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 robustness and usability challenges. In this work we present a novel program synthesis approach which combines the benefits of deductive and enumerative synthesis strategies, yielding a semi-supervised technique with which concise programs expressible in standard languages can be synthesized from very few examples. We demonstrate improvement over existing techniques in terms of overall accuracy, number of examples required, and program complexity. Our method has been deployed 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. 2020. Web data extraction using hybrid program synthesis: a combination of top-down and bottom-up inference. In *SIGMOD 2020*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

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 [21]. 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 [21], STALKER [26], Lixto [4], Mozenda [18], import.io [16], SelectorGadget [39]), 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, as is evident from numerous online discussions in help forums 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 the schema of each 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 data manipulation operations, the system automatically develops a program (in the lightweight "M" programming language) which the user can re-execute on different datasets or manually edit for more advanced tasks. However, web extraction support has again been severely

SIGMOD '20, June 14–19, 2020, Portland, OR

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Stack Overflow question:

Target webpage:

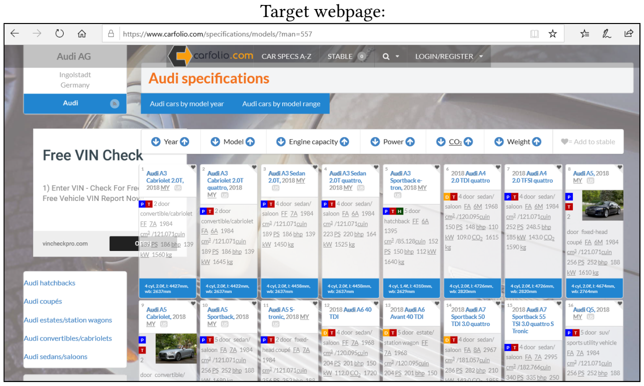
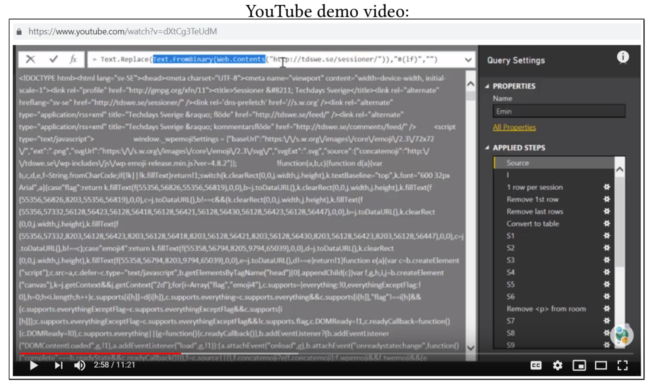


Figure 1: Forum question: web extraction in Python

YouTube demo video:



Target webpage:

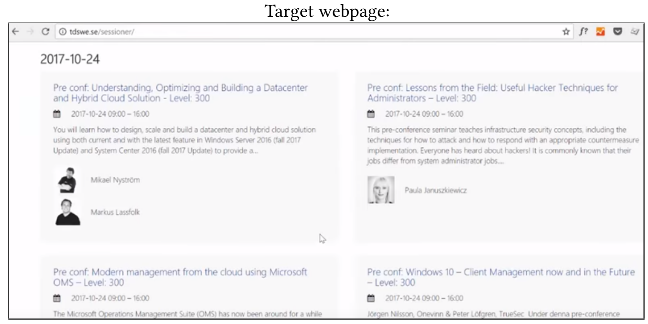


Figure 2: Demo video: web extraction in Power BI

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 from examples that addresses the difficulties faced by analysts in performing web extraction tasks. We discuss some of the goals in developing such a system, which include robust inference from a small number of examples, learning concise readable programs in standard web languages, and inference from text-based examples.

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. Since a user cannot be expected to provide all of these as examples even on a single webpage, 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 to ensure that all of the data has been extracted. Existing approaches not aimed at minimizing examples often conservatively favour more specific programs [2, 23, 28, 29] which can overfit and require many examples to sufficiently generalize, which can lead to significant burden on the user. For example, it would be difficult to identify random elements missing from 1671 items in Figure 1. In general, correct inference from a few examples in the majority of cases would help to instill user confidence in the robustness of the system.

