Web data extraction using hybrid program synthesis: a combination of top-down and bottom-up inference

Mohammad Raza
Microsoft Corporation
Redmond, Washington
moraza@microsoft.com

Sumit Gulwani
Microsoft Corporation
Redmond, Washington
sumitg@microsoft.com

ABSTRACT

Automatic synthesis of web data extraction programs has been explored in a variety of settings, but in practice there remain various usability challenges around robustness, the amount of training effort required, the complexity of programs synthesized, as well as the ease of interaction in limited UI environments. In this work we address these challenges based on a novel program synthesis approach which combines the benefits of deductive (top-down) and enumerative (bottom-up) synthesis strategies. This yields a semi-supervised technique with which concise web data extraction programs expressible in standard XPath/CSS can be synthesized from a small number of user-provided examples. We demonstrate the effectiveness of our method in comparison to existing techniques in terms of overall accuracy, robust inference from a small number of examples, as well as inference of concise programs comparable to hand-written selectors. Our method has been deployed as a feature in the Microsoft Power BI product and released to millions of users.

CCS CONCEPTS

• **Information systems** → **Web mining**; *Markup languages*; • **Software and its engineering** → *Automatic programming*.

KEYWORDS

web data extraction, program synthesis, wrapper induction

ACM Reference Format:
Mohammad Raza and Sumit Gulwani. 2019. Web data extraction using hybrid program synthesis: a combination of top-down and bottom-up inference. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Since the early days of the web, the idea of automated synthesis of web data extraction programs from examples (or *wrapper induction*) has been explored in various forms to enable users to extract the semi-structured information available on the web into a structured format [25]. In the current age of big data, the emerging persona particularly interested in this area is that of data scientists, business intelligence analysts and other knowledge workers who regularly need to explore and extract information from various websites and incorporate such actions into their analysis workflows.

Although many specialized automated web extraction tools and services have become available in recent years (e.g. WIEN [25], STALKER [31], Lixto [6], Mozenda [22], import.io [20], SelectorGadget [43]), such technologies have generally targeted web extraction as an isolated task in specialized tools and have seen little adoption within the environments that data analysts commonly work in. In such environments, web extraction tasks may be interleaved with other data extraction, cleaning, integration and analysis steps that are part of workflows that must be flexible (quick and easy experimentation with data from different sources or websites) and re-executable on different datasets. Data analysts usually fall under the persona of users who are not hardcore programmers, but also not complete novices: they can manage lightweight scripting tasks for various data processing activities, but the more automation that can be provided, the better. The difficulties they face in web extraction tasks is evident from numerous online discussions in help forums or articles, as well as requests made to product teams.

For example, data scientists working in Python environments (e.g. Pandas dataframes in Jupyter Notebooks) commonly resort to using HTML parsing libraries (e.g. *Beautiful Soup* or *Scrapy*) which require them to hand-write code such as XPath or CSS expressions to extract data from webpages as part of their analysis scripts. This requires knowledge of these HTML query languages as well as the time and effort to examine and experiment with the schema of every

new website. Moreover, since the website schemas change frequently, the analysis scripts are also fragile and must be updated regularly. Figure 1 shows an example of a question posted on Stack Overflow where the user would like to extract information from a car catalog site into a Pandas dataframe but is unable to specify a correct selection logic for the data required. Web extraction support is similarly (if not more) limited in spreadsheet environments, e.g. Microsoft Excel or Google Spreadsheets limit web extraction to only explicit HTML tables or lists. Microsoft Power BI is a more advanced tool targeting business intelligence analysts, which also provides a spreadsheet-like interface but with support for automating analysis scripts: as the user performs various actions such as data extraction or transformation in the UI, the system automatically develops a program (in the lightweight "M" programming language) which the user can re-execute on different datasets and also manually inspect and edit for more advanced tasks. However, web extraction support has again been severely limited: Figure 2 illustrates a YouTube video in which an expert is demonstrating how to extract information from a web page in Power BI, through a complex and fragile process of first extracting the HTML source as text and then performing a long sequence of text manipulation operations, as seen in the "Applied Steps" section of the UI which shows the analysis script being created.

In this work we present a program synthesis approach for inferring web data extraction programs that addresses the difficulties faced by analysts in performing web extraction tasks. Although our approach is not limited to benefiting only this class of users and we show improvement over existing synthesis tools in general, it in particular addresses the features that are especially important in analytics environments. This includes robust inference from a small number of examples, learning concise readable programs in standard web languages, and inference from text-based examples. We now discuss these goals in more detail, most of which were identified in discussions with the Power BI product team.

Inference from few examples/robustness. An important usability challenge is for the system to make robust inference from a small number of examples provided by the user. For example, the webpage in Figure 1 contains 1671 data records. A user cannot be expected to provide all of these as examples even on a single webpage instance (as would be required in some previous wrapper induction approaches [35]). Instead, we must support robust inference from a handful of manually provided examples. The number of examples required should also be as small as possible as this indicates the robustness of the system and how much burden of verification/risk there is on the user: making sure that all of the data has been extracted and to identify any missing items. Existing approaches not aimed at minimizing examples often conservatively favour more specific programs [3, 27, 33, 34]

Stack Overflow question:

Target webpage:

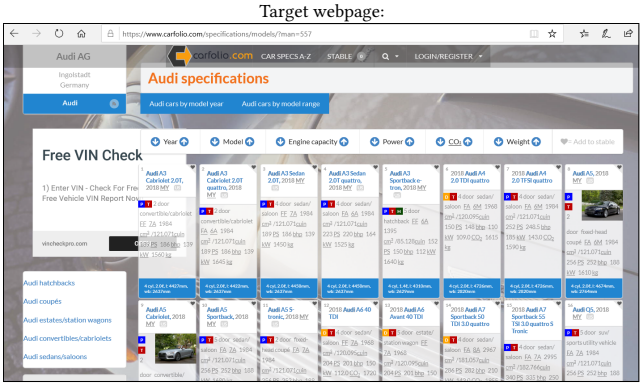
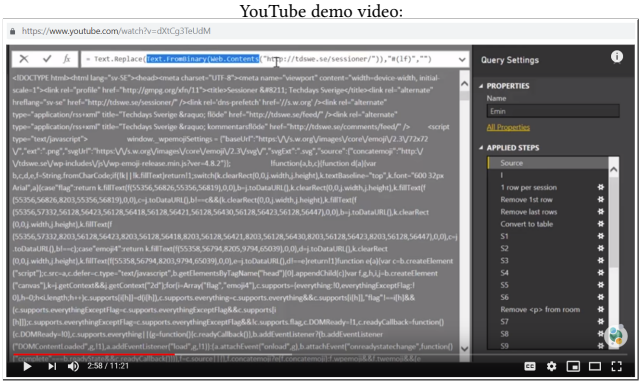


Figure 1: Forum question: web extraction in Python

which can overfit and require many examples to sufficiently generalize. This can lead to significant effort by the user to identify missing records to provide as additional examples at each interaction round, which can also be easy to overlook. For example, it will be difficult for the user to identify random elements missing from 1671 items in Figure 1. In general, correct inference from 2 examples for sequence extraction tasks in the majority of cases would help to instill user confidence in the robustness of the system.

Inferring simple programs in standard, lightweight languages. Another common requirement in practice is that the synthesized programs be represented in a standard lightweight and commonly available language, such as DOM query languages like XPath or CSS (which are W3C standards), so that users are likely to be or can easily become familiar with the language and its syntax and can also understand and manipulate the synthesized programs in common HTML tools or modern web browsers. In the case of the Power BI team, they were able to easily incorporate CSS selector expressions into their M language for analysis scripts.

YouTube demo video:



Target webpage:

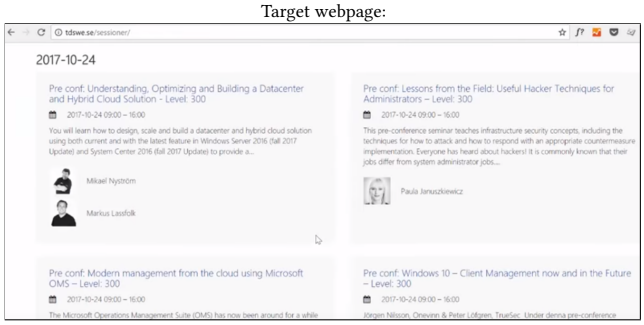


Figure 2: Demo video: web extraction in Power BI

This requirement is in contrast to wrapper induction approaches that often employ more complex extraction models, such as incorporating visual or semantic features or specialized treatment of particular verticals [7, 11, 18, 24, 36]. However, even when we consider standard web extraction languages, the simplicity and readability of the synthesized programs is also an important requirement. Existing synthesis approaches do not aim for concise programs that are easy for humans to understand, and their focus on improving accuracy and generality often yields very complex programs. For example, existing approaches for inferring XPath data selectors based on path alignment or least general generalizations [3, 33, 34, 37, 41] can lead to long path sequences from the root element or large conjunctions of predicates. Other approaches that aim to improve recall can lead to a large number of disjunctive path expressions to guard against overfitting [35, 40]. Such expressions can be difficult to interpret by users and are more complex than the simple selectors that a human expert may write for the same extraction task.

Text-based examples Many wrapper induction systems provide a visual interface in which users can select regions of the webpage using point-and-click actions to give examples of data items that are of interest. Such visual interfaces may not always be easily integrated into data analysis environments that commonly employ text-based interaction paradigms. For example, data scientists commonly write Python scripts in IDEs with intellisense or Jupyter Notebooks that employ a REPL (read-eval-print loop) interaction model, where text-based examples would be more natural. Other benefits of the text-based interaction paradigm include: *(1) Wide-scale adoption.* Web extraction support may be more easily integrated in different products and services as the text-based paradigm alleviates the need for heavy UI investment. *(2) Bypassing limitations of visual UIs.* Modern websites often employ dynamic scripts on webpages that alter the contents or layouts depending on user actions such as hovering or clicking on regions in the page, which is an obstacle for visual UIs. Another difficulty is the ambiguity that is often present in the DOM structure itself when text content is nested inside multiple nodes that visually appear in the same region. *(3) Robustness to changing webpage schemas/formats.* Analysis scripts can be more robust to site format changes: by maintaining the textual data examples, if a previously learned extraction program fails due to format changes then the analysis script can be automatically refreshed to learn a new program in the updated schema. *(4) Advanced explorations.* While not a focus of this work, the ability to infer extraction programs from purely text-based examples removes dependence to particular website formats and opens possibilities for more general explorations, e.g. integrating a search engine to infer data completions from arbitrary websites [1].

