Lossless Separation of Web Pages into Layout Code and Data

Adi Omari, Benny Kimelfeld, Eran Yahav
Technion
{omari,bennyk,yahave}@cs.technion.ac.il

Sharon Shoham
Tel Aviv University
sharon.shoham@gmail.com

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.

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.

KDD '16, August 13-17, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-4232-2/16/08...$15.00
DOI: http://dx.doi.org/10.1145/2939672.2939858

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 [1, 3, 6, 8, 18, 19, 24–28, 32–35, 37, 42]. Techniques have also been presented for *wrapper induction*, using the template structure of a page to produce a wrapper that extracts particular data elements [7, 10, 22]. 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 [2, 21, 24, 29, 42] and template-extraction [4, 9, 13–15, 20, 36]. 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 [31] principle. Our approach could be applied to any form of tree-structured data, and can be used for applications such as tree retrieval [23].

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.

2 Related Work

There has been a considerable amount of work on page-level data extraction (e.g., [1, 5, 12, 19, 33, 34, 40]) and record-level extraction (e.g., [3, 6, 11, 24, 26, 27, 32, 35, 37, 42]). 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 [19] 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 [34] extracts data, but does not extract the template or the schema of the data. Trinity [33] 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 [7] 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 [1] uses the concept of equivalence classes and "differentiating roles" to discover a template, which is a regular expression. TPC [28] 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 [35] 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 [24] and DEPTA [42] use tag strings representation of DOM nodes to compare individual nodes and combinations of adjacent nodes. Similar individual nodes or node combinations are considered as generalized nodes, and sequences of generalized nodes

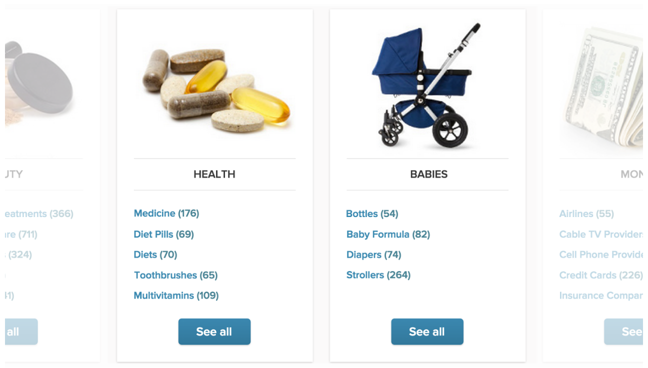


Figure 1: A simple webpage.

```
<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 2: A sample static html snippet that we would like to separate into code and data.

are considered as a data region. DEPTA [42] uses partial tree alignment to align generalized nodes and extract their data. DeLa [37] automatically generate regular expression wrappers based on the page HTML-tag structures to extract data objects from the result pages. IEPAD [6] 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., [19, 28, 35, 42]), 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.

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.

3.1 Motivating Example

Fig. 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

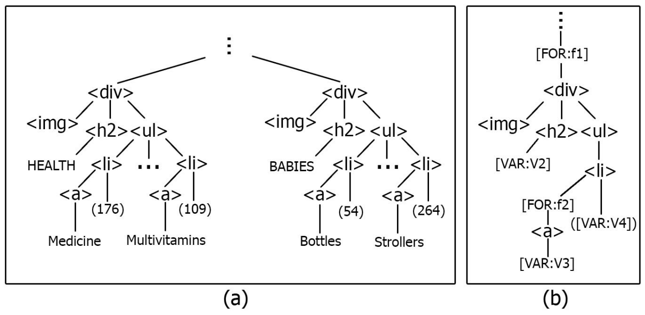


Figure 3: (a) The DOM tree of the original HTML document, and (b) the layout tree produced by our approach from this DOM tree.

```
1  <% for(var loop1:f1){ %>
2  <div>
3  <img src="/<%=loop1.v1%>.jpg"/><h2><%=loop1.v2%></h2>
4  <ul class="stripped">
5  <% for(var loop1:loop2:f2){
6  <li><a href="/<%=loop2.v3%>">
7  <%=loop2.v3%></a>(<%=loop2.v4%>)</li>
8  <% } %>
9   </ul>
10  </div>
11 <% } %>
```

Figure 4: Code synthesized for the given static HTML.