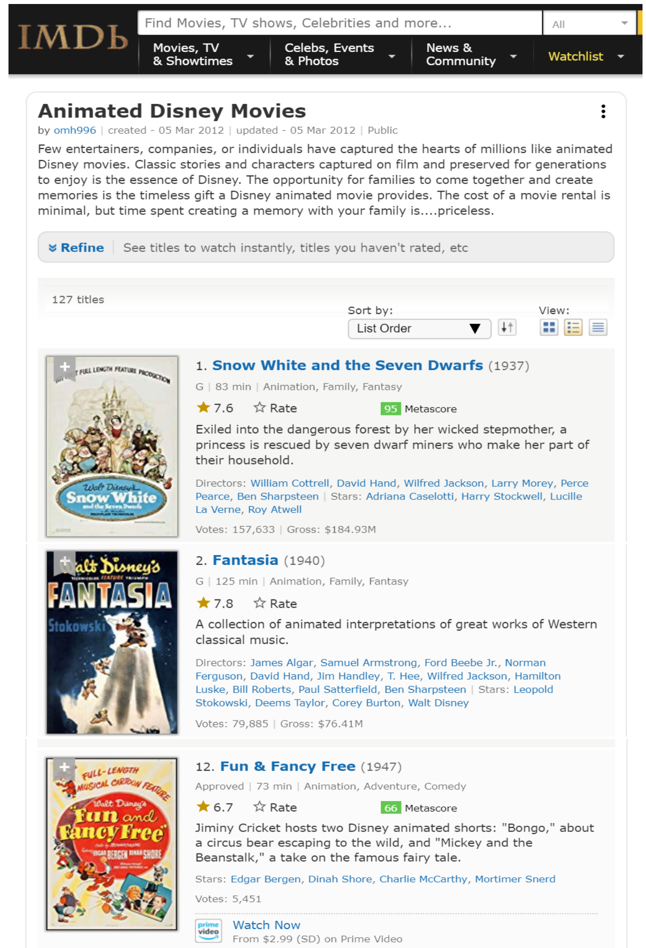
Inferring simple programs in standard, lightweight languages. Another common requirement in practice is that the synthesized programs be represented in a standard and lightweight language, such as DOM query languages like XPath or CSS (which are W3C standards), so that users are likely to be or can easily become familiar with the language and can also understand or manipulate the synthesized programs in common HTML tools or modern web browsers. This requirement is in contrast to wrapper induction approaches that often employ more complex extraction models, such as incorporating visual/semantic features or specialized treatment of particular verticals [5, 9, 14, 20, 31]. Moreover, even with standard 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 often yields very complex programs, e.g. long path sequences or large conjunctions in path alignment or least general generalization approaches [2, 28, 29, 32, 37], or large disjunctive expressions to guard against overfitting [30, 35]. Such expressions are difficult to interpret as compared to the simple selectors that a human expert may write for the same task.

Text-based examples Many wrapper induction systems provide a visual interface in which users can point-and-click actions to give examples of data items of interest on a webpage. Such visual interfaces may not always be easily integrated into data analysis environments that commonly employ text-based interaction paradigms, e.g. Python scripts in IDEs with intellisense or Notebooks that employ a REPL (read-eval-print loop) interaction model. Other benefits of text-based examples include: *(1) Wide-scale adoption.* Web extraction support may be more easily integrated in different products and services without the need for heavy UI investment. *(2) Bypassing limitations of visual UIs.* Modern websites often employ dynamic scripts on webpages that alter the content or layout depending on user actions such as hovering over regions in the page, which is an obstacle for visual UIs. *(3) Robustness to changing site formats.* The text examples may be used to relearn a new program if a previously learned program fails due to format changes.