1.1 Key ideas and contributions

In this paper we present an approach to automatic synthesis of web data extraction programs that addresses the challenges that we have discussed above. We describe a method for inferring programs from text-based examples in a domain-specific language (DSL) that is expressible in XPath and CSS, and in contrast to existing approaches we demonstrate how the synthesized programs are simple (comparable to hand-written selectors) and can be inferred from a small number of examples. Our approach is based on the following key technical contributions:

Combination of top-down and bottom-up program synthesis. Our approach is based in the field of program synthesis, which has seen rising interest and progress in recent years [2, 10, 16, 26, 28, 37, 39, 44]. Given a fixed DSL (domain-specific language) the aim of a supervised program synthesis system is to find a program in the DSL that satisfies input-output examples given by the user. An efficient way of performing this search is *top-down*, where examples constraints are recursively propagated from candidate DSL operators to their parameters until a satisfying program is found [21, 27, 37, 40]. Such deductive approaches have the benefit of efficiently constraining the search to only those

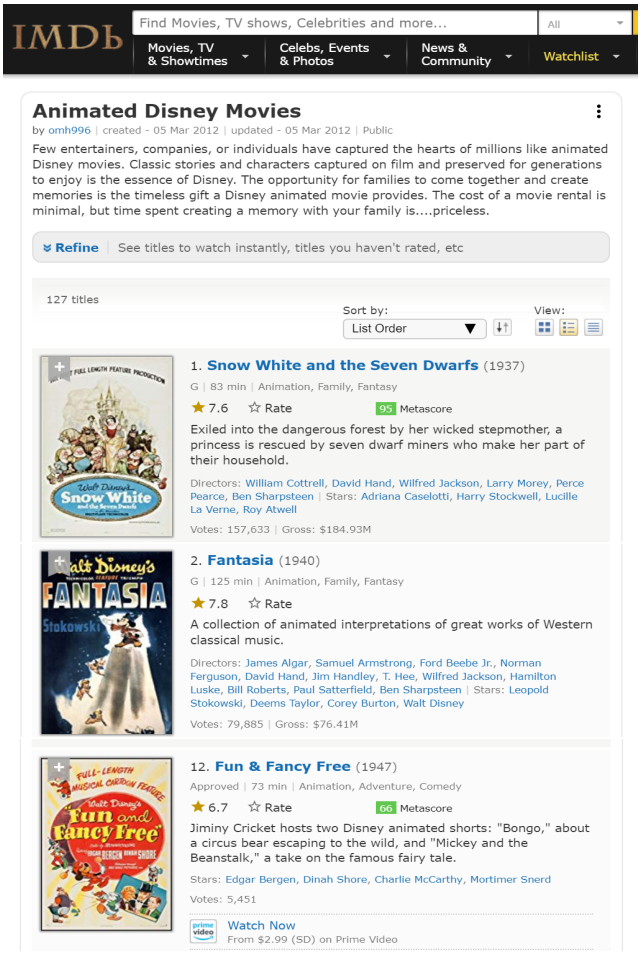


Figure 3: IMDB movies list page

```
<html>
  ...
  <div class="list_item odd">
      <div class="image"> ... </div>
      <div class="number">1.</div>
      <div class="info">
          <b>
              <a href="/title/tt0029583/">Snow White and the Seven Dwarfs</a>
              <span class="year_type">(1937)</span>
          </b>
          <div class="rating rating-list"> ... </div>
          <div class="item_description">
              Snow White, pursued by a jealous queen, hides with ...
              <span>(83 mins.)</span>
          </div>
          <div class="secondary">Director: William Cottrell, David Hand, ... </div>
          <div class="secondary">Stars: Adriana Caselotti, Harry Stockwell, ... </div>
          <div> ... </div>
      </div>
  </div>
  <div class="list_item even">
      <div class="image"> ... </div>
      <div class="number">2.</div>
      <div class="info">
          <b>
              <a href="/title/tt0032455/">Fantasia</a>
              <span class="year_type">(1940)</span>
          </b>
          <div class="rating rating-list"> ... </div>
          <div class="item_description">
              A collection of animated interpretations of great ...
              <span>(125 mins.)</span>
          </div>
          <div class="secondary">Director: James Algar, Samuel Armstrong, ... </div>
          <div class="secondary">Stars: Leopold Stokowski, Deems Taylor, ... </div>
          <div> ... </div>
      </div>
  </div>
  ...
  <div class="list_item even">
      <div class="image"> ... </div>
      <div class="number">12.</div>
      <div class="info">
          <b>
              <a href="/title/tt0039404/">Fun &amp; Fancy Free</a>
              <span class="year_type">(1947)</span>
          </b>
          <div class="rating rating-list"> ... </div>
          <div class="item_description">
              Jiminy Cricket hosts two Disney animated shorts: "Bongo," about ...
              <span>(70 mins.)</span>
          </div>
          <div class="secondary">Stars: Edgar Bergen, Dinah Shore, ... </div>
          <div> ... </div>
      </div>
  </div>
  ...
</html>
```

Figure 4: Simplified source HTML of IMDB movies list

DSL fragments relevant to the examples provided by the user. However, recent work has explored a *bottom-up* approach [39] for *unsupervised* program synthesis for web extraction where extraction programs are inferred without any examples from the user, e.g. automatically inferring a program to extract tabular data from a webpage without user examples. This approach proceeds by enumerating many DSL programs and then detecting alignment patterns between node selections produced by these programs, in order to find prominent data extraction patterns on the webpage without user-provided examples.

The two approaches have different strengths and weaknesses, and in this work we develop an integration of the bottom-up and top-down strategies. This amounts to a semi-supervised program synthesis approach that utilises the bottom-up analysis to improve the top-down inference of selectors. We illustrate this concept with a practical example which we shall discuss further as we describe our technique in more detail. Figure 3 shows a sample from an IMDB page containing a list of 100 movies, and Figure 4 shows a simplified version of the source HTML markup. In this case a bottom-up enumeration method was able to detect extraction programs for all 100 movie records and their various fields such as the movie year, running time, description, etc. However, notably the movie title could not be detected as it required a more complex selector. A purely top-down approach supported a more expressive DSL in which the title extraction was supported. However, given the first two example titles ("Snow Whilte and the Seven Dwarfs" and "Fantasia"), there are many possible extraction programs that can satisfy these two examples. The top-down approach yields a logic of selecting any <b> element that is the 6th child of its parent when counting from the end. This logic works for the first two movies, and all other movies except the 12th one ("Fun & Fancy Free"), because in this case the title is the 5th child from last since the director field is missing in this record. In our hybrid approach, we utilise the bottom-up analysis that detected all 100 movie records and use it to improve the top-down inference so that it synthesizes a better program generalizing to all records. This yields an

improved alternative selection logic that selects all <b> elements that are children of elements of class ".info". Thus a global bottom-up document analysis helps disambiguate between many possible alternative selection logics that may satisfy a small number of examples. To our knowledge, ours is the first *semi-supervised* approach to synthesis of XPath-style programs for web data extraction.

Predicate inference beyond least general conjunctions. The choice of which node selection logic to infer from a small set of example nodes is a key challenge in program synthesis methods. This is because there can be many properties that a small number of example nodes may share, and any conjunction of a subset of these properties (from the exponential possible subsets) can be a valid generalization of the examples. Common approaches usually adopt heuristic predicate preferences or favour largest conjunctions of all the common properties (least general generalizations) [3, 27, 34, 37, 41]. Apart from creating syntactically complex programs, these largest conjunctions easily cause overfitting by constraining to too many shared properties of the examples, and hence require many examples to infer correct programs. Apart from utilizing the bottom-up analysis to infer predicates that are consistent with global structure, we also describe a method to address such overfitting based on *soft negative examples*, which are nodes that are less likely to be part of the target selection (e.g. occurring outside common ancestors of example nodes). We show how concise predicate conjunctions can be inferred using a maximal set cover approach avoiding such negative examples.

Text-to-node disambiguation. For cases where node-based examples cannot be provided, our algorithm supports inference from text-only examples. In general there is no unique correspondence between a set of text examples and nodes in the webpage DOM, e.g. a flight search results page may contain the same airline name or flight times in many flight options or journey legs, or product search pages may contain the same price/brand names/date values for multiple products in the list. A set of a few text examples can combinatorially lead to hundreds of matching node combinations. In this work we address the text-to-node disambiguation problem by ranking possible node combinations using a number of structural features (e.g. common node attributes, topological similarities, and uniform node distances), as well as the global bottom-up document analysis.

