WebRelate: Integrating Web Data with Spreadsheets using Examples

JEEVANA PRIYA INALA, MIT, USA

RISHABH SINGH, Microsoft Research, USA

Data integration between web sources and relational data is a key challenge faced by data scientists and spreadsheet users. There are two main challenges in programmatically joining web data with relational data. First, most websites do not expose a direct interface to obtain tabular data, so the user needs to formulate a logic to get to different webpages for each input row in the relational table. Second, after reaching the desired webpage, the user needs to write complex scripts to extract the relevant data, which is often conditioned on the input data. Since many data scientists and end-users come from diverse backgrounds, writing such complex regular-expression based logical scripts to perform data integration tasks is unfortunately often beyond their programming expertise.

We present WEBRELATE, a system that allows users to join semi-structured web data with relational data in spreadsheets using input-output examples. WEBRELATE decomposes the web data integration task into two sub-tasks of i) URL learning and ii) input-dependent web extraction. We introduce a novel synthesis paradigm called "Output-constrained Programming By Examples", which allows us to use the finite set of possible outputs for the new inputs to efficiently constrain the search in the synthesis algorithm. We instantiate this paradigm for the two sub-tasks in WEBRELATE. The first sub-task generates the URLs for the webpages containing the desired data for all rows in the relational table. WEBRELATE achieves this by learning a string transformation program using a few example URLs. The second sub-task uses examples of desired data to be extracted from the corresponding webpages and learns a program to extract the data for the other rows. We design expressive domain-specific languages for URL generation and web data extraction, and present efficient synthesis algorithms for learning programs in these DSLs from few input-output examples. We evaluate WEBRELATE on 88 real-world web data integration tasks taken from online help forums and Excel product team, and show that WEBRELATE can learn the desired programs within few seconds using only 1 example for the majority of the tasks.

CCS Concepts: • **Software and its engineering** → **Programming by example**;

Additional Key Words and Phrases: Program Synthesis, Data Integration, Spreadsheets, Web Mining

1 INTRODUCTION

Data integration is a key challenge faced by many data scientists and spreadsheet end-users. Despite several recent advances in techniques for making it easier for users to perform data analysis [Kandel et al. 2011], the inferences obtained from the analyses is only as good as the information diversity of the data. Therefore, more and more users are enriching their internal data

Authors' addresses: Jeevana Priya Inala, CSAIL, MIT, 32 Vassar Street, Cambridge, MA, 02139, USA, jinala@csail.mit.edu; Rishabh Singh, Microsoft Research, Redmond, WA, USA, risin@microsoft.com.

in spreadsheets with the rich data available on the web. However, the web data sources (websites) are typically semi-structured and in a format different from the original spreadsheet data format, and hence, performing such integration tasks requires writing complex regular-expression based data transformation and web scraping scripts. Unfortunately, a large fraction of end-users come from diverse backgrounds and writing such scripts is beyond their programming expertise [Gualtieri 2009]. Even for experienced programmers and data scientists, writing such data integration scripts can be difficult and time consuming.

Example 1.1. To make the problem concrete, consider the integration task shown in Fig. 1. A user had a spreadsheet consisting of pairs of currencies and the dates of transaction (shown in Fig. 1(a)), and wanted to get the exchange rates from the web. Since there were thousands of rows in the spreadsheet, manually performing this task was prohibitively expensive for the user.

	Cur 1	Cur 2	Date	Exchange Rate
1	EUR	USD	03, November, 16	
2	USD	INR	01, November, 16	
3	AUD	CAD	07, October, 16	

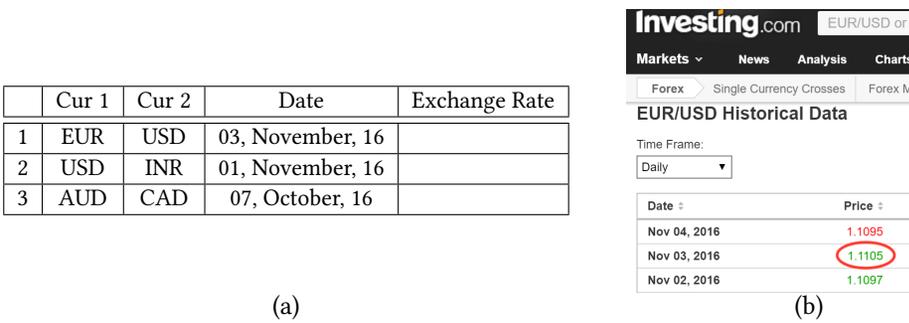(a)   (b)

Fig. 1. (a) A spreadsheet containing pairs of currency symbols and dates of transaction. (b) The webpage for EUR to USD exchange rate for different dates.

Previous works have explored two different strategies for automating such data integration tasks. In DataXFormer [Abedjan et al. 2015], a user provides a few end-to-end input-output examples (such as $row1 \rightarrow 1.1105$ in the above example), and the system uses a fully automatic approach by searching through a huge database of web forms and web tables to find a transform that is consistent with the examples. However, many websites do not expose web forms and do not have the data in a single web table. On the other end are programming by demonstration (PBD) systems such as WebCombine [Chasins et al. 2015] and Vegemite [Lin et al. 2009] that rely on users to demonstrate how to navigate and retrieve the desired data for a few example rows. Although the PBD systems can handle a broader range of webpages compared to DataXFormer, they tend to put an additional burden on users to perform exact demonstrations to get to the webpage, which has been shown to be problematic for users [Lau 2009].

In this paper, we present an intermediate approach where a user provides a few examples of URLs of webpages that contain the data as well as examples of the desired data from these webpages, and the system automatically learns how to perform the integration task for the other rows in the spreadsheet. For instance, in the above task, the example URL for the first row is http://www.investing.com/currencies/eur-usd-historical-data and the corresponding webpage is shown in Fig. 1(b) where the user highlights the desired data. There are three main challenges for a system to learn a program to automate this integration task.

First, the system needs to learn a program to generate the desired URLs for the other input rows. In the above example, the intended program for learning URLs needs to select the appropriate columns from the spreadsheet, perform casing transformations, and then concatenate them with appropriate
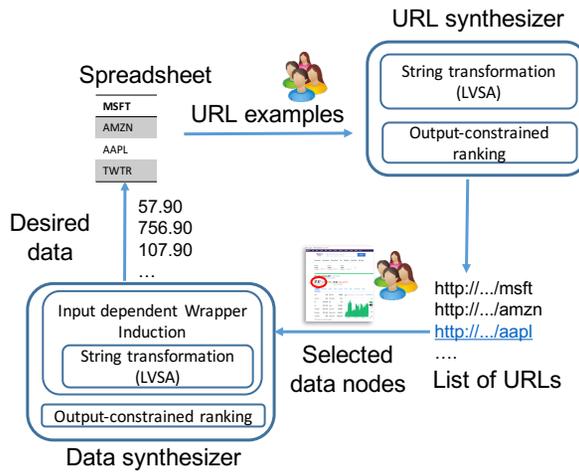
Fig. 2. An overview of the workflow of the WebRelate system. A user starts with providing a few examples for URL strings corresponding to the desired webpages of the first few spreadsheet rows. The URL synthesizer then learns a program consistent with the URL examples, which is executed to generate the desired URLs for the remaining spreadsheet rows. The user then opens the URLs for the first few spreadsheet rows in an adjoining pane (one at a time), and highlights the data items that need to be extracted from the webpage. The Data synthesizer then learns a data extraction program consistent with the examples to extract the desired data for the remaining spreadsheet rows.

constant strings. Moreover, many URL generation programs require using string transformations on input data based on complex regular expressions.

The second challenge is to learn an extraction logic to retrieve the relevant data, which depends on the underlying DOM structure of the webpage. Additionally, for many integration scenarios, the data that needs to be extracted from the web is conditioned on the data in the table. For instance, in the above example, the currency exchange rate from the web page should be extracted based on the Date column in the table. There are efficient systems in the literature for wrapper induction [Anton 2005; Dalvi et al. 2009; Kushmerick 1997; Muslea et al. 1998] that can learn robust and generalizable extraction programs from a few labeled examples, but, to the best of our knowledge, there is no previous work that can learn data extraction programs that are conditioned on the input.

The third challenge is that the common data key to join the two sources (spreadsheet and web data) might be in different formats, which requires writing additional logic to transform the data before performing the join. For instance, in the example above, the format of the date in the web page "Nov 03, 2016" is different from the format in the spreadsheet "03, November, 16".

To address the above challenges, we present WebRelate, a system that uses input-output examples to learn a program to perform the web data integration task. The high-level overview of the system is shown in Fig. 2. It breaks down the integration task into two sub-tasks. The first sub-task uses the *URL synthesizier* module to learn a program to generate the URLs of the webpages that contain the desired data from few example URLs. The second sub-task uses the *Data synthesizer* module to learn a data extraction program to retrieve the relevant data from these URLs given a set of example data selections on the webpages.

WebRelate is built on top a novel program synthesis paradigm of "Output-constrained Programming By Examples" (O–PBE). This formulation is only possible in PBE scenarios where it is possible to compute a finite set of possible outputs for new inputs (i.e. for inputs other than the inputs in

the input-output examples provided by the user). Previous PBE systems such as FlashFill [Gulwani 2011; Gulwani et al. 2012] and its extensions do not have such a property as any output string is equally likely for the new inputs and there is no way to constrain the possible set of outputs. The O-PBE paradigm allows us to develop a new efficient synthesis algorithm that combines both the *output uniqueness constraint* (program should match only 1 output string that is provided by the user for the inputs in the examples) as well as the *generalization constraint* (output is within the set of possible outputs for the other inputs). We instantiate the O-PBE paradigm for the two sub-tasks in WebRelate: i) URL generation, and ii) input-dependent web extraction. For URL learning problems, the constraint is that the output should be a valid URL or the output should be from the list of possible URLs obtained using the search engine for that particular input. For data extraction problems, the constraint is that the output program for every input should result in a non-empty node in the corresponding HTML document.

We design an expressive domain-specific language (DSL) for the URL generation programs. The DSL is built on top of regular expression based substring operations introduced in FlashFill [Gulwani 2011]. We then present a synthesis algorithm based on *layered version space algebra* (LVSA) to efficiently learn URL programs in the DSL from few input-output examples. We also show that this algorithm is significantly better than existing VSA based techniques. Learning URLs as a string transformation program raises an additional challenge since some URLs may contain additional strings such as unique identifiers that are not in the spreadsheet table and are not constants (Ex 2.2 illustrates this challenge). The O-PBE framework allows us to handle such scenarios by having *filter* programs in the language for URLs. These filter programs produce regular expressions for URLs as opposed to concrete strings. Then, WebRelate leverages a search engine to get a list of relevant URLs for every input and selects the one that matches the regular expression.

Similar to URL learning, WebRelate uses examples to learn data extraction programs. A user can select a URL to be loaded in a neighboring frame, and then highlight the desired data element to be extracted. WebRelate records the DOM locations of all such example data and learns a program to perform the desired extraction task for the other rows in the table. We present a new technique for *input-dependent wrapper induction* that involves designing an expressive DSL built on top of XPath constructs and regular expression based substring operations. We, then, present a synthesis algorithm that uses *predicates graphs* to succinctly represent a large number of DSL expressions that are consistent with a given set of I/O examples. For the currency exchange example, our system learns a program that first transforms the date to the required format and then, extracts the conversion rate element whose left sibling contains the transformed date value from the webpage.

This paper makes the following key contributions:

- We present WebRelate, a system to join web data with relational data, which divides the data integration task into URL learning and web data extraction tasks.
- We present a novel program synthesis paradigm called "Output-constrained Programming By Examples" that allows for incorporating output uniqueness and generalization constraints for an efficient synthesis algorithm (§ 3). We instantiate this paradigm for the two sub-tasks of URL learning and data extraction.
- We design a DSL for URL generation using string transformations and an algorithm based on *layered version space algebra* to learn URLs from a few examples (§ 4).
- We design a DSL on top of XPath constructs that allows input-dependent data extractions. We present a synthesis algorithm based on a *predicates graph* data structure to learn extraction programs from examples (§ 5).