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. Our approach is based on the following key technical contributions:

Combination of top-down and bottom-up synthesis. The field of program synthesis has seen rising interest and progress in recent years [1, 8, 12, 22, 24, 32, 34, 40]. 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 [17, 23, 32, 35]. Such deductive approaches have the benefit of efficiently constraining the search to only those DSL fragments relevant to the examples provided by the user. However, recent work has explored a *bottom-up* approach [34] for *unsupervised* program synthesis, where extraction programs are inferred without any examples from the user. This is done by systematically enumerating DSL programs




```
<div class="list_item odd">
  <div class="image">...</div>
  <div class="number">1.</div>
  <div class="info">
    <b><a href="...">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 ... <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="...">Fantasia</a><span class="year_type">(1940)</span></b>
    <div class="rating rating-list"> ... </div>
    <div class="item_description">A collection of animated ... <span>(125 mins.)</span></div>
    <div class="secondary">Director: James Algar, ... </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="...">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 ... <span>(70 mins.)</span></div>
    <div class="secondary">Stars: Edgar Bergen, Dinah Shore, ... </div>
    <div> ... </div></div></div>
```

Figure 3: IMDB movies list page & simplified HTML

and detecting alignment patterns between node selections produced by these programs.

In this work we observe that the bottom-up and top-down strategies have complimentary strengths and weaknesses, and we develop a hybrid program synthesis approach that utilises the bottom-up analysis to improve top-down inference. 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 a very simplified version of its 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 used a more expressive DSL in which the title extraction was supported. However, given the first two example titles ("Snow White 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 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 synthesis methods. This is because there can be many properties that a small number of example nodes may share, and any of the exponential possible subsets of these properties 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) [2, 23, 29, 32, 37]. Apart from creating syntactically complex programs, this easily causes overfitting by constraining to too many shared properties of the examples. We 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) and show how concise predicate conjunctions can be inferred using a maximal set cover approach to exclude negative examples.

Text-to-node disambiguation. For cases where node-based examples cannot be provided, our system 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 multiple times, or product search pages may contain the same prices or brand names for different products in the list. A set of a few text examples can combinatorially lead to hundreds of matching node combinations. We address the text-to-node disambiguation problem by ranking possible node combinations using a number of structural features, as well as utilising the global bottom-up document analysis.

Some details of this work are in our technical report [36].

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

Figure 4: The DSL $\mathcal{L}$ for HTML node selection

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 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 4 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 XPath and CSS (we define the DSL independently of the syntax of these languages to keep the synthesis formulation generic). 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 $\mathsf{Filter}(p, s)$ or a disjunction $\mathsf{Disj}(f, ..., f)$ of any number of filter expressions (disjunction is equivalent to the union operator "|" in XPath or "," in CSS). A simple filter expression $\mathsf{Filter}(p, s)$ 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 the particular atomic node-level predicates, and these may vary in different environments (e.g. some attributes such as text content may be expressible in XPath but not in CSS, and we also avoid 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 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 4, we have distinguished two notable fragments of the DSL: we refer to as $\mathcal{L}_t$ the fragment that excludes operators with a dotted underline, and $\mathcal{L}_b$ 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 better suited to efficient bottom-up enumeration of programs, as shown in [34]. Its use of the descendant operator also allows a bottom-up search to explore more expressive logics rather than the limited node neighborhoods accessible to the direct child operator. In contrast, $\mathcal{L}_t$ is a richer fragment but uses the direct child operator that is better suited to top-down deductive inference as constraints can be tractably propagated through the node levels [17, 32, 35]. In this work, the richness of our full DSL $\mathcal{L}$ which includes both $\mathcal{L}_t$ and $\mathcal{L}_b$ enables the inference of more concise programs (e.g. a single descendant expression rather than a long sequence of child steps) as well as handling tasks that may otherwise not be expressible in either approach.

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 page. The example specification may optionally include the precise nodes on the webpage which

1: **function** SynthProg(d, E)
2: $\quad P \leftarrow$ SynthFilterProg(d, E)
3: $\quad$ **if** $P \neq$ `null` **return** P
4: $\quad E_1 \leftarrow$ Max$(\{E' \mid E' \subseteq E \wedge \text{SynthFilterProg}(d, E') \neq \text{null}\})$
5: $\quad P_1 \leftarrow$ SynthFilterProg(d, E_1)
6: $\quad P_2 \leftarrow$ SynthProg$(d, E \backslash E_1)$
7: $\quad$ **return** Disj(P_1, P_2)