2 WEB EXTRACTION LANGUAGE

In this section we describe the domain specific language (DSL) that we use for data extraction from webpages. Apart from the design consideration of programs being expressible in standard webpage query languages, another technical

$$
\begin{aligned}
\textbf{start}\ \text{Node[]}\ f &:= \text{Filter}(p, s) \mid \text{Disj}(f, ..., f) \\
\text{Node[]}\ s &:= \text{AllNodes}() \\
&\mid \underline{\text{ChildrenOf}(f)} \\
&\mid \underline{\text{DescendantsOf}(f)} \mid \underline{\text{RightSiblingsOf}(f)} \\
\text{Node} \rightarrow \text{Bool}\ p &:= \text{Tag}(t) \mid \text{Class}(t) \mid \text{Id}(t) \mid \text{ItemProp}(t) \\
&\mid \underline{\text{NthChild}(k) \mid \text{NthLastChild}(k)} \\
&\mid \underline{\text{IdSub}(t) \mid \text{Style}(t, t) \mid \text{Attr}(t, t)} \\
&\mid \text{Conjunction}(p, p) \\
\textbf{input}\ \text{DomTree}\ inp &\qquad \text{string}\ t \qquad \text{int}\ k
\end{aligned}
$$

Figure 5: The DSL $\mathcal{L}$ for HTML node selection

trade-off in the DSL design in any program synthesis approach is between expressivity of the language and tractability of the synthesis algorithm, as too much expressivity can severely affect the performance of synthesis. Figure 5 shows the context-free grammar of the DSL $\mathcal{L}$ that we use for extracting nodes from an HTML document. It defines programs that are based on path expressions and filter predicates, and can be directly translated to common DOM query languages including both XPath and CSS selectors (we define the DSL independently of the syntax of these languages to keep the synthesis formulation generic and permissive of DSL variations). The terminal input symbol *inp* indicates the input to a program which is the DOM tree of the entire HTML document. The start symbol f of the grammar indicates the output of any complete program, which is a sequence of nodes selected from the input tree.

A complete program can either be a simple filter expression Filter(p, s) or a disjunction Disj($f, ..., f$) of any number of filter expressions (disjunction is equivalent to the union operator "|" in XPath or "," in CSS). A simple filter expression Filter(p, s) applies a filtering condition p on a selection of nodes s. The selection s can be all the nodes in the document (AllNodes) or obtained as the immediate children (ChildrenOf), any descendants (DescendantsOf), or right siblings (RightSiblingsOf) of a set of nodes obtained from a previous filter operation. The condition p used for filtering is a boolean function on nodes that is either an atomic predicate or conjunction Conjunction(p, p) of any number of atomic predicates. Atomic predicates include checks for the tag type of the node (Tag), its class (Class), ID (full match Id or substring match IdSub), item property from microdata (ItemProp), sibling index (NthChild from left or NthLastChild from right), as well as arbitrary key-value checks on styles (Style) and attributes (Attr).

In principle, our approach is independent of which particular atomic node-level predicates to include, and these

can be added/removed to adapt to different environments (e.g. some attributes such as text content may be expressible in XPath but not in CSS, and we also avoid other attributes that may cause overfitting such as href). As an example, a program to select any node of class "c1" that is the second child of any "DIV" element that occurs under the node with ID "mydata" is given as:

Filter(Conjunction(Class("c1"), NthChild(2)), ChildrenOf(Filter(Tag("DIV"), DescendantsOf(Filter(AllNodes(), Id("mydata"))))))

An inductive translation exists for any program in our DSL to the CSS or XPath languages. The above program directly translates to the CSS selector "#mydata DIV > .c1:nth-child(2)".

In Figure 5, we have distinguished two notable fragments of the DSL. We refer to as $\mathcal{L}_t$ the fragment that excludes the operators with a dotted underline, and $\mathcal{L}_b$ as the fragment that excludes operators with full underline. These fragments are notable in the way they are better suited to different synthesis strategies. $\mathcal{L}_b$ is a very limited language with fewer predicate operators, and is thus better suited to efficient bottom-up enumeration of programs, as shown in [39]. Also, given the program size limits in the bounded enumeration, the descendant operator in $\mathcal{L}_b$ allows a bottom-up search to explore more expressive logics rather than the localized neighborhoods of nodes accessible to the direct child operator. On the other hand, $\mathcal{L}_t$ uses only the direct child operator, as this is better suited to top-down deductive inference because constraints can be more tractably propagated through the node levels (in contrast to the combinatorial number of possibilities encountered for the descendent operator, which is why top-down synthesis approaches commonly avoid or strongly limit its use [21, 37, 40]). In this work our full DSL $\mathcal{L}$ supports both kinds of operators as it includes both $\mathcal{L}_t$ and $\mathcal{L}_b$. This richness in the language enables the inference of more concise programs (e.g. a single descendant expression rather than a long sequence of child steps) as well as the handling of tasks that may not otherwise be expressible in either approach. As we describe in the next section, our combined synthesis approach utilizes the benefits of the two language fragments in different ways.

3 PROGRAM SYNTHESIS ALGORITHM

In this section we describe the algorithm for synthesizing programs in the DSL $\mathcal{L}$ given a web document and examples specification provided by the user. An examples specification provides a sequence of text values from the webpage that is a prefix of some long sequence of data that the user would like to extract from the webpage. If available, the example specification may also include the precise nodes on the webpage which contain each of the text values. Given such a specification, the algorithm synthesizes a DSL program such that when this program is applied to the webpage DOM it returns a sequence of nodes that satisfy all of the given examples. If node information is missing, the algorithm performs additional inference to attempt to infer the correct nodes from the text-only specification.

```
1:  function SynthProg(d, E)
2:      P ← SynthFilterProg(d, E)
3:      if P ≠ null return P
4:      E1 ← Max({E' | E' ⊆ E ∧ SynthFilterProg(d, E') ≠ null})
5:      P1 ← SynthFilterProg(d, E1)
6:      P2 ← SynthProg(d, E\E1)
7:      return Disj(P1, P2)
1:  function SynthFilterProg(d, E)
2:      E ← EnumerateBottomUp(d)
3:      G ← TopAlignmentGroups(E)
4:      N ← TopNodeCombinations(d, E, G)
5:      for each N ∈ N until max iterations bound do
6:          Pt ← SynthTopDown(d, N, null)
7:          if Pt ≠ null then
8:              Ph ← SynthHybrid(d, N, G, Pt)
9:              if Ph ≠ null return Ph else return Pt
10:     return SynthBottomUp(d, E, G, E)
```

Figure 6: Program synthesis algorithm

For instance, Figure 3 shows a sample of an IMDB page containing 100 movies, which are formatted in arbitrary DIV elements rather than table or list tags. To extract all the movie names the user can provide the first two examples:

$$[(\text{"Snow White and the Seven Dwarfs"}, n_1), (\text{"Fantasia"}, n_2)]$$

where n_1 and n_2 can be null if examples are text-only, or they may be the nodes in the webpage that contain those text values, if such nodes can be detected using a visual point-and-click UI for instance. Given this specification, with or without node information, the algorithm generates a program represented by the CSS selector ".info > B > A" which extracts all 100 movie names from the page. Formally, for a given web document d and example specification $E = [(t_1, n_1), \ldots, (t_k, n_k)]$, the algorithm learns a DSL program $P \in \mathcal{L}$ such that $[\![P]\!](d) = [n'_1, ..., n'_k, ..., n'_s]$, where n'_i.Text = t_i and if $n_i \neq$ null then $n_i = n'_i$ for all $1 \leq i \leq k$. We write Satisfies(P, E, d) when a program P satisfies an example specification E on a document d in this way.

In summary, the algorithm implements a semi-supervised combination of top-down and bottom-up program synthesis. It uses bottom-up exploration in an unsupervised manner to infer programs aligned with the global *structure* on the webpage, independent of any user-provided examples. This global analysis is used as a bias to improve the supervised top-down synthesis in order to favour those programs whose results align with the inferred webpage structure. We first

give a general outline of the algorithm and then describe particular features in detail.

The top-level algorithm is shown in Figure 6. The main function SynthProg(d, E) returns a program in $\mathcal{L}$ that satisfies examples E on document d. This function attempts to synthesize a simple filter program that satisfies the examples (lines 2-3), and if no such program is found then it returns a minimal disjunction of simple filter expressions that satisfies all the examples, using a greedy approach to compute maximal example satisfaction sets (lines 4-7). The function SynthFilterProg(d, E) synthesizes filter expressions using our hybrid approach. It starts by performing an unsupervised analysis of the web document independently of the examples provided by the user. This is done by the bottom-up enumeration of a large number of programs and obtaining groups of highly aligned programs from this set (lines 2-3). This information is used in various ways in the remainder of the algorithm. We next infer the node combinations that match the text-based examples if node examples are not given (line 4). We then try each of the node combinations until a valid program can be found using top-down synthesis (line 6). If the top-down inference is successful on a node combination, then we perform the *hybrid synthesis* that checks if top-down inference can be improved with the bottom-up analysis and return the better program (lines 7-9). Finally, if no satisfying programs could be found in this way, then we fall back to a purely bottom-up synthesis (line 10). Before describing each of these steps in more detail, we first state some simple definitions. For nodes n, n' in a document, we say IsAnc(n, n') when n' is an ancestor of n. For a sequence of nodes N we define LCA(N) to be the node in the document that is the lowest common ancestor of all nodes in N. For node sequences N, N' we say N' is an ancestor sequence of N, stated IsAncSeq(N, N'), iff $N = [n_1, \ldots, n_k]$ and there exists a subsequence $[n'_1, \ldots, n'_k]$ of N' such that each n'_i is an ancestor of n_i.

