given set G of already selected representatives measures how unique d is compared to the pages in G. Formally:

Definition 5.5.2. Given a set of already selected representatives G, the *novelty* of a page d with respect to G is calculated as follows:

$$Nov(d, G) = \prod_{d_i \in G} (1 - Sim(d, d_i))$$

Note that for $d \in G$, $Nov(d, G) = 0$. Namely, the novelty of a page that already belongs to the guide set is 0.

The support of a page d in a sample D given a set of already selected representatives G is the average amount of *additional coverage* it contributes to other *novel* pages in D relative to G. Formally:

Definition 5.5.3. Given a set of pages D and a set of already selected representatives G, we calculate the *support* of a page d as follows

$$Supp(d, D, G) = \frac{\sum_{d_i \in D \setminus G} Sim(d, d_i)^{1+c} * Nov(d_i, G)}{\sum_{d_i \in D \setminus G} Sim(d, d_i)^{c}}$$

The support score is used by our greedy algorithm for representatives (guide set) selection to decide which additional representative to select at each iteration, given the set of D pages and the set of already selected representatives G. The higher the support the page has, the greater its contribution to the coverage of D is. Therefore, the more likely it is to be selected and added to the guide set.

The c parameter A page can cover a large number of pages that are moderately similar to it, adding a small coverage to each one of them. Alternatively, ir can also cover a small set of pages that are highly similar to it (completely covering this small set of pages). The question “what is considered a better coverage” may have different answers depending on the application. The c parameter gives a user control over what to choose. A lower c-value gives priority to the former possibility (where a large number of supporters can compensate for lower similarity), while a higher c-value gives higher priority to the latter (where the similarity of the supporters is more important than their number).

Algorithm 3: Guide Set Selection

```
$G = \emptyset$;
**foreach** $d_i \in D$ **do**
    $Nov(d_i, G) = 1$
**repeat**
    $d = \arg\max_{d \in D \setminus G} Supp(d, D, G)$;
    $G = G \cup \{d\}$;
    **foreach** $d_i \in D \setminus G$ **do**
        $Nov(d_i, G) = Nov(d_i, G \setminus \{d\}) * (1 - Sim(d, d_i))$
**until** $Supp(d, D, G) < threshold$;
**return** $G$;
```

Guide Set Selection Algorithm

Given a set of pages D , we use (the greedy) Algorithm 3 to build a guide set. The algorithm starts with an empty guide set $G = \emptyset$ and with the novelty of each page $d_i \in D$ set to 1. It then gradually adds pages that have the highest support score, while continually updating the novelty score of the remaining pages in $D \setminus G$. Namely, in each iteration the algorithm selects the page $d \in D \setminus G$ with the highest support score, adds it to the guide set G , and penalizes all pages similar to d by reducing their novelty score by the amount of their similarity to the newly added page. Note that the algorithm does not update the novelty of the existing representatives in G, nor does it update the novelty of the new representative $d \in G$ to 0, since in any case pages in G do not affect the support of other pages.

5.5.2 Website Separation based on Guide Set

Given the guide set G, we synthesize layout code templates, and data, for the pages in D by grouping them according to their similarity to the pages from G.

To that end, we first partition the pages of D to clusters by classifying each page in D to the cluster of the most similar page from G. Similarity is measured by the same function as before. The result is a set of clusters, where each cluster contains exactly one page from G.

Within each cluster C, we use the unique page from G as a starting point, and align it with the other pages in C. The result of these alignments is a layout code template π_C and a set of environments $\{\mathcal{E}_1^C, \ldots, \mathcal{E}_{|C|}^C\}$, one for each of the aligned pages. Technically, we apply the alignment on the page level separations of the pages in the cluster. Algorithm 4 summarizes the cluster-level separation algorithm. In the algorithm, the initial layout-code π^* is the layout-code of the representative $d_1 \in G$ of C, and the single data component is its environment. In each alignment step, the separation $(\pi_i, \mathcal{E}_i)$ of another page $d_i \in C$ is aligned with π_C, while updating all the existing environments $\{\mathcal{E}_1^C, \ldots, \mathcal{E}_{i-1}^C\}$ and appending a new one $\mathcal{E}_i^C$ for the newly aligned page.

Algorithm 4: Cluster-level Separation of C

```
Let $C = \{d_1, \ldots, d_{|C|}\}$ s.t. $\{d_1\} = C \cap G$ ;
for $i = 1 \ldots |C|$ do
    Let $(\pi_i, \mathcal{E}_i)$ be the page-level separation of $d_i$
$\pi_C = \pi_1$, $\mathbf{E}_C = \mathcal{E}_1$;
for $i = 2 \ldots |C|$ do
    $(\pi_C, \mathbf{E}_C) = align((\pi_C, \mathbf{E}_C), (\pi_i, \mathcal{E}_i))$
return $(\pi_C, \mathbf{E}_C)$;
```

Finally, the separation of the given set D of pages consists of the cluster-level separations $(\pi_C, \mathbf{E}_C)$ of all clusters.

5.6 Evaluation

In this section we evaluate the effectiveness of our approach. Our experiments in this work focus on three main aspects: (i) evaluating our representative set selection algorithm, and (ii) evaluating our page classification and the resulting clustering, and (iii) evaluating the separation quality

5.6.1 Implementation

We implemented our approach in a tool called SAPPORO in java, and used it separate multiple sets of web pages from different real-life websites. For comparing the performance of our tool in each one of the aspects, we use the following baselines: For representative selection comparison, we have implemented a perfect ranker baseline. The perfect ranker is aware of the real templates of the provided HTML files and aware of the scoring metric -unlike our solution which has no such prior knowledge-, it ranks the representatives in a way the guarantees maximal (perfect) score in the ranking metrics. We have implemented page-level alignment algorithm presented in [OKYS16], and used it as a baseline to evaluate the effectiveness of our site-level alignment algorithm. In addition, we have implemented two custom web page clustering algorithms to compare our resulting clustering to. They are based on Expectation-Maximization (EM) clustering with two different representations: tag based representation [But04] and XPath based representation [RMS06].