- We evaluate WEBRELATE on 88 real-world data integration tasks. It takes on average less than 1.2 examples and 0.15 seconds each to learn the URLs, whereas it takes less than 5 seconds and 1 example to learn data extraction programs for 93% of tasks (§ 6).

2 MOTIVATING EXAMPLES

In this section, we present a few additional motivating scenarios for web data integration tasks of varying complexity and illustrate how WEBRELATE can be used to automate these tasks using few input-output examples.

Example 2.1. **[Stock prices]** A user wanted to retrieve the stock prices for hundreds of companies (Company column) as shown in Fig. 3.

	Company	URL	Stock price
1	MSFT	https://finance.yahoo.com/q?s=msft	59.87
2	AMZN	**https://finance.yahoo.com/q?s=amzn**	**775.88**
3	AAPL	**https://finance.yahoo.com/q?s=aapl**	**113.69**
4	TWTR	**https://finance.yahoo.com/q?s=twtr**	**17.66**
5	T	**https://finance.yahoo.com/q?s=t**	**36.51**
6	S	**https://finance.yahoo.com/q?s=s**	**6.31**
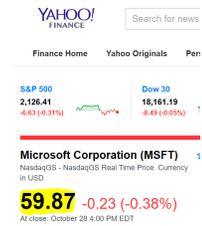
(a)   (b)

Fig. 3. Joining stock prices from web with company symbols using WEBRELATE. Given one example URL and data extraction from the webpage for the first row, the system learns a program to automatically generate the corresponding URLs and extracted stock prices for the other rows (shown in bold).

In order to perform this integration task in WEBRELATE, a user can provide an example URL such as https://finance.yahoo.com/q?s=msft that has the desired stock price for the first row in the spreadsheet (Fig. 3(a)). This web-page then gets loaded and is displayed to the user. The user can then highlight the required data from the web-page (Fig. 3(b)). WEBRELATE learns the desired program to perform this task in two steps. First, it learns a program to generate the URLs for remaining rows by learning a string transformation program that combines the constant string "https://finance.yahoo.com/q?s=" with the company symbol in the input. Second, it learns a web data extraction program to extract the desired data from these web-pages.

Example 2.2. **[Weather]**. A user had a list of addresses in a spreadsheet and wanted to get the weather information at each location as shown in Fig. 4. The provided example URL is https://weather.com/weather/today/l/Seattle+WA+98109:4:US#!.

There are two challenges in learning the URL program for this example. First, the addresses contain more information than just the city and state names such as street names and house numbers. Therefore, the URL program first needs to learn regular expressions to extract the city-name and state from the address and then concatenate them appropriately with constant strings to get the desired URL. The second challenge is that the URL contains zip code that is not present in the input, meaning that there is no simple program that concatenates constants and sub-strings to learn the URLs for the remaining inputs. For supporting such cases, the DSL for URL learning also supports *filter* programs that use regular expressions to denote unknown parts of the URL. A possible *filter* program for this example is https://weather.com/weather/today/l/{**Extracted city name**}+{**Extracted state name**}+{**AnyStr**}:4:US#!, where **AnyStr** can match any non-empty string. Then, for every other row in the table, WEBRELATE leverages a search engine to get a

	Address	Weather
1	742 17th Street NE,Seattle,WA	59
2	732 Memorial Drive,Cambridge,MA	**43**
3	Apt 12, 112 NE Main St.,Boston,MA	**42**
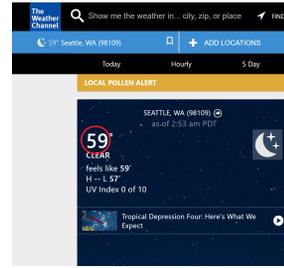
(a)   (b)

Fig. 4. Joining weather data from web with addresses. Given one example row, the system automatically extracts the weather for other row entries (shown in bold).

list of possible URLs for that row and selects the top URL that matches the filter program. By default, we use the words in input row as the search query term and set the target URL domain to be the domain of the given example URLs. WebRelate also allows users to provide search query terms (such as "Seattle weather") as additional search query examples and it learns the remaining search queries for other inputs using a string transformation program. Using the search query and a filter program, WebRelate learns the following URL for the second row: https://weather.com/weather/today/l/Cambridge+MA+02139:4:US#!.

Example 2.3. **[Citations]** A user had a table containing author names and titles of research papers, and wanted to extract the number of citations for each article using Google Scholar (as shown in Fig. 5).

In this case, the example URL for Samuel Madden's Google Scholar page is https://scholar.google.com/scholar?q=samuel+madden and the corresponding web page is shown in Fig. 5(b). The URL can be learned as a string transformation program over the Author column. The more challenging part of this integration task is that the data in the web page is in a semi-structured format and the required data (# citations) should be extracted based on the Article column in the input. Our data extraction DSL is expressive enough to learn a program that captures this complex dependency. This program generates the entire string "Cited by 2316" and WebRelate learns another transformation program on top of this to extract only the number of citations "2316" from the results.

	Author	Article	# citations
1	Samuel Madden	TinyDB: an acquisitional ...	2316
2	HV Jagadish	Structural joins: A primitive ...	**1157**
3	Mike Stonebraker	C-store: a column-oriented ...	**1119**

(a)   (b)

Fig. 5. (a) Integrating a spreadsheet containing authors and article titles with the number of citations obtained from Google Scholar. (b) The google scholar page for the first example.

3 OUTPUT-CONSTRAINED PROGRAMMING BY EXAMPLE

We first define the abstract *Output-constrained* Programming By Example (O-PBE) problem, which we then instantiate for both the URL synthesizer and the data extraction synthesizer. Let $E = \{(i_1, o_1), \cdots, (i_m, o_m)\}$ denote the list of m input-output examples provided by the user and $U = \{i_{m+1}, \cdots, i_{m+n}\}$ denote the list of n inputs with unknown outputs. We use L to denote the DSL that describes the space of possible programs. The traditional PBE problem is to find a program P in the DSL that satisfies the given input-output examples i.e.

$$\exists P \in L. \forall k \leq m. P(i_k) = o_k$$

On the other hand, the O-PBE problem formulation takes advantage of the existence of an *oracle* O that can generate a finite list of possible outputs for any given input. For instance, in the URL learning scenario, this oracle is a search engine that lists the top URLs obtained for a search query term based on the input. For the data extraction learning scenario, this oracle is the web document corresponding to each input where any node in the web document is a candidate output. The existence of this oracle can benefit the PBE problem in two ways. First, we can solve problems with noisy input-output examples where there is no single program in the language that can *generate* all the desired outputs for the inputs in the examples. For instance, the URL learning task in Ex 2.2 is a synthesis problem with noisy input-output examples that traditional PBE approaches cannot solve. However, the presence of oracles can solve this problem because it is now sufficient to just learn a program that, given an input and list of possible outputs, can discriminate the desired output from the other possible outputs. This property is called the *output-uniqueness constraint*. The second benefit of oracles is that they impose additional constraints on the learned program. In addition to satisfying the input-output examples, we want the learned program to produce valid outputs for the other inputs. We refer to this as the *generalization constraint*. Using this constraint, the O-PBE approach can efficiently learn the desired program using very few input-output examples. Thus, we can now define the O-PBE problem formally as follows:

$$\exists P \in L. \forall k \leq m. P(i_k, O(i_k)) = o_k \quad \textit{(output-uniqueness constraint)}$$
$$\wedge \forall k \leq n. P(i_{m+k}, O(i_{m+k})) \in O(i_{m+k}) \quad \textit{(generalization constraint)}$$

where program P is now a more expressive higher-order expression in the language L that takes as input a list of possible outputs $O(i)$ (in addition to the input i) and returns one output among them i.e. $P(i, O(i)) = o \ s.t. \ o \in O(i)$.

At a high level, to solve this synthesis problem, WEBRELATE first learns the set of all programs in the language L that satisfies the given input-output examples E (not necessarily uniquely). This set is represented succinctly using special data structures. Then, WEBRELATE uses an *output-constrained ranking scheme* to eliminate programs that do not uniquely satisfy the given examples E or are inconsistent with the unseen inputs U.

We now describe the two instantiations of the abstract O-PBE problem for the URL and data extraction synthesizers in more detail.

4 URL LEARNING

We first present the domain-specific language for the URL generation programs and then present a synthesis algorithm based on *layered version space algebra* to efficiently learn programs from few input-output examples.

4.1 URL Generation Language L_u

Syntax. The syntax of the DSL for URL generation L_u is shown in Fig. 6. The DSL is built on top of regular expression based substring constructs introduced in FlashFill [Gulwani 2011; Gulwani

$$
\begin{array}{rcl}
\text{URL String } u & ::= & \boxed{\texttt{Filter}(\phi)} \\
\text{Predicate } \phi & ::= & \texttt{Concat}(f_1, \cdots, f_n) \\
\text{Atomic expr } f & ::= & r \mid b \\
\boxed{\text{Regex expr } \texttt{r}} & ::= & \boxed{\texttt{AnyStr}} \\
\text{Base expr } b & ::= & \texttt{ConstStr}(s) \mid \texttt{SubStr}(k, p_l, p_r, c) \mid \boxed{\texttt{Replace}(k, p_l, p_r, c, s_1, s_2)} \\
\text{Position } p & ::= & (\tau, k, \texttt{Dir}) \mid \texttt{ConstPos}(k) \\
\text{Direction } \texttt{Dir} & ::= & \texttt{Start} \mid \texttt{End} \\
\text{Token } \tau & ::= & s \mid T \\
\text{Regex Token } T & ::= & \texttt{CAPS} \mid \texttt{ProperCase} \mid \texttt{lowercase} \mid \texttt{Digits} \mid \texttt{Alphabets} \mid \texttt{AlphaNum} \\
\text{Case } c & ::= & \texttt{lower} \mid \texttt{upper} \mid \texttt{prop} \mid \texttt{iden}
\end{array}
$$

Fig. 6. The syntax of the DSL L_u for regular expression based URL learning. Here, s is a string, and k is an integer.

et al. 2012] with the key differences highlighted. The top-level URL program is a filter expression $\texttt{Filter}(\phi)$ that takes as argument a predicate ϕ, where ϕ is denoted using a concatenation of base atomic expressions or $\texttt{AnyStr}$ expressions. The base atomic expression b can either be constant string, a regular expression based substring expression that takes an index, two position expressions and a case expression, or a replace expression that takes two string arguments in addition to the arguments of substring expression. A position expression can either be a constant position index k or a regular expression based position $(\tau, k, \texttt{Dir})$ that denotes the $\texttt{Start}$ or $\texttt{End}$ of kth match of token τ in the input string.

Semantics. The semantics of the DSL for L_u is shown in Fig. 7. We use the notation $[\![x]\!]_i$ to represent the semantics of x when evaluated on an input i (a list of strings from the table row). The semantics of a filter expression $\texttt{Filter}(\phi)$ is to use a URL list generator oracle O_u to obtain a ranked list of URLs for the input i, and select the first URL that matches the regular expression generated by the evaluation of the predicate ϕ. The default implementation for O_u runs a search engine on the words derived from the input i (with the domain name of URL examples) and returns the URLs of the top results. However, a user can also provide a few search query examples, which WebRelate uses to learn another string transformation program to query the search engine for the remaining rows. The semantics of a predicate expression ϕ is to first evaluate each individual atomic expression in the arguments and then return the concatenation of resulting atomic strings. The semantics of $\texttt{AnyStr}$ expression is Σ^+ that can match any non-empty string. The semantics of a substring expression is to first evaluate the position expressions to obtain left and right indices and then return the corresponding substring for the kth string in i. The semantics of a replace expression $\texttt{Replace}(k, p_l, p_r, c, s_1, s_2)$ is to first get the substring corresponding to the position expressions and then replace all occurrences of string s_1 with the string s_2 in the substring. We allow strings s_1 and s_2 to take values from a finite set of delimiter strings such as " ", "-", "_", and "#".

