Disjunctive Program Synthesis: a Robust Approach to Programming by Example

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

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

Abstract

Programming by example (PBE) systems allow end users to easily create programs by providing a few input-output examples to specify their intended task. The system attempts to generate a program in a domain specific language (DSL) that satisfies the given examples. However, a key challenge faced by existing PBE techniques is to ensure the robustness of the programs that are synthesized from a small number of examples, as these programs often fail when applied to new inputs. This is because there can be many possible programs satisfying a small number of examples, and the PBE system has to somehow rank between these candidates and choose the correct one without any further information from the user. In this work we present a different approach to PBE in which the system avoids making a ranking decision at the synthesis stage, by instead synthesizing a *disjunctive program* that includes the many possible top-ranked programs as possible alternatives and selects between these different choices upon execution on a new input. This delayed choice brings the important benefit of comparing the possible outputs produced by the different disjuncts on a given input at execution time. We present a generic framework for synthesizing such disjunctive programs in arbitrary DSLs, and describe two concrete implementations of disjunctive synthesis in the practical domains of data extraction from plain text and HTML documents. We present an evaluation showing the significant increase in robustness achieved with our disjunctive approach, as illustrated by an increase from 59% to 93% of tasks for which correct programs can be learnt from a single example.

Introduction

The vast majority of computer users and knowledge workers today are not computer programmers, and yet commonly face challenges for which programming skills are required. Many examples of such tasks can be seen in the domain of data manipulation, such as extracting substrings of text from a column in a spreadsheet, or extracting important data fields from a collection of richly formatted emails or web pages. Such tasks may be performed by professional programmers by writing custom extraction scripts using regular expressions, Excel macros or CSS expressions, but are out of the reach of the common computer user. This is where Programming by example (PBE) approaches (Lieberman 2001) can be extremely beneficial in helping end users to automate the generation of such scripts. A PBE system allows the user to specify their intent by giving a few input-output examples of the desired task, from which the system attempts to automatically generate a program in an underlying domain-specific language (DSL) that satisfies the given examples. Instead of opaquely automating one-off tasks, PBE produces lightweight scripts in common programming languages that can saved and reused by users in different environments, independently of the learning techniques used to infer those scripts.

PBE approaches have seen significant interest and progress in recent years (Lau et al. 2003; Gulwani, Harris, and Singh 2012; Devlin et al. 2017; Balog et al. 2017; Manshadi, Gildea, and Allen 2013; Singh 2016; Raza and Gulwani 2017), as well as notable commercial successes such as the *Flash Fill* feature in Microsoft Excel (Gulwani 2011). However, a key challenge faced by current systems is that the programs inferred from a few examples generally lack robustness and easily fail on new inputs. This is because the state space of possible programs (defined by the DSL) is large, since we need to support expressive programs covering different tasks, and hence there can be many possible programs satisfying a given set of examples provided by the user. This expressivity vs. correctness trade-off is a central challenge in the design of PBE systems, which is why there has been a strong effort in the synthesis community to improve the *ranking* used in PBE systems to choose the *most likely* program the user may want out of the large set of candidates that logically satisfy the given examples (Singh and Gulwani 2015; Singh 2016; Ellis and Gulwani 2017).

However, no matter how good the ranking technique is, this approach by definition restricts the system to choose a single option from a set of many likely possibilities (and this is true for synthesis at every sub-program level, not just for final programs). In this work we explore a philosophically different approach to this issue of robustness based on the idea that we need not prematurely commit to a single choice of program at synthesis time. Instead, it would be beneficial to delay this ranking decision to the execution of the program when we have more information: the actual input on which the program is to be executed. We illustrate this idea with scenarios from two practical application domains

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Input	Desired output
12 units PID 24122 Laptop	PID 24122
43 units PID 98311 Wireless keyboard	PID 98311
7 units PID 21312 Memory card	PID 21312
22 units PID 23342 Docking station v2	PID 23342
6 units PID 64232 Mouse with pad	PID 64232
...	...

Figure 1: Text extraction task to extract product ID

of text extraction and web extraction.

Figure 1 shows a data cleaning scenario where the user would like to remove the arbitrary description that occurs after the product ID in the input column, as shown in the desired output column. Let us assume that the user provides the first row as an input-output example. With this one example a PBE system based on regular expressions such as Flash Fill can generate many possible satisfying programs that employ different substring extraction logics.

For instance, the possible logics to detect the end of the extracted substring could include "the end of the last number" (P_1), "the end of the second number" (P_2) , "the end of the first number that occurs after capital letters and whitespace" (P_3) and many other more complex expressions. As simplicity is commonly favoured by the ranking in most PBE systems, in this case the system chooses the logic P_1. However, this particular logic fails on row 4, where there happens to be another number occurring at the end of the input, and so the system incorrectly outputs the string "PID 23342 Docking station v2" in this case. Hence the programs P_2 or P_3 would have been better suited to this task, but we only become aware of this when we encounter the non-conforming input on row 4, which in practice could have been anywhere in the data or even in a different dataset. Moreover, the user has no way of knowing which examples to provide to the system to ensure that it learns better programs.

