Web Question Answering with Neurosymbolic Program Synthesis

Qiaochu Chen
University of Texas at Austin
Austin, Texas, USA
qchen@cs.utexas.edu

Aaron Lamoreaux
University of Texas at Austin
Austin, Texas, USA
lamoreauxaj@gmail.com

Xinyu Wang
University of Michigan
Ann Arbor, Michigan, USA
xwangsd@umich.edu

Greg Durrett
University of Texas at Austin
Austin, Texas, USA
gdurrett@cs.utexas.edu

Osbert Bastani
University of Pennsylvania
Philadelphia, Pennsylvania, USA
obastani@seas.upenn.edu

Isil Dillig
University of Texas at Austin
Austin, Texas, USA
isil@cs.utexas.edu

Abstract

In this paper, we propose a new technique based on program synthesis for extracting information from webpages. Given a natural language query and a few labeled webpages, our method synthesizes a program that can be used to extract similar types of information from other unlabeled webpages. To handle websites with diverse structure, our approach employs a neurosymbolic DSL that incorporates both neural NLP models as well as standard language constructs for tree navigation and string manipulation. We also propose an optimal synthesis algorithm that generates all DSL programs that achieve optimal F_1 score on the training examples. Our synthesis technique is compositional, prunes the search space by exploiting a monotonicity property of the DSL, and uses transductive learning to select programs with good generalization power. We have implemented these ideas in a new tool called WebQA and evaluate it on 25 different tasks across multiple domains. Our experiments show that WebQA significantly outperforms existing tools such as state-of-the-art question answering models and wrapper induction systems.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; • **Information systems** → *Data extraction and integration.*

Keywords: Program Synthesis, Programming by Example, Web Information Extraction

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454047

ACM Reference Format:
Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021. Web Question Answering with Neurosymbolic Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454047

1 Introduction

As the amount of information available on the web proliferates, there is a growing need for tools that can extract relevant information from websites. Due to the importance of this problem, there has been a flurry of research activity on *information extraction* [39, 44] and *wrapper induction* [5, 10, 18, 27, 32, 37, 42, 50]. In particular, most recent research from the natural language processing (NLP) community focuses on unstructured text documents and employs powerful neural models to automate information extraction and question answering (QA) tasks. On the other hand, most wrapper induction work focuses on semi-structured documents and aims to synthesize programs (e.g., XPath queries) to extract relevant nodes from the DOM tree. While such wrapper induction techniques work well when the target webpages have a shared global schema (e.g., Yelp pages or LinkedIn profiles), they are not as effective on structurally heterogeneous websites such as faculty webpages. On the other hand, ML-based techniques from the NLP community are, in principal, applicable to heterogeneous websites; however, by treating the entire webpage as unstructured text, they fail to take advantage of the inherent tree structure of HTML documents.

In this paper, we propose a new information extraction approach —based on *neurosymbolic program synthesis* — that combines the relative strengths of wrapper induction techniques for webpages with the flexibility of neural models for unstructured documents. Our approach targets structurally heterogeneous websites with no shared global schema and

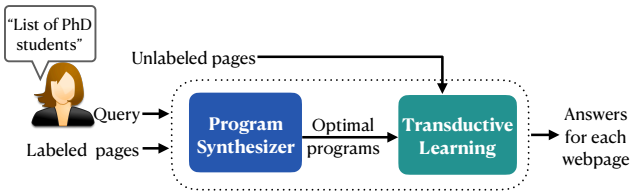

Figure 1. Schematic overview of our approach

can be used to automate many different types of information extraction tasks. Similar to prior program synthesis approaches[37, 57], our approach can learn useful extractors from a *small* number of labeled webpages.

As illustrated in Figure 1, our approach takes three inputs, including (1) a natural language query, (2) a small number of labeled webpages, and (3) a much bigger set of unlabeled webpages from which to extract information. For instance, if the task is to extract PhD students from faculty webpages, the input might consist of a question such as "Who are the PhD students?" as well as keywords like "advisees" and "PhD students". In addition, the user would also provide a set of target faculty webpages, together with labels (i.e., names of PhD students) for a few of these. Given this input, the goal of our technique is to generate a program that can be used to extract the desired information from all target webpages.

To solve this challenging problem, we employ a multi-pronged solution that incorporates three key ingredients:

- **Neurosymbolic DSL:** To combine the relative strengths of wrapper induction techniques with the flexibility of language models, we design a new neurosymbolic domain-specific language targeted for web question answering. Our DSL combines pre-trained neural modules for natural language processing with standard programming language constructs for string processing and tree traversal.
- **Optimal program synthesis:** To utilize this DSL for automated web information extraction, we describe a new program synthesis technique for finding DSL programs that best fit the labeled webpages. However, since it is often impossible to find programs that *exactly* fit the provided labels, we instead search for programs that optimize F_1 score[1]. Our proposed optimal synthesis method is compositional and leverages a monotonicity property of the DSL to aggressively prune parts of the search space that are guaranteed *not* to contain an optimal program.
- **Transductive program selection:** During synthesis, there are often *many* (e.g., hundreds of) DSL programs with optimal F_1 score on the labeled data. However, not all of these candidate programs perform well on test data, and standard heuristics (e.g., based on program size) are not effective at distinguishing between these programs. We address this challenge using transductive learning: it generates *soft labels* for the test data based on all candidate programs and then chooses the "consensus" program whose output most closely matches the soft labels.

We have implemented our proposed approach in a tool called WebQA and evaluate it across several different tasks and many webpages. Our evaluation demonstrates that WebQA yields significantly better results compared to existing baselines, including both question answering models and wrapper induction systems. We also perform ablation studies to evaluate the relative importance of our proposed techniques and show that all of these ideas are important for making this approach practical.

In summary, this paper makes the following contributions:

- We propose a new technique for web question answering that is based on optimal neurosymbolic program synthesis.
- We present a DSL for web information extraction that combines pre-trained NLP models with traditional language constructs for string manipulation and tree traversal.
- We describe a compositional program synthesis technique for finding all programs that achieve optimal F_1 score on the labeled webpages. Our synthesis algorithm prunes the search space by exploiting a monotonicity property of the DSL with respect to recall.
- We present a transductive learning technique for choosing a good program for labeling the target webpages.
- We implement our approach in a tool called WebQA and evaluate it on 25 different tasks spanning four domains and 160 webpages.

2 Motivating Example

In this section, we present a motivating scenario for WebQA and highlight salient features of our approach.

Usage scenario. Suppose that the PC chair for a conference needs to form a program committee, and she has access to the websites of many researchers. To help her form a good committee, she wants to extract program committees that each researcher has served on (which is often available on their websites). Since there are too many websites, extracting this information manually is too laborious. Our proposed system, WebQA, is useful in scenarios like this that require collecting information from many structurally heterogeneous websites.

To use WebQA, the user starts by providing a question (e.g., "Which program committees has this researcher served on?") and a set of keywords (e.g., "PC", "Program Committee", "Service"). Then, given a target set of websites, WebQA asks the user to provide labels for a small number of webpages. For instance, Figure 2 presents two (hypothetical) websites that WebQA may show to the user, with the user-provided labels highlighted in blue. Observe that both of these webpages are semi-structured in the sense that they contain clearly-delineated sections (e.g., Students, Service); however, they

[1] F_1 score is computed as $2 \cdot \frac{\texttt{precision} \cdot \texttt{recall}}{\texttt{precision} + \texttt{recall}}$. It is a common evaluation metric in information extraction.

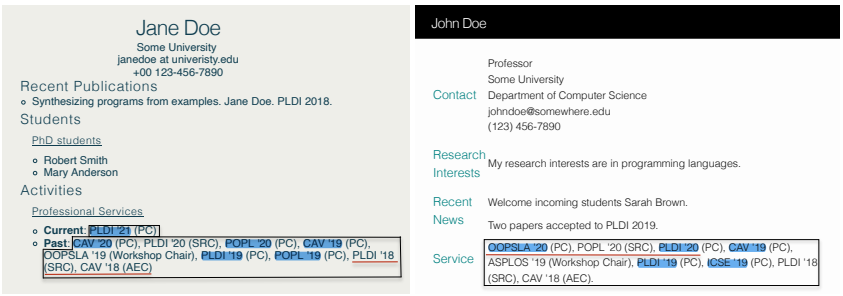

Figure 2. Sample faculty websites with their program committee information. Correct answers are in blue; the output of a QA model is underlined in red.

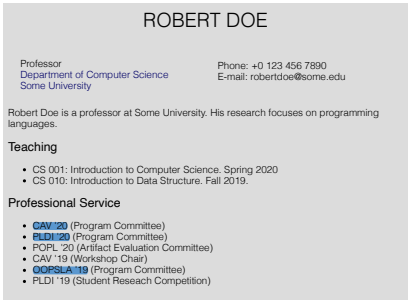

Figure 3. The synthesized program also works on this website; highlighted text are the extracted output.

differ both in terms of their high-level structure and what information they contain.

Limitations of existing approaches. We now use this simple motivating example to illustrate why existing approaches are not effective for this type of tasks. As mentioned in Section 1, there are two classes of techniques, namely *program induction* and *question answering*, that could potentially be useful in this setting.

Like our approach, program induction techniques aim to extract information from webpages based on a small number of user-provided training examples [35, 50]. Specifically, given a few labeled webpages, these techniques learn XPath expressions to locate relevant nodes in the DOM tree. However, as illustrated in Figure 2, researcher webpages typically do not have a uniform structure. Furthermore, even for webpages that are structurally somewhat similar, they exhibit minor variations (e.g., different section names, relative ordering of sections etc.) that make it very difficult to learn XPath expressions that generalize well to unseen websites. In addition, almost all existing techniques in this space focus on extracting relevant nodes in the DOM tree; however, they do not attempt to perform any further text processing within that node. As illustrated by both webpages in Figure 2, extracting the desired information requires further processing at the text level, such as extracting relevant substrings.

An alternative approach for automating this task is to use a state-of-the-art question answering (QA) system that treats the entire webpage as a raw sequence of words. However, in practice, such approaches perform poorly since they are not designed to leverage the tree structure of the document. Furthermore, because they treat text across different DOM nodes as natural language, they have difficulty dealing with more structured information like long comma-delineated lists or formatting with parentheticals. For instance, for the two webpages from Figure 2 and the question "Which program committees has this researcher served on?", a BERT-based QA system [19] yields the suboptimal answers underlined in red in Figure 2. In particular, it either outputs incorrect spans or includes text that should not be part of the answer (e.g. "POPL'20 (SRC)" in the second webpage).