Examples. A program in L_u to perform the URL learning task in Ex 1.1 is: $\texttt{Filter}(\texttt{Concat}(\texttt{ConstStr}("http://www.investing.com/currencies/"), \texttt{Substr}(0, \texttt{ConstPos}(0), \texttt{ConstPos}(-1), \texttt{lower}), \texttt{ConstStr}("-"), \texttt{Substr}(1, \texttt{ConstPos}(0), \texttt{ConstPos}(-1), \texttt{lower}), \texttt{ConstStr}("-historical-data")))$. The program concatenates a constant string, the lowercase transformation of the first input string (-1 index in $\texttt{ConstPos}$ denotes the last string index), the constant hyphen , the lowercase transformation of the second input string, and finally, another constant string.

$$
\begin{aligned}
[\![\texttt{Filter}(\phi)]\!]_i &= u \text{ s.t. } u \in O_u(i) \text{ and } [\![\phi]\!]_i \models u \\
[\![\texttt{Concat}(f_1, \cdots, f_n)]\!]_i &= \texttt{Concat}([\![f_1]\!]_i, \cdots, [\![f_n]\!]_i) \\
[\![\texttt{AnyStr}]\!]_i &= \Sigma^+ \\
[\![\texttt{ConstStr}(s)]\!]_i &= s \\
[\![\texttt{SubStr}(k, p_l, p_r, c)]\!]_i &= \texttt{ToCase}(i_k[[\![p_l]\!]_i..[\![p_r]\!]_i], c) \\
[\![\texttt{Replace}(k, p_l, p_r, c, s_1, s_2)]\!]_i &= \texttt{ToCase}(i_k[[\![p_l]\!]_i..[\![p_r]\!]_i], c)[s_1 \leftarrow s_2] \\
[\![\texttt{ConstPos}(k)]\!]_i &= k > 0?\, k : \texttt{len}(s) + k \\
[\![(\tau, k, \texttt{Start})]\!]_i &= \text{Start of } k^{\text{th}} \text{match of } \tau \text{ in i} \\
[\![(\tau, k, \texttt{End})]\!]_i &= \text{End of } k^{\text{th}} \text{match of } \tau \text{ in i}
\end{aligned}
$$

Fig. 7. The semantics of the DSL L_u. O_u is a URL list oracle that generates a ranked list of URLs for the input i by using a search engine.

A DSL program for the URL learning task in Ex 2.2 is: $\texttt{Filter}(\phi)$, where $\phi \equiv \texttt{Concat}(b_1, b_2,$ $\texttt{ConstStr}(\text{"+"}), b_3, \texttt{ConstStr}(\text{"+"}), \texttt{AnyStr}, \texttt{ConstStr}(\text{":4:US\#!"}))$, $b_1 \equiv \texttt{ConstStr}(\text{"https://weather.com} \text{/weather/today/l/"})$, $b_2 \equiv \texttt{SubStr}(0, (\text{","},\text{-2},\texttt{End}),(\text{","},\text{-1},\texttt{Start}),\texttt{iden})$, and $b_3 \equiv \texttt{SubStr}(0, (\text{","},\text{-1},\texttt{End}),$ $\texttt{ConstPos}(\text{-1}),\texttt{iden})$. Here, b_2 and b_3 are regular expression based substring expressions to derive the city and the state strings from the input address.

4.2 Synthesis Algorithm

We now present the synthesis algorithm to learn a URL program that solves the O-PBE problem.

4.2.1 Background: Version Space Algebra. This section presents a brief background on version space algebra (VSA), a technique used by existing string transformation synthesizers such as FlashFill. We refer the readers to [Gulwani et al. 2012] for a detailed description. The key idea of VSA is to use a directed acyclic graph (DAG) to succinctly represent an exponential number of candidate programs using polynomial space. For the string transformation scenario, a DAG $\mathcal{D}$ is defined as a tuple $(\mathcal{V}, v_s, v_t, \mathcal{E}, \mathcal{W})$ where $\mathcal{V}$ is the set of vertices, $\mathcal{E}$ is the set of edges, v_s is the starting vertex and v_t is the target vertex. Each edge in the DAG represents a set of atomic expressions (the map $\mathcal{W}$ captures this relation) whereas a path in the DAG represents the concatenation of the atomic expressions of the edges. Given a synthesis problem, a DAG is constructed in such a way that any path in the DAG from v_s to v_t is a valid program in L that satisfies the examples. This is achieved by iteratively constructing a DAG for each example and performing an automata-like-intersection on these individual DAGs to get the final DAG. For example, a sample DAG is shown in Fig. 8. The nodes in this DAG (for each I/O example) correspond to the indices of the output string. An edge from a node i to a node j in the DAG represents the set of expressions that can generate the substring between the indices i and j of the output example string when executed on the input data.

4.2.2 Layered Version Space Algebra. It is challenging to use VSA techniques for learning URLs because the URL strings are long, and the run time and the DAG size of the existing algorithms explode with the length of the output. To overcome this issue, we introduce a synthesis algorithm based on a layered version space algebra for efficiently searching the space of consistent programs. The key idea is to perform search over increasingly expressive sub-languages $L_1 \subseteq L_2 \cdots \subseteq L_k$, where L_k corresponds to the complete language L_u. The sub-languages are selected such that the earlier languages capture the portion of the search space that is highly probable and at the same
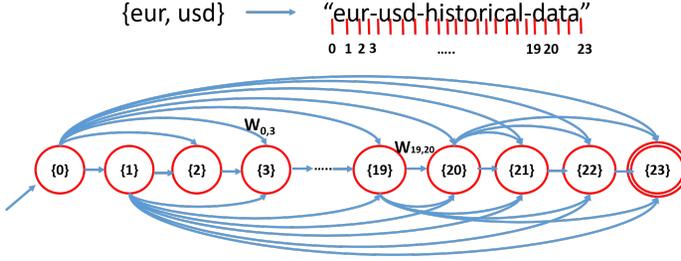
Fig. 8. An example DAG using Version space algebra (VSA) to represent the set of transformations consistent with the example. For example, here, $\mathcal{W}_{0,3}$ is a list of three different expressions {ConstStr(eur), SubStr(0, ConstPos(0), ConstPos(−1), lower), AnyStr}; $\mathcal{W}_{19,20}$ = {ConstStr(d), SubStr(1, ConstPos(2), ConstPos(3), lower), AnyStr}.

time, it is easier to search for a program in these sub-languages. For example, it is less likely to concatenate a constant with a sub-string within a word in the URL and hence, earlier sub-languages are designed to eliminate such programs. For instance, consider the URL in Ex 1.1. In this case, given the first I/O example, programs that derive the character d in data from the last character in the input (USD) are not desirable because they do not generalize to other inputs.

The general synthesis algorithm GenProg for learning URL string transformations in a sublanguage L_i is shown in Fig. 9. This algorithm is similar to the VSA based algorithm, but the key difference is that instead of learning all candidate programs (which can be expensive), the algorithm only learns the programs that are in the sub-language L_i. The GenProg algorithm takes as input the I/O examples $\{i_k, o_k\}_k$, and three Boolean functions $\lambda_s, \lambda_c, \lambda_a$ that parameterize the search space for the language L_i. The output is a program that is consistent with the examples or ⊥ if no such program exists. The algorithm first uses the GenDag procedure to learn a DAG consisting of all consistent programs (in the given sub-language) for the first example. It then iterates over the other examples and intersects the corresponding DAGs to compute a DAG representing programs consistent with all the examples. Finally, it searches for the best program in the DAG that satisfies the O-PBE constraints and returns it.

The GenDag algorithm is also shown in Fig. 9, where the space of programs is constrained by the parameter Boolean functions λ_s, λ_c, and λ_a. Each function $\lambda : \text{int} \rightarrow \text{int} \rightarrow \text{string} \rightarrow \text{bool}$ takes two integer indices and a string as input and returns a Boolean value denoting whether certain atomic expressions are allowed to be added to the DAG. The algorithm first creates $\text{len}(o) + 1$ nodes, and then adds an edge between each pair of nodes $\langle k, l \rangle$ such that $0 \leq k < l \leq \text{len}(o)$. For each edge $\langle k, l \rangle$, the algorithm learns all atomic expressions that can generate the substring $o[k..l]$. For learning SubStr and Replace expressions, the algorithm enumerates different argument parameters for positions, cases, and delimiter strings, whereas the ConstStr and AnyStr expressions are always available to be added. The addition of SubStr and Replace atomic expressions to the DAG are guarded by the Boolean function λ_s whereas the addition of ConstStr and AnyStr atomic expressions are guarded by the Boolean functions λ_c and λ_a, respectively.

Fig. 10 shows how the different layers are instantiated for learning URL expressions. For the first layer, the algorithm only searches for URL expressions where each word in the output is either generated by a substring or a constant or an AnyStr. The onlyWords (oW) function is defined as:

$$\text{oW} = (i, j, o) => \neg\text{isAlpha}(o[i-1]) \wedge \neg\text{isAlpha}(o[j+1]) \wedge \forall\, k : i \leq k \leq j\ \text{isAlpha}(o[k])$$

where isAlpha(c) is true if the character c is an alphabet. The second layer allows for multiple words in the output string to be learned as a single substring. The function multipleWords (mW) is

```
GenProg({(i_k,o_k)}_k, λ_s, λ_c, λ_a)
 Dag d = GenDag(i_1,o_1,λ_s,λ_c,λ_a)
 for k from 2 to m:
    if d = ⊥: return ⊥
    Dag d' = GenDag(i_k,o_k,λ_s,λ_c,λ_a)
    d = d.Intersect(d')
 return SearchBestProg(d)
```

```
GenDag(i, o, λ_s, λ_c, λ_a)
 𝒱 = {0, ..., len(o)}, v_s = {0}, v_t = {len(o)}
 ℰ = {⟨k,l⟩ : 0 ≤ k < l ≤ len(o)}
 𝒲 = maps each edge to set of atomic exprs
 foreach 0 ≤ k < l ≤ len(o):
    w = ∅
    if λ_s(k,l,o): w.add(GenSubstr(k,l, i, o))
    if λ_s(k,l,o): w.add(GenReplace(k,l, i, o))
    if λ_c(k,l,o): w.add(ConstStr(o[k..l]))
    if λ_a(k,l,o): w.add(AnyStr)
    𝒲.add(⟨k,l⟩, w)
 return Dag(𝒱, v_s, v_t, ℰ, 𝒲)
```

Fig. 9. Synthesis algorithm for URL generation programs, parameterized by $λ_s$, $λ_c$, and $λ_a$.

```
LearnURL({(i_k,o_k)}_k)
 if(p := GenProg({(i_k,o_k)}_k, oW, oW, oW) ≠ ⊥: return p // Layer 1
 if(p := GenProg({(i_k,o_k)}_k, mW, oW, oW) ≠ ⊥: return p // Layer 2
 if(p := GenProg({(i_k,o_k)}_k, iW ∨ mW, oW, oW) ≠ ⊥: return p // Layer 3
 return GenProg({(i_k,o_k)}_k, T, T, T) // Layer 4
```

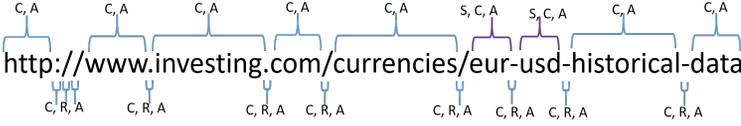Fig. 10. Layered version spaces for learning a program in L_u.