We also note that it is not just a matter of improving the ranking system to choose a better program, because in general there is no single ranking strategy that can work in all cases. Depending on the variability in the data set, it is possible that there is no single logic that can satisfy all rows, but that different logics may apply to different rows to solve the overall task. For example, on an input row such as "22 units PID 23342 Docking station VERSION 2" the program P_3 would fail, or on a row "22 to 25 units PID 23342 Docking station v2" the program P_2 would fail. Hence, in such situations any one ranking scheme chosen by the PBE system would fail on some inputs.

Thus we observe that the fragility comes from restricting to a particular choice of extraction logic at the synthesis stage. In this paper we propose that rather than prematurely enforcing a single choice P_i from the list of top-ranked programs $P_1, \ldots, P_n$ in the synthesis algorithm, we can instead learn a more robust *disjunctive* program that includes all of the choices $P_1, \ldots, P_n$ and selects between these different options when it is executed on a new input. Hence, with this disjunctive approach, we can delay the choice of which particular logic to use until we have the important information of what the execution input is and can compare the outputs that are produced by the different logics on this input.

We consider another example from the domain of web extraction. In this case the user's extraction task may be to extract a particular data field from a structured HTML document, such as extracting the flight number from an email containing a flight itinerary. In existing PBE systems for web extraction (Le and Gulwani 2014; Polozov and Gulwani 2015) the user can perform such a task by giving examples of the desired field on a few emails, from which the system will synthesize node selection expressions based on languages such as XPath or CSS selectors to extract nodes in the HTML DOM (Document Object Model). For example, assume the user example of the desired field refers to the following SPAN element node in the HTML markup:

```
<span style="color: gray" class="c1">
   BA052
</span>
```

The challenge comes in choosing which properties of the example nodes to include or exclude in the node selection logic, as it is not known which of these properties would be varying in other inputs. For example, in this case we may have a choice between selection logics "color is gray" (P_1), "class is c1" (P_2) or the more restrictive conjunction of the two properties "color is gray and class is c1" (P_3). Ranking schemes often choose to be conservative and choose the most specific logics, but these logics tend to impose too many constraints and overfit the examples, and therefore fail to return any result on new inputs. Choosing more general logics has the pitfall of losing important properties and identifying incorrect nodes.

Thus again we observe that it will be beneficial to use a disjunctive approach to maintain a set of these different possible selection logics in the synthesized program and only choose between them at execution time when we have the new input available. For example, the program may first try the most specific logic and if that fails to return any result then it may try less restrictive logics. However, another issue in this case is that the set of possible logics grows exponentially with the number of properties that a node may have, as possible conjunctions of the predicates correspond to all possible subsets. Including all of these possibilities in a disjunctive program would not be viable, both in terms of performance of the learning algorithm as well as the size and execution time of the learnt program. In this work we describe a technique to compute the maximal (the most specific) as well as a set of minimal (least restrictive) programs that satisfy the example extractions and together cover all the properties of the example nodes. We show how these candidates covering all of the observed properties can be chosen as effective disjuncts by the synthesis algorithm to yield efficient and robust disjunctive programs.

The scenarios we describe in both the text and web extraction domains illustrate how PBE systems generate programs that can easily fail on new inputs. However, we also note that is not simply a matter of providing more examples to learn better programs. Even if users are willing to give many

$$
\begin{aligned}
\textbf{@start}\ \mathsf{string}\ res &:= sub \mid \mathsf{Const}(s) \mid \mathsf{Concat}(res, res)\\
\mathsf{string}\ sub &:= \mathsf{Substr}(p, p)\\
\mathsf{int}\ p &:= \mathsf{Pos}(r, r, k) \mid \mathsf{Pos}(k)\\
\mathsf{Regex}\ r &:= \epsilon \mid t \mid \mathsf{ConcatReg}(r, r)\\
\textbf{@input}\ \mathsf{string}\ inp &\quad \mathsf{int}\ k \quad \mathsf{string}\ s \quad \mathsf{Regex}\ t
\end{aligned}
$$