Dataset To evaluate our approach and the different baselines, we have constructed a data set that contains 318 pages from ten real-life websites, these pages are manually grouped in folders according to their template. For each site, the dataset contains pages from 2-4 different templates. This dataset is used in the evaluation of all the different aspects of the site. It is available for download at: goo.gl/hII1n4.

5.6.2 Representative set selection evaluation

In this section we evaluate the performance of our representative selection algorithm by evaluating the order in which it selects representatives. Given an ordered sequence of representatives as selected by out algorithm, we calculate the amount of additional coverage each representative in the sequence adds, and compare it to the optimal possible order under the same metric.

Evaluation metrics

Normalized discounted cumulative gain ($\alpha NDCG$) [CKC+08b, CCSA11] is a ranking quality metric designed specifically for evaluation of novelty based ranking. Given a sequence of representatives, the metric calculates for each representative its additional coverage gain (template coverage) normalized by its position, while promoting yet uncovered by penalizing already covered templates by a factor of (1-α).

$$\alpha DCG = \sum_{j=1}^{k} (1-\alpha)^{r(t(j))} / (1 + log(j))$$

. The final step is to normalize it against the gain sum of the optimal representatives order to get $\alpha NDCG$. In addition to the original $\alpha NDCG$, we use a modified version of $\alpha NDCG$ that gives different weights to the clusters (templates) according to the number of document they contain in the set.

$$\text{Weighted-}\alpha\text{DCG} = \sum_{j=1}^{k} W(t(j))(1-\alpha)^{r(t(j))} / (1 + log(j))$$

Where W(t(j)) is the number of pages in the cluster covered by the j-th element in the representatives sequence, and r(t(j)) is the number of previous representatives that cover it.

Templates Coverage Given an ordered sequence of representatives, we calculate the number of different templates (original layout codes) covered by each prefix of the sequence up to prefix length of 5. At each length we normalize by the maximal number of templates that can be covered (optimal order).

Document Coverage Calculated and normalized similarly to the template coverage metric, the only difference is that the templates are weighted, each the number of documents associated with it in the data set by our manual clustering.

Representative selection results

The $\alpha NDCG$ performance results for our representative selection algorithm are reported in Table 5.1. The results show the optimality scores of our ranking algorithm (since $\alpha NDCG$ is -by definition- normalized by the optimal possible gain). According to the results in Table 5.1, for $\alpha = 1$ (full penalization of already covered clusters), our solution (TRACY) selects representatives in a close to optimal order according to the (non weighted) $\alpha NDCG$ metric. For the weighted $\alpha NDCG$ metric however, TRACY has better performance when the value of parameter c is closer to 0. This is expected, as a higher c value gives less importance to the amount of similar documents to a representative and more importance to how similar and how novel they are.

	α=0	α=0.2	α=0.4	α=0.6	α=0.8	α=1
$\alpha NDCG$						
c=0	1.00	0.94	0.92	0.93	0.96	0.98
c=0.5	1.00	0.94	0.93	0.94	0.96	0.99
c=1	1.00	0.94	0.92	0.94	0.96	0.99
$Weighted\ \alpha NDCG$						
c=0	1.00	0.93	0.93	0.94	0.96	0.99
c=0.5	0.92	0.91	0.90	0.90	0.91	0.93
c=1	0.92	0.90	0.88	0.89	0.90	0.92

Table 5.1: $\alpha NDCG$ results for our representative selection algorithm, for different values of α and c.

Table 5.2 shows coverage performance results for representative sequences as selected by our tool (SAPPORO), normalized by the coverage of optimal sequence of the same length (according to each metric) as a function of the sequence length. The results show close to optimal coverage by sequences generated by our tool for the non-weighted case. In the weighted case however (where templates with more pages should appear first), our tool has better coverage (close to optimal) when c values are closer to 0 (which is expected as we explained earlier).

5.6.3 Page Classification and Clustering

In this section we evaluate the classification accuracy and the quality of the clustering generated by our approach, and compare it to two clustering baselines: (i) EM-TagCount (tags), which uses tag representation of the pages [But04] and uses expectation-maximization (EM) clustering

	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$
Template Coverage					
c=0	1.00	0.95	0.97	0.97	0.97
c=0.5	1.00	0.95	0.97	1.00	1.00
c=1	1.00	0.95	0.97	1.00	1.00
Pages Coverage					
c=0	1.00	0.97	0.97	0.97	0.97
c=0.5	0.80	0.86	0.92	1.00	1.00
c=1	0.80	0.80	0.92	1.00	1.00

Table 5.2: Template coverage and pages coverage (normalized by optimal) as a function of the representatives sequence length l.

algorithm to cluster them , (ii) EM-PathCount (paths), which does the same, but uses tag-path representation instead [RMS06].

Clustering evaluation metrics

Cluster purity given a set of clusters generated by our method (as representatives with their associated pages) and the other baselines for each site in DS1, we measure the purity of each one by counting the number of correctly assigned pages (have the same template as the representative) and dividing by the cluster size. The average purity over all clusters is then calculated.

Rand index (RI) measures the percentage of pairs of pages that are correctly classified in the same cluster and in different clusters. It is calculated in the following way:

$$RI = \frac{TP + TN}{TP + FP + TN + FN}$$

Where TP is the number of pairs correctly clustered in the same (different for TN) clusters , while FP is falsely clustered together (in different clusters for FN) pairs.

Clusters number We compare resulting number of clusters of our approach and the other baselines, and the real number of templates for each site. Our approach uses threshold on the support value to decide when to stop adding representatives. We test the resulting clusters number for multiple different low threshold values to assess the sensitivity of the clusters number to this parameter.

