Automated Data Extraction Using Predictive Program Synthesis

Mohammad Raza  
Microsoft Corporation  
One Microsoft Way  
Redmond, Washington, 98052  
moraza@microsoft.com

Sumit Gulwani  
Microsoft Corporation  
One Microsoft Way  
Redmond, Washington, 98052  
sumitg@microsoft.com

Abstract

In recent years there has been rising interest in the use of programming-by-example techniques to assist users in data manipulation tasks. Such techniques rely on an explicit input-output examples specification from the user to automatically synthesize programs. However, in a wide range of data extraction tasks it is easy for a human observer to predict the desired extraction by just observing the input data itself. Such predictive intelligence has not yet been explored in program synthesis research, and is what we address in this work. We describe a predictive program synthesis algorithm that infers programs in a general form of extraction DSLs (domain specific languages) given input-only examples. We describe concrete instantiations of such DSLs and the synthesis algorithm in the two practical application domains of text extraction and web extraction, and present an evaluation of our technique on a range of extraction tasks encountered in practice.

1 Introduction

With the ever-increasing growth of data in the world, recent years have seen rising interest from both academia and industry in the use of automatic programming techniques to address the problem of data wrangling: the challenge faced by data scientists to cope with data in a multitude of formats from different sources and to bring this raw data into a form that is amenable to their analysis tools. Such data preprocessing is a time-consuming activity (responsible for up to 80% of time in some cases (Kandel et al. 2011)) and most often requires programming skills in order to write robust extraction or transformation scripts.

This is where automated generation of such programs can be extremely beneficial - both to speed up the wrangling process and also to make it accessible to a wide range of data analysts and knowledge workers who are not skilled programmers. In comparison to more opaque automated inference techniques for data processing where the user has little understanding of the inferred transformations, automatic program synthesis has the advantage of reusability (one-off learning of lightweight scripts that can be stored and applied to similar datasets in future), as well as transparency and editability: transformations are programs inspired from classical programming languages that can be manually edited if required.

Towards this goal, many recent works have explored various programming-by-example (PBE) approaches (Lieberman 2001; Gulwani, Harris, and Singh 2012; Raza, Gulwani, and Milic-Frayling 2014; Manshadi, Gildea, and Allen 2013; Lau et al. 2003). In PBE, users can specify their intended task by giving some input-output examples, from which the system attempts to automatically generate a program in some domain-specific language (DSL) that satisfies the given examples. A notable commercial success in this area is the Flash Fill feature in Microsoft Excel that is based on the PBE technique of (Gulwani 2011).

However, PBE approaches depend on an explicit intent specification from the user about the task that they want to perform, which often requires the user to correctly understand the examples that will help the system to infer the correct program, and can also require significant manual effort from the user in many kinds of tasks. In this work, we identify and address a wide range of data extraction tasks where the system can operate without explicit examples specifications and generate extraction scripts in a purely predictive manner, based on the properties of the input data alone. This can hence be seen as inference from input-only examples rather than input-output examples. We first illustrate such extraction scenarios in two concrete application domains that we investigate here, before discussing the predictive approach and its benefits in more detail.

Text extraction. Figure 1 shows a text extraction scenario, where the input data set (shown at the top) contains entries from a web server log. Each row is a text string that contains values such as the client IP address, date, etc, which are separated by various delimiting regions that are particular to this log format. The goal is to extract these values into separate columns as shown in the table at the bottom of the figure, where the columns representing the delimiting regions are highlighted.

In simple text-based formats such as CSV (Comma-Separated Values), such an extraction task is relatively easy as there is usually a fixed single-character delimiter used to separate data values. But in general, as shown in Figure 1, there can be any number of arbitrary string delimiters used in the same dataset, and a particular string may even be used
"I have a column that looks like the below however consistently changes in number of characters and letter. I need to be able to separate the numbers from the letters as they are the unit of measurement. I believe the mid or right function would work together however I just can't figure it out. Any help would be greatly appreciated. I would love to be able to isolate both the numbers and letters into their own cells."


Web extraction. Another automated data extraction domain is the extraction of tabular information from web pages, especially in cases where there is no explicit visual (row by column) table present in the web page. For example, Figure 3 shows the task of extracting the results of an Amazon product search, where each result item has numerous fields such as title, date, various prices, etc that are not laid out in an explicit row by column table. These fields, some of which are missing for some items, are represented in the DOM (Document Object Model) tree structure of the web page using different formatting properties rather than simple HTML table tags. And some fields, such as “new” and “used” prices, actually have no difference in formatting and can only be distinguished by examining the text content.