Figure 2: The DSL $\mathcal{L}_t$ for text processing

$$
\begin{aligned}
\textbf{@start}\ \mathsf{Node}\ n &:= \mathsf{Select}(fl, k)\\
\mathsf{Node}[]\ fl &:= \mathsf{Filter}(sel, c)\\
\mathsf{Node}[]\ sel &:= \mathsf{All}() \mid \mathsf{Children}(fl)\\
\mathsf{Func}\langle \mathsf{Node}, \mathsf{bool}\rangle\ c &:= \mathsf{Tag}(s) \mid \mathsf{Class}(s) \mid \mathsf{ID}(s) \mid \mathsf{Text}(r)\\
&\quad \mid \mathsf{NthChild}(k) \mid \mathsf{NumChild}(k)\\
&\quad \mid \mathsf{Style}(s, s) \mid \mathsf{Attr}(s, s) \mid \mathsf{Conj}(c, c)\\
\textbf{@input}\ \mathsf{DomTree}\ inp &\quad \mathsf{string}\ s \quad \mathsf{int}\ k \quad \mathsf{Regex}\ r
\end{aligned}
$$

Figure 3: The DSL $\mathcal{L}_w$ for node extraction from HTML

examples, they cannot know which examples to give: if they are representative of their target data and are the ones required by the PBE system to learn a correct program. This is an especially important concern in situations where the user cannot recover from the problem by attempting to relearn a different program when the failure case is encountered. This is because the failure may occur at any point in future in a different execution environment (or for a different user of the program) where access to the learning system is no longer possible.

The key idea behind our disjunctive synthesis approach is to move the top-program selection from ranking in the synthesis phase to the semantics of the program itself, by maintaining multiple disjuncts in the final program. The main challenges we address in implementing this idea is the technique for deciding which disjuncts to include, as well as the nature of the selection mechanism that decides between the disjuncts at execution time. These design choices are guided by the goals of correctness as well as performance of the synthesized programs. As well as concrete implementations of the disjunctive approach for the text and web application domains, in this work we also present a generic framework in which arbitrary DSLs and synthesis algorithms can be extended to support the learning of disjunctive programs and we formulate the components required for such a system in an abstract setting.

In the next section we begin with the general form of DSLs we consider and present concrete DSLs for the text and web domains that are based on the standard regular expression and CSS selector languages. We then describe the extensions required for arbitrary DSLs to support disjunctive programs, and illustrate this with the disjunctive extensions we make to the text and web DSLs. In the following section we describe the generic synthesis algorithm, as well as the implementations of domain-specific components of the algorithm for particular DSLs. We then describe the evaluation of our technique over a set of extraction scenarios obtained from a product team. We illustrate the improvement in robustness by considering the number of scenarios in which a correct program can be learnt from a single example, and show how this improves from 59.6% to 93.6% from the non-disjunctive to the disjunctive case. We end with a discussion of related work and conclusions.

Domain Specific Languages

The design of the domain specific language (DSL) is an important requirement for any synthesis system. It needs to strike the right balance between expressiveness (to handle a range of common tasks in the target domain) and tractability (for the synthesis algorithm to learn correct programs efficiently). In this section we describe a general form of DSLs and provide instances of two concrete DSLs that we use for the text and web domains. We then describe the notion of a disjunctive program operator at an abstract level, and describe how to extend an arbitrary DSL with disjunctive support for any of its components. We illustrate with examples of disjunctive components in both the text and web DSLs.

A DSL is defined as a context-free grammar of the form $(\tilde{\psi}_N, \tilde{\psi}_T, \psi_{st}, \psi_{in}, \mathcal{R})$, where $\tilde{\psi}_N$ is a set of non-terminal symbols, $\tilde{\psi}_T$ is the set of terminal symbols, ψ_{st} is the start symbol, ψ_{in} is the input symbol and $\mathcal{R}$ is the set of production rules of the grammar. Each symbol ψ is semantically interpreted as ranging over a set of values $[\![\psi]\!]$, which can be standard programming types such as integers, strings, arrays, lambda expressions, etc. Each production rule $r \in \mathcal{R}$ represents an operator 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 $\psi_i \in \tilde{\psi}_N \cup \tilde{\psi}_T$ and returns a value of type given by the head symbol $\psi_h \in \tilde{\psi}_N$. Hence the formal semantics of the DSL is given by an interpretation of each rule r as a function

$$[\![r]\!] : [\![\psi_1]\!] \times \ldots \times [\![\psi_n]\!] \rightarrow [\![\psi_h]\!]$$

where ψ_h is the head symbol and $\psi_1, ..., \psi_n$ are the body symbols of the rule operator. A program P of type ψ is any concrete syntax tree defined by the DSL grammar with root symbol ψ. A *complete program* has the root symbol ψ_{st}. Any derivation from a non-root symbol is a *sub-program*. The input symbol ψ_{in} is a terminal symbol that represents the input to the overall program, and is a global variable that is available to the semantics of all operators in the programming language.