3.1 Bottom-up synthesis

The bottom-up synthesis provides an unsupervised analysis of the webpage that is used in various parts of the algorithm, including text-node disambiguation, improving the top-down inference as well as resorting to a purely bottom-up search in the final step. The bottom-up synthesis method we use is based on [39], where an unsupervised analysis aims to automatically extract tabular information (a sequence of records with various fields) that may be represented in arbitrary formatting patterns on the page. This is done by enumerating numerous programs in the DSL up to a bounded size, and then finding a group of these programs that exhibit strong alignment patterns that indicate that each program in the group may extract a particular field of a record sequence. The program enumeration is done by the method EnumerateBottomUp(d) in Figure 6, which returns a set of states $\mathcal{E}$, where each state is a pair (P, N) of a program and the sequence of nodes it selects from the document, that is $[\![P]\!](d) = N$. We perform an efficient enumeration in the DSL fragment $\mathcal{L}_b$ by following the approach of [39] to recursively apply the rules of the grammar starting from terminal states using lifting functions and other optimizations such as semantic equivalence. After enumeration, the TopAlignmentGroups$(\mathcal{E})$ function is used to find the list $\mathcal{G}$ of the top ranked *alignment groups* of programs. An alignment group is of the form $(P_a, (P_1, \ldots, P_n))$, where for $[\![P_a]\!](d) = N_a$ and $[\![P_i]\!](d) = N_i$ we have N_a and all N_i are minimal sequences of nodes in the sense that no node in the sequence is an ancestor of any other node in the sequence, and for each N_i, we have $|N_i| = |N_a|$ and IsAncSeq(N_i, N_a). We refer to P_a as the common ancestor program for the alignment group, and the other programs as field programs. We compute alignment groups by performing a pairwise quadratic-time comparison of the enumerated states $\mathcal{E}$ with each other to check interleaving, and then ranking states by the highest number of interleavings to find the largest alignment groups. This notion of alignment is similar to [39], except we do the additional step of finding an ancestor state from $\mathcal{E}$ for each alignment group. Considering the IMDB example from Figures 3 and 4, the enumeration and alignment analysis yields a highly ranked alignment group with ancestor program ".list_item" that selects the 100 DIV elements that contain all the information about a movie. The field programs extract various properties such as the movie year (".year_type"), the running time (".item_desc SPAN"), etc. However, not all fields are captured in the restricted DSL, e.g. the movie title requires a selector ".info > B > A" which lies outside the bottom-up DSL $\mathcal{L}_b$. In section 3.4 we show how detecting this alignment group helps to infer the title using our hybrid synthesis approach.

Supervised bottom-up. Although the bottom-up synthesis is mainly used for an unsupervised analysis of the webpage, in the final step of the main algorithm (Figure 6) we resort to purely bottom-up search for a program that satisfies the text-based examples if no such program is found in the top-down DSL. The function SynthBottomUp$(d, E, \mathcal{G}, \mathcal{E})$ searches for such a program first within the top-ranked alignment groups *alignGroups*, and then all remaining enumerated states $\mathcal{E}$.

3.2 Text-node disambiguation

In this section we describe our method for determining which nodes in the webpage correspond to the text-based examples, in the case where node information is not provided in the

```
1:  function TopNodeCombinations(d, E, 𝒢)
2:      let E = [(t_1, n_1), ..., (t_k, n_k)]
3:      if n_i ≠ null for all i = 1 ... k then
4:          N ← [n_1, ..., n_k]
5:          return [N]
6:      let T = [t_1, ..., t_k]
7:      let [S_1, ..., S_k] such that S_i = {n ∈ d | n.Text = t_i}
8:      𝒩 ← S_1 × ... × S_k
9:      return 𝒩 where N ∈ 𝒩 are ordered lexically by
10:         BottomUpAlignment(N, 𝒢),
11:         UniformTags(N, d),
12:         ExtremalNodes(N, d),
13:         UniqueCommonAncestor(N, d),
14:         UniformTagClassHierarchy(N, d),
15:         NodeDistanceDeviation(N, d)
```

Figure 7: Node combinations from text specification

```
1:  function SynthTopDown(d, N_e, N_a)
2:      N_p ← {n ∈ d | n is parent of some n' ∈ N_e}
3:      P_p ← SynthTopDown(d, N_p, N_a)
4:      𝒫_s = { ChildrenOf(P_p), AllNodes() }
5:      𝒫 ← ∅
6:      for each P_s ∈ 𝒫_s do
7:          N_s ← [[P_s]](d)
8:          P_c ← SynthPredicate(d, N_s, N_e, N_a)
9:          P = Filter(P_c, P_s)
10:         if N_a = null ∨ SatisfiesAncSeq([[P]](d), N_e, N_a) then
11:             𝒫 ← 𝒫 ∪ {P}
12:     return ArgMin_{P∈𝒫} (|[[P]](d)|, Size(p))

1:  function SatisfiesAncSeq(N, N_e, N_a)
2:      if N_a is an ancestor sequence of N_e then
3:          return ∀n ∈ N.∃n' ∈ N_a. IsAnc(n, n')
4:      else
5:          return ∀n ∈ N_a.∃n' ∈ N. IsAnc(n, n')
```

Figure 8: Top-down synthesis

examples specification. As there can be many valid node combinations matching a given set of text values, we need to rank the combinations and can only consider a top few: in practice a bound of at most 4 iterations of the loop at line 5 in Figure 6 kept within acceptable performance limits. Figure 7 shows the function for ranking node combinations which returns a ranked list $\mathcal{N}$, where each $N \in \mathcal{N}$ is a sequence of nodes such that $|N| = |T|$ and $N[k]$.Text $= T[k]$ for all k. If the node information is already provided in the examples specification, then we simply return it. Otherwise, we infer combinations matching the text examples by performing a cartesian product over the sets of all matching nodes for each text value, and then ranking them using a number of features based on both the bottom-up analysis as well as structural properties of the document. The first feature BottomUpAlignment prefers node combinations that are consistent with any of the alignment groups created by the bottom-up analysis. Formally, we have BottomUpAlignment$(N, \mathcal{G}, d)$ if and only if there exists some $(P_a, (P_1, \ldots, P_n)) \in \mathcal{G}$ such that IsAncSeq$(N, [\![P_a]\!](d))$. This is a powerful feature that utilises consistency with respect to the global unsupervised page analysis. The remaining features are based on uniformity of node attributes and structural properties of the nodes in the document. These include UniformTags (all nodes in the combination have the same tag), ExtremalNodes (nodes are either all maximal or all minimal in ancestor hierarchy), UniqueCommonAncestor (any common ancestor of 2 or more nodes is a common ancestor of all nodes), UniformTagClassHierarchy (all nodes have same pattern of tags and class names in their ancestor hierarchy), and NodeDistanceDeviation (the nodes occur at uniform distances between each other, in terms of their document order position). In our evaluations we found a simple lexical preference over these features to produce effective results, though it may also be interesting to explore statistical techniques with weighted combinations of features.

As an example, one of the pages in our test scenarios was a Kayak flight search results page, where the airline name "United" appears 44 times in different regions of the page: in many onward/return journey legs, under the price in the second result, and even *before* the main search results in the left margin of the page. Despite such ambiguity, for the task of extracting the onward flight name for each flight result, from just the first 2 examples ["United", "United"] (both happened to be the same airline in this case), using the above features our method matches the correct nodes required to infer the program for extracting airline names.

3.3 Top-down & predicate inference

Given the ranked list of candidate node combinations, the synthesis algorithm attempts to infer a program from a node-based examples specification (line 6 of SynthFilterProg function in Figure 6). This is done using a top-down program synthesis technique which can explore the richer fragment $\mathcal{L}_t$ of the full DSL that includes the more expressive operators required for most practical tasks. The SynthTopDown(d, N_e, N_a) function in Figure 8 returns a program P in $\mathcal{L}_t$ such that $[\![P]\!](d) = N'$ where N_e is a prefix subsequence of N'. The parameter N_a to the function imposes an optional *ancestor constraint* on N' that $\forall n \in N_a.\exists n' \in N'$.IsAnc$(n', n)$. This parameter is optional (ignored by passing N_a = null), and we make use of it in the hybrid synthesis described in the next section. Following standard top-down approaches [21, 37, 40], the function follows the DSL structure assuming

```
1:  function SynthPredicate(d, N_s, N_e, N_a)
2:      P_atm ← InferAtomicPredicates(N_e)
3:      N_n ← {n ∈ N_s \ N_e | ¬IsAnc(n, LCA(N_e)) ∨ ∃n' ∈ N_e.n < n'}
4:      P_r ← ∅
5:      while |P_atm| > 0 do
6:          P_min ← {P ∈ P_atm | has minimal |[[P]](N_n)|}
7:          for each P_m ∈ P_min do
8:              P_c ← P_m
9:              while |[[P_c]](N_n)| > 0 do
10:                 N_c ← [[P_c]](N_n)
11:                 P ← {P ∈ P_atm | P ∉ P_c}
12:                 if N_a ≠ null then
13:                     for each P ∈ P do
14:                         N ← [[Conjunction(P_c, P)]](d)
15:                         if ¬∀n ∈ N_a.∃n' ∈ N.IsAnc(n,n') then
16:                             P ← P − {P}
17:                 P_p ← ArgMin_{P∈P} |[[P]](N_c)|
18:                 P_c ← Conjunction(P_c, P_p)
19:             P_r ← P_r ∪ {P_c}
20:         P_atm ← P_atm − {P | ∀P' ∈ P_r. P ∉ P'}
21:     if ∃P ∈ P_r. |[[P]](N_n)| = 0 then
22:         P_r ← {P ∈ P_r | |[[P]](N_n)| = 0}
23:     return ArgMin_{P∈P_r} Size(p)
```

Figure 9: Predicate inference in top-down synthesis

```
1:  function SynthHybrid(d, N, G, P_t)
2:      N_t ← [[P_t]](d)
3:      N_a ← ∅
4:      for each (P_a, (P_1, ..., P_k)) ∈ G do
5:          N_a ← [[P_a]](d)
6:          if IsAncSeq(N_t, N_a) ∧ LCA(N_t) = LCA(N_a) then
7:              for i = 1...k do
8:                  N_i ← [[P_i]](d)
9:                  if ∀j = 1...|N|. N_i[j] = N[j] return P_i
10:             N_a ← N_a ∪ {N_a}
11:     for each N_a ∈ N_a do
12:         P_h ← SynthTopDown(d, N, N_a)
13:         if P_h ≠ null return P_h
14:     return null
```

Figure 10: Hybrid program synthesis

the final program will be of the form $\mathsf{Filter}(P_c, P_s)$ and proceeds by inferring constraints for each of the parameters for the predicate condition and the set. For the set parameter P_s, the two options provided by the DSL are either AllNodes or a ChildrenOf expression synthesized by a recursive call on the parent nodes of the example nodes (lines 2-3). Corresponding predicate conditions for each of these set expressions are synthesized by calling the predicate synthesis (lines 7-8), and the generated filter program is added it to the set of possible programs if it satisfies the ancestor constraint. The final ranking criteria is to return the program that selects fewer nodes and is smaller in size (line 12).