Clustering evaluation results

Clustering quality results of our tool (SAPPORO) and the two clustering baselines are reported in Table 5.3. For each clustering tool, the table contains the quality of its resulting clustering for each site in our data set individually, and its aggregated average score. The results indicate that our tool has higher purity and IR scores compared to the two other baselines. Where among the two baselines, the EM with tag-path based representation has a better clustering quality.

Table 5.4 shows the number of resulting clusters created by our approach for different threshold values compared to the other baselines. For low threshold values (0.1 and 0.3) our

	Purity			Rand index		
Tool	Paths	Tags	Our	Paths	Tags	Our
idealo	0.50	0.50	1.00	0.50	0.50	1.00
aria.co.uk	0.83	0.83	1.00	0.88	0.88	1.00
pricerunner	0.96	0.67	0.89	0.30	0.30	0.79
cclonline	0.36	0.36	0.91	0.75	0.52	0.95
priceme	0.72	0.83	1.00	0.78	0.88	1.00
shopbot	1.00	1.00	1.00	0.95	1.00	1.00
direct	1.00	1.00	1.00	1.00	1.00	1.00
pricespy	1.00	0.69	0.96	0.96	0.86	1.00
currys	1.00	1.00	1.00	1.00	0.76	0.96
ebuyer	0.50	0.50	1.00	0.50	0.50	1.00
average	0.79	0.74	0.98	0.76	0.72	0.97

Table 5.3: Cluster purity and RI values for our tool and the baselines.

approach results in perfect number of clusters, this shows that our clustering is not too sensitive to the threshold parameter, and will not be hard to train it to find a global good threshold value. The other baselines have less accurate number of clusters.

Count	GT	Path	Tag	our-0.1	our-0.3	our-0.5
idealo	2	1	1	2	2	2
aria.co.uk	4	3	3	4	4	3
pricerunner	3	3	1	3	3	3
cclonline	4	1	1	4	4	3
priceme	4	3	3	4	4	4
shopbot	3	4	3	3	3	3
direct	2	2	2	2	2	2
pricespy	4	4	3	4	4	3
currys	3	4	4	3	3	2
ebuyer	2	1	1	2	2	2
difference		0.9	1.1	0	0	0.4

Table 5.4: Number of clusters created by our approach with different support threshold values, and the two clustering baselines compared to the ground truth (GT).

5.6.4 Separation Quality

In this section we evaluate the quality of the resulting separation by checking its validity and comparing the size of resulting separation to the original non-separated representation and to page-level separation produced by [OKYS16].

We have used our representative selection and classification algorithms to cluster the web pages of each one of the websites in the dataset, each cluster containing files with similar template. We then aligned pages in each cluster, producing a single template file and multiple data files, one for each page of the original pages.

We have implemented the page-level separation algorithm presented in [OKYS16] and used it to separate pages of each website into layout-code and data. We use the resulting page-level separation as a reference point and compare the gain in reduction in representation size between our site-level approach and page-level one.

Separation quality results

The size of representations produced by our approach and the baselines are reported in Table 5.5. The results show that our site-level separation approach reduces the representation size to 42.3% of the original representation size, while the size of representations produced by page-level separation is 64.6% of the original size. The results show 34.5% improvement in reduction of the representation size by our site-level approach compared to the page-level approach.

size in MB	original	page-level	site-level
idealo	2.43	1.89	1.41
aria.co.uk	2.75	2.03	1.02
pricerunner	6.68	3.59	2.91
cclonline	6.22	4.07	2.57
priceme	4.34	2.48	1.42
shopbot	6.51	1.48	1.1
direct	4.33	4.19	3.11
pricespy	5.71	4.45	3.14
currys	13.8	9.08	5.04
ebuyer	3.62	3.15	2.12
total	56.39	36.41	23.84
size/original	100	64.6	42.3

Table 5.5: The file sizes in MB of representations of webpages of each site produced by our site-level separation approach compared to page-level approach and original files.

5.6.5 Discussion

The evaluation results show that our approach (SAPPORO) has a close to perfect representative selection order in novelty oriented metrics such as the coverage metrics and $\alpha NDCG$ (with α-values close to 1). In addition to the contribution a good representative selection order can bring to the clustering quality (which we assess separately), it can save manual effort and make more convenient tasks involving a human reviewer, who can review fewer representatives to cover all the different templates in the provided page set. Such task can be, for instance, manual modification or selection the set of clusters, or annotation of less pages for data extraction.

The clustering quality results show that our approach has a superior clustering quality in terms of purity, IR and number of clusters compared to the two other clustering baselines. The clustering quality is important for our approach as it directly affects the quality of the resulting separation. A clustering with poor quality may result in a higher number of layout-code files that are too general, and will result in more complicated data files, whereas a high quality clustering will result in a relatively (compared to a lower quality clustering) small number of clean layout code files, and will result in a relatively simple data sources which makes the resulting representation more efficient and more useful for important applications like data extraction, and traffic reduction.

The separation quality results show that our site-level has an improved reduction in representation size compared to page-level separation. The representation resulting from our site-level separation can be used to automatically generate AJAX versions of websites to optimise traffic.and reduce load times.

5.7 Conclusion

We propose a novel approach for separation of websites into layout code files and data files. The approach uses a novelty based algorithm to rank pages according to their novelty and contribution to the coverage of the different templates. It selects a set of representative pages and uses them to classify the rest of the pages to template clusters. It then aligns pages in each cluster with their representative to synthesize a layout code file and extract a list of structured data sources. We have implemented our approach in a tool called SAPPORO and evaluated its effectiveness. The evaluation shows high effectiveness of our tool for structural summarization, clustering and alignment of web pages.

Chapter 6

Conclusion and Open Questions

6.1 Conclusion

The goal of this work was to address the problem of scalable data-extraction via automatic synthesis of data extraction programs and web applications. Our work explores two different types of solutions: one that uses program synthesis techniques to automatically synthesize web extractors, and another that uses program synthesis to separate a website back into layout code and data.