Fig. 11. DAG for Ex 1.1 constructed using layer 1 where S: SubStr, C: ConstStr, R: Replace, and A: AnyStr.

defined as:

$$mW = (i, j, o) => ¬isAlpha(o[i − 1]) ∧ ¬isAlpha(o[j + 1])$$

The third layer, in addition, allows for words in the output string to be a concatenation of multiple substrings, but not a concatenation of substrings with constant strings (or AnyStr). The function insideWords (iW) is defined as:

$$iW = (i, j, o) => ∀ k : i ≤ k ≤ j \, isAlpha(o[k])$$

The final layer allows arbitrary compositions of constants, AnyStr and substring expressions by setting the functions $λ_s$, $λ_c$ and $λ_a$ to always return True (T).

Example 4.1. Consider the currency exchange example where the example URL for the input {EUR, USD, 03, November 16} is "http://www.investing.com/currencies/eur-usd-historical-data". The first layer of the search will create a DAG as shown in Fig. 11. We can observe that the DAG eliminates most of the unnecessary programs such as those that consider the d in data to come from the D in USD and the DAG is much smaller (and sparser) compared to the DAG in Fig. 8.

Example 4.2. For the same example, assume that we want to get the currency exchange values from http://finance.yahoo.com/q?s=EURUSD=X. For this example, layers 1 and 2 can only learn the string EURUSD as a ConstStr which will not work for the other inputs, or a AnyStr which is
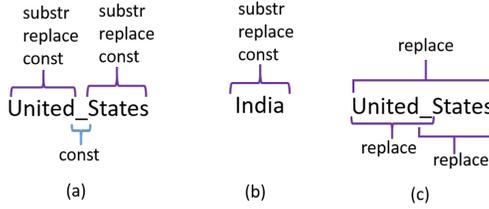
Fig. 12. (a) and (b) shows a portion of the DAGs for Ex 4.3 using layer 1 of hierarchical search. (c) shows additional edges from layer 2 of the hierarchical search.

too general. So, the layered search moves to layer 3 which allows SubStr inside words. Now, the system can learn EUR and USD separately as two SubStr expressions and concatenate them.

Example 4.3. Consider another example where we want to learn the URL https://en.wikipedia. org/wiki/United_States from the input `United  States` and the URL https://en.wikipedia.org/ wiki/India from the input `India`. Fig. 12(a) and (b) show portions of the DAGs for these two examples when using the first layer[1]. Here, there is no common program that can learn both these examples together. In such situations, the layered search will move to the second layer. Fig. 12(c) shows the extra edges added in layer 2. Now, these examples can be learned together as Filter(Concat(ConstStr("https"),···,ConstStr("/"), Replace(0, ConstPos(0),ConstPos(-1), iden, " ", "_"))).

4.2.3 Output-constrained Ranking. The LearnURL algorithm learns multiple programs that match the example URLs. However, not all of these programs are desirable for two reasons. First, some filter programs are too general and hence, fail the output-uniqueness constraint. For instance, AnyStr is one of the possible predicates that will be learned by the algorithm, but in addition to matching the desired example URLs, this predicate will also match any URL in the search results. Hence, we need to carefully select a consistent program from the DAG. Second, not all programs are equally desirable as they may not generalize well to the unseen inputs. For instance, consider Ex 2.2. If we have an example URL such as https://weather.com/weather/today/l/Seattle+WA+98109:4:US#!, then the programs that make the zip code 98109 to be a constant instead of AnyStr are not desirable. On the other hand, we want strings such as today and weather to be constants. We overcome both of these issues by devising an output-constrained ranking scheme.

Fig. 13 shows our approach where we search for a consistent program in tandem with finding the best program. The algorithm takes as input a DAG d, a list of examples E, and a list of unseen inputs U, and the outcome is the best program in the DAG that is consistent with the given examples (if such a program exists). The algorithm is a modification to Dijkstra's shortest path algorithm. The algorithm maintains a ranked list of at-most κ *prefix programs* for each node v in the DAG where a prefix program is a path in the DAG from the start node to v and the rank of a prefix program is the sum of ranks of all atomic expressions in the path. The atomic expressions are ranked as SubStr = Replace > ConstStr > AnyStr. Initially, the set of prefix programs for every node is $\emptyset$. The algorithm then traverses the DAG in reverse topological order and for each outgoing edge, it iterates through pairs of prefixes and atomic expressions based on their ranks. For each pair, the algorithm checks if the partial path satisfies the output-uniqueness constraint (Line 8) and the generalization constraint (Line 9). Whenever a consistent pair is found, the concatenation of the atomic expression with the prefix is added to the list of prefixes for the other node in the edge.

---

[1]We omit the DAGs for the first part of the URLs as they are similar to the previous example.

```
1 SearchBestProg(d, E, U)
2 foreach v ∈ 𝒱(d):
3    v.prefixes = ∅,
4 foreach v ∈ 𝒱(d) in reverse topological order:
5    foreach e: ⟨v, v′⟩ ∈ ℰ(d):
6       foreach φ_sofar in v.prefixes.RankedIterator:
7          foreach f in 𝒲(e).RankedIterator:
8             if Consistent(φ_sofar + f, v′, E):
9                if Generalizes(φ_sofar + f, v′, U):
10                   v′.prefixes.add(φ_sofar + f, φ_sofar.score + f.score)
11                   if f ≠ AnyStr: Goto Line 12
12          if AnyStr ∉ φ_sofar: Goto Line 5
13 return d.v_t.prefixes[0]
```

Fig. 13. Algorithm to find the best consistent program from the DAG learned by LearnURL.

Finally, the algorithm returns the highest ranked prefix for the target node of the DAG. In the limit $\kappa \to \infty$, the above algorithm will always find a consistent program if it exists. However, in practice, we found that a smaller value of κ is sufficient because of the two pruning steps at Line 11 and Line 12 and because of the ranking that gives least preference to AnyStr among the other atomic expressions.

THEOREM 4.4 (SOUNDNESS OF GENPROG). *The* GenProg *algorithm is sound for all* λ_s, λ_c *and* λ_a *i.e. given a few input-output examples* $\{(i_k, o_k)\}_k$, *the learned program* P *will always satisfy* $\forall k.\ [\![P]\!]_{i_k} = o_k$.

Proof sketch: This holds because the GenDag algorithm only learns programs that are consistent for each input and Intersect preserves this consistency across multiple inputs. For learned programs that contain AnyStr expressions, the SearchBestProg algorithm ensures soundness because it only adds an atomic expression to a prefix if the combination is consistent with respect to the given examples.

THEOREM 4.5 (COMPLETENESS OF GENPROG). *The* GenProg *algorithm is complete when* λ_s = *True,* λ_c = *True,* λ_a = *True, and in the limit* $\kappa \to \infty$ *where at-most* κ *prefixes are stored for each node in the* SearchBestProg *algorithm. In other words, if there exists a program in* L_u *that is consistent with the given set of input-output examples, then the algorithm will produce one.*

Proof sketch: This is because when λ_s, λ_c, and λ_a are True, GenDag will learn all atomic expressions for every edge that satisfy the examples. Since, the DAG structure allows all possible concatenations of atomic expressions, the GenDag algorithm is complete in this case. Intersect also preserves completeness, and in the limit $\kappa \to \infty$, the SearchBestProg will try all possible paths in the DAG to find a consistent program. Thus, GenProg does not drop any consistent program and hence, complete.

THEOREM 4.6 (SOUNDNESS OF LEARNURL). *The* LearnUrl *algorithm is sound.*

Proof sketch: This is because every layer in the layered search is sound using Theorem 4.4.

THEOREM 4.7 (COMPLETENESS OF LEARNURL). *The* LearnUrl *algorithm is complete in the limit* $\kappa \to \infty$ *where at-most* κ *prefixes are stored for each node in the* SearchBestProg *algorithm.*

Proof sketch: This follows because the last layer in the layered search is complete using Theorem 4.5.

$$
\begin{aligned}
\text{Prog } P &\ \ ::=\ \ (\text{name}, \{\pi_1, \cdots, \pi_r\}) \\
\text{Pred } \pi &\ \ ::=\ \ \pi_n \mid \pi_{\text{path}} \\
\text{NodePred } \pi_n &\ \ ::=\ \ \pi_a \mid \pi_c \\
\text{AttrPred } \pi_a &\ \ ::=\ \ [\text{attr}(\text{name}) == \boxed{\phi}\,] \\
\text{CountPred } \pi_c &\ \ ::=\ \ [\text{count}(\text{axis}) == k] \\
\text{PathPred } \pi_{\text{path}} &\ \ ::=\ \ [p] \\
\text{Path } p &\ \ ::=\ \ n_c \mid n_s \mid n_a/n_s \mid p/n_c \\
\text{Node } n &\ \ ::=\ \ (\text{name}, \text{axis}, \pi_{\text{pos}}, \{\pi_{n_1}, \cdots, \pi_{n_r}\}) \\
\text{PosPred } \pi_{\text{pos}} &\ \ ::=\ \ [\text{pos } (== \mid \leq)\, k] \mid \bot \\
\text{Axis } \text{axis} &\ \ ::=\ \ \texttt{Child} \mid \texttt{Ancestor} \mid \texttt{Left} \mid \texttt{Right} \\
\text{String } \boxed{\phi} &\ \ ::=\ \ \boxed{\text{Same as } \phi \text{ in Fig. 6}}
\end{aligned}
$$

Fig. 14. The syntax for the extraction language L_w, where name is a string, k is an integer, and $\bot$ is an empty predicate.

5 DATA EXTRACTION LEARNING

Once we have a list of URLs, we now need to synthesize a program to extract the relevant data from these web pages. This data extraction is usually done using a query language such as XPath [Berglund et al. 2003] that uses path expressions to select an HTML node (or a list of HTML nodes) from an HTML document. Previous systems such as DataXFormer [Abedjan et al. 2015; Morcos et al. 2015] have considered the absolute XPath obtained from the examples to do the extraction on other unseen inputs. An absolute XPath assumes that all the elements from the root of the HTML document to the target node have that same structure. This assumption is not always valid as web-pages are very dynamic. For instance, consider the weather data extraction from Ex 2.2. The web pages sometimes have an alert message to indicate events such as storms, and an absolute XPath will fail to extract the weather information from these web pages. More importantly, an absolute XPath will not work for input-dependent data extractions as shown in Ex 1.1.

Learning an XPath program from a set of examples is called *wrapper induction*, and there exist many techniques [Anton 2005; Dalvi et al. 2009; Kushmerick 1997; Muslea et al. 1998] that solve this problem. However, none of the previous approaches applied wrapper induction in the context of data-integration tasks that requires generating input-dependent XPath programs. We present a DSL that extends the XPath language with input-dependent constructs. Then, we present an O-PBE based synthesis algorithm that can learn robust and generalizable extraction programs using very few examples. Our synthesis algorithm uses a VSA based technique that allows us to seamlessly integrate with complex string transformation learning algorithms from § 4 that are necessary for learning input-dependent extractions.