Hence, every such website uses different representations of tabular information, dedicated extraction scripts are required in each case. Although there has been much work on automatically detecting tables based on particular tags or visual properties (Krüpl, Herzog, and Gatterbauer 2005; Gatterbauer and Bohunsky 2006; Chu et al. 2015), extraction of arbitrary non-visual tables has mostly been explored with the help of user-provided examples, such as (Tengli, Yang, and Ma 2004) and the Flash Extract system (Le and Gulwani 2014).

Predictive program synthesis. The extraction scenarios described above have been addressed by the various PBE approaches that require the user to specify their intention through explicit examples of the desired extraction. However, in all of these scenarios it is easy for a human to predict the desired extraction by just observing the input data itself, without any need to be told what to extract. Such predictive intelligence has not yet been explored in program synthesis research, and is what we address in this work: automatic learning of extraction programs from input-only examples. The predictive approach we propose here has a number of advantages over previous PBE-based techniques:

- **Reducing user effort in giving examples.** In the scenarios in Figures 1 and 3 there are more than 10 extracted fields, and in practice it is normal for some log files to have around 50 fields. PBE systems such as Flash Fill and Flash Extract can normally require 2 or 3 examples per field, which can therefore lead to significant manual effort on the part of the user to complete an extraction task.

- **Less reliance on user understanding of system requirements.** In PBE approaches users are not aware of which examples would be most useful to the learning system, given all the variability in the data. For example, in Flash Fill users usually give examples on the top few rows, from which the system can often learn a program that is over-specific to the given examples, and therefore fails at a later row somewhere in the dataset.

- **Using all of the input data for learning.** PBE approaches work from a very small number of input-output examples to reduce the manual effort for users. In contrast, the predictive approach we describe here can utilise the many more inputs available in the data to infer the common patterns as well as the variability present in the data.

- **Allowing batch processing.** The need for manual intervention from the user in PBE approaches prevents the possibility of large scale automation of data processing tasks. For example, if the user has to process a large collection of datasets in different formats or pages from different websites, then they would need to manually provide examples for every new kind of format that is encountered.

In the next section, we begin by defining a general form of DSLs for performing data extraction, in which programs are structured as a combination of independent sub-programs...
for different data fields. We illustrate this with the concrete DSLs that we have designed for the text and web extraction domains, which are based on classical languages such as regular expressions and CSS (Cascading Style Sheets) selectors and can express the range of transformations illustrated in the scenarios above. We then describe our novel predictive synthesis algorithm for inferring programs in an extraction DSL given an input dataset. This is a domain-agnostic algorithm which operates by generating programs up to semantic equivalence in an efficient bottom-up fashion, and uses the notion of a correspondence relation between sub-programs as the central ranking principle. We describe concrete instantiations of the algorithm and ranking relations for the text and web domains, and in the following section we describe the evaluation of our technique on practical test scenarios obtained from log files, real users and the web. We end with a discussion of conclusions and future work.

2 Data Extraction DSLs

The design of the domain specific language (DSL) is an important component of any program synthesis approach, as there is always a delicate trade-off between expressivity (ability to address a range of practical use cases) and tractability (efficiently generating correct programs). In this section we describe a general form of DSLs for performing data extraction tasks, and then illustrate this with the concrete DSLs that we have designed for the text and web extraction domains. An extraction DSL is defined as a context-free grammar of the form

\[
\left( \psi_N, \psi_T, \psi_{start}, \mathcal{R} \right)
\]

where \( \psi_N \) is a set of non-terminal symbols, \( \psi_T \) is the set of terminal symbols, \( \psi_{start} \) is the start symbol and \( \mathcal{R} \) is the set of non-terminal production rules of the grammar. Every symbol \( \psi \) is semantically interpreted as ranging over a set of values \( \left[ \psi \right] \), which can be standard types such as integers, strings, arrays, etc. Each production rule \( r \in \mathcal{R} \) represents a program in the programming language, and is of the form

\[
\psi_h := Op(\psi_1, \ldots, \psi_n)
\]

where \( Op \) is the name of the operator, which takes parameter types given by the body symbols \( \psi_i \in \psi_N \cup \psi_T \) and returns a value of type given by the head symbol \( \psi_h \in \psi_N \). Hence the formal semantics of the DSL is given by an interpretation of each rule \( r \) as a function

\[
\left[ r \right] : \left[ \psi_1 \right] \times \ldots \times \left[ \psi_n \right] \rightarrow \left[ \psi_h \right]
\]

where \( \psi_h \) is the head symbol and \( \psi_1, \ldots, \psi_n \) are the body symbols of the rule operator. A program \( P \) of type \( \psi \) is any concrete syntax tree defined by the DSL grammar with root symbol \( \psi \). A complete program has the root symbol \( \psi_{start} \). Any derivation from a non-root symbol is a sub-program.