```
1  f1:{
2   {v1:"health",v2:"HEALTH",
3    f2:{{v3:"Medcicine",v4:"176"},{v3:"Diet␣Pills",v4:"69"},
4         {v3:"Diets",v4:"70"},{v3:"Toothbrushes",v4:"65"},
5         {v3:"Multivitamins",v4:"109"}}
6   },
7   {v1:"babies",v2:"BABIES",
8    f2:{{v3:"Bottles",v4:"54"},{v3:"Baby␣Formula",v4:"82"},
9         {v3:"Diapers",v4:"74"},{v3:"Strollers",v4:"264"}}
10 }}
```

Figure 5: The data extracted for the given static HTML.

of a program that formats data into a web page. We formally define layout trees in Section 4.

Fig. 2 shows the HTML document of Fig. 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 [39]. Fig. 3(a) shows the DOM tree for the HTML document of Fig. 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. 4 shows the code synthesized by our technique for the page of Fig. 2. The code can also be viewed in tree form as the layout tree shown in Fig. 3(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 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) 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.

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

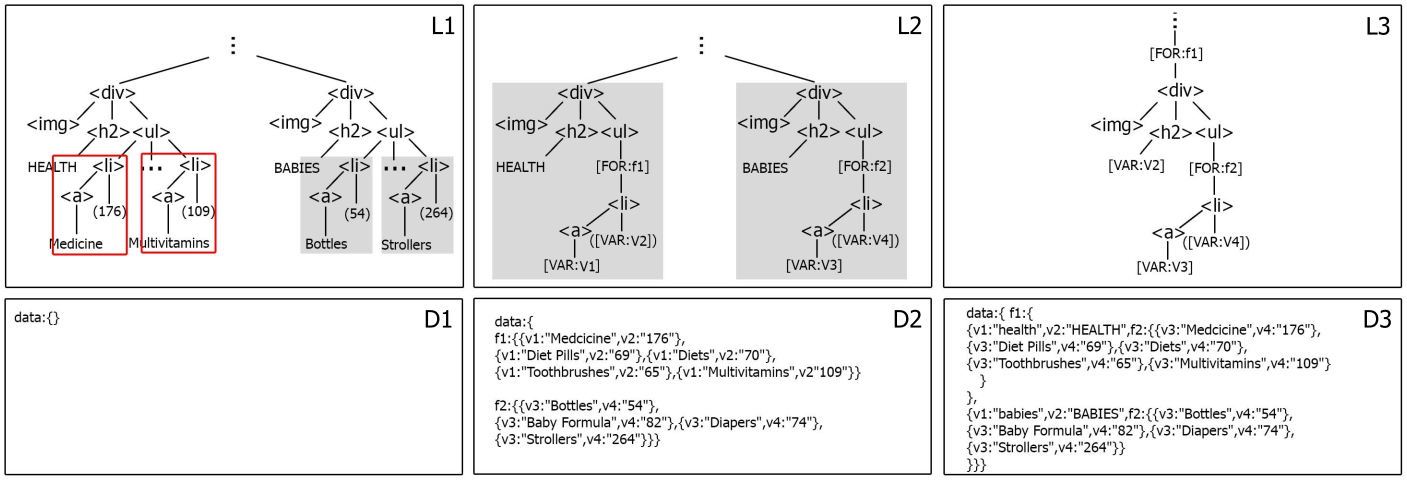
We address both of these questions using a novel *alignment* algorithm. We use the alignment algorithm as a building block for deciding when to fold subtrees, and also for computing the separation into layout tree and data when subtrees are folded.

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 1. *Fig. 6 shows a few steps of our algorithm applied to the (partial) tree of Fig. 3. Initially, the layout tree is the original DOM tree (L1), with no folding, and no extracted data (D1).*



```
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 6: Example steps of the separation algorithm.

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. 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. 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. 4. A graphical representation is shown in Fig. 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.

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 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.

[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.

5 Problem Definition

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

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.

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

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.