Predicate inference. Unlike previous techniques, we use a novel predicate inference method that alleviates overfitting and improves learning from fewer examples. The function $\mathsf{SynthPredicate}(d, N_s, N_e, N_a)$ in Figure 9 infers a predicate P that is true for all the example nodes N_e from a set of nodes N_s. The first step at line 2 is to synthesize all the atomic predicates satisfied by all examples, which is done by a simple analysis of all common attributes of nodes in N_e. At this point, the key question is what combination of these predicates to choose, as any conjunction of a subset of these atomic predicates is a valid predicate that generalizes over the examples but selects different nodes from N_s. One approach that is often taken is to infer the *least general generalization* which is simply to return the conjunction of all the atomic predicates. Apart from creating syntactically complex programs, these largest conjunctions easily cause overfitting and require many examples to infer correct programs. The approach we use here is to formulate the predicate inference as a minimal set cover problem in which we aim to find the smallest set of predicates that satisfy as many *negative* node examples as possible. The implicit negative examples we choose to avoid is a heuristic choice: all the nodes that lie outside the LCA of the example nodes (we assume generalization to within the LCA) and those that occur before any of the example nodes (since the example nodes are a prefix of the desired sequence). Note that this heuristic imposes only a preference bias to avoid as many of the negative example nodes as possible, and is as such a *soft constraint*. Having computed the soft negative examples set N_n at line 3, the remainder of the algorithm implements a greedy approximation algorithm to the set cover problem [8], as this is an NP-complete problem. This produces smallest conjunctions by incrementally adding atomic predicates that cover the largest number of negative examples that have so far not been covered by the conjunction. The one additional check we make is to exclude any predicates that do not satisfy the ancestor constraint N_a if one is provided (lines 12 to 16).

3.4 Hybrid synthesis

Top-down synthesis is a purely supervised approach that aims to synthesize a program in a rich DSL guided only by the few examples provided by the user. It is hence prone to overfitting. On the other hand, the bottom-up approach provides a global unsupervised analysis of the webpage that yields maximal alignment patterns on the page, but it is limited by the very restricted DSL. In this section we describe

our *hybrid synthesis* approach which improves the top down inference by combining it with the bottom-up analysis.

In the main synthesis algorithm SynthFilterProg in Figure 6, if the top-down synthesis succeeds on a particular node combination, then we attempt to improve this program using the hybrid approach (line 8). The SynthHybrid$(d, N, \mathcal{G}, P_t)$ function returns a program that satisfies the node specification N. The function synthesizes this program using the top-ranked alignment groups $\mathcal{G}$ created by the bottom-up synthesis and the top-ranked program P_t created by the top-down synthesis. The key idea underlying the hybrid approach is that whenever the top-down program P_t produces results that are consistent with any of the alignment groups in $\mathcal{G}$, then we try to ensure the alignment is *completely* satisfied: that P_t is not missing some of the records of the alignment group by overfitting to the examples. Hence, in the main loop at line 4, for each alignment group we check if the ancestor program of the group is also an ancestor program for the top-down result and they share a common LCA. If so, then the first preference is to simply prefer any program in the alignment group if it directly satisfies the examples, since this program satisfies the full alignment group and the examples and is also within the simpler DSL $\mathcal{L}_b$ (lines 7-9). But such a program may not exist in the alignment group because of the restricted bottom-up language. In this case we collect the ancestor programs for all the satisfying groups in $\mathcal{N}_a$ (line 11). We then try each of these ancestor programs $N_a \in \mathcal{N}_a$ as an ancestor constraint in a top-down synthesis in order to find a program that can generalize to all of the records of the alignment group (lines 12 to 14).

We illustrate the hybrid approach with the example movies page in Figure 3 and the corresponding simplified HTML source in Figure 4. In section 3.1 we described how the bottom-up analysis detects the correct alignment group with all 100 result records, and some fields such as movie runtimes, years, etc. However, the movie title field was not detected in this limited DSL. To extract the movie titles, if we provide the first 2 examples to the purely top-down algorithm we get the program "B:nth-last-child(6) > A". This extracts 99 of the 100 titles: all except the 12th one, which is different because it is missing one of the fields (the director) as observed by the missing DIV element in Figure 4. Hence the nth-last-child logic fails in this case. However, considering our hybrid synthesis approach, this program is consistent with the correct alignment group with 100 records. Since none of the field programs in the group directly satisfy the examples, we reperform the top-down synthesis using the group ancestor program as the ancestor constraint. This forces a generalization to all records in the group and gives the improved program ".info > B > A" which is expressible in the top-down DSL and correctly extracts all 100 movie titles.

4 EVALUATION

In this section we describe an evaluation of our method with respect to different aspects of quality. We first demonstrate improvement in overall accuracy (precision/recall) of our system in comparison to the current state-of-the-art approaches, using labelled training sets of documents and measuring accuracy on different documents in the test set. This is one indication of the inference power of a system, but it does not address the important usability aspect which is the number of examples per document the user would need to give in practice, as it assumes the user must provide *all* of the desired nodes from a document when training the system. We demonstrate how our system requires the fewest number of examples for task completion per document as compared to other similar systems. We also evaluate two other important usability aspects of our system: the low complexity of the programs it synthesizes and the ability to learn from ambiguous text-based examples.

4.1 Overall accuracy across documents

We compared the overall accuracy of an implementation of our hybrid synthesis approach (HYB) with existing approaches for program synthesis as well model-based approaches. These include the recent work [35] on synthesis of *forgiving data extractors* (FX), their corresponding non-forgiving synthesis method (NFX), the C4.5 classifier of [38] (C4.5), a naive bayes classifier [23] (NB), XPath alignment-based synthesis method [34] (XA), and synthesis using least general generalizations [27, 37] (LGG). The particular implementations we used for FX, NFX, C4.5, NB, and XA were from [35] (some using Weka [17]), and LGG is from [37].[1].

Datasets. We evaluated the systems using three datasets which contain extraction tasks over a broad range of verticals, websites and attributes (all our datasets are available from: https://app.box.com/s/vi4c976afptq39524y1pofz7fw995qf9). We used the **DS1** dataset from [35] which contains 166 manually annotated pages from 30 websites ranging over 4 verticals (books, shopping, hotels and movies), with 2-3 attributes per vertical (e.g. title, author or price from book pages, or the title, genre or list of actors from movie pages). For each webpage, ground truth is given as nodes that are annotated in the page (marked with *"userselected"* attribute in the HTML). However, the labels in the DS1 dataset include significant redundancy in terms of multiple labelled nodes representing the same attribute value: e.g. if the title of the book occurs in multiple nodes in different regions of a book webpage, then all of these nodes are marked as the ground truth. Such redundant extraction is often not the case in practice: one would expect to extract a single data item representing the title from a book page, or a list of actors from a movie page without

[1] many thanks to the authors for providing these

duplicate occurrences of the same actor. For this reason, we created the dataset **DS1-b**, which uses the same webpages from DS1 but where corresponding extraction tasks do not contain such redundancy. To consider a wider range of verticals and websites, we used another bigger dataset **SWDE** which consists of 626 annotated webpages from 80 websites ranging over 8 verticals (auto, book, camera, job, movie, NBA player, restaurant, university), with 3-5 attributes per vertical and 10 websites for each vertical. We obtained SWDE as a subset of the structured web data extraction dataset from [18] which contains many more webpage instances but where the ground truth is only given as text values rather than node locations in the webpage. Hence we needed to manually annotate the ground truth nodes as there were multiple matching node occurrences in many cases. Given the manual effort involved in such annotation, we annotated the first two pages for each (vertical, site, attribute) combination in the original dataset where the attribute values were non-null. Hence SWDE maintains the same variety of (vertical, site, attribute) combinations as the original dataset (only smaller in terms of the number of webpage instances per combination), and is therefore useful for a comparative evaluation of the systems over different domains.

Experiments. We performed experiments on each system using the three datasets DS1, DS1-b and SWDE. In each case, we trained the system on the webpages and the desired ground truth nodes from the training set, and then tested the accuracy of the synthesized extractor (or classifier) on the webpages and ground truth from the test set. In our experiments we used standard precision, recall and F1 calculations based only on the set of ground truth nodes (e.g. we do not consider children or other nodes as permissible in precision calculations as in [35]). The results of our experiments are shown in Figure 11, where we show for each system the precision, recall and F-measure averaged across all tasks, and for each metric we also include the 95% confidence interval (CI). The main result is that over all datasets combined (top table), our system HYB was the best performer with the highest average F1 score of 0.86 and this is a statistically significant improvement over all other systems at the 95% confidence level (there is no overlap between the CIs). The same is also true for the precision of our system which was also highest with statistical significance (average of 0.86 with no overlap in CIs from other systems). For recall, FX and NB were significantly higher but they were the lowest ranking systems overall, while NFX had slightly higher recall than HYB but with overlapping CIs.

On dataset DS1, the best performing systems were HYB, FX and NFX, but with large overlap in the CIs. Forgiving selectors are highly disjunctive in nature, which may explain the better performance of FX on the dataset that has more redundancy in the outputs. On dataset DS1-b which has less redundancy, HYB and NFX were the best performers while FX degraded severely in precision and its recall hit maximum. On the large dataset SWDE, the best performers were HYB and XA with statistical significance, while NFX degraded considerably mainly due to precision. As SWDE mostly contains tasks with very few extractions per page, most baselines suffered on precision here likely due to overgeneralization, while XA performed well on this dataset but worse on others likely due to insufficient generalization. Though never among the top performers, we note that LGG was the third-best performing system overall after HYB and NFX, and showed very stable metrics with averages of 0.73-0.75 in all of the datasets. Overall, HYB was the only system that was among the best performers in each of the datasets individually, and it was the single best performer overall with statistical significance.