In our first work (Chapter 2), we present an automatic synthesis approach for generating extraction web crawlers for a group of websites. This chapter introduces a new cross-supervised crawler synthesis algorithm that extrapolates crawling schemes from one web-site to another. The main idea is to automatically label data in one site based on others and synthesize a crawler from the labeled data. Since the annotation is done automatically, we cannot assume that all annotations are correct (hence some of the examples might be false positives), and we cannot assume that un-annotated data is noise (hence we have no negative examples). We use a voting approach and the notion of containers to overcome these difficulties. The idea of cross-supervised synthesis allows us to use a single handcrafted web-crawler to automatically synthesize a set of web crawlers for a group of websites from the same category. In addition, it allows us to automatically regenerate a new web crawler for a website in case its web-crawler broke as a result of structural changes of the web-site template.

In chapter 3 we have presented and implemented a novel approach for automatically synthesizing forgiving-xpaths. These forgiving xpaths combine the benefits of XPath queries with the generality of classifier-based extractors. The synthesis process uses a modified decision tree construction algorithm to build general yet accurate XPath. It then iteratively prunes the resulting tree producing a sequence of decision trees with decreasing precision and increasing recall. These threes are then translated to XPaths and combined together into a forgiving XPath. In our evaluation we found that extractors produced using this approach outperform not only other pattern-based extractors, but also classifier-based extractors which are typically more suited for handling unseen sites. This accuracy was achieved while maintaining the readability and efficiency of the XPath.

In chapters 4 and 5 we presented a technique for separating a website into layout code and data. The separation has many possible application, including page-level and record-level data extraction. In chapter 4 we propose a technique for separation of a single page. The separation that our method calculates is lossless, which means that running the extracted code on the extracted data reproduces the original page. Because there are many ways to separate a webpage into layout code and data, we make sure that our separation is efficient by aiming to minimize the joint description length of code and data. Despite the fact that our approach is not specifically tailored towards data-extraction, it outperforms state of the art data extraction tools, and achieves impressive compression ratios. In chapter 5 we present a technique for separating a website into a group of layout pages and structured data sources. The approach uses a novelty based algorithm to select a set of representative pages and uses them to classify the rest of the pages to template clusters. It then aligns pages in each cluster with their representative to synthesize a layout code file and extract a list of structured data sources. Our evaluation shows high effectiveness of our site-level separation for structural summarization, clustering and alignment of web pages.

6.2 Open Questions

Separation of web-applications into components: We addressed the problem of page-level and site-level separation into layout code and data. We showed that the resulting separation has a lot of interesting applications, like representation compression and data-extraction. While, this separation is useful for extraction from websites, it could be less efficient for extracting data in more complicated cases. Data extraction from e-mail, as an example, requires dealing with dynamically changing templates. While these templates change more frequently than web-page templates in websites, they are often built from a pre-defined set of components that do not change (or change much less frequently). Identifying these shared components among different e-mail templates can help produce an efficient and automatic data-extractor that not only deals with seen e-mail templates, but also with future ones. Another interesting application for separation of web-applications into components, is cross-site data extractors: while page templates may differ among different sites, many sites share fragments of their page templates with other sites (thanks to open source publishing platforms and the use of pre-built plugins and components). Learning to identify and understand the structure of these shared components can facilitate cross-site data extraction.

Automatic synthesis of web-scale data extractors:

The goal of web-scale data extraction is to construct a unified Web knowledge base that gives us access to data aggregated from all websites available online. While the methods we proposed in this work address the problems of page-level and record-level extraction from websites, it would be interesting to research the possibility of integrating them (or other solutions inspired by them) inside a system for web-scale data extraction.

Separation beyond web applications: In our separation work in chapter 4 and chapter 3, we address the separation problem on web-pages. What enables such separation is our knowledge about the syntax of these template scripting languages, in addition to the similarities and the functional equivalence of most these scripting languages. Other document types (logs for example) in the other hand, do not have the same level of shared similarities and often lack the well-defined scripting constructs that facilitate the separation of web-applications.

Bibliography

[AGM03] Arvind Arasu and Hector Garcia-Molina. Extracting structured data from web pages. In *SIGMOD*, 2003.

[AGWC07] Yoo Jung An, James Geller, Yi-Ta Wu, and Soon Chun. Semantic deep web: automatic attribute extraction from the deep web data sources. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1667–1672. ACM, 2007.

[Ant05] Tobias Anton. Xpath-wrapper induction by generalizing tree traversal patterns. In *Lernen, Wissensentdeckung und Adaptivitt (LWA) 2005, GI Workshops, Saarbrcken*, pages 126–133, 2005.

[But04] David Buttler. A short survey of document structure similarity algorithms. In *ICOMP*, pages 3–9, 2004.

[BYR02] Ziv Bar-Yossef and Sridhar Rajagopalan. Template detection via data mining and its applications. In *WWW'02*, 2002.

[CCCD16] Chia-Hui Chang, Tian-Sheng Chen, Ming-Chuan Chen, and Jhung-Li Ding. Efficient page-level data extraction via schema induction and verification. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 478–490. Springer, 2016.

[CCDW04] Fabio Ciravegna, Sam Chapman, Alexiei Dingli, and Yorick Wilks. Learning to harvest information for the semantic web. In *The Semantic Web: Research and Applications*, pages 312–326. Springer, 2004.

[CCSA11] Charles L.A. Clarke, Nick Craswell, Ian Soboroff, and Azin Ashkan. A comparative analysis of cascade measures for novelty and diversity. In *WSDM*, 2011.

[CD$^+$99] James Clark, Steve DeRose, et al. Xml path language (xpath). *W3C recommendation*, 16, 1999.

[CDB15] Joseph Paul Cohen, Wei Ding, and Abraham Bagherjeiran. Semi-supervised web wrapper repair via recursive tree matching. *arXiv preprint arXiv:1505.01303*, 2015.

[CDC04] Sam Chapman, Alexiei Dingli, and Fabio Ciravegna. Armadillo: harvesting information for the semantic web. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 598–598. ACM, 2004.