Text extraction

Figure 2 shows the DSL $\mathcal{L}_t$ for text manipulation, which is based on the *Flash Fill* language (Gulwani 2011; Polozov and Gulwani 2015) for string processing using regular expressions. The symbols of the grammar are shown with their associated semantic types with the start and input symbols explicitly marked. The input inp in this case is a string and the output of any complete program res is also a string, e.g., a substring of the input as in the extraction scenario in Figure 1. At the top level, the output is a concatenation Concat

$$\begin{array}{rcl} & \vdots & \\ \textbf{@disj}\{f\}\ \textsf{string}\ sub & := & \textsf{Substr}(p,p) \\ \textsf{Func}\langle\textsf{string},\textsf{obj}\rangle\ f & := & \textsf{NotNull}() \mid \textsf{NumChars}() \\ & & \mid \textsf{NumNonWS}() \mid \textsf{SpecialChars}() \\ & \vdots & \end{array}$$

Figure 4: Disjunctive DSL $\mathcal{DL}_t$ which extends $\mathcal{L}_t$

$$\begin{array}{rcl} & \vdots & \\ \textbf{@disj}\{f\}\ \textsf{Node[]}\ \mathit{fl} & := & \textsf{Filter}(sel,c) \\ \textsf{Func}\langle\textsf{string},\textsf{obj}\rangle\ f & := & \textsf{NonEmpty}() \\ & \vdots & \end{array}$$

Figure 5: Disjunctive DSL $\mathcal{DL}_w$ which extends $\mathcal{L}_w$

of constant strings Const or some substrings of the input defined by sub. A substring expression $\textsf{Substr}(p_1, p_2)$ extracts the substring between two positions p_1 and p_2 in the input string. Position expressions are defined as either constant positions $\textsf{Pos}(k)$, or $\textsf{Pos}(r_1, r_2, k)$ which determines the kth position in the input string whose left and right sides match regular expressions r_1 and r_2 respectively. For example, the program for the task in Figure 1 that extracts the substring from the first capital letter to the last number is given as

$$\textsf{Substr}(\textsf{Pos}(\epsilon, \textsf{Caps}, 0), \textsf{Pos}(\textsf{Num}, \epsilon, -1)) \qquad \langle * \rangle$$

where the negative k indicates occurrence from the end of the string. An alternative program for the same task that extracts until the end of the *second* number rather than the last number is given as

$$\textsf{Substr}(\textsf{Pos}(\epsilon, \textsf{Caps}, 0), \textsf{Pos}(\epsilon, \textsf{Num}, 1)) \qquad \langle ** \rangle$$

Web extraction

Figure 3 shows the DSL $\mathcal{L}_w$ for extracting nodes from an HTML document. $\mathcal{L}_w$ is based on path expressions and filter predicates similar to CSS selectors. The input inp in this case is the DOM tree of the entire HTML document and the output n of any complete program is a particular node in this tree, e.g. the node containing the flight number in a flight itinerary, as in the extraction task described in the introduction.

The top level operator $\textsf{Select}(\mathit{fl}, k)$ selects the kth node from a filtered collection of nodes fl in document order. The $\textsf{Filter}(sel, c)$ operator filters all nodes in a selection sel using a condition c. The sel can be all the nodes in the document (All) or the children of any nodes obtained by another filter operation as given by $\textsf{Children}(\mathit{fl})$. The condition c used for filtering is a boolean function on nodes defined by a range of atomic predicates (constraints including tag type of the node, its ID, class, regular expression matches in the text content, number of children, style and attributes) as well as any conjunctions $\textsf{Conj}(c, c)$ of these predicates.

For example, a program to extract the 3rd node in the document that has class "c1" and is the child of a "TD" element is given as

$$\textsf{Select}(\textsf{Filter}(\textsf{Children}(\textsf{Filter}(\textsf{All}(), \textsf{Tag}(\text{“TD”}))), \textsf{Class}(\text{“c1”})), 3)$$

Disjunctive Programs

In this section we describe the notion of a disjunctive program and how to extend any DSL to support disjunctive programs. The DSL designer can choose to provide disjunctive support for any symbols defined in their DSL. Figure 4 illustrates the extension we have made to the text DSL to support disjunctive programs at the substring operator level. This is indicated by the **@disj**$\{f\}$ annotation before the sub symbol. This annotation includes a reference to a new symbol f that we have added to the grammar, which defines a number of *feature calculators* in the DSL that will be used to choose between different substring logics at execution time. The annotation **@disj**$\{f\}$ indicates that a disjunctive subprogram for symbol sub is of the form:

$$\textsf{Disj}([P_1, \ldots, P_n], [F_1, \ldots, F_m], [v_1, \ldots, v_m])$$

where $[P_1, \ldots, P_n]$ is an array of disjuncts of type sub, $[F_1, \ldots, F_m]$ is an array of feature programs of type f, and $[v_1, \ldots, v_m]$ is an array of objects of type obj. The disjunction operator Disj chooses between possible disjuncts $P_1, \ldots, P_n$ by computing a feature vector on the output of each P_i using the feature functions $[F_1, \ldots, F_m]$, and then choosing the output that has feature vector most similar to the target feature values $[v_1, \ldots, v_m]$. As an example, let us consider the following disjunctive program that addresses the task in Figure 1