1: **function** SynthFilterProg(d, E)
2: $\quad \mathcal{E} \leftarrow$ EnumerateBottomUp(d)
3: $\quad \mathcal{G} \leftarrow$ TopAlignmentGroups$(\mathcal{E})$
4: $\quad \mathcal{N} \leftarrow$ TopNodeCombinations$(d, E, \mathcal{G})$
5: $\quad$ **for each** $N \in \mathcal{N}$ until max iterations bound **do**
6: $\quad\quad P_t \leftarrow$ SynthTopDown$(d, N,$ `null`$)$
7: $\quad\quad$ **if** $P_t \neq$ `null` **then**
8: $\quad\quad\quad P_h \leftarrow$ SynthHybrid$(d, N, \mathcal{G}, P_t)$
9: $\quad\quad\quad$ **if** $P_h \neq$ `null` **return** P_h **else return** P_t
10: $\quad$ **return** SynthBottomUp$(d, E, \mathcal{G}, \mathcal{E})$

Figure 5: Program synthesis algorithm

contain each of the text values (e.g. from a visual point-and-click UI for instance). For instance, Figure 3 shows a sample of an IMDB page containing 100 movies. To extract all the movie names the user can provide the first two examples:

[("Snow White and the Seven Dwarfs", n_1), ("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. 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.

We first give a summary description of the top-level algorithm, which is shown in Figure 5, and shall then describe the main components in more detail. The algorithm implements a combination of top-down and bottom-up program synthesis. It uses bottom-up exploration to infer sets of programs that reveal *alignment patterns* of nodes on the webpage, independent of any user-provided examples. This global analysis is used as a signal to improve a supervised top-down synthesis in order to favour those programs whose results align with the inferred structural patterns. The main function SynthProg(d, E) returns a program in $\mathcal{L}$ that satisfies examples E on document d. This function first attempts to synthesize a filter program that satisfies the examples (lines 2-3), and if no such program is found then it returns a minimal

function TopNodeCombinations($d, E, \mathcal{G}$)
  let $E = [(t_1, n_1), ..., (t_k, n_k)]$
  if $n_i \neq$ null for all $i = 1 \dots k$ **then**
    $N \leftarrow [n_1, ..., n_k]$
    return $[N]$
  let $T = [t_1, ..., t_k]$
  let $[S_1, ..., S_k]$ such that $S_i = \{n \in d \mid n.\texttt{Text} = t_i\}$
  $\mathcal{N} \leftarrow S_1 \times \dots \times S_k$
  return $\mathcal{N}$ where $N \in \mathcal{N}$ are ordered lexically by
   BottomUpAlignment($N, \mathcal{G}$), UniformTags(N, d),
   ExtremalNodes(N, d), UniqueCommonAncestor(N, d),
   UniformTagClass(N, d), NodeDistanceDev(N, d)

Figure 6: Node combinations from text specification

disjunction of filter expressions that cover all the examples (lines 4-7). The function SynthFilterProg(d, E) synthesizes non-disjunctive filter expressions. It starts by performing an unsupervised analysis of the webpage enumerating 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. Next we infer the top-ranked 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 this inference is successful on a node combination, then we perform the *hybrid synthesis* that checks if top-down inference can be improved using the bottom-up analysis (lines 7-9). Finally, if no satisfying programs could be found, then we fall back to a purely bottom-up search (line 10).

We state some basic definitions. For nodes n, n' in a webpage, 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 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 $N = [n_1, \dots, n_k]$ and there exists a subsequence $[n'_1, \dots, n'_k]$ of N' such that each n'_i is an ancestor of n_i.

Bottom-up synthesis. The bottom-up method we use is based on [34], and proceeds by enumerating and finding groups of programs that exhibit strong alignment patterns. Enumeration is done by the method EnumerateBottomUp(d) in Figure 5, 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 webpage, that is $[\![P]\!](d) = N$. It performs efficient enumeration in the DSL fragment $\mathcal{L}_b$ by recursive rule application using lifting functions and other optimizations such as semantic equivalence [34]. 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 $(P_a, (P_1, \dots, P_n))$, 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 rank by largest alignment groups. Unlike [34], we do the additional step of finding an ancestor state from $\mathcal{E}$ for each alignment group. Considering the IMDB example from Figure 3, the enumeration and alignment analysis yields a highly ranked alignment group with ancestor program ".list_item" that selects the 100 DIV elements for each movie. The inferred 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, e.g. the movie title requires a selector ".info > B > A" which lies outside the bottom-up DSL $\mathcal{L}_b$, and we shall describe how our hybrid synthesis approach utilises the alignment group to infer the title selector.