Key idea #1: Neurosymbolic DSL. Our approach combines the relative strengths of machine learning and program induction techniques by synthesizing programs in a neurosymbolic DSL for web information extraction. In particular, our proposed DSL incorporates both pre-trained neural models for question answering, keyword matching, and entity extraction with standard programming language constructs for string processing and tree navigation. The tree navigation constructs allow taking advantage of webpage structure, while making it possible to handle minor variations (e.g., exact section names) using pre-trained neural models. Furthermore, the presence of string processing constructs in the DSL allows our method to extract fine-grained information within individual tree nodes.

In more detail, a program in our DSL is structured to first locate relevant nodes in the tree representation of a webpage (see Figure 4) and then perform additional information extraction from each tree node. For example, the following code snippet in our DSL can be used to locate the relevant parts of the webpages from Figure 2:

$$\texttt{GetLeaves}(\texttt{GetDescendents}(r, \lambda z.\texttt{matchKeyword}(z, K))) \quad (1)$$

Here, r is the root node of the input webpage, and the construct $\texttt{GetDescendants}(r, \phi)$ returns all tree nodes whose content satisfies predicate ϕ. In the code snippet above, the predicate $\lambda z.\texttt{matchKeyword}(z, K)$ is implemented by a neural network that has been pre-trained for keyword matching. Thus, this program first locates all tree nodes whose content matches any of the provided keywords K and returns all of their leaf nodes. For example, given the tree in Figure 4 representing the top webpage from Figure 2, the `GetDescendants` sub-program will match node 11, and `GetLeaves`[2] will return nodes 14 and 15, which are leaf nodes of the subtree

[2]Actually, there is no explicit `GetLeaves(v)` construct in our DSL; this is just syntactic sugar for `GetDescendants(v, λ n. isLeaf(n))`.

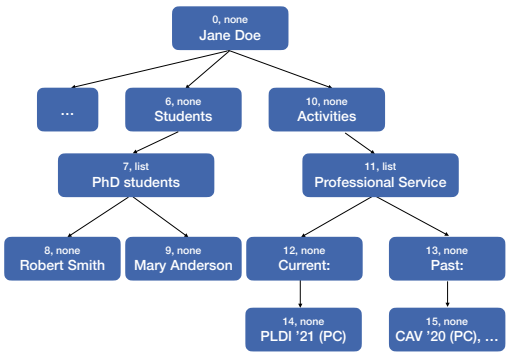

Figure 4. Tree representation of top webpage from Figure 2. Each node contains the node id, type and its content.

rooted at node 11. For the webpages in Figure 2, this program yields the tree nodes annotated using black boxes.

Next, given the tree nodes returned by the above code snippet, we can extract the desired information from these nodes using the following code snippet in our DSL:

$$\lambda x.\texttt{GetEntity}(\texttt{Filter}(\texttt{Split}(\texttt{ExtractContent}(x), \texttt{COMMA}), \lambda z.\texttt{matchKeyword}(z, K)), \texttt{ORG}) \quad (2)$$

In particular, this code snippet first retrieves the content of tree node x using `ExtractContent` and then splits it into a set of (comma-separated) strings using `Split`. Then, it filters those elements that do not match the provided keywords and finally extracts substrings that correspond to an organization entity[3]. Thus, assuming sufficiently good neural models for keyword matching and entity recognition, the output of this program would be exactly the highlighted text for the webpages from Figure 2.

It is worth noting that the extraction logic described above generalizes fairly well across websites with quite different layouts. In particular, the same DSL program can be used to extract the desired information from Figure 3 even though this webpage looks quite different from those in Figure 2.

Key idea #2: Allowing imperfect solutions. In our example so far, we were able to find a DSL program that produces exactly the highlighted text from examples in Figure 2. However, suppose that the pre-trained network for entity extraction is unable to recognize computer science conference names as organizations. In that case, the output of the extraction program from Eq. 2 would not exactly match the user-provided labels. In fact, there is *no* program in our DSL that would produce exactly the desired output.

To deal with this difficulty, our synthesis algorithm aims to find programs that maximize F_1 score rather than looking for solutions that exactly match the user-provided labels. Thus, we frame our problem as *optimal program synthesis*, where the goal is to find programs that maximize some optimization objective (F_1 score in our case). This optimality requirement makes the synthesis problem harder because we need to exhaustively explore the search space.

Key idea #3: Transductive learning. An additional difficulty in our context is that there may be hundreds or even thousands of optimal solutions for a synthesis task. In particular, given the scarcity of training examples, many different DSL programs yield the same F_1 score on the labeled webpages. For instance, for the two webpages from Figure 2, there are actually 85 optimal DSL programs that achieve the same F_1 score. Existing techniques in the synthesis literature deal with the under-constrained nature of input-output examples by using heuristics to distinguish different candidate solutions. However, standard heuristics (e.g., based on AST size) do not work well in our setting because there are still *many* programs that are tied with respect to such heuristics.

Our approach deals with this challenge using *transductive learning*. In particular, given all programs that yield optimal F_1 score on the labeled data, it generates *soft labels* for unlabeled webpages by running these programs on the unlabeled webpages and aggregating their outputs. Then, among these programs, we choose the one whose outputs most closely match the soft labels for the unlabeled webpages. In other words, transductive learning allows our method to utilize the unlabeled data to choose a most promising program and obviates the need for complex hand-crafted heuristics.

3 Preliminaries

In this section, we discuss how we represent webpages as trees. Our representation is different from the standard Document Object Model (DOM) and represents the nesting relationship between text elements on the *rendered* webpage to better facilitate web question answering.

Definition 3.1. (Webpage) A *webpage* is a tree (N, E, n_0) with root node $n_0 \in N$, nodes N and edges E. An edge is a pair (n, n') where n is the parent of n', and each node is a triple (id, text, type) where text is the string content of that node and type $\in \{list, table, none\}$ indicates whether the node corresponds to an HTML list, table, or neither.

Intuitively, an edge (n, n') indicates that the text of node n is the header for that of node n' — i.e., text of n' is nested inside that of n on the rendered version of the webpage. For instance, given an HTML document with title "Title" and body text "Text", our representation introduces an edge (n, n') where n has text "Title" and n' contains "Text".

In our representation, internal nodes can represent structured HTML elements like lists (both ordered and unordered) as well as tables. For a node n representing an HTML list (resp. table), n's children correspond to elements in the list (resp. rows of the table).

Example 3.2. Our method represents the first webpage in Figure 2 as the tree shown in Figure 4.

[3]Note that there is no explicit `GetEntity` on our DSL; this is a syntactic sugar for $\texttt{Substring}(e, \lambda z.\texttt{hasEntity}(z, \texttt{ORG}), 1)$.

$$
\begin{array}{rll}
\text{Program } p ::= & \lambda Q, K, W.\{\psi_1 \to \lambda x.e_1, \ldots, \psi_n \to \lambda x.e_n\} \\
\text{Guard } \psi ::= & \texttt{Sat}(\nu, \lambda z.\phi) \mid \texttt{IsSingleton}(\nu) \\
\text{Extractor } e ::= & \texttt{ExtractContent}(x) \\
\mid & \texttt{Substring}(e, \lambda z.\phi, k) \\
\mid & \texttt{Filter}(e, \lambda z.\phi) \\
\mid & \texttt{Split}(e, c) \\
\text{Section locator } \nu ::= & \texttt{GetRoot}(W) \\
\mid & \texttt{GetChildren}(\nu, \lambda n.\varphi) \\
\mid & \texttt{GetDescendants}(\nu, \lambda n.\varphi) \\
\text{Node filter } \varphi ::= & \texttt{isLeaf}(n) \mid \texttt{isElem}(n) \\
\mid & \texttt{matchText}(n, \lambda z.\phi, b) \\
\mid & \top \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \\
\text{NLP predicate } \phi ::= & \texttt{matchKeyword}(z, K, t) \\
\mid & \texttt{hasAnswer}(z, Q) \\
\mid & \texttt{hasEntity}(z, l) \\
\mid & \top \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi
\end{array}
$$

Figure 5. DSL for WebQA. Here, c denotes a character (e.g., a delimiter like comma), and l is an entity type (e.g. Person). Also, $k \in \mathbb{Z}$, b is a boolean, $t \in [0, 1]$ is a threshold.

4 DSL for Web Question Answering

In this section, we describe our domain-specific language called WebQA for web information extraction. At a high level, this DSL combines pre-trained neural models for standard NLP tasks (i.e., question answering, entity extraction, and keyword matching) with symbolic constructs for manipulating strings and navigating the tree structure of the webpage. As shown in Figure 5, a program in this DSL takes as input a question Q, keyword(s) K, and a webpage W, and it returns a *set* of strings that collectively answer the question.

As shown in Figure 5, each WebQA program is a sequence of *guarded expressions* of the form $\psi_i \to \lambda x.e_i$ where the *guard* ψ_i locates relevant tree nodes and checks whether they satisfy some property, and the *extractor* takes as input a set of tree nodes (computed by the guard) and returns a set of strings (see Figure 6 for their types). The program returns the result of expression e_i if the corresponding guard ψ_i is true and all previous guards $\psi_1, \ldots, \psi_{i-1}$ evaluate to false. Intuitively, the guards are used to determine the webpage "schema" and locate the relevant tree nodes from which to extract information. Then, the corresponding expression e_i extracts the relevant text from those nodes.

In more detail, guards ψ in a WebQA program locate the relevant sections N of the webpage using so-called *section locators* and check whether nodes N satisfy some predicate. If they do, the located sections N are bound to variable x of the corresponding extractor expression e, and the result of evaluating e on N is returned. On the other hand, if a guard evaluates to false, then the next guard is evaluated, and this process continues until one of the guards evaluates to true. If all guards evaluate to false, the return value of the program is $\varnothing$. Next, we explain the WebQA constructs in more detail.