Note that it is not always possible to achieve input-dependent data extraction by using a two-phase approach, which first extracts all relevant data from the web-page into a structured format and then, extracts the input-dependent components from this structured data. This is because the data-dependence between the input and the webpage is sometimes hidden, e.g. a stock div element might have id "msft-price", which is not directly visible to the users. In these scenarios, it is not possible for the users to identify and provide examples regarding what data should be extracted into the structured format in the intermediate step before the second step of data extraction. Hence, a more integrated approach is required for learning input-dependent extractions.

$$
\begin{aligned}
[\![(\text{name}, \{\pi_1, \cdots, \pi_r\})]\!]_i(w) &= \texttt{Filter}(\lambda\gamma.\gamma.\text{name} == \text{name and } \wedge_{k=1}^{r} [\![\pi_k]\!]_i(\gamma), \texttt{AllNodes}(w)) \\
[\![[\texttt{attr}(\text{name}) == \phi]]\!]_i(\gamma) &= \gamma.\texttt{Attr}(\text{name}) == [\![\phi]\!]_i \\
[\![[\texttt{count}(\text{axis}) == k]]\!]_i(\gamma) &= \texttt{Len}(\texttt{Neighbors}(\gamma, \text{axis})) == k \\
[\![[p]]\!]_i(\gamma) &= \texttt{Len}([\![p]\!]_i(\{\gamma\})) > 0 \\
[\![p/n]\!]_i(\{\gamma_k\}_k) &= [\![n]\!]_i([\![p]\!]_i(\{\gamma_k\}_k)) \\
[\![n]\!]_i(\{\gamma_k\}_k) &= \texttt{Filter}(\lambda\gamma.\texttt{Check}_i(n, \gamma), \\
&\qquad \texttt{Flatten}(\texttt{Map}(\lambda\gamma.\texttt{Neighbors}(\gamma, n.\text{axis}, n.\pi_{\text{pos}}), \{\gamma_k\}_k))) \\
[\![\phi]\!]_i &= \texttt{Same as in Fig. 7} \\
\texttt{Check}_i(n, \gamma) &\equiv \gamma.\text{name} == n.\text{name and } \wedge_k [\![n.\pi_{n_k}]\!]_i(\gamma)
\end{aligned}
$$

Fig. 15. The semantics of L_w, where AllNodes, Len and Neighbors are macros with expected semantics.

5.1 Data Extraction Language L_w

Syntax. Fig. 14 shows the syntax of L_w. At the top-level, a program is a tuple containing a name and a list of predicates, where name denotes the HTML tag and predicates denote the constraints that the desired "target" nodes should satisfy. There are two kinds of predicates—NodePred and PathPred. A NodePred can either be an AttrPred or a CountPred. An AttrPred has a name and a value. We treat the text inside a node as yet another attribute. The attribute values are not just strings but are string expressions from the DSL L_u in Fig. 6, which allow the attributes to be computed using string transformations on the input data. This is one of the key differences between the XPath language and L_w. A CountPred indicates the number of neighbors of a node along a particular direction. Predicates can also be PathPreds denoting existence of a particular path in the HTML document starting from the current node. A path is a sequence of nodes where each node has a name, an axis, a PosPred, and a list of NodePreds (possibly empty). The name denotes the HTML tag, axis is the direction of this node from the previous node in the path, and PosPred denotes the distance between the node and the previous node along the axis. A PosPred can also be empty ($\perp$) meaning that the node can be at any distance along the axis. In L_w, we only consider paths that have at-most one node along the Ancestor axis (n_a) and at-most one sibling node along the Left or the Right axis (n_s). Moreover, the ancestor and sibling nodes can only occur at the beginning of the path.

Semantics. Fig. 15 shows the semantics of L_w. A program P is evaluated under a given input data i, on an HTML webpage w, and it produces a list of HTML nodes that have the same name and satisfy all the predicates in P. In this formulation, we use γ to represent an HTML node, and it is not to be confused with the node n in the DSL. A predicate is evaluated on an HTML node and results in a Boolean value. Evaluating an AttrPred checks whether the value of the attribute in the HTML node matches the evaluation of the string expression under the given input i. A CountPred verifies that the number of children of the HTML node along the axis (obtained using the Neighbors macro) matches the count k. A PathPred first evaluates the path which results in a list of HTML nodes and checks that the list is not empty. A path is evaluated step-by-step for each node, where each step evaluation is based on the set of HTML nodes obtained from the previous steps. Based on the axis of the node in the current step evaluation, the set of HTML nodes is expanded to include all their neighbors along that axis and at a position as specified by the PosPred. Next, this set is filtered according to the name of the node and its node predicates (using the Check macro).

$$\text{LearnTarget}(\gamma) \quad = \quad \text{LearnAnchor}(\gamma); \text{LearnChildren}(\gamma); \text{LearnSiblings}(\gamma); \text{LearnAncestors}(\gamma)$$

$$\text{LearnChildren}(\gamma) \quad = \quad \forall \gamma' \text{ in Neighbors}(\gamma, \text{Child}).\ \text{LearnAnchor}(\gamma'); \text{LearnChildren}(\gamma')$$

$$\text{LearnSiblings}(\gamma) \quad = \quad \forall \gamma' \text{ in Neighbors}(\gamma, \{\text{Left}, \text{Right}\}).\ \text{LearnAnchor}(\gamma'); \text{LearnChildren}(\gamma')$$

$$\text{LearnAncestors}(\gamma) \quad = \quad \forall \gamma' \text{ in Neighbors}(\gamma, \text{Ancestor}).\ \text{LearnAnchor}(\gamma'); \text{LearnSiblings}(\gamma')$$

$$\text{LearnAnchor}(\gamma) \quad = \quad \text{Anchor}(\gamma.\text{name}, \text{LearnPreds}(\gamma))$$

$$\text{LearnAttrPred}(\gamma) \quad = \quad \forall \text{attr in } \gamma.\ \text{AttrPred}(\text{attr.name}, \text{GenDag}(\text{attr.value}))$$

$$\text{LearnCountPred}(\gamma) \quad = \quad \forall \text{dir}.\ \text{CountPred}(\text{dir}, \text{Len}(\text{Neighbors}(\gamma, \text{dir})))$$

Fig. 16. Algorithm to transform an HTML document into a predicates graph.

Example. A possible program for the currency exchange rate extraction in Ex 1.1 is (td, [(td, Left,[pos == 1])/ (text,Child,[attr("text") == ⟨**Transformed Date**⟩])])]). This expression denotes the extraction of an HTML node (γ_1) with a td tag. The path predicate states that there should be another td HTML node (γ_2) to the left of γ_1 at a distance of 1 and it should have a text child with its text attribute equal to the transformed date that is learned from the input.

Design choices. This DSL is only a subset of the XPath language that has been chosen so that it can handle a wide variety of data extraction tasks and at the same time, enables an efficient synthesis algorithm. For example, our top-level program is a single node whereas the XPath language would support arbitrary path expressions. Moreover, paths in XPath do not have to satisfy the ordering criteria that L_w enforces and in addition, the individual nodes in the path expression in L_w cannot have recursive path predicates. However, we found that most of these constraints can be expressed as additional path predicates in the top-level program and hence, does not restrict the expressiveness.

5.2 Synthesis Algorithm

We now describe the synthesis algorithm for learning a program in L_w from examples. Here, the list of input-output examples is denoted as $E = \{(i_1, w_1, \gamma_1), (i_2, w_2, \gamma_2), \cdots (i_m, w_m, \gamma_m)\}$, where each example is a tuple of an input i, a web page w, and a target HTML node γ, and the list of pairs of unseen inputs and web pages is denoted as $U = \{(i_{m+1}, w_{m+1}), (i_{m+2}, w_{m+2}), \cdots, (i_{m+n}, w_{m+n})\}$. In this case, the O-PBE synthesis task can be framed as a search problem to find the right set of predicates ($\{\pi_1, \pi_2, \cdots, \pi_r\}$) that can sufficiently constrain the given target example nodes.

At a high-level, the synthesis algorithm has three key steps: First, it uses the first example to learn all possible predicates for the target node in that example. Then, the remaining examples are used to refine these predicates. Finally, the algorithm searches for a subset of these predicates that uniquely satisfies the given examples and also generalizes well to the unseen inputs.

5.2.1 Learning all predicates. For any given example HTML node, there are numerous predicates (especially path predicates) in the language that constrain the target node. In order to learn and operate on these predicates efficiently, we use a graph data structure called *predicates graph* to compactly represent the set of all predicates. This data structure is inspired by the tree data structure used in Anton [2005], but it is adapted to our DSL and algorithm.

Similar to Anton [2005], to avoid confusion with the nodes in the DSL, we use the term Anchor to refer to nodes and the term Hop to refer to edges in this graph. Hence, a predicates graph is a tuple (A, H, T) where A is the list of anchors, H is the list of hops and $T \in A$ is the target anchor. An anchor is a tuple (n, P) where n is the name of the anchor and P is a list of node predicates in the L_w language. An edge is a tuple (a_1, a_2, x, d) where a_1 is the start anchor, a_2 is the end anchor,

x is the axis and d is a predicate on the distance between a_1 and a_2 measured in terms of number of hops in the original HTML document.

Fig. 16 shows the algorithm for transforming an HTML document into a predicates graph. We will explain this algorithm based on an example shown in Fig. 17 where the input HTML document is shown on the left, and the corresponding predicates graph is shown on the right; the target node is the text node (T_3) shown in red. First, it is important to note the difference between these two representations. Although both the HTML document and the predicates graph have a tree like structure, the latter is more centered around the target anchor. In the predicates graph, all anchors are connected to the target anchor using a minimum number of intermediate anchors that is allowed by the DSL. The algorithm first creates an anchor for the target and then learns the anchors for its children, siblings, and ancestors recursively. Learning a child or a sibling will also learn its children in a recursive manner, whereas learning an ancestor will only learn its siblings recursively. Finally, when creating an anchor for an HTML node, all the node predicates of the HTML node are inherited by the anchor, but if there are any attribute predicates, their string values are first converted to DAGs using the GenDag method in § 4.2.2.

After the above transformation, a path $p = a_1/a_2/\cdots a_r$ in the predicates graph (where a_1 is the target node) represents many different predicates in L_w corresponding to different combinations of the node predicates in each anchor a_k. We use $\texttt{predicates}(p)$ to denote the set of all such predicates, e.g. for the path from the target (T_3) to the text node T_2 in Fig. 17, $\texttt{predicates}(p) = \{\pi_{p_1}, \pi_{p_2}, \pi_{p_3} \cdots\}$ where:

$\pi_{p_1} = $ [(p, Ancestor, [pos == 1])/(div, Left, [pos == 1])/ (text, Child, [attr("text") = dag$_2$.BestProg])]

$\pi_{p_2} = $ [(p, Ancestor, [pos == 1])/(div, Left)/(text, Child, [attr("text") = dag$_2$.BestProg])]

$\pi_{p_3} = $ [(p, Ancestor)/(div, Left)/(text, Child)]

We can define a partial ordering among predicates generated by a path p as follows: $\pi_1 \sqsubseteq \pi_2$ if the set of all node predicates in π_1 is a subset of the set of all node predicates in π_2. In the above example, we have $\pi_{p_3} \sqsubseteq \pi_{p_2} \sqsubseteq \pi_{p_1}$.

Definition 5.1 (Minimal path predicate). Given a path p in the predicates graph, a minimal path predicate is the predicate π_p encoded by this path such that $\nexists \pi_{p'}. \pi_{p'} \sqsubseteq \pi_p$.

Definition 5.2 (Maximal path predicate). Given a path p in the predicates graph, a maximal path predicate is the predicate π_p such that $\nexists \pi_{p'}. \pi_p \sqsubseteq \pi_{p'}$

In other words, a *minimal path predicate* is the one that does not have any node predicates for any anchor in the path and a *maximal path predicate* is the one that has all node predicates for every anchor in the path. For the above example, π_{p_3} is the *minimal path predicate* and π_{p_1} is the *maximal path predicate* assuming the nodes do not have any other predicates.

LEMMA 5.3. *Any predicate expressed by the predicates graph, Π, for an example (i, w, γ) will satisfy the example i.e.* $\forall \text{path } p \in \Pi. \forall \pi \in \texttt{predicates}(p). [\![\pi]\!]_i(\gamma) = \texttt{True}.$

Proof Sketch: This lemma is true because LearnTarget constructs the predicates graph by only adding those nodes and predicates that are in the original HTML document.

LEMMA 5.4. *The predicates graph, Π, for an example (i, w, γ) can express all predicates in L_w that satisfy the example i.e.* $\forall \pi \in L_w. [\![\pi]\!]_i(\gamma) = \texttt{True} \implies \exists \text{path } p \in \Pi. \pi \in \texttt{predicates}(p).$

Proof Sketch: This lemma is true because the traversal of nodes when constructing the predicates graph covers all possible paths in the HTML document that can be expressed in L_w.