Although the bottom-up synthesis is mainly used for an unsupervised analysis of the webpage, in the final step of the main algorithm (Figure 5) we resort to a purely bottom-up search if no satisfying program is found in the top-down DSL. The function SynthBottomUp($d, E, \mathcal{G}, \mathcal{E}$) searches for a satisfying program first within the top-ranked alignment groups $\mathcal{G}$, and then all remaining enumerated states $\mathcal{E}$.

Text-node disambiguation. Figure 6 shows the function for inferring top-ranked node combinations that match text-based examples. It returns a ranked list $\mathcal{N}$, where each $N \in \mathcal{N}$ is a sequence of nodes such that $|N| = |T|$ and $N[k].\texttt{Text} = T[k]$ for all k. It 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 leveraging both the bottom-up analysis as well as structural properties of the document. The BottomUpAlignment feature prefers node combinations that are consistent with any of the alignment groups created by the bottom-up analysis. Formally, BottomUpAlignment($N, \mathcal{G}, d$) if and only if there exists some $(P_a, (P_1, \dots, P_n)) \in \mathcal{G}$ such that IsAncSeq($N, [\![P_a]\!](d)$). The remaining features are based on uniformity of node attributes and structural properties of the nodes in the document. They 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), UniformTagClass (all nodes have same pattern of tags and class names in their ancestor hierarchy), and NodeDistanceDev (the nodes occur at uniform distances between each other, in terms of document ordering).

```
function SynthTopDown(d, N_e, N_a)
    N_p ← {n ∈ d | n is parent of some n′ ∈ N_e}
    P_p ← SynthTopDown(d, N_p, N_a)
    𝒫_s = { ChildrenOf(P_p), AllNodes() }
    𝒫 ← ∅
    for each P_s ∈ 𝒫_s do
        N_s ← ⟦P_s⟧(d)
        P_c ← SynthPredicate(d, N_s, N_e, N_a)
        P = Filter(P_c, P_s)
        if N_a = null ∨ SatisfiesAncSeq(⟦P⟧(d), N_e, N_a) then
            𝒫 ← 𝒫 ∪ {P}
    return ArgMin_{P∈𝒫} (|⟦P⟧(d)|, Size(p))

function SatisfiesAncSeq(N, N_e, N_a)
    if IsAncSeq(N_e, N_a) return ∀n∈N.∃n′∈N_a. IsAnc(n, n′)
    else return ∀n ∈ N_a.∃n′ ∈ N. IsAnc(n, n′)
```

Figure 7: Top-down synthesis

```
function SynthHybrid(d, N, 𝒢, P_t)
    N_t ← ⟦P_t⟧(d)
    𝒩_a ← ∅
    for each (P_a, (P_1, ..., P_k)) ∈ 𝒢 do
        N_a ← ⟦P_a⟧(d)
        if IsAncSeq(N_t, N_a) ∧ LCA(N_t) = LCA(N_a) then
            for i = 1...k do
                N_i ← ⟦P_i⟧(d)
                if ∀j = 1...|N|. N_i[j] = N[j] return P_i
        𝒩_a ← 𝒩_a ∪ {N_a}
    for each N_a ∈ 𝒩_a do
        P_h ← SynthTopDown(d, N, N_a)
        if P_h ≠ null return P_h
    return null
```

Figure 8: Hybrid program synthesis

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.

Top-down synthesis & predicate inference. The function SynthTopDown(d, N_e, N_a) in Figure 7 returns a program P in the DSL fragment $\mathcal{L}_t$ 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 $\forall n \in N_a.\exists n' \in N'.\mathsf{IsAnc}(n', n)$ (this is an optional parameter we make use of in hybrid synthesis). Following standard top-down approaches [17, 32, 35], the function proceeds by propagating examples constraints for each of the parameters of the applicable DSL operators (in this case Filter, AllNodes and ChildrenOf). However, unlike previous techniques, we use a novel predicate inference method that alleviates overfitting and infers concise conjunctions.