Pre-trained NLP Models. Our DSL contains three pre-trained neural models for extracting information from webpages. These pre-trained models are used inside predicates ϕ and include the following primitives:

p :: Question × Keywords × Webpage → Set<String>
ψ :: Bool × Set<Node>
e :: Set<String> z :: String
x :: Set<Node> n :: Node
ν :: Set<Node> φ, ϕ :: Bool

Figure 6. Types of different symbols in the WebQA grammar

- **Keyword match:** Given string z, the $\texttt{matchKeyword}(z, K, t)$ predicate evaluates to true if the semantic similarity between z and keyword k exceeds threshold $t \in [0, 1]$ for some keyword $k \in K$.
- **Question answering:** The $\texttt{hasAnswer}(z, Q)$ predicate returns true if a pre-trained neural network for textual question answering can find the answer to the given question Q in input string z.
- **Entity matching:** Given string z, $\texttt{hasEntity}(z, l)$ returns true if a neural model for entity matching decides that z contains an entity of type l (e.g., person, location).

These neural primitives draw on standard NLP modeling tools for each of their respective tasks. By using standard tools, we can exploit not only pre-trained vectors [47] and models such as BERT [19], but we can take advantage of training sets created for other tasks like question answering [49]. This design choice allows us to leverage neural components despite the lack of substantial training data.

Section locators. Our WebQA DSL includes so-called *section locator* constructs ν for identifying tree nodes from which to extract information. Section locators allow navigating the tree structure and identifying nodes that satisfy a given predicate. In particular, given a webpage W, $\texttt{getRoot}(W)$ returns the root node of the webpage, and the recursive `getChildren` and `getDescendants` constructs return respectively the children and descendant nodes satisfying a certain predicate φ. Predicates on nodes allow testing whether a given node is a leaf (`isLeaf`), whether it is a list/table element (`isElem`), or whether the text contained in that node matches NLP predicate ϕ (`matchText`). Note that the third boolean argument of `matchText` specifies whether to consider only text within that node ($b = \texttt{false}$) or whether to consider the text in the entire subtree ($b = \texttt{true}$).

Guards. As mentioned earlier, guards in our DSL are used for locating relevant sections within a webpage and testing their properties. In particular, a guard ψ uses section locators to identify relevant nodes N and then checks their properties via the `IsSingleton` and `Sat` predicates. As its name indicates, `IsSingleton` tests whether N contains a single node. Intuitively, this predicate is useful because existing textual question answering systems like `hasAnswer` are more likely to be effective if the desired information can be found within a single block of text. On the other hand, the `Sat` predicate is used to test whether *any* of the nodes $n \in N$ satisfy some neural classifier ϕ — i.e., $\texttt{Sat}(N, \lambda z.\phi)$ checks whether text z of node n satisfies ϕ for some $n \in N$.

1: **procedure** Synthesize($\mathcal{E}, Q, K$)
input: training examples $\mathcal{E}$, question Q, and keywords K.
output: all WebQA programs with optimal F_1 score.
2: $R \leftarrow \bot$; $opt \leftarrow 0$;
3: **for all** $P \in$ Partitions($\mathcal{E}$) **do**
4: $bs \leftarrow []$;
5: **for all** $\mathcal{E}_i \in P$ **do**
6: $B \leftarrow$ SynthesizeBranch($\mathcal{E}_i, P \setminus \cup_{j=1}^{i}\mathcal{E}_j, Q, K$);
7: $bs.append(B)$;
8: **if** $F_1(bs, \mathcal{E}) > opt$ **then**
9: $opt = F_1(bs, \mathcal{E})$; $R \leftarrow \{bs\}$;
10: **else if** $F_1(bs, \mathcal{E}) = opt$ **then**
11: $R \leftarrow R \cup \{bs\}$;
12: **return** R;

Figure 7. Top-level synthesis algorithm.

Extractors. The extractor constructs are used to extract text from relevant sections N of a webpage. Note that these relevant sections are determined by the corresponding guard and bound to variable x referenced in the extraction construct. In the simplest case, the `ExtractContent` function returns the string content of each node $n \in N$. The remaining constructs are recursive and allow (a) extracting substrings, (b) filtering elements from a set, and (c) splitting a string into multiple strings. In particular, `Substring`$(n, \lambda z.\phi, k)$ returns the top-k substrings satisfying neural classifier ϕ on n's contents. Similarly, `Filter`$(N, \lambda z.\phi)$ filters those nodes n whose content does not satisfy ϕ from set N. Finally, `Split`(n, c) generates multiple new substrings by splitting n's content based on the provided delimiter c (e.g., comma).

5 Optimal Neurosymbolic Synthesis

In this section, we describe our algorithm for synthesizing *all* programs that achieve optimal F_1 score on a given set of training examples. At a high level, our method is based on enumerative search but employs two ideas that allow it to scale better: First, we *decompose* the task of synthesizing extractors from that of synthesizing guards; this decomposition significantly reduces the space of programs we need to consider. Second, we exploit a certain *monotonicity property* of our DSL to prune programs that are guaranteed to be sub-optimal in terms of their F_1 score.

Our top-level synthesis algorithm is presented in Figure 7. Given a few training examples $\mathcal{E}$, a question Q, and keywords K, Synthesize returns a *set* of programs that achieve optimal F_1 score on $\mathcal{E}$. At a high level, the algorithm considers all possible ways of partitioning the training examples and synthesizes optimal programs for each partition.[4] Intuitively, each partition corresponds to a different way of assigning guards to webpages in the training set, and the overall synthesis algorithm chooses a partition that yields the best F_1 score among all partitions.

[4]Since our technique only requires a small set of labeled examples, considering all partitions of $\mathcal{E}$ is computationally tractable.

In more detail, the Synthesize procedure works as follows. It first generates all possible partitions of the training examples, and then, for each partition $P = [\mathcal{E}_1, \ldots, \mathcal{E}_n]$, it generates a set of (optimal) programs of the form:

$$\psi_1 \rightarrow \lambda x.e_1, \ldots, \psi_n \rightarrow \lambda x.e_n$$

such that examples $\mathcal{E}_i$ satisfy the i'th guard ψ_i and the corresponding extractor e_i achieves optimal F_1 score for $\mathcal{E}_i$. We represent the set of optimal programs for partition P as a list $bs = [B_1, \ldots, B_n]$, where each B_i represents an optimal set of programs for the i'th branch.

In particular, a *branch program* $b \in B_i$ is a pair (ψ, e) consisting of a guard and an extractor, and we represent a set of branch programs as a mapping B_i from guards to a set of extractors E. Thus, B_i represents all branch programs (ψ, e) satisfying the following three properties:

1. The guard ψ evaluates to true for all examples in $\mathcal{E}_i$.
2. The guard ψ evaluates to false for $\mathcal{E} \setminus (\mathcal{E}_1 \cup \ldots \cup \mathcal{E}_i)$.[5]
3. The extractor e achieves optimal F_1 score for examples $\mathcal{E}_i$.

Synthesizing branch programs. Next, we consider the SynthesizeBranch procedure (Figure 8) for generating optimal branch programs for a given set of examples. As mentioned earlier, there are two important ideas underlying this algorithm: First, we decompose the branch synthesis problem into two separate sub-problems (one for synthesizing guards, and one for synthesizing extractors). Second, we prune the search space by inferring an upper bound on the optimal F_1 score that can be achieved by partial branch programs.

In more detail, the SynthesizeBranch procedure works as follows. For a given set of positive examples $\mathcal{E}^+$ and negative examples $\mathcal{E}^-$, it first synthesizes a guard ψ that separates $\mathcal{E}^+$ from $\mathcal{E}^-$ (line 4) and then generates the set of all optimal extractors using ψ (line 8). Note that there may be multiple guards in our DSL that distinguish $\mathcal{E}^+$ from $\mathcal{E}^-$. While our algorithm considers all possible guards (loop in lines 3–12), it does so *lazily* — i.e., it only synthesizes the next guard after synthesizing optimal extractors for the previous guards. As we will see shortly, such lazy enumeration strategy is useful because it improves the pruning power of the guard synthesis algorithm.

Now, let us consider each iteration of loop in lines 3–12. First, given a guard ψ separating $\mathcal{E}^+$ and $\mathcal{E}^-$ (line 4), our technique infers an *upper bound* on the F_1 score of any branch program using ψ as its guard. In particular, we can do this because the extractors in our DSL are monotonic with respect to recall: If extractor e' appears as a sub-expression of e, then the recall that can be achieved by extractor e cannot be more than that of e'. Furthermore, since the extractor operates over the tree nodes N returned by its corresponding guard, the recall can only decrease with respect to N's contents.

[5]Since a guard is only evaluated if previous guards evaluate to false, we only require ψ to differentiate between the current set of examples and the examples that have not yet been considered.

1: **procedure** SynthesizeBranch($\mathcal{E}^+, \mathcal{E}^-, Q, K$)
input: Pos/neg examples $\mathcal{E}^+, \mathcal{E}^-$; question Q; keywords K
output: Branch programs represented as a mapping R from guards to extractors such that for each $(\psi, E) \in R$ (1) ψ classifies $\mathcal{E}^+, \mathcal{E}^-$ and (2) E achieves maximum F_1 score for $\mathcal{E}^+$.
2: $R \leftarrow \bot$; $opt \leftarrow 0$;
3: **while** true **do**
4: $\psi \leftarrow$ GetNextGuard($\mathcal{E}^+, \mathcal{E}^-, Q, K, opt$);
5: **if** $\psi = \bot$ **then break**;
6: **if** UB($\psi.\nu, \mathcal{E}^+$) $< opt$ **then continue**;
7: $\mathcal{E}' \leftarrow$ PropogateExamples($\mathcal{E}^+, \psi, Q, K$);
8: (E,F_1) $\leftarrow$ SynthesizeExtractors($\mathcal{E}', Q, K, opt$);
9: **if** $F_1 > opt$ **then**
10: $opt \leftarrow F_1$; $R \leftarrow \{(\psi, E)\}$;
11: **else if** $F_1 = opt$ **then**
12: $R[\psi] \leftarrow E$
13: **return** R

Figure 8. Algorithm for synthesizing branch programs.