For the implementation, we only construct a portion of the predicates graph that captures nodes that are within a distance $r = 5$ from the target node.
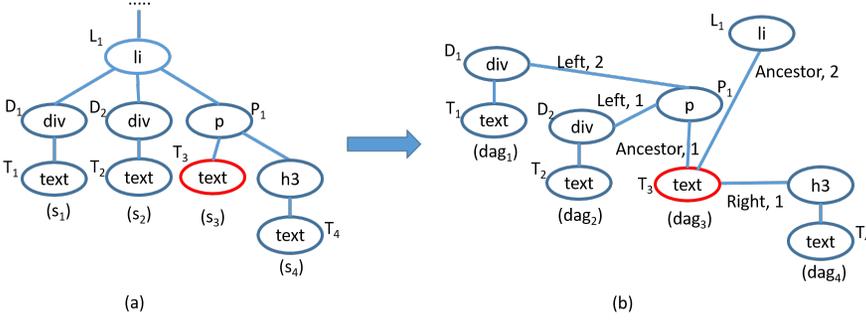
Fig. 17. An example demonstrating the transformation from an HTML document to a predicates graph.

$$
\begin{aligned}
\text{IntersectPath}(p, q) &= \{a_k\}_k \text{ where } a_k = \text{IntersectAnchor}(a_{p_k}, a_{q_k}) \\
\text{IntersectAnchor}(a_p, a_q) &= \text{Anchor}(a_p.\text{name}, \{\pi_k\}_k) \text{ where } \pi_k = \text{IntersectPred}(\pi_{p_k}, \pi_{q_k}) \\
\text{IntersectAttrPred}(\pi_p, \pi_q) &= \text{AttrPred}(\pi_p.\text{name}, \text{Intersect}(\pi_p.\text{DAG}, \pi_q.\text{DAG})) \text{ if } \pi_p.\text{name} = \pi_q.\text{name} \\
\text{IntersectPosPred}(\pi_p, \pi_q) &= \text{PosPred}(==, \pi_p.k) \text{ if } \pi_p.k = \pi_q.k \text{ else PosPred}(\le, \max(\pi_p.k, \pi_q.k)) \\
\text{IntersectCountPred}(\pi_p, \pi_q) &= \text{CountPred}(\pi_p.k) \text{ if } \pi_p.k = \pi_q.k
\end{aligned}
$$

Fig. 18. Algorithm to intersect two paths in two predicates graphs.

5.2.2 Handling multiple examples. We, now, have a list of all predicates that constrain the target HTML node for the first example. However, not all of these predicates will satisfy the other examples provided by the user. A simple strategy to prune the unsatisfiable predicates is to create a predicates graph for each example and perform a full intersection of all these graphs. However, this operation is very expensive and has a complexity of N^m where N is the number of nodes in the HTML document and m is the number of examples. Moreover, in the next subsection, we will see that the algorithm for searching the right set of predicates (Fig. 20) will try to add these predicates one-by-one based on a ranking scheme and stops after a satisfying set is obtained. Therefore, our strategy is to refine predicates in the predicates graph in a lazy fashion for one path at a time (rather than the whole graph) when the path is required by the SearchBestProg algorithm.

We will motivate this refinement algorithm using the example from Fig. 17. Suppose that the first example E_1 in this scenario has $i_1 =$ "10/16/16" and w_1 as shown in Fig. 17(a) with $s_{1_1} =$ "10-16-2016" and $s_{2_1} =$ "foo". As described earlier, one of the possible predicates for this example is: $\pi = $ [(p,Ancestor,[pos==1])/(div,Left,[pos==1])/ (text,Child, [attr("text")=dag$_2$.BestProg])]. Now, suppose that the best program of dag$_2$ extracts the date (16) in the text of s_{1_1} from the year (16) (rather than the date 16) in the input i_1. Also, assume that the second example E_2 has $i_2 =$ "10/15/16"and w_2 similar to w_1 but with $s_{1_2} =$ "bar" and $s_{2_2} =$ "10-15-2016". Clearly, the predicate π will not satisfy this new example and hence, we need to refine the path for π using the other examples. The path for π, $p = T_{3_1}/P_{1_1}/D_{2_1}/T_{2_1}$, is shown in Fig. 19(a). The algorithm for refining the path is done iteratively for one example at a time. For any new example E_k, the algorithm will first check if the maximal path predicate of p satisfies the new example. If it does, then there is no need to refine this path. Otherwise, the algorithm gets all paths in the predicates graph corresponding to E_k that satisfies the minimal path predicate of p. For the example E_2, there are two such paths as shown in Fig. 19(b). The algorithm will then intersect p with each of these paths.
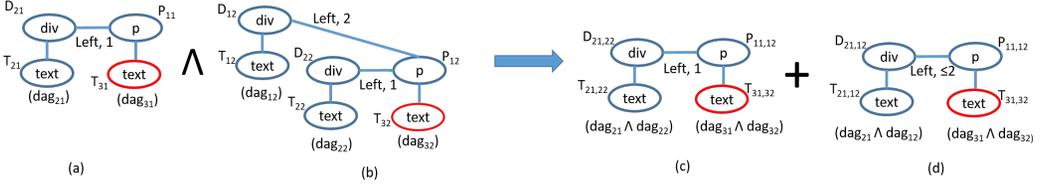
Fig. 19. Example demonstrating refining a path in predicates graph using other examples.

```
SearchBestProg(Π, E, U)
1   Π′ = φ; RankP(Π)
2   foreach π in Π.SortedIterator:
3       𝒫 = Refine(Path(π), E)
4       foreach π′ in 𝒫
5           done = TestNAdd(π′, Π′, E, U)
6           if (done): return (E[0].γ.Name, Π′)
```

Fig. 20. The algorithm for computing the best predicate set using output-constrained ranking.

Fig. 18 shows the algorithm for intersecting two paths. The algorithm goes through all the anchors in the two paths and intersects each of their node predicates. Note that, since the two paths have the same minimal path predicate, the anchors in the two paths will have the same sequence of tags. For intersecting attribute predicates, the algorithm will intersect their respective DAGs corresponding to the values if their attributes have the same name. For intersecting position predicates, the algorithm will take the maximum value of k and update the operation accordingly. For intersecting count predicates, the algorithm will return the same predicate if the counts (k) are the same. For the example in Fig. 19, Fig. 19(c)-(d) are the two new resulting paths after the intersection, whose predicates are:

$\pi_1' = [(p, \text{Ancestor}, [\text{pos}==1])/(\text{div}, \text{Left}, [\text{pos}==1])/(\text{text}, \text{Child}, [\text{attr}("text")=(\text{dag}_{2_1} \wedge \text{dag}_{2_2}).\text{TopProg}])]$

$\pi_2' = [(p, \text{Ancestor}, [\text{pos}==1])/(\text{div}, \text{Left}, [\text{pos}\leq 2])/(\text{text}, \text{Child}, [\text{attr}("text")=(\text{dag}_{2_1} \wedge \text{dag}_{1_2}).\text{TopProg}])]$

The worst case complexity of this lazy refinement strategy is N^m, but in practice, it works well because the algorithm usually accesses few paths in the predicates graph because of the ranking scheme and moreover, the path intersection is only required if the original path cannot satisfy the new examples.

The following lemmas regarding the intersection algorithm hold true by construction. Let p, q be two paths in the predicates graphs of two different examples $\{(i_1, w_1, \gamma_1), (i_2, w_2, \gamma_2)\}$ that have the same minimal path predicate, and let r be the path obtained after the intersection.

LEMMA 5.5. *Any predicate obtained after intersecting two paths satisfies all the examples. Formally,* $\forall \pi \in \text{predicates}(r). [\![\pi]\!]_{i_1}(\gamma_1) = \text{True} \wedge [\![\pi]\!]_{i_2}(\gamma_2) = \text{True}.$

LEMMA 5.6. *Intersecting two paths preserves all predicates that satisfy the examples. Formally,* $\forall \pi. \pi \in \text{predicates}(p) \wedge \pi \in \text{predicates}(q) \implies \pi \in \text{predicates}(r).$

5.2.3 Output-constrained ranking of predicates. We now have a list of predicates that satisfy the first example and we have an algorithm to refine predicates based on the other examples provided by the user. However, only some of these predicates are desirable as some predicates might not

generalize to unseen inputs and some others might not be required to constrain the target nodes. Fig. 20 shows the algorithm that uses *output-constrained ranking* to find a subset of the predicates that generalizes well. The algorithm takes as inputs a set of predicates Π, a list of input-output examples E, and a list of unseen inputs U. It uses the unseen inputs as a test set to prune away predicates that do not generalize well. The algorithm iterates through the list of all predicates based on a ranking scheme and adds the predicate if there is at-least one node in each test document that satisfies the predicates added so far. This process is stopped if we find a set of predicates that uniquely constrains the target nodes in the provided examples.

Ranking Scheme. We use the following criterion that gives higher priority to predicates that best constrain the set of target nodes and at the same time, capture user intentions: (i) Attribute predicates with more sub-string expressions on input data are preferred over other attribute predicates, (ii) Left siblings nodes are preferred over right siblings because usually websites contain descriptor information to the left of values and in most cases, these descriptor information tend to be the same across websites, and (iii) Nodes closer to target are preferred over farther nodes.

THEOREM 5.7 (SOUNDNESS). *The data extraction synthesis algorithm is sound i.e. given some input-output examples $\{(i_k, w_k, \gamma_k)\}_k$, the program P that is learned by the algorithm will always satisfy $\forall k. [\![P]\!]_{i_k}(w_k) = \{\gamma_k\}$.*

Proof Sketch: This theorem follows directly from Lemmas 5.3 and 5.5.

THEOREM 5.8 (COMPLETENESS). *The data extraction synthesis algorithm is complete i.e. if there exists a program in L_w that satisfies the given I/O examples, then the algorithm is guaranteed to find one program that satisfies the examples.*

Proof Sketch: This holds because the predicates graphs created for the examples are complete using Lemma 5.4. And the algorithm for refining a path intersects the path with all similar paths (that satisfy the minimal path predicate) in the other examples and `IntersectPath` preserves all predicates that can be satisfied by the examples (Lemma 5.6).

Note, that the `SearchBestProg` algorithm cannot influence the soundness or completeness argument because the set of all predicates obtained after refining every path in the predicates graph is a sound and complete solution. `SearchBestProg` only influences how well it generalizes to unseen inputs.

A Note on adding new rows and changing data sources. When adding new rows to the spreadsheet after the integration task has been performed, there might be concerns about whether the joined data for the old rows will be obsolete. For example, in Ex 2.1 or Ex 2.2, the stock values or the weather information would change presumable every second. However, the user still does not need to re-provide updated examples. This is because WEBRELATE learns programs in the DSLs and it can just re-execute the learned program directly and compute results for the new rows. Only in cases if the website DOM structure changes (which does not happen too often for major websites), the user would need to re-update the previously provided examples.

6   EVALUATION

In this section, we evaluate WEBRELATE on a variety of real-world web integration tasks. In particular, we evaluate whether the DSLs for URL learning and data extraction are expressive enough to encode these tasks, the efficiency of the synthesis algorithms, the number of examples needed to learn the programs, and the impact of layered version spaces and output-constrained ranking.

Benchmarks. We collected 88 web-integration tasks taken from online help forums and the Excel product team. These tasks cover a wide range of domains including finance, sports, weather, travel, academics, geography, and entertainment. Each benchmark has 5 to 32 rows of input data in the spreadsheet. These 88 tasks decompose into 62 URL learning tasks and 88 extraction tasks. For some benchmarks, we had scenarios with examples and webpages provided by the Excel team. For other benchmarks, we chose alternate sources for data extraction and provided examples manually, but they were independent of the underlying learning system. The set of benchmarks with unique URLs is shown in Fig. 21.