The SynthPredicate(d, N_s, N_e, N_a) function infers a predicate P that satisfies all the example nodes N_e from a set of nodes N_s and satisfies the ancestor constraint N_a. The first step 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 conjunction of these predicates to choose, as any subset of the predicates is a valid generalization but selects different nodes from N_s. Approaches that simply take the conjunction of all predicates (*least general generalization*) create syntactically complex programs that also overfit and require many examples to generalize. 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 avoid as many *implicit negative examples* as possible. The implicit negative examples we choose is a heuristic choice that prefers to avoid nodes that lie outside the LCA of the example nodes (prefer 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). We use a greedy approximation algorithm to the set cover problem [6] to produce the smallest conjunctions, by incrementally adding predicates that exclude the largest number of uncovered negative examples. The one additional check we make is to exclude any predicates that do not satisfy the ancestor constraint N_a if one is provided (details available in our full technical report [36]).

Hybrid synthesis. We now describe our *hybrid synthesis* approach which combines the top down inference (which is prone to overfitting given the expressive DSL and a few examples) with the bottom-up analysis (which infers global structural patterns but it is limited by the very restricted DSL). In the main synthesis algorithm SynthFilterProg in Figure 5, 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 is shown in Figure 8. It synthesizes a program that satisfies the node specification N, 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 10). 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 11-14).

We illustrate the hybrid approach with the example movies page in Figure 3. We have 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 the director field as shown in Figure 3. 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 re-perform the top-down synthesis using the group ancestor nodes 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.

Overall accuracy across documents. We first demonstrate improvement in overall accuracy of our hybrid synthesis approach (HYB) in comparison to the current state-of-the-art approaches. These include the recent work [30] on *forgiving data extractors* (FX), their corresponding non-forgiving synthesis method (NFX), the C4.5 classifier of [33] (C4.5), a naive bayes classifier [19] (NB), XPath alignment-based synthesis [29] (XA), and synthesis using least general generalizations [23, 32] (LGG). The implementations for FX, NFX, C4.5, NB, and XA were from [30] (some using Weka [13]), and LGG is from [32] (many thanks to the authors).

	Precision	Recall	F1
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

Figure 9: Precision, recall & F1 with 95% C.I.

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 [30] which contains 166 manually annotated pages from 30 websites ranging over 4 verticals (books, shopping, hotels and movies). However, we observed that the ground truth in the DS1 dataset included significant redundancy in terms of multiple labelled nodes for 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. Since such redundant extraction is often not the case in practice, we created the dataset **DS1-b**, which uses the same webpages and tasks from DS1 but without duplicates. 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). We obtained SWDE as a subset of the larger structured web data extraction dataset from [14]. Since the original dataset contained text values rather than node annotations, we manually annotated the first two pages for each (vertical, site, attribute) combination so that SWDE maintains the same variety as the original dataset but fewer page instances.

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 from the training set, and measured accuracy of the synthesized extractor on the pages from the test set. Figure 9 shows the precision, recall and F-measure averaged across all tasks in all datasets, along with the corresponding 95% confidence interval (CI). The main result is that our system HYB had the highest average F1 score of 0.86 and this is a statistically significant improvement over all other systems at the 95% confidence level (no overlap between the CIs). HYB also had the highest precision with significance. For recall, FX and NB were significantly higher but they were the lowest ranking systems overall, while NFX was slightly higher but with overlapping CIs. The top performers on each dataset individually were DS1: HYB, FX and NFX; DS1-b:

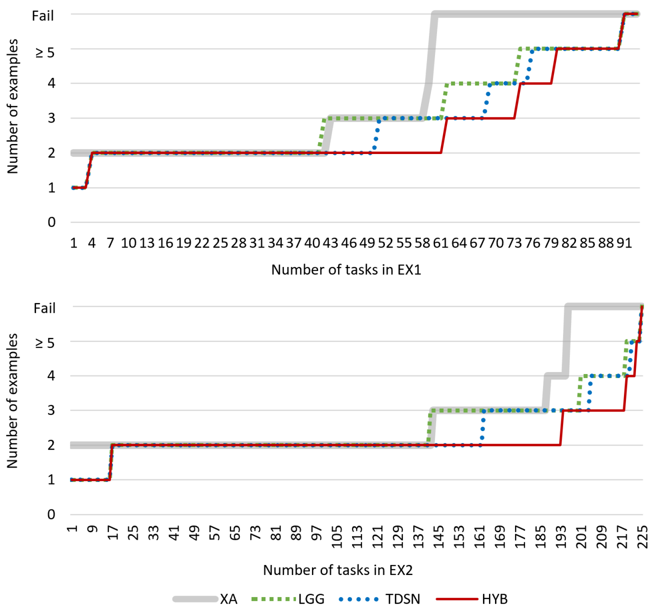


Figure 10: Number of examples for tasks in EX1 & EX2

HYB and NFX; SWDE: HYB and XA (detailed results in our full technical report [36]). 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. Overall, HYB was the only system that was among the top performers in each of the datasets individually, and it was the single best performer overall with statistical significance.

Number of examples per document. Measuring accuracy across documents using training/test sets assumes the user must provide *all* of the desired nodes from each document when training the system. As this can be difficult and error-prone for *list pages* where a single page may contain hundreds of nodes to extract, systems supporting *partial examples specifications* allow the user to only give a small subset of the desired nodes in a page. Here we evaluate how many examples are required to give to the system before it can learn the full extraction. The first dataset we used is **EX1**, consisting of all list-like extraction tasks from the datasets DS1, DS1-b and SWDE where there are at least 5 nodes to extract (since inference from few examples would be trivial for smaller extractions). This gave a total of 93 tasks (average of 12.8 nodes per task). We also used a bigger dataset **EX2** containing 225 list extraction tasks from 66 webpages (average of 56.9 nodes per task). These were representative customer scenarios from the Power BI product team, as well as real use cases we collected from online forums (links to the original sources are included with our dataset).

We compared against the two baseline systems LGG and XA that also support partial examples (FX, NFX, C4.5 and NB do not support partial examples by design, as they assume all non-example nodes are negative examples). We also compared with TDSN, which is our top-down system using the greedy soft negative examples heuristic but not hybrid synthesis. For each task, we provided examples to the system incrementally according to document order of nodes in the webpage, until all nodes were extracted. Figure 10 shows the number of examples required for completion of tasks in EX1 and EX2. The main result is that for both datasets, our system HYB completed the most tasks with 2 examples or less. The relative performance of the systems followed a similar pattern in both datasets: the proportion 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%. More examples were required by all systems for EX1, which is likely due to the high redundancy in tasks from DS1. Improvement in TDSN over LGG or XA (~9%) shows the effectiveness of the greedy set cover heuristic in reducing the number of examples. The more significant improvement in HYB over TDSN (~13%) shows the greater benefits obtained with our hybrid approach using bottom-up analysis. To compare against the purely unsupervised bottom-up approach that works without examples: the top table from such a system [34] failed on 68.8% of tasks in EX1 and 31.6% for EX2.

Program complexity. The complexity of synthesized programs is another important usability aspect, as programs with numerous expressions can be difficult for users to understand or edit if required. We compared the number of operators used in the CSS selectors synthesized by our system HYB with the other systems that also synthesize CSS selectors (LGG and TDSN). Across all datasets, 13.4% of programs from LGG had three operators or fewer, while this increased to 68.4% for programs from HYB. The average number of operators were LGG: 7.9, TDSN: 3.8, HYB: 3.6. The complexity of HYB programs is also comparable to human-written CSS selectors which usually contain about 3 to 4 operators for most extraction tasks. For some qualitative illustration, the following table shows the CSS selectors synthesized by the three systems for a sample task, where HYB could create a much simpler selector by using the descendant operator rather than long child paths created by the other systems:

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)

As we cannot compare number of operators with XPath synthesis methods, for approximate comparison we give the string size 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 complex expressions were created by FX (many disjuncts) and XA (long paths).