Our algorithm uses this observation at line 6 of SynthesizeBranch by using the UB function for computing an upper bound on branch programs using guard ψ. In particular, let ν denote the section locator used in guard ψ. Then, we can obtain an upper bound for any branch program over ψ using the following formula:

$$\mathsf{UB}(\nu, \mathcal{E}) = \frac{2 \cdot \mathsf{Recall}(\nu, \mathcal{E})}{1 + \mathsf{Recall}(\nu, \mathcal{E})} \tag{3}$$

where $\mathsf{Recall}(\nu, \mathcal{E})$ for a section locator ν and examples $\mathcal{E}$ is defined as follows:

$$\frac{\{t \mid t \in \mathtt{ExtractContent}(\nu(W)), W \in \mathcal{E}_{\text{in}}\} \cap \{t \mid t \in \mathcal{E}_{\text{out}}\}}{\{t \mid t \in \mathcal{E}_{\text{out}}\}}$$

where t represents a token.

That is, our upper bound computation assumes maximum possible precision (i.e., 1) and maximum recall for any extractor using section locator ν. Since $\mathsf{UB}(\nu)$ gives an upper bound on the F_1 score of any branch program with guard ψ, we do not need to consider extractors for ψ if $\mathsf{UB}(\nu)$ is less than the maximum F_1 score encountered so far (line 6).

Assuming ψ is not provably sub-optimal, SynthesizeBranch proceeds to construct optimal extractors for the synthesized guard ψ (lines 7–12). To decompose extractor synthesis from guard inference, we first compute separate input-output examples for the extractor by calling PropagateExamples at line 7. In particular, this procedure executes the synthesized section locator ν on the input webpages to obtain new input-output examples $\mathcal{E}'$ for the extractor and invokes SynthesizeExtractors on $\mathcal{E}'$. Finally, if the branch programs associated with guard ψ improve upon (or yield the same) F_1 score, the result set R is updated.[6]

[6]Since branches that use guards with the same section locator have the same set of optimal extractors, the calls to SynthesizeExtractors can be memoized across different iterations within the SynthesizeBranch procedure. We omit this to simplify presentation.

1: **procedure** SynthesizeExtractors($\mathcal{E}, Q, K, opt$)
input: Examples $\mathcal{E}$;question Q; keywords K
input: Lower bound opt on F_1
output: Extractors E_o with optimal F_1 score s_o on $\mathcal{E}$.
2: $E_o \leftarrow \emptyset$; $s_o \leftarrow opt$;
3: $\mathcal{W} \leftarrow \{\mathtt{ExtractContent}(x)\}$;
4: **while** $\mathcal{W} \neq \emptyset$ **do**
5: $e \leftarrow \mathcal{W}.remove()$; $s \leftarrow F_1(e, \mathcal{E})$;
6: **if** $s > s_o$ **then** $E_o \leftarrow \{e\}$; $s_o \leftarrow s$;
7: **else if** $s = s_o$ **then** $E_o.add(e)$;
8: **for all** $e' \in$ ApplyProduction(e) **do**
9: **if** UB($e', \mathcal{E}$) $\geq s_o$ **then** $\mathcal{W}.add(e')$;
10: **return** (E_o, s_o);

Figure 9. Optimal extractor synthesis.

Extractor synthesis. Next, we describe the SynthesizeExtractors procedure (Figure 9) for finding extractors with optimal F_1 score for a given set of input-output examples. This procedure uses *bottom-up* enumeration with pruning based on F_1 scores to reduce the search space. In particular, we use bottom-up rather than top-down enumeration because doing so allows us to more easily exploit the monotonicity property of the DSL with respect to recall.

In more detail, SynthesizeExtractors maintains a worklist $\mathcal{W}$ of complete extractors; and, in each iteration, it dequeues one extractor and expands it by applying all possible grammar productions for `Substring`, `Filter`, and `Split` (line 8). A new extractor e' is added to the worklist only if $\mathsf{UB}(e', \mathcal{E})$ (i.e., F_1 score upper bound for e') is greater than or equal to the previous upper bound s_o (line 9). As described earlier, we compute an upper bound on extractors generated from e' by using 1 for precision and the recall of e' on the given set of examples. As before, this pruning strategy exploits the fact that if e_1 is a subprogram of e_2, then $\mathsf{Recall}(e_1, \mathcal{E}) \geq \mathsf{Recall}(e_2, \mathcal{E})$ for any set of examples $\mathcal{E}$.

Lazy synthesis of guards. The final missing piece of our synthesis algorithm is the GetNextGuard procedure (Figure 10) for lazy guard synthesis. In particular, this algorithm is *lazy* in the sense that it yields a single guard at a time rather than returning the set of all guards separating $\mathcal{E}^+$ from $\mathcal{E}^-$. Since the guard synthesis algorithm also prunes its search space by computing an upper bound on F_1 scores, this lazy enumeration strategy improves pruning power as the optimal F_1 score improves over time. However, despite the lazy nature of the guard synthesis algorithm, our technique is still guaranteed to return all optimal programs.

The guard synthesis algorithm (Figure 10) is similar to SynthesizeExtractors and also performs bottom-up search with pruning. In particular, it maintains a worklist $\mathcal{W}$ of section locators. In each iteration, it dequeues one of the section locators ν and generates all possible guards using ν (up to some bound). If any of the resulting guards ψ is a classifier between $\mathcal{E}^+$ and $\mathcal{E}^-$, then it is returned as the next

1: **procedure** GetNextGuard($\mathcal{E}^+, \mathcal{E}^-, Q, K, opt$)
input: Pos/neg examples $\mathcal{E}^+, \mathcal{E}^-$
input: Question Q; keywords K; lower bound opt on F_1 score
output: Next guard that classifies $\mathcal{E}^+$ and $\mathcal{E}^-$.
2: $\mathcal{W} \leftarrow \{\texttt{GetRoot}(W)\}$;
3: **while** $\mathcal{W} \neq \emptyset$ **do**
4: $v \leftarrow \mathcal{W}.remove()$;
5: **for all** $\psi \in \text{GenGuards}(v)$ **do**
6: **if** $\forall e_{in} \in \mathcal{E}^+.[\![\psi(e_{in}, Q, K)]\!] \wedge$ $\forall e_{in} \in \mathcal{E}^-.\neg[\![\psi(e_{in}, Q, K)]\!]$ **then yield** ψ;
7: **for all** $v' \in \text{ApplyProduction}(v)$ **do**
8: **if** $\text{UB}(v', \mathcal{E}^+) \geq opt$ **then** $\mathcal{W}.add(v')$;
9: **return** $\perp$;

Figure 10. Lazy enumeration of guards.

viable guard. Once the algorithm enumerates all possible classifiers using the section locator v, it then generates all possible section locators derived from v by using the productions `GetChildren` and `GetDescendants` (lines 7-8). As in the previous algorithms, GetNextGuard also computes an upper bound on F_1 score and adds a new section locator v' to the worklist if $\text{UB}(v', \mathcal{E}^+)$ is no worse than the previous optimal F_1 score opt (line 8).

Theorem 5.1. *Let $\mathcal{E}, Q, K$ be inputs to the* Synthesize *procedure and let p be a* WebQA *program. Then, the set of programs returned by* Synthesize*$(\mathcal{E}, Q, K)$ includes p if and only if, for any other* WebQA *program p', $F_1(p) \geq F_1(p')$.*

6 Program Selection via Transductive Learning

Our optimal synthesis algorithm outputs all programs that have optimal F_1 score on the labeled training data. However, not all of these programs generalize well to new inputs. In this section, we present a technique based on transductive learning [61] that selects a program that generalizes well beyond the training examples.

We describe our algorithm (summarized in Figure 11) for selecting a program that generalizes well to the test set. At a high level, our selection method must satisfy two objectives. First, it should select a program that generalizes well in terms of F_1 score. Empirically, we observe that a large fraction of optimal programs achieve good F_1 score on the test set, so a randomly chosen program has good F_1 score on average. However, we also want to minimize variance—in many cases, a sizable fraction of programs perform quite poorly, so a randomly chosen program may have poor F_1 score. Our selection method is designed to select a good program while avoiding these poorly performing programs.

The key concept underlying our approach is to use an *ensemble* of the optimal programs Π^* generated by our synthesis algorithm; that is, we aggregate predictions over a

1: **procedure** Select($\mathcal{E}, \mathcal{I}, \Pi^*$)
input: training examples $\mathcal{E}$, unlabeled input examples $\mathcal{I}$, optimal programs Π^*.
output: optimal program π^* on the transductive learning objective.
2: draw i.i.d. samples $\pi_1, ..., \pi_{\mathcal{N}} \sim \Pi^*$;
3: construct ensemble $\Pi_E = \{\pi_1, ..., \pi_{\mathcal{N}}\}$;
4: compute the output $O_j \in \Pi_E$ for π_j according to Eq. 8;
5: compute $L(\pi) = \sum_{j=1}^{\mathcal{N}} L(\pi; \mathcal{I}, O_j)$ for each $\pi \in \Pi_E$;
6: **return** $\pi^* = \arg\max_{\pi \in \Pi_E} L(\pi)$;

Figure 11. Program Selection Algorithm.

large random sample of optimal programs. Such an ensemble would address both of the above points. Ensembles of multiple models typically generalize better than the individual models since the errors made by individual models tend to average out [43]. For the same reason, they also tend to reduce the variance in performance [20]. However, directly using an ensemble instead of an individual program has a few drawbacks. First, an ensemble is significantly less interpretable than an individual model. Second, since an ensemble includes many programs, there is a large computational cost to using the ensemble if the learned model is to be used over and over again.

Thus, our algorithm first builds the ensemble and then *compresses* it into a single program by leveraging the *unlabeled* training data. In particular, it constructs an ensemble Π_E by sampling $\mathcal{N}$ optimal programs returned by the synthesis algorithm (lines 2–3). Then, it uses the ensemble to generate soft labels for the unlabeled webpages and returns the program π^* that minimizes loss $L(\pi^*)$ with respect to these soft labels. We describe our approach in more detail below. For conciseness, we give a high-level sketch of our derivations, and provide details in the Appendix.