Experimental Setup. We implemented WebRelate in C#. All experiments were performed using a dual-core Intel i5 2.40 GHz CPU with 8GB RAM. For each component, we incrementally provided examples until the system learned a program that satisfied all other inputs, and we report the results of the final iteration.

6.1 URL learning

For each URL learning task, we run the layered search using 4 different configurations for the layers: 1. L1 to L4, 2. L2 to L4, 3. L3 to L4, and 4. Only L4 where L1, L2, L3, and L4 are as defined in Fig. 10. The last configuration essentially compares against the VSA algorithm used in FlashFill (except for the new constructs).

Fig. 22(a) shows the synthesis times[2] required for learning URLs. We categorize the benchmarks based on the layer that has the program required for performing the transformation. We can see that for the L1 to L4 configuration, WebRelate solves all the tasks in less than 1 second. The L2 to L4 configuration performs equally well whereas the performance of only L4 configuration is much worse. Only 28 benchmarks complete when given a timeout of 2 minutes. Note that none of the URL learning tasks need the L4 configuration in our benchmarks, but we still allow for using L4 for two reasons: i) completeness of the synthesis algorithm for our DSL for other benchmarks in future, and ii) comparison against a no layered approach (a baseline similar to FlashFill like VSA algorithms).

For these tasks, the length of the URLs is in the range of 23 to 89. Regarding DSL coverage, about 50% of the benchmarks require AnyStr, 88% require SubStr, 38% require Replace, and all benchmarks require ConstStr expressions. Thus, all DSL features are needed for different subsets of benchmarks. Moreover, note that the 50% of the benchmarks that require AnyStr expressions cannot be learned by traditional PBE techniques and hence, require the O-PBE formulation.

We also perform an experiment to evaluate the impact on generalization with the *output-constrained ranking* scheme. For this experiment, we use the L1 to L4 configuration for the layered search and report the results in Fig. 22(b). With output-constrained ranking, 85% of the benchmarks require only 1 example whereas without it, only 29% of benchmarks can be synthesized using a single example. Note that the L1 to L4 layered search already has a strong prior that the programs in earlier layers are more desirable, and the output-constrained ranking is able to further reduce the number of examples required.

To evaluate the scalability with respect to the number of examples, we performed an experiment where we took a benchmark that has 32 rows in the spreadsheet and incrementally added more examples. The results are shown in Fig. 24(a). Although the theoretical complexity of the algorithms is exponential in the number of examples, in practice, we observed that the performance scales almost linearly. We attribute this to the sparseness of the DAGs learned during the layered search.

---

[2]excluding the time take to run search queries on the search engine.

#	Description	#R	Example data item(s)	Website Domain
1-2	ATP players to ages/countries	5	(age 29)	Serbia and Montenegro	wikipedia
3-4	ATP players to latest tweet/total tweets	5	Long tweet message	2,324	twitter
5-7	ATP players to single titles/ranking/W-L	20	7	ATP Rank #1	742-152	espn
8	ATP players to number of singles based on year	7	7	espn.go
9	ATP players to rankings	5	1	atpworldtour
10	ATP players to rankings	5	#1	tennis
11	ATP players to career titles	5	66 (7th in the Open Era)	wikipedia
12	Addresses to population	7	668,342 (100% urban, 0% rural).	city-data
13-14	Addresses to population/zipcode	7	786,130	98101 Zip Code	zipcode
15-16	Addresses to weather/weather based on date	7	51°	51°	accuweather
17	Addresses to weather	7	57°F	timeanddate
18	Addresses to weather	7	52	weather
19-23	Addresses to weather stats	6	49|Partly sunny|74°|62|68°	accuweather
24	Airport code to airport name	5	Boston Logan International Airport (BOS)	virginamerica
25-26	Airport code to delay status/name	5	Very few delays	Logan International	flightstats
27	Airport code to terminal information	5	Terminal B - Gates B4 - B19	aa
28	Albums to genre	5	Rock	wikipedia
29-31	Authors to paper citations, max citation, title	5	Cited by 175	Cited by 427	Syntax-guided	scholar.google
32-35	Authors to different data from DBLP	5	Demo...|Alvin Cheung(12)|PLDI(9)|POPL(2)	dblp
36	Cities to population	6	21,357,000	worldpopulation
37	Company symbols to 1 year target prices	6	65	nasdaq
38	Company symbols to stock prices	6	59.87	finance.yahoo
39	Company symbols to stock prices	6	59.87	money.cnn
40	Company symbols to stock prices	6	59.87	quotes.wsj
41	Company symbols to stock prices	6	59.87	google
42	Company symbols to stock prices	6	59.00	marketwatch
43	Company symbols to stock prices	6	59.95	nasdaq
44	Company symbols to stock prices	6	$59.87	google
45	Company symbols to stock prices on date	6	60.61	google
46	Company symbols to stock prices on date	6	60.61	yahoo
47	Country names to population	6	1,326,801,576	worldometers
48	Country names to population	6	1,336,286,256	wikipedia
49	Cricket results for teams on different dates	5	Australia won by 3 wickets	espncricinfo
50-52	Cricket stats for two different teams	5	90	24	31.67	espncricinfo
53	Currency exchange values	8	66.7870	finance.yahoo
54	Currency exchange values	8	INR/USD = 66.84699	moneyconverter
55	Currency exchange values	8	66.7800	bloomberg
56	Currency exchange values	8	66.8043 INR	investing
57	Currency exchange values	8	66.779	xe
58	Currency exchange values	8	66.7914	exchange-rates
59	Currency exchange values based on date	8	64.1643 INR	investing
60	Currency exchange values based on date	8	66.778	exchange-rates
61	Currency exchange values based on date	8	64.231	investing
62	Flight distance between two cities	5	2,487 Miles	cheapoair
63-65	Flight fares between two cities/cheapest/airline	5	$217	Wednesday	$237	farecompare
66	Flight fares between two cities	5	$257	cheapflights
67	Flight travel time between two cities	5	6 hrs 11 mins	travelmath
68	Flight travel time between two cities	5	5 hours, 38 minutes	cheapflights
69	NFL teams to rankings	5	4-5, 3rd in AFC East	espn
70	NFL teams to rankings	5	26th	cbssports
71	NFL teams to rankings	5	26.7	nfl
72	NFL teams to rankings	5	#5	teamrankings
73	NFL teams to stadium names	32	New Era Field	espn
74	Nobel prize for different subjects based on year	5	Richard F. Heck	wikipedia
75	Nobel prize for different years based on subject	5	Richard F. Heck	nobelprize
76	Nobel prize for different years based on subject	5	Richard F. Heck	nobelprize
77-78	Novels to authors/genre	7	Charles Dickens	Historical novel	wikipedia
79	Novels to authors	8	Charles Dickens	goodreads
80	# of coauthored papers between two authors	10	Rastislav Bodik (6)	dblp
81	Number of daily flights between two cities	5	26	cheapoair
82	People to profession	5	Principal Researcher	zoominfo
83-84	Real estate properties to sale prices and stats	5	$1,452	1,804 sqft	zillow
85	Stock prices with names from same webpage	6	7.60	marketwatch
86-88	Video names to youtube stats	5	11,473,055	2,672,818,718 views	Jul 15, 2012	youtube

Fig. 21. The set of web data integration benchmarks with a brief description along with number of input rows (#R), the example data item(s), and the website domain from which the data is extracted.
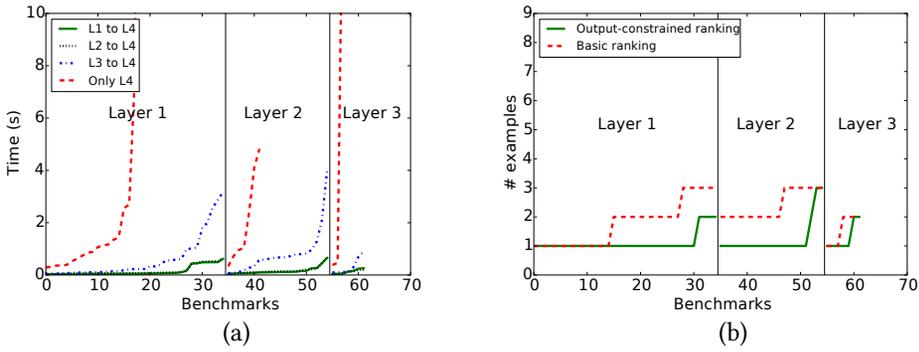
Fig. 22. (a) The synthesis times and (b) the number of examples required for learning URL programs. The benchmarks are categorized based on the layer that contains the program required for performing the transformation.
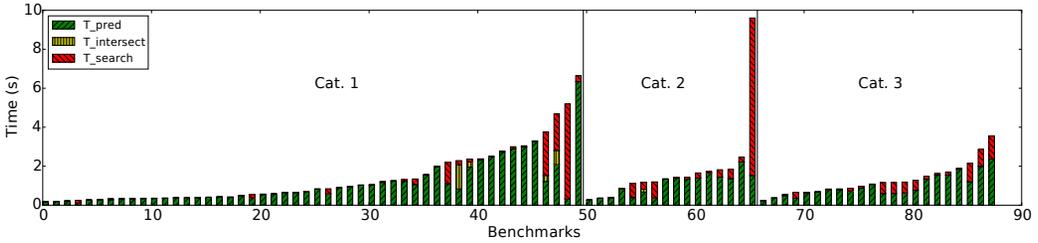

Fig. 23. The synthesis times for learning data extraction programs by WebRelate on the 88 benchmarks.
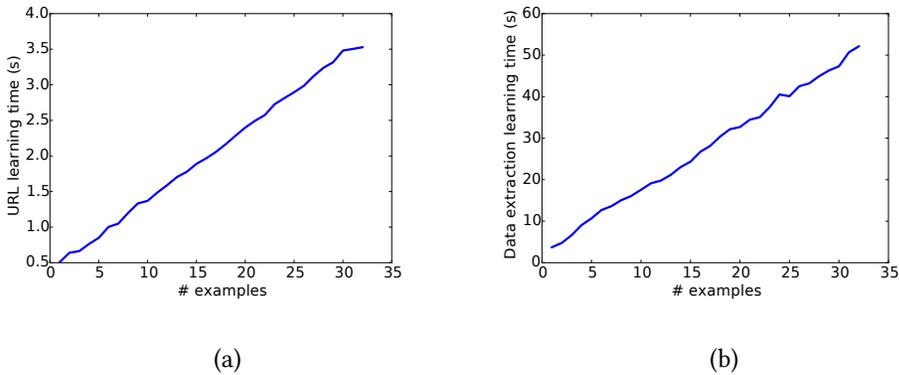

Fig. 24. The synthesis time vs the number of examples for URL learning (a) and data extraction learning (b).

6.2 Data Extraction

We now present the evaluation of our data extraction synthesizer. We categorize the benchmarks into three categories. The first category consists of benchmarks where the data extraction can be learned using an absolute XPath. The second category includes the benchmarks that cannot be learned using an absolute XPath, but by using relative predicates (especially involving nearby strings). The last category handles the cases where the data extraction is input-dependent.
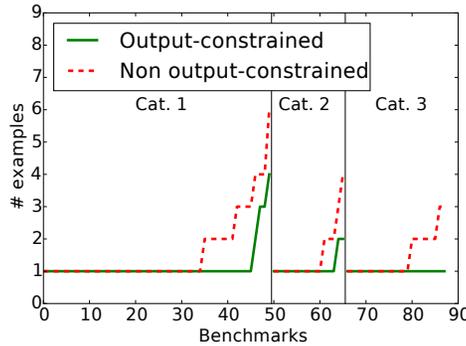
Fig. 25. The number of examples needed to learn the data extraction programs.