[CH04] Shui-Lung Chuang and JY-j Hsu. Tree-structured template generation for web pages. In *Web Intelligence, 2004. WI 2004. Proceedings. IEEE/WIC/ACM International Conference on*, pages 327–333. IEEE, 2004.

[CJL$^+$11] Olivier Chapelle, Shihao Ji, Ciya Liao, Emre Velipasaoglu, Larry Lai, and Su-Lin Wu. Intent-based diversification of web search results: metrics and algorithms. *Information Retrieval*, 14(6):572–592, 2011.

[CKC$^+$08a] Charles LA Clarke, Maheedhar Kolla, Gordon V Cormack, Olga Vechtomova, Azin Ashkan, Stefan Büttcher, and Ian MacKinnon. Novelty and diversity in information retrieval evaluation. In *SIGIR*, 2008.

[CKC$^+$08b] Charles L.A. Clarke, Maheedhar Kolla, Gordon V. Cormack, Olga Vechtomova, Azin Ashkan, Stefan Büttcher, and Ian MacKinnon. Novelty and diversity in information retrieval evaluation. In *SIGIR*, 2008.

[CKGS06] Chia Hui Chang, Mohammed Kayed, M.R. Girgis, and K.F. Shaalan. A survey of web information extraction systems. *IEEE Trans. on Knowledge and Data Engineering*, 18(10), 2006.

[CKP07] Deepayan Chakrabarti, Ravi Kumar, and Kunal Punera. Page-level template detection via isotonic smoothing. In *Proc. of the international conf. on World Wide Web*, 2007.

[CL] Chia-Hui Chang and Shao-Chen Lui. IEPAD: Information extraction based on pattern discovery. In *WWW '01*.

[CL01] Chia-Hui Chang and Shao-Chen Lui. IEPAD: information extraction based on pattern discovery. In *WWW*, 2001.

[CMM01] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, pages 109–118, 2001.

[CMM$^+$02] David Carmel, YS Maarek, Y Mass, N Efraty, and GM Landau. An extension of the vector space model for querying xml documents via xml fragments. In *SIGIR Workshop on XML and Information Retrieval*, 2002.

[CYWM03] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. Vips: a vision-based page segmentation algorithm. Technical report, Microsoft technical report, MSR-TR-2003-79, 2003.

[DBS09a] Nilesh Dalvi, Philip Bohannon, and Fei Sha. Robust web extraction: An approach based on a probabilistic tree-edit model. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 335–348, New York, NY, USA, 2009. ACM.

[DBS09b] Nilesh Dalvi, Philip Bohannon, and Fei Sha. Robust web extraction: an approach based on a probabilistic tree-edit model. In *SIGMOD*, pages 335–348. ACM, 2009.

[DC] Miss Poonam Rangnath Dholi and KP Chaudhari. Template extraction from heterogeneous web pages using MDL principle.

[DCWS06] Theodore Dalamagas, Tao Cheng, Klaas-Jan Winkel, and Timos Sellis. A methodology for clustering xml documents by structure. *Information Systems*, 31(3):187–228, 2006.

[DEG$^+$03] Stephen Dill, Nadav Eiron, David Gibson, Daniel Gruhl, R Guha, Anant Jhingran, Tapas Kanungo, Sridhar Rajagopalan, Andrew Tomkins, John A Tomlin, et al. Semtag and seeker: Bootstrapping the semantic web via automated semantic annotation. In *Proceedings of the 12th international conference on World Wide Web*, pages 178–186. ACM, 2003.

[DKS11] Nilesh Dalvi, Ravi Kumar, and Mohamed Soliman. Automatic wrappers for large scale web extraction. *Proceedings of the VLDB Endowment*, 4(4):219–230, 2011.

[FDMFB14] Emilio Ferrara, Pasquale De Meo, Giacomo Fiumara, and Robert Baumgartner. Web data extraction, applications and techniques: A survey. *Knowledge-based systems*, 70:301–323, 2014.

[FFT05] Bettina Fazzinga, Sergio Flesca, and Andrea Tagarelli. Learning robust web wrappers. In *Database and Expert Systems Applications*, pages 736–745. Springer, 2005.

[FK04] Aidan Finn and Nicholas Kushmerick. *Multi-level boundary classification for information extraction*. Springer, 2004.

[FWB$^+$11a] Fabio Fumarola, Tim Weninger, Rick Barber, Donato Malerba, and Jiawei Han. Extracting general lists from web documents: A hybrid approach. In *IEA/AIE'11*, 2011.

[FWB$^+$11b] Fabio Fumarola, Tim Weninger, Rick Barber, Donato Malerba, and Jiawei Han. Hylien: a hybrid approach to general list extraction on the web. In *WWW*, pages 35–36. ACM, 2011.

[GF14] Bo Gao and Qifeng Fan. Multiple template detection based on segments. In *Advances in Data Mining. Applications and Theoretical Aspects*, pages 24–38. Springer, 2014.

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *ACM SIGPLAN Notices*, volume 46, pages 62–73. ACM, 2011.

[GM07] Evgeniy Gabrilovich and Shaul Markovitch. Computing semantic relatedness using wikipedia-based explicit semantic analysis. In *IJCAI*, volume 7, pages 1606–1611, 2007.

[GM13] Filippo Geraci and Marco Maggini. A fast method for web template extraction via a multi-sequence alignment approach. In *Knowledge Discovery, Knowledge Engineering and Knowledge Management*, pages 172–184. Springer, 2013.

[GMM$^+$11] Pankaj Gulhane, Amit Madaan, Rupesh Mehta, Jeyashankher Ramamirtham, Rajeev Rastogi, Sandeepkumar Satpal, Srinivasan H Sengamedu, Ashwin Tengli, and Charu Tiwari. Web-scale information extraction with vertex. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1209–1220. IEEE, 2011.