		Precision	Recall	F1
All datasets	HYB	0.86 ± 0.03	0.87 ± 0.03	0.86 ± 0.03
	FX	0.21 ± 0.03	0.97 ± 0.01	0.25 ± 0.03
	NFX	0.77 ± 0.03	0.89 ± 0.03	0.78 ± 0.03
	C4.5	0.61 ± 0.04	0.86 ± 0.03	0.65 ± 0.04
	NB	0.32 ± 0.03	0.97 ± 0.01	0.38 ± 0.03
	XA	0.73 ± 0.04	0.76 ± 0.04	0.73 ± 0.04
	LGG	0.74 ± 0.04	0.75 ± 0.04	0.74 ± 0.04
DS1	HYB	0.85 ± 0.07	0.88 ± 0.07	0.85 ± 0.07
	FX	0.84 ± 0.07	0.93 ± 0.05	0.85 ± 0.06
	NFX	0.83 ± 0.07	0.88 ± 0.07	0.84 ± 0.07
	C4.5	0.74 ± 0.08	0.85 ± 0.07	0.76 ± 0.07
	NB	0.40 ± 0.08	0.92 ± 0.06	0.47 ± 0.08
	XA	0.49 ± 0.11	0.51 ± 0.11	0.48 ± 0.10
	LGG	0.73 ± 0.09	0.75 ± 0.09	0.73 ± 0.09
DS1-b	HYB	0.92 ± 0.05	0.95 ± 0.05	0.93 ± 0.05
	FX	0.11 ± 0.03	1.00 ± 0.00	0.18 ± 0.04
	NFX	0.92 ± 0.05	0.95 ± 0.05	0.93 ± 0.05
	C4.5	0.75 ± 0.08	0.87 ± 0.07	0.78 ± 0.08
	NB	0.43 ± 0.08	0.95 ± 0.04	0.49 ± 0.08
	XA	0.56 ± 0.11	0.57 ± 0.11	0.56 ± 0.11
	LGG	0.73 ± 0.09	0.74 ± 0.09	0.73 ± 0.09
SWDE	HYB	0.85 ± 0.04	0.85 ± 0.04	0.84 ± 0.04
	FX	0.07 ± 0.01	0.97 ± 0.02	0.12 ± 0.01
	NFX	0.71 ± 0.05	0.88 ± 0.03	0.73 ± 0.04
	C4.5	0.54 ± 0.05	0.87 ± 0.04	0.58 ± 0.05
	NB	0.27 ± 0.04	0.98 ± 0.01	0.34 ± 0.04
	XA	0.84 ± 0.04	0.88 ± 0.03	0.84 ± 0.04
	LGG	0.75 ± 0.05	0.75 ± 0.05	0.74 ± 0.05

Figure 11: Precision, recall & F1 with 95% C.I.

4.2 Number of examples per document

Measuring the overall accuracy across documents using training/test sets assumes the user must provide *all* of the desired nodes from each document when training the system. This may not be difficult for extraction from *details pages* where a single or small number of nodes are desired from a document (e.g. extracting the title or authors from a webpage about a particular book), but it can be a significant challenge for *list pages* where a single document may contain hundreds of nodes to extract (e.g. getting titles or prices of all books listed on a search result page). The number of examples required from the user is an important usability aspect as giving numerous examples can be error-prone and difficult. Inferring correct inferences from fewer examples indicates the robustness of the system and lessens the risk and burden of verification for the user. This issue can be addressed by supporting the ability to learn from *partial examples specifications*: the user need only give a small subset of the desired selection of nodes in a document, e.g. providing the titles of the first two or three books from a search result page and the system can infer the correct program to extract all titles. In this section we provide an evaluation of our approach compared with other systems supporting partial examples specifications. Given a document from which there are N nodes to extract, the goal of our evaluation is to find out how many examples are required to give to the system before it can learn a program that extracts all N nodes from the document. Since such evaluation is not appropriate for tasks where N itself is very small (showing inference from few examples would be trivial), we only consider extraction tasks where there are 5 or more nodes to extract from a document.

Datasets. The first dataset we use is EX1, which consists of all the tasks from the datasets DS1, DS1-b and SWDE where there are 5 or more nodes to extract from the webpage. This gave a total of 93 tasks (48 from DS1, 43 from DS1-b and 2 from SWDE). As most of the tasks in these datasets were for attribute extraction from *details pages*, the number of tasks in EX1 is small and the average number of nodes to extract per task is 12.8. We therefore used a bigger dataset EX2 that targeted list extraction tasks. This consists of 225 tasks from 66 webpages, with an average of 56.9 nodes to extract per task. These scenarios were mostly provided by the Power BI product team as their representative customer scenarios, as well as real use cases we collected from online help forums or videos where users describe their real world web scraping tasks (we have included with our dataset the links to the original online sources for the webpages used in these scenarios). Unlike EX1, webpages in EX2 generally contain large amounts of tabular information in arbitrary formats, from sites in different domains, e.g. Amazon, eBay, Kayak, Craigslist, Ikea, IMDB and other lesser known sites.

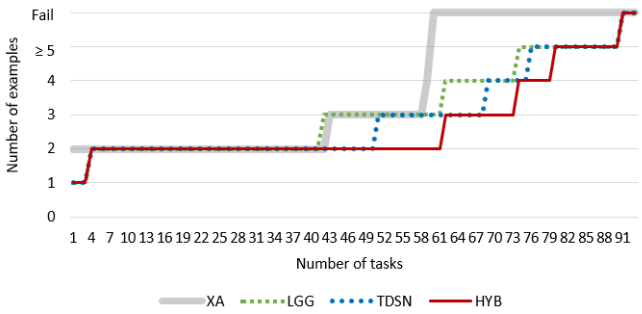


Figure 12: Number of examples for tasks in EX1

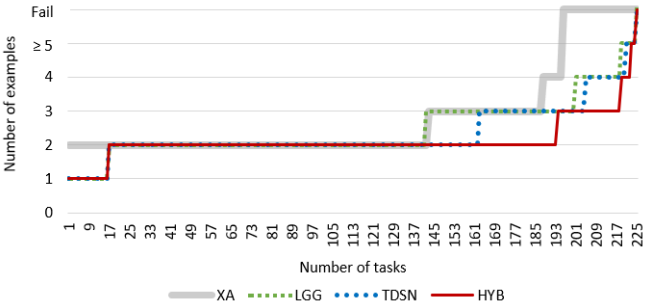


Figure 13: Number of examples for tasks in EX2

Experiments. Using datasets EX1 and EX2, we compared our system with the other baseline systems supporting partial examples specifications. FX, NFX, C4.5 and NB do not support partial examples by design, because they implicitly assign all of the nodes in a document that are not annotated by the user as negative examples. For empirical verification, when we applied NFX (the best performing baseline from the last section) to tasks in EX1, only 6 out of the 93 tasks could be completed with partial examples. All other tasks required all nodes from the page to learn the correct extraction, or else they were over-fitting to the partial examples (we speculate that the 6 cases succeeded with partial examples probably because a selector could not be found that generalized only to the given examples). Since it is not sensible to compare against these systems, in our experiments we considered the two baselines LGG and XA that do support partial examples. We also compared with another baseline system TDSN, which is our top-down system using the greedy set cover heuristic with soft negative examples but not using the hybrid approach. For each task, given a document and sequence of nodes to extract, we provided examples to the system incrementally in prefix order over the target node sequence, until the complete sequence could be extracted by the system. Figures 12 and 13 show the number of examples that were required for task completion in EX1 and EX2. The top line

HYB	.b_algo CITE
TDSN	.b_algo > .b_caption > .b_attribution > CITE
LGG	LI.b_algo > DIV.b_caption:nth-child(2):nth-last-child(1) > DIV.b_attribution:nth-child(1) > CITE:nth-child(1)
HYB	.a-size-small + .a-size-small
TDSN	.a-col-right > .a-spacing-small > .a-row > :nth-child(2)
LGG	DIV.a-fxd-lft-grd-col.a-col-rht[style*="float left"][style*="padding-left 2%"]:nth-child(2) :nth-last-child(1) > DIV.a-row.a-spacing-small:nth-last-child(2) > DIV.a-row.a-spacing-none:nth-last-child(1) > SPAN.a-size-small.a-color-second:nth-child(2):nth-last-child(1)
HYB	.above-button .price
TDSN	.Theme-featured > A.booking-link[id*="-booking-link"][tabindex="0"] [role="option"][target="_blank"]:nth-child(1):nth-last-child(1) > .price
LGG	DIV.Common-Booking-MultiBookProvider.featured-provider.Theme-featured .multi-row[id*="-price-mb-aE"][aria-hidden="false"]:nth-child(1):nth-last-child(1) > A.booking-link[id*="-booking-link"][tabindex="0"][role="option"][target="_blank"] :nth-child(1):nth-last-child(1) > SPAN.price.option-text:nth-child(1):nth-last-child(2)

Figure 14: CSS selectors created for 3 sample tasks

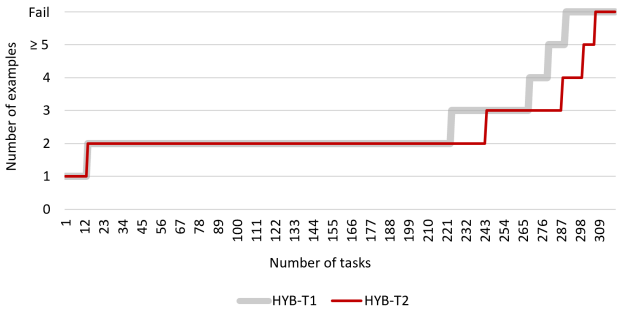


Figure 15: Number of text examples for EX1 & EX2

indicates the number of cases where the correct node selection could not be inferred even with all examples. The main result is that for both datasets, our system HYB completed the highest number of tasks with 2 examples or less. The relative performance of all the systems followed a similar pattern in both datasets: the percentage of tasks completed with under 2 examples in EX1 was XA (45.2%), LGG (44.1%), TDSN (53.8%), HYB (65.6%), while for EX2 it was XA (63.1%), LGG (62.7%), TDSN (72.0%), HYB (85.8%). We observe that in general a higher number of examples was required by all systems for tasks in EX1. This is explained by the high redundancy in tasks in DS1, as duplicate occurrences of text values in different regions of the webpage require more examples (all tasks requiring more than 5 examples or failing for HYB in EX1 were from DS1). Overall, XA and LGG performed similarly but XA had significantly more failing cases. The improvement in TDSN over LGG or XA (~9%) indicates the gain in robustness obtained by using the greedy set cover heuristic (for predicate inference using soft-negative examples) over the least general generalization or path alignment approaches. The more significant improvement in HYB over TDSN (~13%) illustrates the greater benefits obtained with our hybrid approach that utilizes the bottom-up analysis. To compare against the purely unsupervised bottom-up approach that works without examples: the top table from such a system [39] failed on 68.8% of tasks in EX1 and 31.6% for EX2.