$$\textsf{Disj}([P_1, P_2], [\textsf{NotNull}(), \textsf{NumChars}()], [\textsf{True}, 9])$$

where P_1 and P_2 are the substring programs defined in $\langle * \rangle$ and $\langle ** \rangle$ respectively (P_1 extracts up to the last number in the input while P_2 extracts up to the second number). In this case, the disjunctive program is using only two feature calculators: NotNull() which computes a boolean value indicating if the output substring is non-null, and NumChars() which computes the number of characters in the output substring. The target feature values expected for these two features are True and 9, since the output string should be non-null and the product ID strings are of length 9. Hence, when executing this disjunctive program on the inputs in Figure 1, both disjuncts P_1 and P_2 will produce equally scoring outputs on the first three inputs, but on the fourth input it will be P_2 that produces the higher scoring output for the target feature vector.

Note that a disjunctive program need not use all the feature calculators available in the DSL as only some of them may be relevant to the particular situation. The above example program uses two of the calculators, though the DSL also supports other features such as NumNonWS() and SpecialChars() which compute the number of non-whitespace characters and the set of special characters occuring in the output. The DSL designer can choose any feature calculators to include in the language, but as we de-

scribe in the next section, it is the task of the synthesis algorithm to select which features and the target value vector to include in the disjunctive program based on the provided examples.

We next consider another example of a disjunctive DSL in Figure 5, which shows the extensions made to the web DSL. As discussed in the introduction, in this case a great uncertainty lies in what logic to use to filter nodes down to the target extraction. Hence we choose to support disjunctive programs for the *fl* symbol in order to consider alernative filtering logics in the final program. In this case we have a single feature calculator NonEmpty() that checks if the filtered set contains at least one node.

As an example, the following disjunctive program addresses the flight number extraction task described in the introduction:

$$\mathsf{Disj}([P_1, P_2, P_3], [\mathsf{NonEmpty}()], [\mathsf{True}])$$

where

$P_1 = \mathsf{Filter}(\mathsf{All}(), \mathsf{Conj}(\mathsf{Style}(\text{“color”},\text{“gray”}),\mathsf{Class}(\text{“c1”})))$

$P_2 = \mathsf{Filter}(\mathsf{All}(), \mathsf{Class}(\text{“c1”}))$

$P_3 = \mathsf{Filter}(\mathsf{All}(), \mathsf{Style}(\text{“color”}, \text{“gray”}))$

The idea behind this disjunctive program is to try the most restrictive conditions first, and if they fail to be satisfied by any nodes then we consider less restrictive conditions. This example also illustrates that the Disj() operator uses the ordering on the disjuncts to select the first disjunct if there are ties in the feature scores for different disjuncts.

Although our sample DSLs for the both the text and web domains have disjunctive support for only one symbol in each DSL, in general our framework supports any number of disjunctive symbols in the language. One example of a DSL including disjunctive support for two different symbols is the language $\mathcal{DL}_{tw}$, which combines both grammars $\mathcal{DL}_t$ and $\mathcal{DL}_w$ along with a top-level operator that first extracts the node from an HTML document and then performs a substring extraction on the text content of this node.

Formal definition

We now describe the formal semantics of the Disj() operator and the formulation of disjunctive DSLs in the abstract framework. A disjunctive DSL is of the form $(\tilde{\psi}_N, \tilde{\psi}_T, \psi_{st}, \psi_{in}, \mathcal{R}, \Delta)$, which extends the definition of a standard DSL given in the last section with the one additional parameter Δ. This parameter $\Delta : \tilde{\psi}_N \rightharpoonup \tilde{\psi}_N$ maps certain symbols for which we require disjunctive support to the symbols that represent their feature calculators. The well-formedness constraint on this map is that if $\Delta[\psi] = \psi_f$ then the type of ψ_f is $\mathsf{Func}\langle\psi, \mathsf{obj}\rangle$. For example, for the text DSL we have $\Delta[sub] = f$ mapping the single substring symbol to its feature calculator symbol in the grammar.