We impose two structural constraints on extraction DSLs that are oriented towards the data extraction task. Firstly, there exists a global variable that is available to the semantics of all operators in the programming language, which holds the input data on which the extraction task is being performed. This input variable has a fixed type \( \mathcal{I} \). For instance, this type can be a text string in the text extraction domain or the DOM tree of a webpage in web extraction. The semantics of each rule \( r \) in an extraction DSL with head symbol \( \psi_h \) and body symbols \( \psi_1, \ldots, \psi_n \) is defined with respect to the input variable:

\[
\left[ r \right] : \left[ \mathcal{I} \right] \times \left[ \psi_1 \right] \times \ldots \times \left[ \psi_n \right] \rightarrow \left[ \psi_h \right]
\]

Secondly, we require that there is a unique top-level rule in the DSL that has the start symbol as the head, and that this rule is of the form

\[
\psi_{start} := Op_h(\psi_1, \ldots, \psi_f)
\]

for some \( Op_h \) and \( \psi_f \). This models an extraction task as a program that consists of a top-level operator \( Op_h \) that combines the results of different field-level sub-programs that work at the level of individual fields in the input data. For example, in the text extraction task, the field-level programs identify the logic for detecting particular delimiters between data values, while the top-level operator combines these different delimiters to produce the list of extracted values. In the case of web extraction, the field-level sub-programs identify the selection logic for particular fields in the webpage, and the top-level operator combines these individual selection lists into an aligned table of values. We next illustrate these concepts with the concrete extraction DSLs we have designed for text and web extraction.

Text Extraction DSL

Figure 3 shows the DSL \( \mathcal{L}_t \) for text extraction, which is based on delimiters and regular expressions for detecting extraction patterns. The symbols of the grammar are shown
with their associated semantic types and the start symbol is explicitly marked. The input is a text string containing values possibly separated by delimiting regions, and the output (start symbol) of a program is an array of the extracted substrings. The top-level operator in this case is the SplitByDelimiters function which uses a number of different delimiter programs to produce the final splitting of the string. Each delimiter program computes a sequence of delimiting regions represented as a pair of start and end positions in the input string.

A delimiter program is either a match of a constant string (exact matches with tions in the input string).

Limiting regions are represented as a pair of start and end positions. Note that the DSL does not include operators for getting descendant nodes, right siblings, and filtering nodes by tag type, class name, ID, text content and child node positions. Note that the DSL does not bias towards any particular tag types such as table or list elements. For example, the selector for the first column in

```plaintext
@start string [] spl := SplitByDelimiters(d, ..., d)
Pair(int, int) [] d := c | LookAround(r, c, r)
Pair(int, int) [] c := Str(r) | StrWs(r)
Pair(int, int) [] r := Empty() | Tok(t) | Concat(r, r)
string s
RegexToken t
@end
```

**Figure 4:** The DSL $\mathcal{L}_t$ for text extraction

```plaintext
@start string [] tbl := ExtractTable(c, ..., c)
DomNode [] c := AllNodes() | Descendants(c) | RightSiblings(c) | filter
DomNode [] filter := Tag(tg, c) | Class(cl, c) | ID(id, c)
| Text(t, c) | NthChild(n, c) | NthLastChild(n, c)
string tg, cl, id, t
int n
@end
```

**Figure 5:** The DSL $\mathcal{L}_w$ for web extraction

the extracted table in Figure 3 is a simple class name filter

```plaintext
Class(“s-access-detail-page”, AllNodes())
```

while the selector for the ratings (column 8) requires a deeper tree search:

```plaintext
Class(“a-icon-alt”, Descendants(Class(“a-row”, AllNodes())))
```

### 3 Predictive Synthesis Algorithm

In this section we describe the predictive program synthesis algorithm that generates an extraction program from input-only examples. The general algorithm is parametric in a number of domain-specific properties (including the DSL) which can be provided as configuration parameters for particular domain instantiations.