Transductive learning objective. Given (i) labeled examples $\mathcal{E}$, (ii) unlabeled inputs $\mathcal{I}$, and (iii) optimal programs Π^* returned by Synthesize, our algorithm finds a program $\pi \in \Pi^*$ that minimizes the following objective:

$$\tilde{L}(\pi; \mathcal{E}, \mathcal{I}) = \mathbb{E}_{p(O|\mathcal{I}, \mathcal{E})}[L(\pi; \mathcal{I}, O)]. \quad (4)$$

The expression $L(\pi; \mathcal{I}, O)$ is a loss function we wish to minimize in a standard supervised learning fashion using $(\mathcal{I}, O)$ as the training dataset—e.g., we could take it to be the negative F_1 score. However, the difficulty is that we do not know the labels O for the inputs $\mathcal{I}$. Thus, we take the expectation with respect to a distribution $p(O \mid \mathcal{I}, \mathcal{E})$ that leverages information from the labeled examples. As described in detail below, this distribution is constructed by using an ensemble of programs synthesized based on $\mathcal{E}$ to assign soft "pseudo-labels" to $\mathcal{I}$.

Generating labels via program ensembling. Next, we describe how to construct the distribution $p(O \mid \mathcal{I}, \mathcal{E})$. We do this in two steps: first, by defining a distribution $p(\pi' \mid$

$\mathcal{E}$) over (optimal) programs conditioned on the input data, then using the fact that these programs are deterministic to compute $p(O \mid \pi', \mathcal{I})$.

We define our distribution $p(\pi' \mid \mathcal{E})$ to assign probability mass *only* to programs in Π^*, those that best satisfy the given examples $\mathcal{E}$.[7] Ideally, we could use the uniform distribution over optimal programs Π^*. However, a key difficulty is that summing over all programs $\pi \in \Pi^*$ is intractable since the cardinality of Π^* is too large in practice. Thus, we instead approximate this distribution by constructing an ensemble $\Pi_E = \{\pi_1, ..., \pi_{\mathcal{N}}\}$, where $\pi_i \sim \text{Uniform}(\Pi^*)$ are i.i.d. samples and where $\mathcal{N} \in \mathbb{N}$ is a hyperparameter, and then using

$$p(\pi \mid \mathcal{E}) = \frac{1(\pi \in \Pi_E)}{\mathcal{N}} \tag{5}$$

Finally, given this distribution, we have

$$\begin{aligned} p(O \mid \mathcal{I}, \mathcal{E}) &= \sum_{\pi' \in \Pi} p(\pi' \mid \mathcal{E}) \cdot p(O \mid \pi', \mathcal{I}) \qquad (6) \\ &= \sum_{\pi' \in \Pi} p(\pi' \mid \mathcal{E}) \cdot \prod_{k=1}^{K} 1(o_k = \pi'(i_k)) \end{aligned}$$

where the second step follows from the fact that our programs are deterministic and each place probability 1 over a single output.

Program selection. Finally, our algorithm aims to select

$$\pi^* = \arg\min_{\pi \in \Pi_E} \tilde{L}(\pi; \mathcal{E}, \mathcal{I}). \tag{7}$$

i.e., the program that minimizes the loss with respect to the ensemble. We now describe how to evaluate $\tilde{L}(\pi; \mathcal{E}, \mathcal{I})$.

First, we precompute the possible outputs

$$O_j = (\pi_j(i_1), ..., \pi_j(i_K)) \qquad (\forall \pi_j \in \Pi_E), \tag{8}$$

in which case we have

$$p(O \mid \mathcal{I}, \mathcal{E}) = \frac{1}{\mathcal{N}} \sum_{\pi' \in \Pi_E} \prod_{k=1}^{K} 1(o_k = \pi'(i_k)) = \frac{1}{\mathcal{N}} \sum_{j=1}^{\mathcal{N}} 1(O = O_j).$$

In other words, when evaluating $p(O \mid \mathcal{I}, \mathcal{E})$, we only need to account for outputs O_j according to programs $\pi_j \in \Pi_E$. Thus, we have:

$$\tilde{L}(\pi; \mathcal{E}, \mathcal{I}) = \sum_{O} p(O \mid \mathcal{I}, \mathcal{E}) \cdot L(\pi; \mathcal{I}, O) = \frac{1}{\mathcal{N}} \sum_{j=1}^{\mathcal{N}} L(\pi; \mathcal{I}, O_j). \tag{9}$$

Substituting into Eq. 7, our algorithm selects the program

$$\pi^* = \arg\min_{\pi \in \Pi_E} \sum_{j=1}^{\mathcal{N}} L(\pi; \mathcal{I}, O_j), \tag{10}$$

which is equivalent to Eq. 7 since $\mathcal{N}$ is a positive constant.

7 Implementation

In this section, we provide implementation details about different components of WebQA.

[7] We note that other choices are possible (e.g., prioritizing smaller programs, including some probability on erroneous programs, etc.); we found this choice to work well empirically.

Parsing. As explained in Section 3, WebQA represents each webpage as a tree that captures relationships between text elements on the renderd version of the webpage. Thus, WebQA first parses a given HTML document into our internal representation. To do this, we first extract the DOM tree representation using the *BeautifulSoup4* HTML parser and remove unnecessary elements such as images and scripts. Then, when converting to our tree representation, we follow the standard HTML header hierarchy. In particular, the H1 header corresponds to the root node, and H_{i+1} headers are represented as children nodes of the H_i headers.

Interactive labeling. Rather than asking users to directly provide labeled webpages, WebQA actually interacts with users and suggests webpages to label. The goal here is to minimize the number of user annotations while ensuring that the labels achieve good coverage of different schemas in the test set. To do so, WebQA clusters webpages based on various features, including which section locator constructs in our DSL yield non-empty answers, the type of entities contained in the extracted sections, the layout of extracted sections etc. We then identify webpages that are similar to and different from the webpages labeled so far and ask the user to label these additional webpages. In practice, we restrict the number of user queries to at most five.

Neural modules in the DSL. Our tool leverages several existing natural language processing frameworks and models to implement the neural modules. For QA-related constructs, we use the BERT QA system [19] as the underlying model. Specifically, we use the version that has been fine-tuned on the SQUAD dataset[8][49]. We use Sentence-BERT [52] to generate sentence embeddings for keyword similarity, and we employ Spacy[9][31] for named entity extraction and sentence segmentation. Since the keyword matching module requires a real-valued threshold $t \in [0, 1]$, our synthesis algorithm discretizes it using a step size of 0.05.

Transductive learning loss. Recall that our transductive program selection technique from Section 6 is parametrized over a loss function $L(\pi; \mathcal{I}, O)$. In our implementation, we take our loss function to be the Hamming distance between the sets of words extracted by each program. In particular, our loss function is: $L(\pi; \mathcal{I}, O) = \texttt{Hamming}(\pi(\mathcal{I}), O)$.

Hyperparameters. The WebQA system has certain hyperparameters that control the maximum depth of synthesized programs. By default, the hyper-parameter for guard depth is set to 7 and the one for extractor depth is set to be 5. There is also another hyper-parameter (with a default value of 1000) that controls the number of programs used to construct an ensemble during transductive learning.

[8] The link to the model: https://huggingface.co/bert-large-uncased-whole-word-masking-finetuned-squad.

[9] https://spacy.io/. Specifically, we use the "en_core_web_md" model.

Table 1. Description of the tasks used in evaluation.

Domain	Description
Faculty	Extract current PhD students Extract conference publications at PLDI Extract courses they have taught Extract those papers that received a Best Paper Award Extract program committees they have served on Extract conference papers they published in 2012 Extract co-authors among all papers published at PLDI Extract formerly advised students
Conference	Extract program committee members Extract program chairs Extract the topics of interest Extract the paper submission deadlines Extract whether the conference is single-blind or double-blind Extract institutions PC members are from
Class	Extract the name of instructors Extract the time of the lectures Extract the name of teaching assistants Extract the date of the exams Extract information about textbooks Extract information on how grades are assigned
Clinic	Extract the doctors or providers Extract the provided services Extract the types of treatments they specialize in Extract the accepted insurances Extract the locations

8 Evaluation

In this section, we describe a series of experiments that are designed to answer the following research questions:

- **RQ1.** How does WebQA's performance compare against other question answering and information extraction tools?
- **RQ2.** How important are the synthesis techniques proposed in Section 5?
- **RQ3.** Is the program selection technique based on transductive learning (Section 6) useful in practice?

Benchmarks. To answer these questions, we evaluate WebQA on 25 different tasks across four different domains, namely faculty profiles, computer science conferences, university courses, and clinic websites. For each domain, we collect approximately 40 webpages and evaluate the performance of each tool in terms of F_1 score, precision, and recall. For each task, out of around 40 webpages, around 5 of them are used for training (i.e. synthesis) and the remaining is the test set. Table 1 describes the 25 tasks used in our evaluation.

Experimental Setup. All of our experiments are conducted on a machine with Intel Xeon(R) W-3275 2.50 GHz CPU and 16GB of physical memory, running the Ubuntu 18.04 operating system with a NVIDIA Quadro RTX8000 GPU.

8.1 Comparison with Other Tools

To answer our first research question, we compare WebQA against the following baselines:

- BERTQA [19]: This is a state-of-the-art textual question answering system that takes as input an entire webpage and a question and outputs the answer. [10].

[10]We also tried fine-tuning this model using the labels in our training examples; however, we do not report results for the fine-tuned model since its result is actually worse compared to [19].

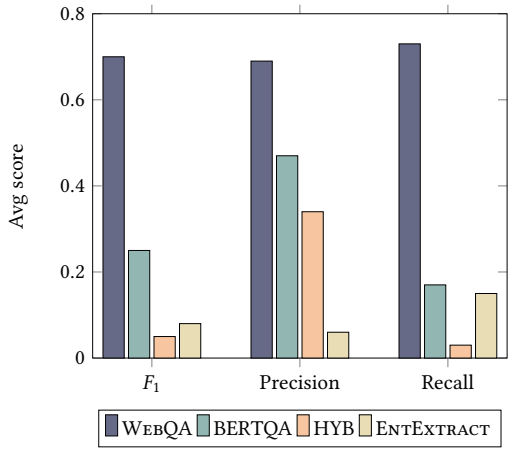

Figure 12. Comparison between WebQA and other tools

- HYB [50]: This is a programming-by-example system that takes a set of webpages as input and synthesizes XPath programs for data extraction.
- EntExtract [44]: This is a zero-shot entity extraction tool for webpages using a natural language query as input.

Note that these baselines do not address *exactly* the same problem addressed by WebQA in that they take fewer inputs. Thus, while our comparison is not completely apples-to-apples, these systems are the closest ones to WebQA for performing a comparison.