Given ψ for which we have disjunctive support with $\Delta[\psi] = \psi_f$, let $\mathsf{T} = [\![\psi]\!]$ be the semantic type of ψ. The semantics of the disjunction operator for ψ is defined in Figure 6. The operator takes as parameters the outputs produced by a list of disjuncts ($disjuncts$), the list of feature calculators to compute relevant features on these outputs ($features$) and the target vector v of values for each of the features (which is expected to be of the same length as the $features$ array).

```
Disj (T[] disjuncts, Func⟨T, obj⟩[] features, obj[] v)  {
   for each d in disjuncts
      let s  =  {i | features[i](d) == v[i]}
      let score[d]  =  |s|
   return the first d in disjuncts where score[d] = Max(score)
}
```

Figure 6: Disjunction operator semantics for symbol ψ, where $\mathsf{T} = [\![\psi]\!]$ is the semantic type of ψ

The operator computes the given features on each of the disjunct outputs and returns the output on which the most feature values match the target values in v. While we use this very simple similarity measure in this work, in theory the scoring function used to select between different disjuncts may incorporate weights associated with different features or use more sophisticated similarity measures than simple equality of feature values.

Also, in some situations it could be the case that only minimal thresholds of feature values need to be satisfied to accept a disjunct, in which case we need not necessarily compute all disjunct outputs but use a more performant lazy evaluation in which the first disjunct that satisfies the given thresholds can be selected. In our empirical investigations in this work we did not require such developments, but they may be relevant in other application domains which we leave for future work.

Program Synthesis Algorithm

In this section we describe the synthesis algorithm for learning disjunctive programs from input-output examples, and its instantiations for the text and web DSLs. The algorithm we describe is agnostic of any particular DSL, as it takes the DSL and some other domain-specific properties as configuration parameters for particular domain instantiations.

In summary, the algorithm is based on learning program expressions in the DSL by propagating example-based constraints on any expression to its subexpressions, as in (Polozov and Gulwani 2015), and we describe how to incorporate disjunctive program learning in this abstract setting. The constraint propagation is done using domain-specific properties of DSL operators, which are provided in the configuration parameter. We first discuss these properties before describing the full algorithm.

Domain-specific parameters

The configuration parameter C to the algorithm contains four domain-specific components that are required by the algorithm: DSL, InferSpec, Synth and Ranker. The DSL is the domain-specific language in which programs will be synthesized. InferSpec is a map from operator rules in the DSL to

```
1:  function SynthProg(C, φ, ψ)
2:      let C.DSL = (ψ̃_N, ψ̃_T, ψ_st, ψ_in, R, Δ)
3:      Progs ← ∅
4:      if C.Synth[ψ] is defined for ψ then
5:          Progs ← C.Synth[ψ](φ)
6:      else
7:          for each r ∈ R where r has head ψ do
8:              let r be ψ := Op(ψ_1, ..., ψ_n)
9:              S ← {[]}
10:             for i = 1 ... n do
11:                 S' ← ∅
12:                 for each [P_1, ..., P_{i-1}] ∈ S do
13:                     φ_i ← C.InferSpec[r](φ, ψ_i, [P_1, ..., P_{i-1}])
14:                     P̃ ← SynthProg(C.DSL, φ_i, ψ_i)
15:                     S' ← S' ∪ {[P_1, ..., P_{i-1}, P] | P ∈ P̃}
16:                 S ← S'
17:             Progs ← {Op(P_1, ..., P_n) | [P_1, ..., P_n] ∈ S}
18:     topProgs ← C.Ranker(Progs)
19:     if Δ[ψ] not defined then
20:         return topProgs
21:     else
22:         ψ_f ← Δ[ψ]
23:         F ← []
24:         V ← []
25:         for each r ∈ R where r has head ψ_f do
26:             let r be ψ_f := Op(ψ)
27:             if exists v such that ⟦Op⟧(P(I)) = v for all
28:               P ∈ topProgs and all inputs I in φ then
29:                 f ← ⟦r⟧()
30:                 F ← F · f
31:                 V ← V · v
32:         return {Disj(topProgs, F, V)}
```

Figure 7: Program synthesis algorithm

a *spec inference* function for the rule. Spec inference functions are based on the *witness functions* introduced in more detail in (Polozov and Gulwani 2015).

The spec inference function for an operator rule in the DSL is a function that computes specifications on the parameters of the operator given a specification on the operator itself. For a rule r in the DSL of the form $\psi := Op(\psi_1, \ldots, \psi_n)$ and an output specification ϕ on the output of this operator, $\mathsf{InferSpec}[r]$ is a function of the form $F(\phi, \psi_i, [P_1, ..., P_{i-1}])$ that returns a specification ϕ_i for the ith parameter of the operator given the program expressions already computed for the first $i-1$ parameters.

For example, for the $\mathsf{Concat}(res, res)$ operator in the text DSL, assume we have a specification that the output should be s_2 on an input s_1. Then a specification inferred for the first parameter is that it should produce any prefix of s_2. If we learn such a program P producing a prefix s', then we can infer a specification for the second parameter with respect to P: that the second parameter should produce any suffix s'' such that $s_2 = s' + s''$.