Fig. 23 shows the synthesis times[3] for all 88 benchmarks. The figure also splits the synthesis time into the time taken for learning the predicates graphs (T_pred), for intersecting the predicates graphs if there are multiple examples (T_intersect) and finally, for searching the right set of predicates (T_search). WebRelate can learn the correct extraction for all the benchmarks in all the three categories showing that the DSL is very expressive. Moreover, it can learn them very quickly— 97% of benchmarks take less than 5 seconds. Fig. 25 shows the number of examples required and also compares against non output-constrained ranking. It can be seen that with output-constrained ranking, 93% of benchmarks require only 1 example.

Most of the synthesis time is actually spent in generating the predicates graphs and this time is proportional to the number of attribute strings in the HTML document since WebRelate will create a DAG for each string. In our examples, the number of HTML nodes we consider for the predicates graph is in the range of 30 to 1200 with 3 to 200 attribute strings. For some benchmarks, the time spent in searching for the right set of predicates dominates. These are the benchmarks that require too many predicates to sufficiently constrain the target nodes. The actual time spent on intersecting the predicates graph is not very significant. For benchmarks that require multiple examples, we found that the time spent on intersection is only 15% of the total time (on average) due to our lazy intersection algorithm.

Fig. 24(b) shows how the performance scales with respect to the number of examples. Similar to URL learning, the performance scales almost linearly as opposed to exponentially. We attribute this to our lazy intersection algorithm.

7   RELATED WORK

Data Integration from Web: DataXFormer [Abedjan et al. 2015; Morcos et al. 2015] performs semantic data transformations using web tables and forms. Given a set of input-output examples, it searches a large index of web tables and web forms to find a consistent transformation. For web forms, it performs a web search consisting of input-output examples to identify web forms that might be relevant, and then uses heuristics to invoke the form with the correct set of inputs. Instead of being completely automated, WebRelate, on the other hand, allows users to identify the relevant websites and point to the desired data on the website, which allows WebRelate to perform data integration from more sophisticated websites. Moreover, WebRelate allows for joining data based on syntactic transformations on inputs, whereas DataXFormer only searches based on exact inputs.

---

[3]excluding the time taken to load the webpages

WebCombine [Chasins et al. 2015] is a PBD web scraping tool for end-users that allows them to first extract logical tables from a webpage, and then provide example interactions for the first table entry on another website. It uses Ringer [Barman et al. 2016] as the backend record and replay engine to record the interactions performed by the user on a webpage. The recorded interaction is turned into a script that can be replayed programmatically. WebCombine parameterizes the recorded script using 3 rules that parameterize xpaths, strings, and frames with list items. Vegemite [Lin et al. 2009] uses direct manipulation and PBD to allow end-users to easily create web mashups to collect information from multiple websites. It first uses a demonstration to open a website, copy/paste the data for the first row into the website, and records user interactions using the CoScripter [Leshed et al. 2008] engine. The learnt script is then executed for remaining rows in the spreadsheet. The XPath learning does not need to be complex since all rows go to the same website and the desired data is at the same location in the DOM.

Instead of recording user demonstrations, WEBRELATE uses examples of URLs (or search queries) to learn a program to get to the desired webpages. The demonstrations-based specification has been shown to be challenging for users [Lau 2009] and many websites do not expose such interactions. These systems learn simpler XPath expressions for data extraction, whereas WEBRELATE can learn input data-dependent Xpath expressions that are crucial for data integration tasks. Moreover, these systems assume the input data is in a consistent format that can be directly used for interactions, whereas WEBRELATE learns additional transformations on the input for both learning the URLs and data-dependent extractions.

Programming By Examples (PBE) for string transformations: Data Wrangler [Kandel et al. 2011] uses examples and predictive interaction [Heer et al. 2015] to create reusable data transformations such as map, joins, aggregation, and sorting. There have also been many recent PBE systems such as FlashFill [Gulwani 2011], BlinkFill [Singh 2016], and FlashExtract [Le and Gulwani 2014] that use VSA [Polozov and Gulwani 2015] for efficiently learning string transformations from examples. Unlike these systems that perform data transformation and extraction from a single document, WEBRELATE joins data between a spreadsheet and a collection of webpages. WEBRELATE builds on top of the substring constructs introduced by these systems to perform both URL learning and data extraction. Moreover, WEBRELATE uses layered version spaces and a output-constrained ranking technique to efficiently synthesize programs.

There is another PBE system that learns relational joins between two tables from examples [Singh and Gulwani 2012]. It uses a restricted set of relational algebra to allow using VSA for efficient synthesis. Unlike learning joins between two relational sources, WEBRELATE learns joins between a relational data source (spreadsheet) and a semi-structured data source (multiple webpages).

Wrapper Induction: Wrapper induction [Kushmerick 1997] is a technique to automatically extract relational information from webpages using labeled examples. There exists a large body of research on wrapper induction with some techniques using input-output examples to learn wrappers [Anton 2005; Dalvi et al. 2009; Hsu and Dung 1998; Kushmerick 1997; Le and Gulwani 2014; Muslea et al. 1998] and others [Chasins et al. 2015; Crescenzi et al. 2001; Zhai and Liu 2005] perform unsupervised learning using pattern mining or similarity functions. Some of these techniques [Anton 2005; Dalvi et al. 2009; Le and Gulwani 2014] are based on Xpath similar to WEBRELATE whereas some techniques such as [Kushmerick 1997] treat webpages as strings and learn delimiter strings around the desired data from a fixed class of patterns. The main difference between any of these techniques and WEBRELATE is that the extraction language used by WEBRELATE is more expressive as it allows richer Xpath expressions that can depend on inputs.

Program Synthesis: The field of program synthesis has seen a renewed interest in recent years [Alur et al. 2013; Gulwani et al. 2017]. In addition to VSA based approaches, several other approaches including constraint-based [Solar-Lezama et al. 2006], enumerative [Udupa et al. 2013],

stochastic [Schkufza et al. 2013], and finite tree automata based techniques [Wang et al. 2017, 2018] have been recently developed to synthesize programs in different domains. Synthesis techniques using examples have also been developed for learning data structure manipulations [Singh and Solar-Lezama 2011], type-directed synthesis for recursive functions over algebraic datatypes [Frankle et al. 2016; Osera and Zdancewic 2015], transforming tree-structured data [Yaghmazadeh et al. 2016], and interactive parser synthesis [Leung et al. 2015]. These approaches are not suited for learning URLs and data extraction programs because the DSL operators such as regular expression based substrings and data-dependent xpath expressions are not readily expressible in these approaches and enumerative approaches do not scale because of large search space.

8   LIMITATIONS AND FUTURE WORK

There are certain situations under which WebRelate may not be able to learn a data integration task. First, since all of our synthesis algorithms are sound, WebRelate cannot deal with noise. For example, if any input data in the spreadsheet is misspelled or has a semantically different format, then WebRelate may not be able to learn such string transformation programs. Our system can handle syntactic data transformations and partial semantic transformations, but not fully semantic transformations. For instance, in Ex 4.3 if the input country name was "US" instead of "United States", then WebRelate would not be able to learn such programs. As future work, we plan to use recent neural program synthesis techniques [Devlin et al. 2017; Parisotto et al. 2017; Singh and Kohli 2017] to learn a noise model based on semantics and web tables [He et al. 2015], and incorporate this model into the synthesis algorithm to make it more robust with respect to noise, small web page discrepancies, and different semantic data formats. In WebRelate, we assume that the webpages containing the desired data have URLs that can be programmatically learned. However, there are also situations that require complex interactions such as traversing through multiple pages possibly by interacting with the webpage UI before getting to the page that has the desired data. In future, we want to explore integrating the techniques in this paper with techniques from record and replay systems such as Ringer [Barman et al. 2016] to enable data-dependent replay.

9   CONCLUSION

We presented WebRelate, a PBE system for joining semi-structured web data with relational data. The key idea in WebRelate is to decompose the task into two sub-tasks: URL learning and data extraction learning. We frame the URL learning problem in terms of learning syntactic string transformations and filters, whereas we learn data extraction programs in a rich DSL that allows for data-dependent Xpath expressions. The key idea in the synthesis algorithms is to use layered version spaces and output-constrained ranking to efficiently learn the programs using very few input-output examples. We have evaluated WebRelate successfully on several real-world web data integration tasks.

REFERENCES

Ziawasch Abedjan, John Morcos, Michael N Gubanov, Ihab F Ilyas, Michael Stonebraker, Paolo Papotti, and Mourad Ouzzani. 2015. Dataxformer: Leveraging the Web for Semantic Transformations.. In *CIDR*.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*. 1–8.

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

Shaon Barman, Sarah Chasins, Rastislav Bodík, and Sumit Gulwani. 2016. Ringer: web automation by demonstration. In *OOPSLA*. 748–764.

Anders Berglund, Scott Boag, Don Chamberlin, Mary F Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. 2003. Xml path language (xpath). *World Wide Web Consortium (W3C)* (2003).

Sarah Chasins, Shaon Barman, Rastislav Bodík, and Sumit Gulwani. 2015. Browser Record and Replay as a Building Block for End-User Web Automation Tools. In *WWW*. 179–182.

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

Nilesh Dalvi, Philip Bohannon, and Fei Sha. 2009. Robust Web Extraction: An Approach Based on a Probabilistic Tree-edit Model. In *SIGMOD*. 335–348.

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*. 990–998.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. In *POPL*. 802–815.

Mike Gualtieri. 2009. Deputize end-user developers to deliver business agility and reduce costs. *Forrester Report for Application Development and Program Management Professionals* (2009).

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL*. 317–330.

Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119.

Yeye He, Kris Ganjam, and Xu Chu. 2015. SEMA-JOIN: Joining Semantically-related Tables Using Big Table Corpora. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1358–1369.

Jeffrey Heer, Joseph M Hellerstein, and Sean Kandel. 2015. Predictive Interaction for Data Transformation.. In *CIDR*.

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.

Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *CHI*. 3363–3372.

Nicholas Kushmerick. 1997. *Wrapper induction for information extraction*. Ph.D. Dissertation. Univ. of Washington.

Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4 (2009), 65–67.

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *PLDI*. 542–553.

Gilly Leshed, Eben M Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *CHI*. 1719–1728.

Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive parser synthesis by example. In *PLDI*. 565–574.

James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A Lau. 2009. End-user programming of mashups with vegemite. In *IUI*. 97–106.

John Morcos, Ziawasch Abedjan, Ihab Francis Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2015. DataXFormer: An Interactive Data Transformation Tool. In *SIGMOD*.

Ion Muslea, Steve Minton, and Craig Knoblock. 1998. Stalker: Learning extraction rules for semistructured, web-based information sources. In *Workshop on AI and Information Integration*. AAAI, 74–81.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed Program Synthesis. In *PLDI*. 619–630.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *ICLR*.

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *OOPSLA*. 107–126.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *ASPLOS*. 305–316.

Rishabh Singh. 2016. BlinkFill: Semi-supervised Programming By Example for Syntactic String Transformations. *PVLDB* 9, 10 (2016), 816–827.

Rishabh Singh and Sumit Gulwani. 2012. Learning Semantic String Transformations from Examples. *PVLDB* 5, 8 (2012), 740–751.

Rishabh Singh and Pushmeet Kohli. 2017. AP: Artificial Programming. In *SNAPL*. 16:1–16:12.

Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing data structure manipulations from storyboards. In *FSE*. 289–299.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial
    sketching for finite programs. In *ASPLOS*. 404–415.
Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013.
    TRANSIT: specifying protocols with concolic snippets. In *PLDI*. 287–296.
Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of Data Completion Scripts using Finite Tree Automata. In
    OOPSLA.
Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program Synthesis using Abstraction Refinement. In *POPL*.
Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchi-
    cally structured data. In *PLDI*. 508–521.
Yanhong Zhai and Bing Liu. 2005. Web Data Extraction Based on Partial Tree Alignment. In *WWW*. 76–85.