Our main results are summarized in Figure 12, and Table 2 shows a more detailed breakdown of results across our four domains. As we can see from Figure 12, WebQA outperforms all three baselines in terms of average F_1 score, precision, and recall, and, accoring to Table 2, these results hold across all four domains. Among the three other tools, BertQA has the best performance; however, it has significantly worse recall and F_1 score compared to WebQA.

Failure analysis for the baselines. We briefly explain why the baseline systems perform poorly in our evaluation. As mentioned earlier, a textual QA system like BERTQA fails to take advantage of the inherent structure in webpages and performs particularly poorly on tasks that require extracting multiple different spans from the input webpage. On the other hand, EntExtract does leverage the tree structure of the webpage but we found that it often returns irrelevant answers (e.g., publications instead of students). We believe this is because EntExtract generates extraction predicates based on XPath queries, but most of our tasks are difficult to solve using simple XPath programs. Finally, HYB tries to synthesize programs that exactly match the provided labels (i.e., perfect F_1 score); however, since such programs do not exist for many tasks, synthesis fails in several cases.

***Failure analysis for* WebQA.** There are two tasks on which WebQA does not significantly outperform the BERTQA

Table 2. Evaluation results for each baseline per domain. P stands for Precision and R means Recall.

Domain	WebQA			BERTQA			HYB			EntExtract		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
Faculty	0.72	0.80	0.75	0.44	0.08	0.18	0.48	0.02	0.04	0.02	0.14	0.04
Conference	0.71	0.69	0.70	0.58	0.31	0.32	0.26	0.02	0.03	0.07	0.20	0.09
Class	0.63	0.77	0.68	0.55	0.26	0.31	0.18	0.04	0.04	0.04	0.09	0.05
Clinic	0.71	0.62	0.66	0.31	0.02	0.04	0.42	0.06	0.09	0.14	0.20	0.16

Table 3. Results of the ablation study. This table shows the average training time and the average speedup that WebQA achieves compared to the other two techniques.

Technique	Avg time (s)	Avg Speedup
WebQA	419	-
WebQA-NoPrune	1351	3.6
WebQA-NoDecomp	931	2.4

baseline. One of these tasks is extracting conference submission deadlines, and the other one is determining whether a conference is double-blind or not. For these two tasks, the program synthesized by WebQA essentially returns the output of the QA model; hence, it does not outperform BERTQA.

8.2 Evaluation of the Synthesis Engine

In this section, we describe an ablation study that quantifies the impact of the proposed synthesis techniques from Section 5. In particular, recall that our synthesis algorithm incorporates two key ideas–decomposition and pruning based on F_1 score. To evaluate the relative importance of these ideas, we consider the following two ablations of WebQA:

- WebQA-NoDecomp: This is a variant of WebQA that synthesizes guards and extractors jointly. In other words, it does not decompose the synthesis problem into two separate guard synthesis and extractor synthesis sub-tasks.
- WebQA-NoPrune: This variant does not compute an upper bound on the F_1 score of partial programs. Thus, it is unable to prune partial programs from the search space.

The results of this ablation study are presented in Figure 3. Here, the first column shows average synthesis time (in seconds) for all three variants, and the second column shows the average speedup of WebQA over its two ablations. As we can see, both decomposition and F_1-based pruning play a significant role in improving synthesis time. In particular, pruning improves synthesis time by a factor of 3.6 and decomposition improves it by a factor of 2.4. Note that we do not report F_1 scores in Table 3 since all variants synthesize the same programs but differ in how long they take to do so.

8.3 Effectiveness of the Transductive Learning

In this section, we investigate the usefulness of the transductive learning technique from Section 6 by comparing against two simpler baselines:

Table 4. Evaluation of transductive learning. This table shows the % of improvement in F_1 and the reduction in variance of WebQA compared to the other two techniques.

Technique	% Improvement in F_1	Reduction in Variance
Random	6.0%	1550×
Shortest	6.3%	1570×

- Random: This baseline chooses uniformly at random one of the optimal programs for the training examples.
- Shortest: This baseline chooses uniformly at random one of the *smallest* programs (in terms of AST size) that optimize F_1 score on the training examples.

Recall that the transductive learning technique from Section 6 is both intended to reduce variance and produce better-quality results on the test set. Thus, we compared WebQA against two baselines in terms of the following two metrics[11].

- Mean: We report percentage improvement of the transductive learning technique in terms of average F_1 score over the two baselines.
- Variance: We also report the average reduction in variance.

As we can see from Table 4, the transductive learning technique dramatically reduces variance and modestly improves average F_1 score. Thus, by using our proposed transductive learning technique, WebQA achieves much more stable performance (in terms of the quality of the synthesized programs) compared to these other approaches.

Remark. Appendix B presents additional ablation studies evaluating the impact of the different input modalities as well as the number of training examples.

9 Related Work

Program synthesis for webpages. There is a large body of prior research on learning extraction rules from HTML documents. In data mining, this problem is known as *wrapper induction* [36], and there is a wide spectrum of proposed solutions [5, 10, 18, 27, 32, 37, 42, 50]. For instance, Vertex [27] uses an *apriori* style algorithm [1] to learn XPath-based rules from human annotated sample pages, and [5] also learns XPath-compatible wrappers. While these techniques can extract HTML elements, they cannot perform finer-grained string processing *inside* HTML elements. In

[11] In the experiment, these two metrics are computed based on 20 runs.

contrast, FlashExtract [37] can perform some text manipulation inside HTML elements[12]; however, unlike our approach, it does not use neural constructs, making it difficult to apply this technique to structurally heterogeneous websites. Recent work [35] targets data extraction from heterogeneous sources by combining ideas from program synthesis and machine learning. However, this approach requires significant number of *manually* labeled samples since it relies on first training an ML model. In contrast, our technique uses pre-trained models and a small amount of training data.

Recent work by Raza and Gulwani [50] proposes HYB, a synthesis-driven web data extraction technique that is now deployed in the Microsoft Power BI product. This technique is also based on program synthesis and uses a combination of top-down and bottom-up search. As shown in our evaluation, WebQA performs significantly better than their approach; we believe this is due to the fact that our method is based on a neurosymbolic DSL.

There is also a line of work [8, 11–13] for learning web automation macros using a programming-by-demonstration approach. These techniques perform scraper synthesis by recording user interactions with a few webpages and then generalize these interactions into a programmatic webpage scraper. In contrast to our approach, these techniques target structurally similar pages (e.g., different Amazon products) and use a different type of input, namely demonstrations.

Information extraction from text. Much of the IE literature (e.g., relation extraction [41]) is confined to a given database schema. Among IE frameworks, "slot-filling" approaches [24, 46] typically require at least medium-sized training sets to work on specific schemas, and most few-shot approaches [7, 30] use a pre-trained model and perform further training on text that expresses the desired relations in a similar fashion to the target domain. In contrast, open information extraction techniques[23] aim to retrieve data in an ontology-free way that can theoretically be used for downstream tasks like question answering [17]. However, this information is extracted primarily from textual relations rather than structured formatting; even graph-based approaches use graphs over textual relations only [48]. Therefore, these approaches do not work well in settings (such as ours) that involve a combination of tree structure and free-form text.

Information extraction from semi-structured data. Recent work has begun to tackle the problem of semi-structured data, particularly interactions between tables and natural language. Prior work looks at extracting lists from the web [44], answering questions from tables [45], verifying facts from tables [15], or generally extracting information from tables [58]. However, much of this work assumes access to large training sets or relies heavily on the structure of tables.

Two recent efforts have tackled the problem of IE from semi-structured data in a setting similar to ours [38, 39]. However, to use these techniques in our setting, we would have to first run their tools to extract a knowledge base, and then interpret our questions into some kind of semantic representation that we can execute against the extracted knowledge base. In contrast to such an approach, our work instead *dynamically* learns the relation to extract from the question, specified keywords, and examples.

Question answering. Beyond the table-based question answering approaches listed above, there is little work on question answering over text that can be directly applied to our setting. BERT-based [19] models applied to datasets such as SQuAD [49] only work well on input that is completely unstructured text. While there are some recent efforts on question answering with more programmatic structures [29] for tasks like DROP [21], these systems are highly specialized to applications like answering numerical questions.

Quantitative program synthesis. There has also been recent work on optimal program synthesis with respect to a quantitative objective. For example, Bornholt et al. [9] introduce a general framework for optimal program synthesis where the search space is represented by a set of sketches. Their technique uses the objective function together with a gradient function to direct the search. In contrast to Bornholt et al. [9], our work specifically targets the web question answering domain, uses a neurosymbolic DSL, and employs an objective function that is based on program semantics. While Bornholt et al. [9], in principle, also support semantic objective functions, they require the objective function to be reducible to a decidable theory, which does not hold in our case due to the use of neural primitives. QuaSi [33] also considers the problem of synthesis with quantitative objectives, but it requires the objectives to be syntactic. Other existing synthesis techniques [28, 55, 56], are mostly *qualitative*, although they implicitly use a ranking function as an inductive bias to help with generalization. However, such ranking functions are quite restricted and mostly syntactic (e.g., based on program size).

Neurosymbolic DSLs. There has been recent interest in neurosymbolic DSLs that include both logical and neural components. For instance, neural module networks [3, 4] dynamically compose DNNs for tasks like predicting object attributes in images [40] or identifying numbers in text [29]. However, these techniques use purely neural components (even for operations like filtering and counting), which significantly increases sample complexity. Recent approaches have trained combinations of neural and logical components by backpropagating through such programs [25, 53, 54]. There has also been work on synthesizing neurosymbolic programs

[12]We were not able to experimentally compare against FlashExtract because their released implementation in Prose does not support text manipulation in HTML elements.

to represent structure in images [22, 59] and reinforcement learning policies [2, 34]. Overall, existing approaches largely focus on simultaneously learning the program structure and the DNN parameters. Hence, they are limited to very simple programs and neural components, as they need to optimize neural network parameters using backpropagation. In contrast, our work is designed to incorporate state-of-the-art DNNs such as BERT, which take significant time to train. In addition, we search over tens of thousands of programs by relying on pretrained DNN models and by developing novel deduction techniques for optimal synthesis.