As another example of spec propagation, say we have a spec ϕ for the substring operator $\mathsf{Substr}(p, p)$ that on input "abcd" it should produce output "bc". In this case, the spec ϕ_1 inferred for the first parameter is that it should produce position 1 on the given input, while the spec ϕ_2 inferred for the second parameter is that it should produce position 3 on that input. As an example in the web DSL, assume we have a spec for the $\mathsf{Select}(fl, k)$ operator that the output should be a node n. The spec inferred for the first parameter fl is that this should be any collection that contains n, while the spec for k depends on the first parameter and requires that k be the index of n in that collection.

Although spec inference functions may be provided for most operators in a DSL, for some DSL symbols, especially terminal symbols, we may have domain-specific learning strategies to infer the programs directly. This is the base case where we no longer propagate specs downwards. For example, for an atomic regular expression token t in the text DSL we may have a spec propagated down from the position operator that the regex must have a match on input s that ends at position i in s. In this base case we will not propagate this spec further, but will simply check the possible atomic regexes in our DSL that satisfy this constraint and return those regexes. Such domain-specific learning strategies are provided in the Synth component of the configuration parameter C, which is a map from certain DSL symbols to functions for directly learning programs for that symbol from the given specification. For such a symbol ψ, we have $\mathsf{Synth}[\psi]$ is a function of the form $F(\phi)$ which returns a set of programs of type ψ that satisfy ϕ.

The final component of the configuration parameter C is the ranking function Ranker, which takes a set of programs and returns a list of top programs ordered according to some ranking criteria. For the text DSL we use ranking criteria as defined in (Gulwani 2011) which generally prefer "simpler" programs. In the web case, previous approaches (Polozov and Gulwani 2015) (and our baseline comparison) have been based on preferring the *most specific* programs for selecting nodes, but we shall describe an improvement over this in more detail after an outline of the main algorithm.

Synthesis algorithm

The synthesis algorithm is defined in Figure 7. The $\mathsf{SynthProg}(\mathsf{C}, \phi, \psi)$ returns a set of top-ranked programs of type ψ in the provided DSL that satisfy the given specification ϕ. To construct complete programs in the DSL we simply choose ψ to be the start symbol ψ_{st}.

In the first phase the algorithm computes the set of programs *Progs* that satisfy the spec (lines 3 to 17). If a specific synthesis function is already provided for this symbol in C then it simply uses that (line 5). Otherwise, it performs synthesis using the spec propagation approach. For each operator rule in the DSL with head ψ, we compute possible programs that use this operator at the top-level (lines 8 to 17). We do this by building the lists of parameter instantiations of the operator (stored in the variable S) until we have instantiations for all parameters of the operator.

For each parameter we infer a specification for that parameter using the instantiations of the previous parameters already computed (line 13). We then use this specification to synthesize programs for this parameter with a recursive call to the synthesis algorithm (line 14), and use these programs to grow the parameter instantiation list S (line 15). At line

17 we add all the possible programs using this rule operator to the set $Progs$. Note that in practice we make various optimizations over this general description, including caching the results of particular synthesis calls for a given specification which may often be repeated, clustering programs together that produce the same outputs, and setting recursion limits to bound the size of programs considered.

The next phase in the algorithm is to choose the top-ranked programs and to construct a disjunctive program if required (lines 18 to 32). We first obtain the top-ranked program from the set $Progs$ using the ranking function. If the symbol ψ is not a disjunctive symbol in the DSL then we simply return this top-ranked set (line 20). Otherwise we synthesize a disjunctive program for the symbol ψ.

We first obtain the feature calculator symbol ψ_f and initialize the feature calculator vector F and the target feature values vector V (lines 22 to 24). The synthesizer must now select the features it will include in the disjunctive program to choose between different disjuncts. These features are selected as all the ones that yield the same value across all the training examples provided to the algorithm. This is because the goal is to synthesize a disjunctive program that will prefer disjuncts whose result is most similar to the training examples.

For example, the text DSL includes the NotNull and NumChars features which indicate whether the output is null and the size of the output string respectively. Hence a program to address the text extraction task in Figure 1 would require NotNull to be True and NumChars to be 9 since those feature values are consistent across any example outputs in Figure 1 that may be given to the synthesizer. Similarly, in the web case, the failure to return any result (or an empty result) is just a feature of the output that is handled in the same way: if all the example outputs given for a task are non-empty, then the synthesizer will include the NonEmpty feature in the disjunctive program with target value of True.

Hence for each feature calculator rule in the DSL with head ψ_f, the synthesizer checks if the top programs yield the same value for this feature in all the given example inputs (line 27). If such a consistent value v exists then we add it to the target value vector V and add the feature calculator function to the vector F. Finally, we return the single disjunctive program with the Disj operator using the vectors F and V, and the top-ranked programs as the disjuncts.