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) [31]. According to MDL, one should favor the model that gives the shortest description of the observed data [17]. 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 [16]. 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: $\mathsf{cost}(\pi, \mathcal{E}) \overset{\text{def}}{=} \mathsf{sizeof}(\pi) + \mathsf{sizeof}(\mathcal{E})$.

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.

6 Our Approach

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

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 $v[t_1, \dots, t_n]$ (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* $s_1, \dots, s_n$ into m chunks $(s_1, \dots, s_{j_1 - 1})$, $(s_{j_1}, \dots, s_{j_2 - 1}), \dots, (s_{j_m}, \dots, s_n)$ where, intuitively, each chunk consists of "similar" separations.
3. *Fold* each chunk $(s_{j_l}, \dots, s_{j_{l+1}-1})$ 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 $\mathsf{fold}(s_1, \dots, s_k)$ the procedure used in step 3 for folding a sequence $s_1, \dots, s_k$ 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).

6.2 Splitting

To split a sequence $s_1, \dots, s_n$ 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 $s_{j_l}, \dots, s_{j_l+q_l}$ of $s_1, \dots, s_n$ where $\sigma(s_i, s_{i+1})$ is larger than some fixed threshold for every $i = j_l, \dots, j_l + q_l - 1$. 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,

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

6.3 Folding

In the rest of the section, we describe the procedure fold. Recall that the input is a sequence $s_1, \dots, s_k$ 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 $\pi_1(\mathcal{E}_1) \cdots \pi_k(\mathcal{E}_k)$.

$\mathsf{fold}(s_1, \dots, s_k)$ 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 $\pi_1, \dots, \pi_k$. The differences between them are captured by conditional and reference nodes in π^c, along with an environment, $\mathcal{E}^c_i$, that is generated for each π_i. The environment $\mathcal{E}^c_i$ 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^c_i. Finally, the output environment $\mathcal{E}$ that accompanies π is constructed by

$$\mathcal{E} = \{x \mapsto \mathcal{E}^c_1(x) \cdot \ldots \cdot \mathcal{E}^c_k(x)\}.$$

REMARK 1. *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 $\mathcal{E}^c_1, \dots, \mathcal{E}^c_k$. This construction is done by applying on the input trees $\pi_1, \dots, \pi_k$ and their environments $\mathcal{E}_1, \dots, \mathcal{E}_k$ 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).

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* $\mathbf{E} = (\mathcal{E}_1, \dots, \mathcal{E}_k)$ 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 $(\Pi(\mathcal{E}_1), \dots, \Pi(\mathcal{E}_k))$ of DOM hedges.

DEFINITION 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:

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

6.4.1 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 2. *Consider the layout tree L3 in Fig. 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`*: $\mathcal{E}_{11} = \{v3 : \text{“}Medcicine\text{”}, v4 : \text{“176”}\}, \dots, \mathcal{E}_{15} = \{v3 : \text{“}Multivitamins\text{”}, v4 : \text{“109”}\}$, along with the four additional environments in the second list, $\mathcal{E}_{21} = \{v3 : \text{“}Bottles\text{”}, v4 : \text{“54”}\}, \dots, \mathcal{E}_{24} = \{v3 : \text{“}Strollers\text{”}, v4 : \text{“264”}\}$.*

6.4.2 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.

6.4.3 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 3. *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. *Consider the folding of the left-most sequence of `<li>` nodes in the layout tree L1 from Fig. 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. Multivitamins). 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 2. *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.*

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).

7.1 Evaluation of Data Extraction

7.1.1 Methodology

To evaluate the effectiveness of our approach for data extraction, we have used the common testbeds TBDW [41] and RISE [30]. We compare our approach to DEPTA [42], a technique that works on single pages, as well as to techniques that handle multiple pages: MDR [24], TPC [28], FivaTech, and Trinity (as reported by [33]).

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 [42], MDR [24], and TPC [28]. 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 [24, 28].

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 [33]. 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 [42] 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 [28] and MDR [24], we use the same partial set of 43 websites containing 213 web pages from TBDW used in [28]. 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 [33].

7.1.2 Results

The results of SYNTHIA when compared to DEPTA on the whole TBDW dataset are reported in Table 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 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 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%

Table 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 [28] and it excludes pages from TBDW containing nested 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 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 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

7.2 Evaluation of Code and Data Separation

7.2.1 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 [9], 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 [15] 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 [29, 36] 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.