Multi-modal program synthesis. There has been growing interest in program synthesis from multiple modalities of specifications. For instance, several works have used a combination of natural language and input-output examples to synthesize regular expressions, data wrangling and string manipulation programs, SQL queries, and temporal logic formulas [6, 14, 16, 26, 51]. Our technique can also be viewed as an instance of multi-modal synthesis that is based on a neurosymbolic programming language.

10 Conclusion

We have presented WebQA, a new synthesis-powered system for extracting information from webpages. We have evaluated WebQA on 25 different tasks spanning four different domains and 160 different webpages and show that WebQA significantly outperforms competing approaches in terms of F_1 score, precision, and recall.

Acknowledgments

We thank our shepherd Uri Alon as well as our anonymous reviewers and members of the UToPiA group for their helpful feedback. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1811865, Grant No. CCF-1762299 and Grant No. CCF-1918889.

A Detailed Derivation of Section 6

In this section, we describe in more detail how we derive Eq. 9 from Eq. 4, including the key step Eq. 6.

Assumptions. We assume the standard probabilistic model from the semi-supervised learning literature [60]:

$$p(i, o, \pi) = p(o \mid \pi, i) \cdot p(\pi) \cdot p(i),$$

where i is an input, o is an output, and π is a program. In addition, we assume that

$$p(o \mid \pi, i) = 1(o = \pi'(i))$$
$$p(\pi) = |\Pi|^{-1},$$

where Π is the space of all possible programs (which is finite since we consider programs of bounded depth). In other words, we assume $p(o \mid \pi, i)$ is only non-zero when o is the output of π. Next, we note that $p(i)$ is the data distribution, so we do not need to model it. In addition, we also assume that two different examples (i, o) and (i', o') are conditionally independent given π—i.e.,

$$p(i, o, i', o', \pi) = p(o \mid \pi, i) \cdot p(o' \mid \pi, i') \cdot p(\pi).$$

Finally, we let Π^* denote the set of programs that are correct for all examples $(i', o') \in \mathcal{E}$—i.e.,

$$\Pi^* = \{\pi \in \Pi \mid \forall (i', o') \in \mathcal{E} \,.\, o' = \pi(i')\}.$$

In practice Π^* may be empty (i.e., if there are no programs that satisfy all the given examples $(i', o') \in \mathcal{E}$), so we approximate it using the set of programs that achieve optimal loss (e.g., according to the F_1 score). This set might be very large, so we additionally approximate it using samples Π_E. This approximation is implicitly used in Section 6.

Theoretical analysis. We show the following result:

Theorem A.1. *Letting*

$$\tilde{L}(\pi; \mathcal{E}, \mathcal{I}) = \sum_{O} p(O \mid \mathcal{I}, \mathcal{E}) \cdot L(\pi; \mathcal{I}, O),$$

then

$$\tilde{L}(\pi; \mathcal{E}, \mathcal{I}) = \frac{1}{\mathcal{N}} \sum_{j=1}^{\mathcal{N}} L(\pi; \mathcal{I}, O_j), \tag{11}$$

where $\mathcal{N} = |\Pi^*|$, *and where*

$$O_j = (\pi_j(i_1), ..., \pi_j(i_K)) \qquad (\forall \pi_j \in \Pi^*).$$

We note that Eq. 11 is identical to Eq. 9, except in Eq. 9 we have taken Π^* to be the set of programs with optimal F_1 score on $\mathcal{E}$, and have furthermore approximated this set using samples Π_E from Π^*.

Proof. First, by our conditional independence assumption, given program π, unlabeled input examples $\mathcal{I}$, candidate output labels O, and labeled examples $\mathcal{E}$, we have

$$\begin{aligned} &p(\mathcal{I}, O, \mathcal{E}, \pi) \\ &= p(O, \pi \mid \mathcal{I}) \cdot p(\mathcal{I}) \cdot p(\mathcal{E} \mid \pi) \cdot p(\pi) \\ &= \left(\prod_{k=1}^{K} p(o_k \mid \pi, i_k) \cdot p(i_k) \right) \cdot \left(\prod_{h=1}^{H} p(o'_h \mid \pi, i'_h) \cdot p(i'_h) \right) \cdot p(\pi), \end{aligned}$$

where $\mathcal{I} = (i_1, ..., i_K)$, $O = (o_1, ..., o_K)$, and $\mathcal{E} = (\mathcal{I}', O')$, and where $\mathcal{I}' = (i'_1, ..., i'_H)$, and $O' = (o'_1, ..., o'_H)$. In other words, $\mathcal{E}$ is conditionally independent of $(\mathcal{I}, O)$ given π.

Now, we proceed with our proof. First, by the law of total probability, we have

$$p(O \mid \mathcal{I}, \mathcal{E}) = \sum_{\pi' \in \Pi} p(\pi' \mid \mathcal{I}, \mathcal{E}) \cdot p(O \mid \pi', \mathcal{I}, \mathcal{E}). \tag{12}$$

To simplify Eq. 12, we show that $p(O \mid \mathcal{I}, \mathcal{E}, \pi') = p(O \mid \mathcal{I}, \pi')$, and that $p(\pi' \mid \mathcal{I}, \mathcal{E}) = p(\pi' \mid \mathcal{E})$. First, to show the former, note that

$$\begin{aligned} p(\pi' \mid \mathcal{I}, \mathcal{E}) &= \frac{p(\mathcal{I}, \mathcal{E} \mid \pi') \cdot p(\pi')}{p(\mathcal{I}, \mathcal{E})} = \frac{p(\mathcal{I}) \cdot p(\mathcal{E} \mid \pi') \cdot p(\pi')}{p(\mathcal{I}) \cdot p(\mathcal{E})} \\ &= \frac{p(\mathcal{E} \mid \pi') \cdot p(\pi')}{p(\mathcal{E})} = p(\pi' \mid \mathcal{E}). \end{aligned}$$

Similarly, to show the latter, we have

$$p(O \mid \mathcal{I}, \mathcal{E}, \pi') = \frac{p(O, \mathcal{E}, \mathcal{I}, \pi')}{p(\mathcal{I}, \mathcal{E}, \pi')} = \frac{p(O \mid \mathcal{I}, \pi') \cdot p(\mathcal{I}) \cdot p(\mathcal{E}, \pi')}{p(\mathcal{I}) \cdot p(\mathcal{E}, \pi')} = p(O \mid \mathcal{I}, \pi').$$

Thus, plugging into Eq. 12, we have

$$p(O \mid \mathcal{I}, \mathcal{E}) = \sum_{\pi' \in \Pi} p(\pi' \mid \mathcal{E}) \cdot p(O \mid \pi', \mathcal{I}).$$

Note that this equation is identical to Eq. 6. Next, by definition of $p(o \mid \pi, i)$, we have

$$p(O \mid \mathcal{I}, \pi') = \prod_{k=1}^{K} 1(o_k = \pi'(i_k)),$$

so it follows that

$$p(O \mid \mathcal{I}, \mathcal{E}) = \sum_{\pi' \in \Pi} p(\pi' \mid \mathcal{E}) \cdot \prod_{k=1}^{K} 1(o_k = \pi'(i_k)). \quad (13)$$

It remains to compute $p(\pi' \mid \mathcal{E})$. To this end, we have

$$p(\pi' \mid \mathcal{E}) = \frac{p(\mathcal{I}', O', \pi')}{p(\mathcal{E})} = \frac{p(O' \mid \pi', \mathcal{I}') \cdot p(\pi') \cdot p(\mathcal{I})}{p(\mathcal{E})} = \frac{\left(\prod_{h=1}^{H} 1(o'_h = \pi'(i'_h))\right) \cdot p(\mathcal{I})}{|\Pi| \cdot p(\mathcal{E})} = \frac{1(\pi' \in \Pi^*) \cdot p(\mathcal{I})}{|\Pi^*| \cdot |\Pi| \cdot p(\mathcal{E})}.$$

Thus, letting $\mathcal{N} = |\Pi| \cdot |\Pi^*| \cdot p(\mathcal{E})/p(\mathcal{I})$, we have

$$p(\pi' \mid \mathcal{E}) = \frac{1(\pi' \in \Pi^*)}{\mathcal{N}}. \quad (14)$$

Note that since $\sum_{\pi' \in \Pi} p(\pi' \mid \mathcal{E}) = 1$, we must have $\mathcal{N} = |\Pi^*|$. Plugging Eq. 14 into Eq. 13, we have

$$p(O \mid \mathcal{I}, \mathcal{E}) = \frac{1}{\mathcal{N}} \sum_{\pi' \in \Pi^*} \prod_{k=1}^{K} 1(o_k = \pi'(i_k)).$$

The remaining steps follow Section 6. In particular, by the definition of O_j, we have

$$p(O \mid \mathcal{I}, \mathcal{E}) = \frac{1}{\mathcal{N}} \sum_{j=1}^{\mathcal{N}} 1(O = O_j),$$

from which it follows that

$$\tilde{L}(\pi; \mathcal{E}, \mathcal{I}) = \sum_{O} p(O \mid \mathcal{I}, \mathcal{E}) \cdot L(\pi; \mathcal{I}, O) = \frac{1}{\mathcal{N}} \sum_{j=1}^{\mathcal{N}} L(\pi; \mathcal{I}, O_j),$$

as claimed. □

B Additional ablation studies

To help readers better understand the design choices behind WebQA, we present additional ablation studies evaluating the impact of the different input modalities used by WebQA as well as its sensitivity to the number of labeled webpages.

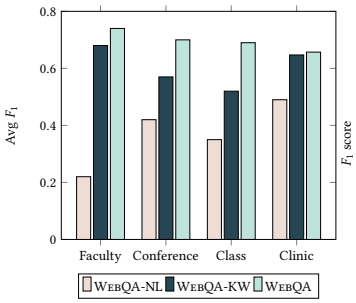

Figure 13. Comparison between WebQA and its variants

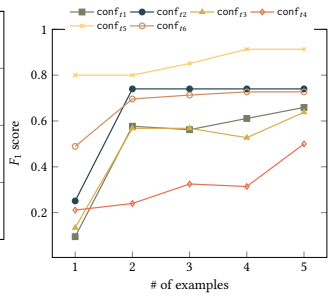

Figure 14. F_1 score achieved in each task of the Conference domains with respect to the number of labeled examples.