In summary, for a given DSL and a set of input examples, the algorithm performs a systematic search to compute the semantically distinct values that can be generated by field-level programs in the DSL up to a certain size bound. This search is similar to previous bottom-up synthesis approaches such as (Katayama 2007; Raza, Gulwani, and Milic-Frayling 2015), but unlike previous approaches it utilizes certain operator-specific functions to gain orders of magnitude improvement in complexity, thereby making the synthesis tractable for practical DSLs such as $\mathcal{L}_t$ and $\mathcal{L}_w$.

After this semantic state space exploration, the final step is to perform a ranking to obtain the collection of field-level programs that will be used by the top-level DSL operator. Unlike previous program synthesis approaches, this ranking of field programs is not based solely on the properties of individual programs, but on correspondences that hold between different programs. Intuitively, in the absence of any output specification, the main ranking criterion is to prefer programs that identify maximal structure in the input data. We do this by finding the largest collection of field-level extractions that align well with one another, for some notion of alignment that is relevant to the data domain.

The general algorithm is shown in Figure 6. The parameter $I$ holds the $m$ input examples $I_1, ..., I_m$. For instance, in the text domain each input example may be a string in a spreadsheet row, while in the web domain each example would be the DOM tree of a single webpage. The other parameter $C$ is the configuration parameter, which defines five
configuration properties for the algorithm: DSL, MaxDepth, ExtractLiterals, LiftFuncs and Rank, which we shall describe in turn.

The DSL is the domain-specific language (as defined in section 2) in which programs will be synthesized. The top-level rule and the field programs symbol \( \psi_f \) is determined at line 4. Line 5 initializes a map \( M \) from symbols to a set of states which will be used to maintain the values generated by different programs on the given inputs. A state \( \sigma \in \Sigma \) of type \( \psi \) is of the form \( (\tilde{v}, P) \), representing a tuple of values \( \tilde{v} = (v_1, \ldots, v_m) \), where each \( v_j \in [\psi] \) is the value generated on input \( I_j \) by program \( P \) of type \( \psi \). We use \( \sigma \) to denote a set of states and denote all states of type \( \psi \) by \( \Sigma(\psi) \). We now describe the algorithm in three phases: state space initialization, search, and ranking.

### Initialization

The state map is initialized with literal values for each of the terminal symbols of the DSL (lines 6 and 7). This is done using the ExtractLiterals function, which computes literal values for each terminal type from the given input data. For instance, for the web DSL \( L_w \), the literals for \( tg, cl, id \) and \( t \) are respectively determined by all the tag types, class names, ids and text content of all the nodes in all of the input webpages, and numeric values for \( n \) are determined by the maximum number of child nodes of any node.

For the text DSL \( L_t \), constant string values for \( s \) can be determined as any substrings of the inputs, but in practice it is more effective to restrict these to special character strings as these are normally used as delimiters. The regular expression tokens \( t \) we consider are standard regex patterns for numbers, lower or upper case letters, special characters, date/time, etc. At line 7, the states for each terminal symbol are initialised with the extracted literal values (where \((v)^m \) represents a value tuple of \( m \) occurrences of the same value \( v \), since a literal has the same value on any input).

### Search

At line 8 we begin the bottom-up exploration of the state space. This is bounded by the MaxDepth configuration parameter, which imposes a bound on the depth of the syntax tree of the programs we consider. We determine the value used for the MaxDepth parameter in a particular domain based on an empirical evaluation, taking the maximum depth that yields sufficiently expressive programs in the given DSL in practical running time. Given the existing set of states at each iteration, for each rule \( r \) other than the top rule, at line 15 we compute the set \( \sigma_{new} \) representing the new distinct values created by application of the rule over the existing values. This rule application is hence a lifting of the rule operator semantics function \( [\rho] \) to sets of states, that is, a function with signature:

\[
[\mathcal{X}]^m \times \mathcal{P}(\Sigma(\psi_1)) \times \cdots \times \mathcal{P}(\Sigma(\psi_n)) \rightarrow \mathcal{P}(\Sigma(\psi_h))
\]

where \( \mathcal{I} \) is the input type of the DSL. A generic way of implementing this lifting function for an arbitrary rule is with the procedure LiftGeneric defined in Figure 7, which simply computes the cross product over all the parameter sets and applies the rule semantics over all combinations of value tuples. Although it is rule-agnostic, this naive combinatorial approach can be prohibitively expensive in practice. Significant complexity reduction can be gained by using specific lifting functions for certain kinds of operators, which can be specified using the configuration parameter LiftFuncs mapping certain rules to their lifting functions. The algorithm uses the lifting function if one exists (line 12), or else defaults to the generic function (line 14).