7.2.2 Results

The results in Table 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: 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%**

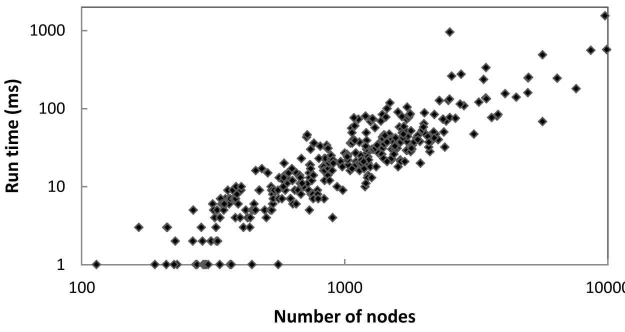


Figure 7: Running times as a function of the nodes count for the different documents in the two datasets

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. 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 $1sec$.

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 [38], 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.

Acknowledgment

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7) under grant agreement no. 615688 - ERC-COG-PRIME and the Israel Ministry of Science and Technology, grant no. 3-9779. The research of Benny Kimelfeld is supported by the Israeli Science Foundation, grant no. 1295/15 and grant no. 1308/15.

9 References

[1] ARASU, A., AND GARCIA-MOLINA, H. Extracting structured data from web pages. In *SIGMOD* (2003).

[2] BAR-YOSSEF, Z., AND RAJAGOPALAN, S. Template detection via data mining and its applications. In *WWW'02*.

[3] CAI, D., YU, S., WEN, J.-R., AND MA, W.-Y. Vips: a vision-based page segmentation algorithm. Tech. rep., Microsoft technical report, MSR-TR-2003-79, 2003.

[4] CHAKRABARTI, D., KUMAR, R., AND PUNERA, K. Page-level template detection via isotonic smoothing. In *Proc. of the international conf. on World Wide Web* (2007).

[5] CHANG, C. H., KAYED, M., GIRGIS, M., AND SHAALAN, K. A survey of web information extraction systems. *IEEE Trans. on Knowledge and Data Engineering 18*, 10 (2006).

[6] CHANG, C.-H., AND LUI, S.-C. IEPAD: information extraction based on pattern discovery. In *WWW* (2001).

[7] CRESCENZI, V., MECCA, G., AND MERIALDO, P. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB* (2001).

[8] DALVI, N., BOHANNON, P., AND SHA, F. Robust web extraction: an approach based on a probabilistic tree-edit model. In *SIGMOD* (2009).

[9] DHOLI, M. P. R., AND CHAUDHARI, K. Template extraction from heterogeneous web pages using MDL principle.

[10] FAZZINGA, B., FLESCA, S., AND TAGARELLI, A. Learning robust web wrappers. In *Database and Expert Systems Applications* (2005), Springer, pp. 736–745.

[11] FUMAROLA, F., WENINGER, T., BARBER, R., MALERBA, D., AND HAN, J. Extracting general lists from web documents: A hybrid approach. In *IEA/AIE'11* (2011).

[12] FUMAROLA, F., WENINGER, T., BARBER, R., MALERBA, D., AND HAN, J. Hylien: a hybrid approach to general list extraction on the web. In *WWW* (2011).

[13] GAO, B., AND FAN, Q. Multiple template detection based on segments. In *Advances in Data Mining. Applications and Theoretical Aspects*. Springer, 2014, pp. 24–38.

[14] GERACI, F., AND MAGGINI, M. A fast method for web template extraction via a multi-sequence alignment approach. In *KIC3K*. Springer, 2013, pp. 172–184.

[15] GIBSON, D., PUNERA, K., AND TOMKINS, A. The volume and evolution of web page templates. In *Special interest tracks and posters of WWW* (2005).

[16] GRÜNWALD, P. D. *The minimum description length principle*. MIT press, 2007.

[17] HANSEN, M. H., AND YU, B. Model selection and the principle of minimum description length. *Journal of the American Statistical Association 96*, 454 (2001), 746–774.