[GPT05] David Gibson, Kunal Punera, and Andrew Tomkins. The volume and evolution of web page templates. In *Special interest tracks and posters of WWW*, pages 830–839. ACM, 2005.

[GRB$^+$14] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663. ACM, 2014.

[Gri13] Tomas Grigalis. Towards web-scale structured web data extraction. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 753–758. ACM, 2013.

[Grü07] Peter D Grünwald. *The minimum description length principle*. MIT press, 2007.

[GZAC13] Anna Lisa Gentile, Ziqi Zhang, Isabelle Augenstein, and Fabio Ciravegna. Unsupervised wrapper induction using linked data. In *Proceedings of the Seventh International Conference on Knowledge Capture*, K-CAP '13, pages 41–48, New York, NY, USA, 2013. ACM.

[HAF$^+$10] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Data structure fusion. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010*, pages 204–221, 2010.

[HCH04] Bin He, Kevin Chen-Chuan Chang, and Jiawei Han. Discovering complex matchings across web query interfaces: a correlation mining approach. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 148–157. ACM, 2004.

[HCPZ11] Qiang Hao, Rui Cai, Yanwei Pang, and Lei Zhang. From one tree to a forest: a unified solution for structured web data extraction. In *SIGIR*, 2011.

[HFH$^+$09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[Hon11] Jer Lang Hong. Data extraction for deep web using wordnet. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 41(6):854–868, 2011.

[HY01] Mark H Hansen and Bin Yu. Model selection and the principle of minimum description length. *Journal of the American Statistical Association*, 96(454):746–774, 2001.

[HZL$^+$11] Raphael Hoffmann, Congle Zhang, Xiao Ling, Luke Zettlemoyer, and Daniel S Weld. Knowledge-based weak supervision for information extraction of overlapping relations. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 541–550. Association for Computational Linguistics, 2011.

[Jac] Paul Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, 11:37–50.

[JGS$^+$10] Susmit Jha, Sumit Gulwani, Sanjit Seshia, Ashish Tiwari, et al. Oracle-guided component-based program synthesis. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 215–224. IEEE, 2010.

[JL95] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, San Mateo, 1995. Morgan Kaufmann.

[JWF$^+$10] Lu Jiang, Zhaohui Wu, Qian Feng, Jun Liu, and Qinghua Zheng. Efficient deep web crawling using reinforcement learning. In *Advances in Knowledge Discovery and Data Mining*, pages 428–439. Springer, 2010.

[KC10] Mohammed Kayed and Chia-Hui Chang. Fivatech: Page-level web data extraction from template pages. *Knowledge and Data Engineering, IEEE Transactions on*, 22(2):249–263, 2010.

[KFN10] Christian Kohlschütter, Peter Fankhauser, and Wolfgang Nejdl. Boilerplate detection using shallow text features. In *Web Search and Data Mining (WSDM)*, WSDM '10, 2010.

[Kle99] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5), 1999.

[KS11] Chulyun Kim and Kyuseok Shim. Text: Automatic template extraction from heterogeneous web pages. *Knowledge and Data Engineering, IEEE Transactions on*, 23(4), 2011.

[Kus00] Nicholas Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1):15–68, 2000.

[KWD] Nicholas Kushmerick, Daniel S. Weld, and Robert B. Doorenbos. Wrapper induction for information extraction. In *IJCAI'97*, pages 729–737.

[LG14] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 55. ACM, 2014.

[LGZ03a] Bing Liu, Robert Grossman, and Yanhong Zhai. Mining data records in web pages. In *KDD*, KDD '03, pages 601–606, 2003.

[LGZ03b] Bing Liu, Robert Grossman, and Yanhong Zhai. Mining data records in web pages. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–606. ACM, 2003.

[LLYZ08] Jianxin Li, Chengfei Liu, Jeffrey Xu Yu, and Rui Zhou. Efficient top-k search across heterogeneous XML data sources. In *Database Systems for Advanced Applications (DASFAA)*, 2008.

[LMM06] Wei Liu, Xiaofeng Meng, and Weiyi Meng. Vision-based web data records extraction. In *Proc. 9th International Workshop on the Web and Databases*, pages 20–25, 2006.

[LMM10] Wei Liu, Xiaofeng Meng, and Weiyi Meng. Vide: A vision-based approach for deep web data extraction. *Knowledge and Data Engineering, IEEE Transactions on*, 22(3):447–460, 2010.

[LPH00] Ling Liu, Calton Pu, and Wei Han. Xwrap: An xml-enabled wrapper construction system for web information sources. In *Proc. of International Conference on Data Engineering*, pages 611–621, 2000.

[LSRT14] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Reducing web test cases aging by means of robust xpath locators. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 449–454. IEEE, 2014.

[LWLY12] Donglan Liu, Xinjun Wang, Hong Li, and Zhongmin Yan. Robust web extraction based on minimum cost script edit model. *Procedia Engineering*, 29:1119–1125, 2012.

[MBDH05] Jayant Madhavan, Philip A Bernstein, AnHai Doan, and Alon Halevy. Corpus-based schema matching. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 57–68. IEEE, 2005.

[MBSJ09] Mike Mintz, Steven Bills, Rion Snow, and Dan Jurafsky. Distant supervision for relation extraction without labeled data. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2*, pages 1003–1011. Association for Computational Linguistics, 2009.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[Mit97] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[MK07] Matthew Michelson and Craig A Knoblock. Unsupervised information extraction from unstructured, ungrammatical data sources on the world wide web. *International Journal of Document Analysis and Recognition (IJDAR)*, 10(3-4):211–226, 2007.

[MTH$^+$09] Gengxin Miao, Junichi Tatemura, Wang-Pin Hsiung, Arsany Sawires, and Louise E. Moser. Extracting data records from the web using tag path clustering. In *WWW*, WWW ’09, 2009.

[NBdT16] Joachim Nielandt, Antoon Bronselaer, and Guy de Tré. Predicate enrichment of aligned xpaths for wrapper induction. *Expert Systems with Applications*, 2016.