Examples of lifting functions exist in both the text and web domains. Most of the filter operators in the web DSL \( L_w \) benefit from specific lifting functions, and an example for the ID operator is shown in Figure 8. In this case, given a set of ID values and a set of node sets, the naive approach would be to filter every node set by every ID. However, the only useful results will be for IDs that actually exist in any of the nodes. This is achieved by the method in Figure 8 by traversing each node set only once, and maintaining a map from all ID values encountered to the nodes that have that ID. Hence formally we go from complexity that is quadratic in the sizes of \( \sigma_1 \) and \( \sigma_2 \) to linear in these sizes. Similar complexity reductions can be made for other filter operators in \( L_w \).

In the text domain, lifting functions play a vital role for the Concat and LookAround operators in \( L_t \). For example,
for Concat we have an existing set of regexes that match on the input strings, and would like to find all concatenations of these regexes that also match on the input strings. While the naive approach is to simply check all pairs, the optimal lifting function traverses the set of regex matches and builds a map from end positions to the regexes matching on those end positions. A second linear traversal can then check for all regexes that start at these end positions, so that only pairs of regexes that actually have adjacent matches on the input strings are ever considered. A similar technique is used in the case of LookAround, where triples instead of pairs of matches are considered.

**Ranking**

The ranking function $\text{Rank}$ is the final configuration parameter, which selects the field-level programs that the algorithm uses to construct the final extraction program with the top-level operator (line 18). The general principle behind our ranking functions is the notion of inter-subprogram correspondence. Unlike previous ranking approaches that examine properties of individual programs, the idea behind the correspondence relations for extraction DSLs is to detect maximal structure in the input data by finding the largest collection of field-level extractions that align well with one another. The ranking function finds such maximal collections of states.

In the web domain, the main correspondence relation that we use is based on an interleaving relationship between the sequence of DOM nodes that are extracted by different field-level programs. Formally, states $(\bar{v}_1, P_1)$ and $(\bar{v}_2, P_2)$ satisfy the interleaving relation if we have $\bar{v}_1 = (s_1, \ldots, s_m)$ and $\bar{v}_2 = (s'_1, \ldots, s'_m)$ and $\text{Interleave}(s_i, s'_i)$ holds for all $i$. Each $s_i$ and $s'_i$ is an array of nodes $\text{DomNode}[]$ that is the result of the field extraction programs $P_1$ and $P_2$ on the $i$th input, and $\text{Interleave}(s, s')$ holds iff

- $\exists N, |s| = |s'| = N$ and
- node $s[j]$ is before $s'[j]$ in document order, for $0 \leq j < N$
- node $s'[j]$ is before $s[j + 1]$, for $0 \leq j < N - 1$

This interleaving relation captures the common pattern on web pages with structured information where a sequence of records occurs in document order, and each record contains nodes for particular fields that are similarly formatted. We note that this strategy assumes that the document order in the DOM tree corresponds to the visual rendering of information on the web page, which is an assumption that holds in a large majority of web pages but not always (in our evaluation we found a single example of such a webpage where the document order did not correspond to the visual order). Also, in addition to the interleaving relationship, we also impose preferences on more structured patterns where the nodes of all the fields share a common ancestor node. This prefers common patterns in webpages where records are grouped under separate DOM nodes, such as DIV of TR tags. Although we give preference to these more structured patterns, we do not depend on them as in many cases such common ancestor nodes do not exist.

In the text domain, the main correspondence relation we use is based on consistent disjoint alignment of delimiters.
the text extraction DSL to perform a splitting of the column data into different field columns (as shown in Figures 1 and 2). However, different users may have different preferences for the degree of splitting they would like (e.g., whether or not to split a date into day, month, year). If data has not been split enough, the user can select an output field column and split it further. On the other hand, the user can highlight any number of adjacent columns and click a “Merge” button to simply merge the columns and undo the unwanted splits.

For web extraction tasks, we implemented a similar Excel add-in called WebExtract. Using this feature, the user can specify the URL of a webpage, and then click an “Extract” button, at which point the system learns an extraction program in the web DSL and displays the extracted table in the spreadsheet (as in Figure 3). WebExtract can be used together with ColumnSplit to perform further text-based splitting on columns that have been extracted by WebExtract.

**Evaluation of ColumnSplit**

For evaluation in the text domain, we collected a set of 20 benchmark cases from product teams, help forums, as well as real users in our organization who provided us with data sets on which they would like to perform extraction. A lot of these datasets come from various log files such as from web servers, but also include other text-based datasets. The goal of our evaluation was to measure the maximum number of fields our system can extract.