B.1 Evaluation on the types of input

Recall that WebQA takes two types of queries as input: a question and a set of keywords. In this section, we evaluate the impact of these two types of inputs on the end-to-end performance of the tool. Specifically, Figure 13 shows the average F_1 score for each evaluation domain for the following two variants of WebQA:

- WebQA-NL: This variant only uses the question but not the keywords.
- WebQA-KW: This variant only uses the keywords but not the question.

As we can see from Figure 13, the system works the best when both modalities of inputs are utilized. We also performed 1-tailed t-tests to check whether the differences in performance are significant and obtained p-values less than 0.01 in the comparison to the two variants. Thus, these results provide evidence that using a combination of questions and keywords as inputs leads to more accurate results.

B.2 Evaluation on the number of labeled webpages

In this section, we evaluate WebQA's sensitivity to the number of labeled examples. For this evaluation, we focus on all 6 tasks in the conference domain and vary the number of training examples from one to five. Specifically, we obtain these examples by removing a subset of the labeled webpages used in our evaluation from Section 8.

Our results are presented in Figure 14. This graph shows the F_1 score (y-axis) with respect to the number of labeled examples (x-axis). As shown in Figure 14, while performance generally gets worse as we reduce the number of examples, sensitivity to the number of examples varies from task to task. For example, for the $\mathtt{conf}_{t5}$ task, WebQA is able to synthesize programs that achieve high F_1 with only a single labeled example, whereas F_1 score drops significantly for $\mathtt{conf}_{t4}$ if we remove even one of the examples.

References

[1] Rakesh Agarwal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference.*

487–499.

[2] Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. 2020. Neurosymbolic Reinforcement Learning with Formally Verified Exploration. In *NeurIPS*.

[3] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Learning to compose neural networks for question answering. In *NAACL*.

[4] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 39–48.

[5] Tobias Anton. 2005. XPath-Wrapper Induction by generalizing tree traversal patterns. In *Lernen, Wissensentdeckung und Adaptivitt (LWA) 2005, GI Workshops, Saarbrcken*. 126–133.

[6] Christopher Baik, Zhongjun Jin, Michael Cafarella, and H. V. Jagadish. 2020. Duoquest: A Dual-Specification System for Expressive SQL Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2319–2329. https://doi.org/10.1145/3318464.3389776

[7] Livio Baldini Soares, Nicholas FitzGerald, Jeffrey Ling, and Tom Kwiatkowski. 2019. Matching the Blanks: Distributional Similarity for Relation Learning.

[8] Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: web automation by demonstration. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 748–764.

[9] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches *(POPL)*. ACM, 775–788.

[10] Chia-Hui Chang and Shao-Chen Lui. 2001. IEPAD: information extraction based on pattern discovery. In *Proceedings of the 10th international conference on World Wide Web*. 681–688.

[11] Sarah Chasins, Shaon Barman, Rastislav Bodik, and Sumit Gulwani. 2015. Browser record and replay as a building block for end-user web automation tools. In *Proceedings of the 24th International Conference on World Wide Web*. 179–182.

[12] Sarah Chasins and Rastislav Bodik. 2017. Skip blocks: reusing execution history to accelerate web scripts. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

[13] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 963–975.

[14] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

[15] Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2020. TabFact: A Large-scale Dataset for Table-based Fact Verification. In *International Conference on Learning Representations (ICLR)*. Addis Ababa, Ethiopia.

[16] Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal Multi-Layer Specification Synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 602–612. https://doi.org/10.1145/3338906.3338951

[17] Eunsol Choi, Tom Kwiatkowski, and Luke Zettlemoyer. 2015. Scalable Semantic Parsing with Partial Ontologies. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, 1311–1320. https://doi.org/10.3115/v1/P15-1127

[18] Valter Crescenzi, Giansalvatore Mecca, Paolo Merialdo, et al. 2001. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, Vol. 1. 109–118.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[20] Thomas G Dietterich. 2000. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*. Springer, 1–15.

[21] Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. DROP: A Reading Comprehension Benchmark Requiring Discrete Reasoning Over Paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 2368–2378. https://doi.org/10.18653/v1/N19-1246

[22] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to infer graphics programs from hand-drawn images. In *Advances in neural information processing systems*. 6059–6068.

[23] Oren Etzioni, Michele Banko, Stephen Soderland, and Daniel S. Weld. 2008. Open Information Extraction from the Web. *Commun. ACM* 51, 12 (Dec. 2008), 68–74. https://doi.org/10.1145/1409360.1409378

[24] Dayne Freitag. 2000. Machine Learning for Information Extraction in Informal Domains. *Mach. Learn.* 39, 2–3 (May 2000), 169–202.

[25] Alexander L Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. 2017. Differentiable programs with neural libraries. In *International Conference on Machine Learning*. 1213–1222.

[26] Ivan Gavran, Eva Darulova, and Rupak Majumdar. 2020. Interactive Synthesis of Temporal Specifications from Examples and Natural Language. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 201 (Nov. 2020), 26 pages. https://doi.org/10.1145/3428269

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

[28] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 317–330.

[29] Nitish Gupta, Kevin Lin, Dan Roth, Sameer Singh, and Matt Gardner. 2020. Neural module networks for reasoning over text. In *ICLR*.

[30] Xu Han, Hao Zhu, Pengfei Yu, Ziyun Wang, Yuan Yao, Zhiyuan Liu, and Maosong Sun. 2018. FewRel: A Large-Scale Supervised Few-Shot Relation Classification Dataset with State-of-the-Art Evaluation.

[31] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. spaCy: Industrial-strength Natural Language Processing in Python. https://doi.org/10.5281/zenodo.1212303

[32] 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.

[33] Qinheping Hu and Loris D'Antoni. 2018. Syntax-guided synthesis with quantitative syntactic objectives. In *International Conference on Computer Aided Verification*. Springer, 386–403.

[34] Jeevana Priya Inala, Yichen Yang, James Paulos, Yewen Pu, Osbert Bastani, Vijay Kumar, Martin Rinard, and Armando Solar-Lezama. 2020. Neurosymbolic Transformers for Multi-Agent Communication. In *NeurIPS*.

[35] Arun Iyer, Manohar Jonnalagedda, Suresh Parthasarathy, Arjun Radhakrishna, and Sriram K Rajamani. 2019. Synthesis and machine learning for heterogeneous extraction. In *Proceedings of the 40th ACM*

SIGPLAN Conference on Programming Language Design and Implementation. 301–315.

[36] Nicholas Kushmerick, Daniel S Weld, and Robert Doorenbos. 1997. *Wrapper induction for information extraction*. University of Washington Washington.

[37] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples *(PLDI)*. ACM, 542–553.

[38] Bill Yuchen Lin, Ying Sheng, Nguyen Vo, and Sandeep Tata. 2020. FreeDOM: A Transferable Neural Architecture for Structured Information Extraction on Web Documents. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery Data Mining* (Aug 2020). https://doi.org/10.1145/3394486.3403153

[39] Colin Lockard, Prashant Shiralkar, Xin Luna Dong, and Hannaneh Hajishirzi. 2020. ZeroShotCeres: Zero-Shot Relation Extraction from Semi-Structured Webpages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 8105–8117. https://doi.org/10.18653/v1/2020.acl-main.721

[40] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. 2019. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In *ICLR*.

[41] Mike Mintz, Steven Bills, Rion Snow, and Daniel Jurafsky. 2009. Distant supervision for relation extraction without labeled data. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*. Association for Computational Linguistics, Suntec, Singapore, 1003–1011. https://www.aclweb.org/anthology/P09-1113

[42] Ion Muslea, Steve Minton, and Craig Knoblock. 1999. A hierarchical approach to wrapper induction. In *Proceedings of the third annual conference on Autonomous Agents*. 190–197.

[43] David Opitz and Richard Maclin. 1999. Popular ensemble methods: An empirical study. *Journal of artificial intelligence research* 11 (1999), 169–198.

[44] Panupong Pasupat and Percy Liang. 2014. Zero-shot Entity Extraction from Web Pages. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Baltimore, Maryland, 391–401. https://doi.org/10.3115/v1/P14-1037

[45] Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, 1470–1480. https://doi.org/10.3115/v1/P15-1142

[46] Siddharth Patwardhan and Ellen Riloff. 2007. Effective Information Extraction with Semantic Affinity Patterns and Relevant Regions. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. Association for Computational Linguistics, Prague, Czech Republic, 717–727. https://www.aclweb.org/anthology/D07-1075

[47] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1532–1543. https://doi.org/10.3115/v1/D14-1162

[48] Yujie Qian, Enrico Santus, Zhijing Jin, Jiang Guo, and Regina Barzilay. 2019. GraphIE: A Graph-Based Framework for Information Extraction. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 751–761. https://doi.org/10.18653/v1/N19-1082

[49] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Austin, Texas, 2383–2392. https://doi.org/10.18653/v1/D16-1264

[50] Mohammad Raza and Sumit Gulwani. 2020. Web Data Extraction using Hybrid Program Synthesis: A Combination of Top-down and Bottom-up Inference. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1967–1978.

[51] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15)*. AAAI Press, 792–800.

[52] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. https://arxiv.org/abs/1908.10084

[53] Ameesh Shah, Eric Zhan, Jennifer J Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. 2020. Learning Differentiable Programs with Admissible Neural Heuristics. In *NeurIPS*.

[54] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. 2018. Houdini: Lifelong learning as program synthesis. In *Advances in Neural Information Processing Systems*. 8687–8698.

[55] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[56] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

[57] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: Filtering Spreadsheet Data using Examples. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 195–213.

[58] Sen Wu, Luke Hsiao, Xiao Cheng, Braden Hancock, Theodoros Rekatsinas, Philip Levis, and Christopher Ré. 2018. Fonduer: Knowledge Base Construction from Richly Formatted Data. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1301–1316.

[59] Halley Young, Osbert Bastani, and Mayur Naik. 2019. Learning neurosymbolic generative models via program synthesis.

[60] Manuela Zanda and Gavin Brown. 2009. A Study of Semi-supervised Generative Ensembles. In *Multiple Classifier Systems*, Jón Atli Benediktsson, Josef Kittler, and Fabio Roli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 242–251.

[61] Xiaojin Zhu. 2005. *Semi-Supervised Learning Literature Survey*. Technical Report 1530. Computer Sciences, University of Wisconsin-Madison.