[NDMBDT14] Joachim Nielandt, Robin De Mol, Antoon Bronselaer, and Guy De Tré. Wrapper induction by xpath alignment. In *6th International Conference on Knowledge Discovery and Information Retrieval (KDIR 2014)*, volume 6, pages 492–500. Science and Technology Publications, 2014.

[OCRS16] Adi Omari, David Carmel, Oleg Rokhlenko, and Idan Szpektor. Novelty based ranking of human answers for community questions. In *SIGIR*. ACM, 2016.

[OKYS16] Adi Omari, Benny Kimelfeld, Eran Yahav, and Sharon Shoham. Lossless separation of web pages into layout code and data. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16. ACM, 2016.

[OSY16] Adi Omari, Sharon Shoham, and Eran Yahav. Cross-supervised synthesis of web-crawlers. In *Proceedings of the 38th International Conference on Software Engineering*, pages 368–379. ACM, 2016.

[OSY17] Adi Omari, Sharon Shoham, and Eran Yahav. Synthesis of forgiving data extractors. In *WSDM*, 2017.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[Qui] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106.

[Qui14] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[RGSL04] Davi De Castro Reis, Paulo Braz Golgher, Altigran Soares Silva, and AF Laender. Automatic web news extraction using tree edit distance. In *WWW*, pages 502–511. ACM, 2004.

[Ris78] Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[RIS98] RISE. Rise: A repository of online information sources used in information extraction tasks. *[http://www.isi.edu/integration/RISE/index.html]*, 1998.

[RM$^+$99] Jason Rennie, Andrew McCallum, et al. Using reinforcement learning to spider the web efficiently. In *ICML*, volume 99, pages 335–343, 1999.

[RMS06] Davood Rafiei, Daniel L Moise, and Dabo Sun. Finding syntactic similarities between xml documents. In *DEXA'06*, 2006.

[SC13] Hassan A Sleiman and Rafael Corchuelo. Tex: An efficient and effective unsupervised web information extractor. *Knowledge-Based Systems*, 39:109–123, 2013.

[SC$^+$14] Hassan Sleiman, Rafael Corchuelo, et al. Trinity: on using trinary trees for unsupervised web data extraction. *IEEE Trans. on Knowledge and Data Engineering*, 26(6), 2014.

[SL05] Kai Simon and Georg Lausen. Viper: augmenting automatic information extraction with visual perceptions. In *Information and knowledge management*, 2005.

[SM02] Torsten Schlieder and Holger Meuss. Querying and ranking xml documents. *Journal of the Association for Information Science and Technology*, 53(6):489–503, 2002.

[SWL$^+$12] Dandan Song, Yunpeng Wu, Lejian Liao, Long Li, and Fei Sun. A dynamic learning framework to thoroughly extract structured data from web pages without human efforts. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, page 9. ACM, 2012.

[TW13] Wachirawut Thamviset and Sartra Wongthanavasu. Information extraction for deep web using repetitive subject pattern. *World Wide Web*, pages 1–31, 2013.

[TW14] Wachirawut Thamviset and Sartra Wongthanavasu. Information extraction for deep web using repetitive subject pattern. *World Wide Web*, 17(5):1109–1139, 2014.

[VdSP$^+$06] Karane Vieira, Altigran S da Silva, Nick Pinto, Edleno S de Moura, Joao Cavalcanti, and Juliana Freire. A fast and robust method for web page template detection and removal. In *Information and knowledge management*, 2006.

[VS05] VG Vinod Vydiswaran and Sunita Sarawagi. Learning to extract information from large websites using sequential models. In *COMAD*, pages 3–14, 2005.

[W$^+$98] Lauren Wood et al. Document object model (dom) level 1 specification. *W3C Recommendation*, 1, 1998.

[WL03] Jiying Wang and Fred H Lochovsky. Data extraction and label assignment for web databases. In *WWW*, 2003.

[WLF15] Shanchan Wu, Jerry Liu, and Jian Fan. Automatic web content extraction by combination of learning and grouping. In *Proc. of the International Conf. on World Wide Web*, 2015.

[WPC$^+$15] Tim Weninger, Rodrigo Palácios, Valter Crescenzi, Thomas Gottron, and Paolo Merialdo. Web content extraction - a meta-analysis of its past and thoughts on its future. *CoRR*, abs/1508.04066, 2015.

[YCNH04] Yasuhiro Yamada, Nick Craswell, Tetsuya Nakatoh, and Sachio Hirokawa. Testbed for information extraction from deep web. In *Proceedings of the international World Wide Web conference on Alternate track papers & posters*, 2004.

[ZL05] Yanhong Zhai and Bing Liu. Web data extraction based on partial tree alignment. In *WWW*, 2005.

[ZNW$^+$06] Jun Zhu, Zaiqing Nie, Ji-Rong Wen, Bo Zhang, and Wei-Ying Ma. Simultaneous record detection and attribute labeling in web data extraction. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, 2006.

[ZSWG09] Shuyi Zheng, Ruihua Song, Ji-Rong Wen, and C Lee Giles. Efficient record-level wrapper induction. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 47–56. ACM, 2009.

במקרה והם קיימים ולהאריך את ארוך חיי שאילתות חילוץ המידע. היכולת לקבוע את הדיוק של השאילות באופן דינמי, מאפשרת להן להסתכל עם שינויים מבניים בלי הצורך לבחור בין דיוק וכלליות מההתחלה ובלי לוותר על דיוק מראש ושלא לצורך.