4.3 Other usability aspects

Program complexity. The complexity of the programs that are synthesized is another important usability aspect, as it affects how easily the user can understand or manipulate the synthesized program if required. This can be difficult if synthesized programs have a large number of expressions, and thus we take the program size as a quantitative indicator of complexity. We examined the number of operators used in the CSS selectors synthesized by our system HYB, and compared with the other systems that also synthesize CSS selectors (LGG and TDSN). We found that across all datasets, 13.4% of programs synthesized by LGG had three operators or fewer, while this increased to 68.4% for programs synthesized by HYB. The average program sizes for the three approaches were: LGG (7.9), TDSN (3.8), HYB (3.6). This shows the reduction in complexity gained by the inference of minimal predicate conjunctions, as well as the use of more expressive operators such as descendants and siblings in the bottom-up DSL $\mathcal{L}_b$ used by our hybrid synthesis approach. From experience, CSS selectors with about 3 to 4 operators is around the number one would expect to write by hand for most extraction tasks, making the complexity of programs synthesized by our system comparable to human-written programs. For some qualitative illustration, we show examples of the CSS selectors synthesized by the different systems for sample tasks in Figure 14. This shows the drastic reduction in complexity in selectors produced by HYB (using descendant and sibling operators) as compared to the other baselines.

Since other previous approaches have focussed on XPath rather than CSS synthesis, we cannot directly compare number of operators as different operators are used in the two languages. But for approximate comparison we give the string size (number of characters) of synthesized expressions. NFX and FX were running out of memory on EX2, but on EX1 the average string sizes were HYB: 129, TDSN: 155, LGG: 345, XA: 272, NFX: 227, FX: 1010. Particularly long and verbose expressions were created by FX (containing many disjuncts) and XA (containing very long paths).

Program complexity is an interesting area which we have begun to explore here using methods such as minimal set cover, ranking by size and using expressive DSL operators. Though the problem is more general than program equivalence (a simpler unequivalent program may be preferable if it satisfies the examples), reduction based on full equivalence may yield further benefits.

Text-based examples. Another novel aspect of our approach is the ability to synthesize programs from text-based

examples when node-based examples cannot be given. As previous approaches have not addressed this problem, we present a comparison with a naive baseline in which we use our system but simply accept the first nodes in the document that match the text examples in document order (HYB-T1). We compare this against our complete text-node disambiguation system (HYB-T2). Figure 15 shows the results of providing these two systems with the text-only examples for all the tasks in datasets EX1 and EX2. We observe the overall improvement in the number of examples gained with our text-disambiguation approach, which succeeded with at most 2 examples in 76.4% of tasks, as compared to 70.1% for the naive baseline. The baseline also failed altogether in 9.1% of tasks as compared to 3.8% for our approach.

Deployment. Our approach has been deployed as a feature in the Microsoft Power BI product used by business intelligence analysts. The Power BI team integrated our approach in their spreadsheet-based UI, in which the user can type in examples and the system infers a CSS selector to complete the task and auto-fills the table. The inferred program becomes part of the analysis script which the user can edit manually or re-execute in the future to refresh the data. The feature is under active development, with a version of the TDSN system currently released for general audience, and development is ongoing to incorporate our complete system. It has been well-received by users as seen in online forums and videos (see the online sources for some of the tasks in our EX2 dataset). One example sentiment: *"I got so excited about this the day of release! I've managed to get so many obscure things working!"*

5 RELATED WORK

Supervised web extraction. Supervised approaches to web data extraction have mainly centered around *wrapper induction* [25], where the goal is to learn extraction rules from HTML pages given sample annotations. Early work in this area mainly focused on string or token-based approaches [19, 25, 31], where the document is viewed as a sequence of characters or tokens, and extraction is based around delimiter patterns. This is in contrast to HTML-aware systems, which exploit the tree-structure of HTML explicitly. This began with some interactive programming approaches where the user provided various structural constraints [6, 32, 42], and since then there has been greater focus on learning wrappers from examples in widely used standard HTML query languages such as XPath or CSS [3, 13, 33–35, 37, 46], which has also been our focus in this work. Approaches based on XPath alignment [33, 34] work by aligning the steps within the XPaths of sample nodes based on edit distances, and merging them to a single generalized XPath. In [46], a record-level wrapper system is proposed which generates complete tag-paths from the root node. Approaches based on least general generalizations [37] produce largest conjunctions of all the properties that example nodes have in common. Such approaches based on alignment or largest conjunctions lead to long path expressions or numerous predicates, which are complex to understand and over-fit on the examples, as shown in our evaluation. Some approaches such as *forgiving XPaths* (FX) [35] attempt to improve the recall and learn cross-site selectors by using multiple disjuncts in the generated selectors, but we show in our evaluation how this leads to severe loss in precision. Machine learning techniques have also been explored such as naive-bayes classifiers [13] and decision trees (NFX system [35]), and we have also shown improvement in robustness over such approaches with our hybrid synthesis method. Other related work has gone beyond the use of standard HTML languages and explored arbitrarily more sophisticated extraction models, such as using visual or semantic features or specialized handling for particular vertical domains [7, 11, 18, 24, 36]. Though beneficial in many scenarios, such approaches are not designed to generate simple selector expressions that users can understand, manipulate, and execute in standard HTML tools. To make an analogy with text extraction, users that require simple regular expressions for common everyday text manipulation tasks will not be served well by complex extraction techniques or deep semantic analyses. Thus in this respect, our problem definition is more specialized than arbitrary wrapper induction or information extraction, as it includes the requirement of inferring concise, readable programs in standard lightweight languages.

Unsupervised, KB-based & distant supervision. Fully automated web extraction approaches attempt to mine recurring patterns in the DOM structure of web pages without any annotated examples [4, 9, 39, 45], and some include generation of wrappers in standard languages, e.g. [39]. These approaches are generally good at finding prominent patterns, but cannot guarantee that all kinds of information desired by different users will be extracted, for which supervised approaches are better-suited. However, in our semi-supervised hybrid approach we have shown how to incorporate such unsupervised analysis which can be very beneficial to quickly converge to the desired extraction. Another interesting area of research has been to leverage existing knowledge bases (KB) and distant supervision to align attribute values in KB to text values on webpages for learning wrappers [12, 14, 29]. Such approaches work well for cases where relevant KBs may exist, but this is not the case in general such as for dynamic web pages, private webpages or other parts of the deep web. However, for scenarios where KBs are applicable, these techniques may be used in place of purely unsupervised methods to further improve our hybrid synthesis approach.

Program synthesis from examples. Programming by example (PBE) has seen significant interest and progress in recent years [2, 5, 10, 15, 16, 26, 28, 30, 37, 39, 44], with notable commercial successes such as the *Flash Fill* feature in Microsoft Excel for automating string transformations [15]. Given a domain-specific language (DSL) and an examples specification, PBE systems aim to find a program in the DSL that satisfies the examples. Approaches can be broadly classed as either bottom-up approaches that enumerate programs following the syntax of the DSL [2, 39], or top-down approaches [16, 37] where examples constraints are recursively propagated through the DSL from the given specification. In this work we have presented the first hybrid approach which combines the benefits of the two approaches into a semi-supervised synthesis system. Although this work has focused specifically in the web domain, the fundamental concepts can be more generally applicable to different document domains if we consider other selection DSLs, e.g. regex-based selectors for plain text documents, or spatial/position based selections for PDF documents, which will be interesting explorations for future work.

6 CONCLUSION

We have described a novel hybrid program synthesis approach for inference of web data extraction programs from examples, which provides inference of concise programs expressible in common languages from a small number of examples, as well as the ability to learn from text-only examples. We have addressed these challenges with three novel ideas: a semi-supervised program synthesis approach effectively combining top-down and bottom-up synthesis methodologies, inferring concise predicates based on soft negative examples and resolving text-node disambiguation based on structural document features. Our evaluation illustrates the effectiveness of our approach in dealing with the usability challenges on real-world datasets, and meets the high bar for shipping in the mass-market Power BI product.

REFERENCES

[1] Ziawasch Abedjan, John Morcos, Michael N. Gubanov, Ihab F. Ilyas, Michael Stonebraker, Paolo Papotti, and Mourad Ouzzani. 2015. Dataxformer: Leveraging the Web for Semantic Transformations.. In *CIDR*. www.cidrdb.org. http://dblp.uni-trier.de/db/conf/cidr/cidr2015.html#AbedjanMGISPO15

[2] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*, Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner (Eds.). NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 1–25. http://dblp.uni-trier.de/db/series/natosec/natosec40.html#AlurBDF0JKMMRSSSSTU15

[3] Tobias Anton. 2005. XPath-Wrapper Induction by generating tree traversal patterns.. In *LWA* (2005-11-14), Mathias Bauer, Boris Brandherm, Johannes Fürnkranz, Gunter Grieser, Andreas Hotho, Andreas Jedlitschka, and Alexander Kröner (Eds.). DFKI, 126–133. http://dblp.uni-trier.de/db/conf/lwa/lwa2005.html#Anton05

[4] Arvind Arasu and Hector Garcia-Molina. 2003. Extracting Structured Data from Web Pages. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, Alon Y. Halevy, Zachary G. Ives, and AnHai Doan (Eds.). ACM, 337–348.