The average performance results of our system on these benchmarks are shown in Figure 9. There were an average of 13.95 fields per dataset, of which 10.3 were detected on the first attempt (primary), 3.45 were extracted by further splitting on some of the columns (not more than three levels of splitting required in any test case), and 0.2 could not be extracted at all. All the non-extracted fields were in a single test case, which was a task involving different numbers of fields on different inputs. With such possibly-missing fields it is in general not possible to determine the desired alignment of fields as there are different alternatives, so such tasks may be better handled by some kind of output specification from the user such as examples. The average execution time per task was 4.2 seconds, although 16 tasks were completed in under 2 seconds. Memory usage was observed to approximately double at each iteration of the synthesis algorithm, which remained under tractable limits as the algorithm maintains the state space of programs up only up to semantic equivalence over the given input states.

For comparison, we also evaluated our system without using the operator-specific lifting functions as described in Section 3, and observed the drastic increase in execution time to an average of 210.73 seconds per task. We also investi-
and Barzilay 2013; Manshadi, Gildea, and Allen 2013; Gulwani and Marron 2014) and various mixed-initiative approaches (Allen et al. 2007; Kandel et al. 2011; Raza, Gulwani, and Milic-Frayling 2015). However, all of these approaches depend on some kind of explicit intent specification from the user e.g. PBNL approaches often suffer from inaccurate or ambiguous user descriptions and vocabulary mismatch. The key difference in the predictive approach we present here is to not rely on explicit intent specification from the user, which works well for extraction tasks. In this respect the predictive approach can be seen as one part of a broad spectrum of approaches for different kinds of tasks: predictive approaches may be effective for extraction, while examples may be useful for transformation tasks and natural language interfaces may be useful for querying.

In another respect, we have yet to explore the combination of the predictive approach together with output examples given by the user. This would be analogous to semi-supervised learning in the PBE setting, as opposed to the fully supervised learning of traditional PBE systems and the purely unsupervised learning we have proposed in this work. PBE techniques, although capable of addressing transformations that go beyond extraction, have generally relied on the small number of input-output examples provided by the user, without making use of all the additional inputs available in the dataset. Only in recent work (Singh 2016) have signals from additional inputs been shown to improve the ranking of a PBE system. However, it is unclear how strong these additional input signals are by themselves, especially in comparison to our predictive approach which can function in a purely unsupervised manner without the need for any output examples. It will therefore be interesting to explore the use of our predictive analysis to infer global alignments and variability in the input data, and use this information to improve the ranking of previous PBE systems. For example, if the predictive analysis yields that a certain delimiter occurs regularly throughout all inputs, then programs utilizing that delimiter may be ranked higher by the PBE system, even if the user has only provided a single output example which does not disambiguate between different candidate delimiters to use.

A different direction for future work is to explore the applicability of our generic framework for predictive synthesis to other practical application domains. Richly formatted documents such as XML-based formats (DOCX, ODF, PPTX, etc) may be addressed using DSLs similar to the X-Path language, perhaps following the technique of least general generalizations over input states as in (Raza, Gulwani, and Milic-Frayling 2014). Extraction from PDF documents may be explored with DSLs that include geometrical co-ordinate constructs. It will also be interesting to investigate the case of plain unstructured text files where record boundaries are not predetermined, as previously addressed by non-predictive PBE approaches such as (Le and Gulwani 2014). While the DSL $L_t$ we presented for text extraction addresses the case of structured log files where row boundaries are given, we may potentially address the case of undefined row boundaries with a generalization of $L_t$, in which records are inferred using an interleaving-style relation similar to the one we used for web extraction with $L_w$.

Semantically-based extractions (Singh and Gulwani 2012a; 2012b) are yet another domain where input data alone may yield useful extraction patterns. In general, it will be interesting to explore the applicability of the predictive approach to most extraction domains previously addressed by PBE techniques.

References


Gatterbauer, W.; and Bohunsky, P. 2006. Table extraction using spatial reasoning on the css2 visual box model. In AAAI, 1313–1318. AAAI Press.


Le, V., and Gulwani, S. 2014. Flashextract: A framework for data extraction by examples. In PLDI.


Raza, M.; Gulwani, S.; and Milic-Frayling, N. 2014. Programming by example using least general generalizations. In AAAI.

Raza, M.; Gulwani, S.; and Milic-Frayling, N. 2015. Compositional program synthesis from natural language and examples. In IJCAI.