בשני הפרקים האחרונים התייחסנו לבעיית החילוץ הבלתי מונחה של מידע. כדי לפתור את בעיית החילוץ הבלתי מונחה של מידע, הצענו פתרון לבעיה היותר כללית של פיצול דפי־אינטרנט חזרה לקוד־תבנית ונתונים. דפי אינטרנט מיוצרים לרוב על ידי הרצת קוד־תבנית על נתונים כדי ליצור מסמך מעוצב שמציג את הנתונים בצורה שנוחה לצריכה על ידי משתמשים אנושיים. בפתרון שלנו אנחנו מתייחסים למשימה ההפוכה: פיצול של דף אינטרנט נתון חזרה למרכיב של תבנית ומרכיב של נתונים. לפיצול זה יש מגוון רחב של אפליקציות חשובות: חילוץ לא מונחה של נתונים, דחיסה של ייצוג דפי אינטרנט לייעול התעבורה, הפשטה של תבניות והתאמתן למשתמש, ועוד.

תקציר

חילוץ מידע מהאינטרנט הוא נושא מחקר חשוב שנחקר בהרחבה ומקבל הרבה צומת לב בשנים האחרונות. כמויות גדולות של נתונים מיוצרות ונצרכות באופן יומי באינטרנט בקצב שהולך וגדל. היכולת לחלץ ולנתח נתונים הפכה ליכולת חיונית למגוון רחב של יישומים, ולשיפור האפקטיביות של עסקים מודרניים. שיטות חילוץ המידע מאפשרות ומקלות על איסוף וניתוח של נתונים על ידי חילוץ נתונים אלה ממקורות שנועדו לצריכה אנושית ומקורות עם ייצוגים שונים של המידע לתוך מסד נתונים עם מבנה מוגדר שמאפשר ניתוח אוטומטי של הנתונים ומקל על הבנתם על ידי מכונות. בעבודה זו אנו מתייחסים לבעיה של חילוץ הנתונים כבעיה של סינתזה של תוכנה. המטרה שלנו היא לסנתז אוטומטית תוכנות לחילוץ המידע. הפופולריות והשימוש הנרחב בשפות השאילתות לכתיבת תוכנות לחילוץ מידע הופכים אותם למטרה טבעית עבור שיטות סינתזה של תוכנה. סיבה נוספת להסתכל על בעיית חילוץ המידע מנקודת מבט של סנתיזה של תוכנה היא העובדה כי מרבית דפי האינטרנט ומקורות המידע נבנים באמצעות קוד תבנית. בחלק מהעובדה אנחנו משתמשים בעובדה זו ובודקים את התרומה של שימוש בשיטות הנדסה לאחור)בנוסף לשיטות אחרות(כדי לאפשר ולשפר חילוץ בלתי מונחה של מידע.

הפרק הראשון של מחקר זה מתמקד בבעיית הסנתיזה האוטומטית של סורקי אינטרנט לחילוץ מידע מקבוצה של אתרים. האתרים בקבוצה שונים מבחינת עיצוב ומבנה אך מכילים אותו סוג של מידע. אנחנו מציעים שיטת סינתזה שמשתמשת בנתונים המשותפים בין אתרים מאותה קטגוריה כדי למזער)והרבה פעמים אפילו לבטל(את הצורך במתכנת או סוקר אנושי. השיטה מקבלת כקלט תוכנה לחילוץ מידע מאחד האתרים בקבוצה, ומשתמשת בנתונים שהיא מחלצת מאתר זה כדוגמאות שבעזרתן היא מזהה מופעים של הנתונים באתרים אחרים)על ידי זיהוי אותן דוגמאות באתרים האחרים(. לאחר מכן מופעים אלה משמים כבסיס ללמידת המיקום שבו מזוהים מופעי הנתונים באתר ולסנתיזה אוטומטית של תוכנה לחילוץ מידע מהאתר בו זוהו. בסוף כל סבב, תוכנות חילוץ המידע החדשות שנלמדו, מופעלות כל אחת על אתר היעד שלה כדי לשלוף נתונים חדשים ולהרחיב את מסד הנתונים של הדוגמאות. התהליך הזה חוזר על עצמו עד שיש תוכנת חילוץ מידע לכל אתר בקבוצה או עד שלא מצליחים לחלץ דוגמאות חדשות.

אספקט נוסף של בעיית הסנתיזה של תוכנות חילוץ מידע שהתייחסנו אליו בעבודה שלנו הוא האספקט של עמידות תוכנת חילוץ המידע בפני שינויים מבניים באתר שבשבילו נוצרו)או נכתבו במקרה של מתכנת אנושי(. בעיית הסנתיזה של תוכנות חילוץ עמידות בפני שינויים מבניים היא ההתמקדות שלנו בפרק השני של מחקר זה. הקלט הוא קבוצה של מקורות מידע)אתרי אינטרנט(מאותה משפחה)מכילים אותו סוג של נתונים(שהמבנה של הדפים שלהם שמכילים את המידע יכול להשתנות עם הזמן כתוצאה משינוי התבנית שבונה אותם. אנו מציעים וממשים את רעיון שאילתות חילוץ המידע הסלחניות, שמסוגלות לשנות את הדיוק שלהן באופן דינמי כדי לטפל ולהסתגל עם השינויים המבניים

המחקר בוצע בהנחייתם של פרופסור ערן יהב ודוקטור שרון שוהם, בפקולטה למדעי המחשב

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי־עת במהלך תקופת מחקר הדוקטורט של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Adi Omari, David Carmel, Oleg Rokhlenko, and Idan Szpektor. Novelty based ranking of human answers for community questions. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 215–224. ACM, 2016.

Adi Omari, Benny Kimelfeld, Eran Yahav, and Sharon Shoham. Lossless separation of web pages into layout code and data. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1805–1814. ACM, 2016.

Adi Omari, Sharon Shoham, and Eran Yahav. Cross-supervised synthesis of web-crawlers. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 368–379. IEEE, 2016.

Adi Omari, Sharon Shoham, and Eran Yahav. Synthesis of forgiving data extractors. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 385–394. ACM, 2017.

תודות

הכרת תודה מסורה לטכניון על מימון מחקר זה.

חילוץ מידע באמצעות סינטזה של תכניות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

עדי עומרי

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
ניסן תשע"ח   חיפה   מרץ 2018

חילוץ מידע באמצעות סינטזה של תכניות

עדי עומרי