[5] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *ICLR*.

[6] Robert Baumgartner, Oliver Frölich, and Georg Gottlob. 2007. The Lixto Systems Applications in Business Intelligence and Semantic Web.. In *ESWC (Lecture Notes in Computer Science)*, Enrico Franconi, Michael Kifer, and Wolfgang May (Eds.), Vol. 4519. Springer, 16–26. http://dblp.uni-trier.de/db/conf/esws/eswc2007.html#BaumgartnerFG07

[7] Andrew Carlson and Charles Schafer. 2008. Bootstrapping Information Extraction from Semi-structured Web Pages.. In *ECML/PKDD (1) (Lecture Notes in Computer Science)*, Walter Daelemans, Bart Goethals, and Katharina Morik (Eds.), Vol. 5211. Springer, 195–210. http://dblp.uni-trier.de/db/conf/pkdd/pkdd2008-1.html#CarlsonS08

[8] V. Chvatal. 1979. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research* 4, 3 (1979), 233–235. https://doi.org/10.2307/3689577

[9] V Crescenzi. 2001. RoadRunner: Towards Automatic Data Extraction from Large Web Sites. *International Conference on Very Large Data Bases (VLDB)* (2001). http://www.vldb.org/conf/2001/P109.pdf

[10] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *ICML*.

[11] Tim Furche, Georg Gottlob, Giovanni Grasso, Ömer Gunes, Xiaonan Guo, Andrey Kravchenko, Giorgio Orsi, Christian Schallhart, Andrew Sellers, and Cheng Wang. 2012. DIADEM : Domain-centric , Intelligent , Automated Data Extraction Methodology Categories and Subject Descriptors. *The World Wide Web Conference* (2012).

[12] Anna Lisa Gentile, Ziqi Zhang, and Fabio Ciravegna. 2015. Early Steps Towards Web Scale Information Extraction with LODIE. *AI Magazine* 36, 1 (2015), 55–64. http://dblp.uni-trier.de/db/journals/aim/aim36.html#GentileZC15

[13] Pankaj Gulhane, Amit Madaan, Rupesh R. Mehta, Jeyashankher Ramamirtham, Rajeev Rastogi, Sandeepkumar Satpal, Srinivasan H. Sengamedu, Ashwin Tengli, and Charu Tiwari. 2011. Web-scale information extraction with vertex.. In *ICDE*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 1209–1220. http://dblp.uni-trier.de/db/conf/icde/icde2011.html#GulhaneMMRRSSTT11

[14] Pankaj Gulhane, Rajeev Rastogi, Srinivasan H. Sengamedu, and Ashwin Tengli. 2010. Exploiting Content Redundancy for Web Information Extraction. *PVLDB* 3, 1 (2010), 578–587. http://dblp.uni-trier.de/db/journals/pvldb/pvldb3.html#GulhaneRST10

[15] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *Principles of Programming Languages (POPL)*. 317–330.

[16] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012).

[17] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. 2009. The weka data mining software: an update. *ACM SIGKDD explorations newsletter 11, 1* (2009).

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

SIGIR, Wei-Ying Ma, Jian-Yun Nie, Ricardo A. Baeza-Yates, Tat-Seng Chua, and W. Bruce Croft (Eds.). ACM, 775–784. http://dblp.uni-trier.de/db/conf/sigir/sigir2011.html#HaoCPZ11

[19] Chun-Nan Hsu and Ming-Tzung Dung. 1998. Generating Finite-State Transducers for Semi-Structured Data Extraction from the Web. *Information Systems* 23, 8 (1998), 521–538.

[20] import.io. 2018. *import.io*. http://www.import.io

[21] Jeevana Priya Inala and Rishabh Singh. 2018. WebRelate: integrating web data with spreadsheets using examples. *PACMPL* 2, POPL (2018), 2:1–2:28. http://dblp.uni-trier.de/db/journals/pacmpl/pacmpl2.html#InalaS18

[22] Mozenda Inc. 2018. *Mozenda*. http://www.mozenda.com/

[23] G. H. John and P. Langley. 1995. Estimating Continuous Distributions in Bayesian Classifiers. *11th Conference on Uncertainty in Artificial Intelligence* (1995), 338–345.

[24] Iraklis Kordomatis, Christoph Herzog, Ruslan R. Fayzrakhmanov, Bernhard Krüpl-Sypien, Wolfgang Holzinger, and Robert Baumgartner. 2013. Web object identification for web automation and meta-search.. In *WIMS*, David Camacho, Rajendra Akerkar, and María Dolores Rodríguez-Moreno (Eds.). ACM, 13. http://dblp.uni-trier.de/db/conf/wims/wims2013.html#KordomatisHFKHB13

[25] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. 1997. Wrapper Induction for Information Extraction. In *IJCAI-97*.

[26] Tessa A. Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1-2 (2003), 111–156. http://dblp.uni-trier.de/db/journals/ml/ml53.html#LauWDW03

[27] Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples.. In *PLDI*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 55. http://dblp.uni-trier.de/db/conf/pldi/pldi2014.html#LeG14

[28] Henry Lieberman (Ed.). 2001. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers.

[29] Colin Lockard, Xin Luna Dong, Arash Einolghozati, and Prashant Shiralkar. 2018. CERES: Distantly Supervised Relation Extraction from the Semi-Structured Web. *PVLDB* 11, 10 (2018), 1084–1096. http://dblp.uni-trier.de/db/journals/pvldb/pvldb11.html#LockardDSE18

[30] Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. 2013. Integrating Programming by Example and Natural Language Programming.. In *AAAI*, Marie desJardins and Michael L. Littman (Eds.). AAAI Press. http://dblp.uni-trier.de/db/conf/aaai/aaai2013.html#ManshadiGA13

[31] Ion Muslea, Steven Minton, and Craig A. Knoblock. 1999. A Hierarchical Approach to Wrapper Induction. In *Autonomous Agents and Multi-Agent Systems*. 190–197. http://dblp.uni-trier.de/db/conf/agents/agents99.html#MusleaMK99

[32] Jussi Myllymaki and Jared Jackson. 2002. IBM Research Report Robust Web Data Extraction with XML Path Expressions. *Technical Report, IBM* (2002).

[33] Joachim Nielandt, Antoon Bronselaer, and Guy De Tré. 2016. Predicate enrichment of aligned XPaths for wrapper induction. *Expert Syst. Appl.* 51 (2016), 259–275. http://dblp.uni-trier.de/db/journals/eswa/eswa51.html#NielandtBT16

[34] Joachim Nielandt, Robin De Mol, Antoon Bronselaer, and Guy De Tré. 2014. Wrapper Induction by XPath Alignment.. In *KDIR*, Ana L. N. Fred and Joaquim Filipe (Eds.). SciTePress, 492–500. http://dblp.uni-trier.de/db/conf/ic3k/kdir2014.html#NielandtMBT14

[35] Adi Omari, Sharon Shoham, and Eran Yahav. 2017. Synthesis of Forgiving Data Extractors.. In *WSDM*, Maarten de Rijke, Milad Shokouhi, Andrew Tomkins, and Min Zhang (Eds.). ACM, 385–394. http://dblp.uni-trier.de/db/conf/wsdm/wsdm2017.html#OmariSY17

[36] Ermelinda Oro, Massimo Ruffolo, and Steffen Staab. 2010. SXPath - Extending XPath towards Spatial Querying on Web Documents. *PVLDB* 4, 2 (2010), 129–140. http://www.vldb.org/pvldb/vol4/p129-oro.pdf

[37] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis.. In *OOPSLA*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 107–126. http://dblp.uni-trier.de/db/conf/oopsla/oopsla2015.html#PolozovG15

[38] Ross Quinlan. 2014. *C4. 5: programs for machine learning*. Elsevier.

[39] Mohammad Raza and Sumit Gulwani. 2017. Automated Data Extraction Using Predictive Program Synthesis.. In *AAAI*, Satinder P. Singh and Shaul Markovitch (Eds.). AAAI Press, 882–890. http://dblp.uni-trier.de/db/conf/aaai/aaai2017.html#RazaG17

[40] Mohammad Raza and Sumit Gulwani. 2018. Disjunctive Program Synthesis: A Robust Approach to Programming by Example.. In *AAAI*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 1403–1412. http://dblp.uni-trier.de/db/conf/aaai/aaai2018.html#RazaG18

[41] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2014. Programming by Example using Least General Generalizations. In *AAAI*.

[42] Arnaud Sahuguet and Fabien Azavant. 1999. Building Light-Weight Wrappers for Legacy Web Data-Sources Using W4F.. In *VLDB* (2002-01-03), Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 738–741. http://dblp.uni-trier.de/db/conf/vldb/vldb99.html#SahuguetA99

[43] SelectorGadget. 2018. *SelectorGadget*. https://selectorgadget.com/

[44] Rishabh Singh. 2016. BlinkFill: Semi-supervised Programming by Example for Syntactic String Transformations. In *PVLDB*. 816–827.

[45] Yanhong Zhai and Bing Liu. 2005. Web data extraction based on partial tree alignment.. In *WWW*, Allan Ellis and Tatsuya Hagino (Eds.). ACM, 76–85. http://dblp.uni-trier.de/db/conf/www/www2005.html#ZhaiL05

[46] Shuyi Zheng, Ruihua Song, Ji-Rong Wen, and C. Lee Giles. 2009. Efficient record-level wrapper induction.. In *CIKM*, David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy J. Lin (Eds.). ACM, 47–56. http://dblp.uni-trier.de/db/conf/cikm/cikm2009.html#ZhengSWG09