We have addressed program complexity using methods like minimal set cover and expressive DSLs. Though the problem is more general than program equivalence (unequivalent programs may be preferable if they satisfy examples), reduction based on full equivalence may yield further benefits.

Text-based examples. As previous approaches have not addressed learning from text-only examples, we compared our text-node disambiguation system with a naive baseline in which we use our system but simply accept the first nodes in the document that match the text examples. We provided each system with the text-only examples for all the tasks in datasets EX1 and EX2. We observed overall improvement in the number of examples gained with our 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. More details are in the full technical report [36].

Deployment. Our approach has been deployed as a feature in the Microsoft Power BI product (under active development, with a version of the TDSN system currently released for general audience). It has been well-received by users as seen from numerous comments in online forums (see the sources for 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 approaches to web data extraction have mainly centered around *wrapper induction* [21], 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 [15, 21, 26], 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 [4, 27, 38], and since then there has been greater focus on learning wrappers from examples in standard HTML query languages such as XPath or CSS [2, 10, 28–30, 32, 42], which has also been our focus in this work. XPath alignment approaches [28, 29] work by aligning and merging the steps within the XPaths of sample nodes based on edit distances, while least general generalization methods [32] produce largest conjunctions of all common node attributes. Such approaches can lead to long path expressions or numerous predicates, which are complex to understand and over-fit to the examples. Some approaches such as *forgiving XPaths* (FX) [30] attempt to improve the recall and learn cross-site selectors by using multiple disjuncts in the generated selectors, but we have shown how this can lead to severe loss in precision. Machine learning techniques have also been explored such as naive-bayes classifiers [10] and decision trees (NFX system [30]), and we have also shown improvement over such approaches with our hybrid synthesis method.

Other related work has gone beyond the use of standard HTML languages and explored more complex extraction models, such as using visual or semantic features or specialized handling for particular vertical domains [5, 9, 14, 20, 31]. Though beneficial in many scenarios, such approaches are not designed to generate simple selector expressions that users can understand. Thus in this respect, our problem definition is more specialized than arbitrary information extraction, as it includes the requirement of inferring concise, readable programs in standard languages.

Fully automated web extraction approaches attempt to mine recurring patterns in the DOM structure of web pages without examples [3, 7, 34, 41]. Such approaches are good at finding prominent patterns, but cannot extract all kinds of information desired by different users. However, in our hybrid approach we have shown how to leverage such unsupervised analysis to quickly converge to the desired extraction.

Program synthesis has seen rising progress in recent years [1, 8, 11, 12, 22, 24, 25, 32, 34, 40], with commercial successes such as the *Flash Fill* feature in Microsoft Excel [11]. Such approaches aim to find a program in a domain-specific language (DSL) that satisfies user examples, usually using either bottom-up approaches that enumerate DSL programs [1, 34], or top-down approaches [12, 32] where constraints are propagated through the DSL structure. We have presented the first hybrid technique that combines the benefits of the two approaches into a semi-supervised synthesis system.

6 CONCLUSION

We have described a novel hybrid program synthesis approach for web data extraction programs, which provides inference of concise programs expressible in common languages from very few examples and text-only examples. 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. Although we have focussed on webpages, the fundamental concepts of hybrid synthesis may be formulated at a more abstract level and applicable to different document domains if we consider other selection DSLs, e.g. regex-based selectors for plain text, or spatial/position based selections for PDF or scanned documents. These will be interesting explorations for future work.

REFERENCES

[1] 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

[2] 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

[3] 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.

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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*.

[9] 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).

[10] 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

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

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

[13] 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).

[14] 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

[15] 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.

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

[17] 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

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

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

[20] 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

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

[22] 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

[23] 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

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

[25] 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

[26] 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

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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

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

[34] 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

[35] 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

[36] Mohammad Raza and Sumit Gulwani. 2019. Microsoft Technical Report: Web data extraction using hybrid program synthesis: a combination of top-down and bottom-up inference. https://www.microsoft.com/en-us/research/uploads/prod/2019/11/websynth-techreport.pdf

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

[38] 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

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

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

[41] 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

[42] 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