[18] HAO, Q., CAI, R., PANG, Y., AND ZHANG, L. From one tree to a forest: a unified solution for structured web data extraction. In *SIGIR* (2011).

[19] KAYED, M., AND CHANG, C.-H. Fivatech: Page-level web data extraction from template pages. *Knowledge and Data Engineering, IEEE Transactions on 22*, 2 (2010), 249–263.

[20] KIM, C., AND SHIM, K. Text: Automatic template extraction from heterogeneous web pages. *Knowledge and Data Engineering, IEEE Transactions on 23*, 4 (2011).

[21] KOHLSCHÜTTER, C., FANKHAUSER, P., AND NEJDL, W. Boilerplate detection using shallow text features. In *Web Search and Data Mining (WSDM)* (2010).

[22] KUSHMERICK, N., WELD, D. S., AND DOORENBOS, R. B. Wrapper induction for information extraction. In *IJCAI'97*.

[23] LI, J., LIU, C., YU, J. X., AND ZHOU, R. Efficient top-k search across heterogeneous XML data sources. In *Database Systems for Advanced Applications (DASFAA)* (2008).

[24] LIU, B., GROSSMAN, R., AND ZHAI, Y. Mining data records in web pages. In *KDD* (2003).

[25] LIU, D., WANG, X., LI, H., AND YAN, Z. Robust web extraction based on minimum cost script edit model. *Procedia Engineering 29* (2012), 1119–1125.

[26] LIU, W., MENG, X., AND MENG, W. Vision-based web data records extraction. In *Proc. 9th International Workshop on the Web and Databases* (2006), pp. 20–25.

[27] LIU, W., MENG, X., AND MENG, W. Vide: A vision-based approach for deep web data extraction. *Knowledge and Data Engineering, IEEE Transactions on 22*, 3 (2010), 447–460.

[28] MIAO, G., TATEMURA, J., HSIUNG, W.-P., SAWIRES, A., AND MOSER, L. E. Extracting data records from the web using tag path clustering. In *WWW* (2009).

[29] REIS, D. D. C., GOLGHER, P. B., SILVA, A. S., AND LAENDER, A. Automatic web news extraction using tree edit distance. In *WWW* (2004).

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

[31] RISSANEN, J. Modeling by shortest data description. *Automatica 14*, 5 (1978).

[32] SIMON, K., AND LAUSEN, G. Viper: augmenting automatic information extraction with visual perceptions. In *Information and knowledge management* (2005).

[33] SLEIMAN, H., CORCHUELO, R., ET AL. Trinity: on using trinary trees for unsupervised web data extraction. *IEEE Trans. on Knowledge and Data Engineering 26*, 6 (2014).

[34] SLEIMAN, H. A., AND CORCHUELO, R. Tex: An efficient and effective unsupervised web information extractor. *Knowledge-Based Systems 39* (2013).

[35] THAMVISET, W., AND WONGTHANAVASU, S. Information extraction for deep web using repetitive subject pattern. *World Wide Web* (2013).

[36] VIEIRA, K., DA SILVA, A. S., PINTO, N., DE MOURA, E. S., CAVALCANTI, J., AND FREIRE, J. A fast and robust method for web page template detection and removal. In *Information and knowledge management* (2006).

[37] WANG, J., AND LOCHOVSKY, F. H. Data extraction and label assignment for web databases. In *WWW* (2003).

[38] WENINGER, T., PALÁCIOS, R., CRESCENZI, V., GOTTRON, T., AND MERIALDO, P. Web content extraction - a meta-analysis of its past and thoughts on its future. *CoRR abs/1508.04066* (2015).

[39] WOOD, L., ET AL. Document object model (dom) level 1 specification. *W3C Recommendation 1* (1998).

[40] WU, S., LIU, J., AND FAN, J. Automatic web content extraction by combination of learning and grouping. In *WWW* (2015).

[41] YAMADA, Y., CRASWELL, N., NAKATOH, T., AND HIROKAWA, S. Testbed for information extraction from deep web. In *Proc. of the WWW conf. - papers & posters* (2004).

[42] ZHAI, Y., AND LIU, B. Web data extraction based on partial tree alignment. In *WWW* (2005).