Disjunct learning

The algorithm that we have described chooses the disjuncts in the final program using the top-ranked programs (in our experiments we found a cut-off of top 10 disjuncts yields an effective balance of performance and robustness in practice). In the case of the text DSL where we performed disjunctive synthesis on the substring operator, we found that the ranking used by the underlying Flash Fill system (Gulwani 2011) produced effective disjuncts. However, in the case of web extraction the existing synthesis systems (Polozov and Gulwani 2015; Le and Gulwani 2014) do not provide adequate disjuncts for the filter operators, as these systems tend to favor the most specific condition possible or other heuristic techniques to choose the filtering condition, which do not work well in practice.

With the disjunctive approach we needed to consider more general conditions as possible disjuncts to guard against over-fitting, but we cannot include all possibilities because, as discussed in the introduction, there may be an exponential number of possible conjunctive conditions c for the filter operator in the web DSL (Figure 3). Instead we describe a learning strategy for disjuncts that considers the maximal (the most specific) as well as a set of minimal (least restrictive) conjunctions that satisfy the specification. This disjunct learning is implemented in the $\mathsf{Synth}[c](\phi)$ function which synthesizes conditions for the filter operator.

The spec ϕ given to this function specifies that the computed condition expression P must hold for a subset N' of the input node set N. The learning function first computes all the atomic predicates $A_1, ..., A_n$ of symbol c that satisfy the spec. The top-ranked disjunct it considers is the most specific condition $P_1 = A_1 \wedge ... \wedge A_n$. It then computes the remaining disjuncts as the smallest conjunctions that produce the same filtered set of nodes as P_1 on the input set (hence are semantically equivalent programs over the given input). As this computation of minimals is equivalent to the minimal set cover problem which is NP-complete, we use a greedy approximation algorithm (Chvatal 1979) to compute the minimals efficiently. We also ensure that every atomic predicate A_i is included in some minimal, in order to produce an effective set of disjuncts that cover all predicates in the disjunctive program.

Evaluation

We have implemented our DSLs and algorithm for disjunctive program synthesis for both the text and web extraction domains using the Microsoft PROSE framework for program synthesis[1]. In this section we report our results over a set of benchmarks obtained from a product team that is interested in a PBE technology for extracting information from emails. Users often get emails regularly from the same provider such as flight itineraries, hotel bookings or purchase receipts, and the product team would like a PBE feature that allows users to create work flows for extracting important content whenever a new email arrives. Users can give examples to the PBE system by highlighting important fields in existing emails. However, in this scenario the web extraction DSL alone does not suffice, as many of the provided benchmarks include substring extractions from nodes in the HTML. Hence we addressed these benchmarks with the disjunctive DSL $\mathcal{DL}_{tw}$ which incorporates both the web and text DSLs, so the extraction program first selects a node and then performs a substring extraction on its text content.

The benchmarks we obtained covered 47 extraction tasks from 8 different providers. 34 tasks required substring extractions on the node content rather than just node selections. For each extraction task, we had around 10 test instances of an input and the desired output. In each instance the input was an email and the output was a text string of the field to be extracted, e.g. the flight number from a flight confirmation email.

[1]https://microsoft.github.io/prose/

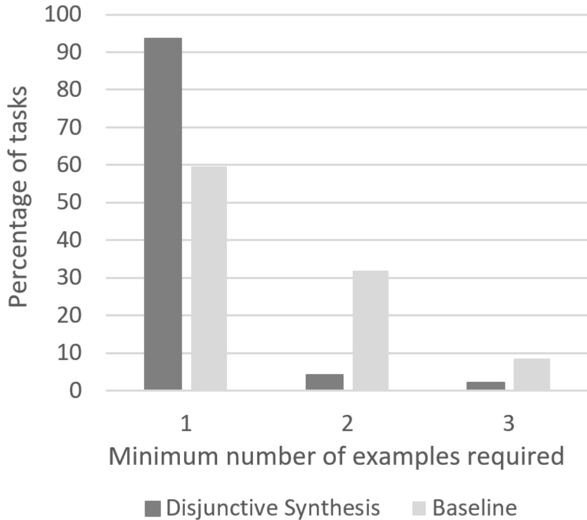

Figure 8: Number of examples required for correct synthesis

An important consideration for the product team is for the system to learn robust programs that do not easily fail on new inputs. The measure of robustness we use is the *minimum number of examples* required to learn a correct a program for a given task. More precisely, the minimum number of examples is the minimum number of input-output instances that were required as examples to be given to the system such that the program it synthesized succeeded on all the remaining instances.

We evaluated our disjunctive learning system against the baseline that does not include any disjunctive operators in the same DSL. Figure 8 shows the comparison of the minimum number of examples required by our disjunctive approach and the baseline. In summary, the proportion of tasks for which a correct program was learnt from one example increased from 59.6% to 93.6% with our disjunctive approach.

We note that these results show the *minimum* number of examples required to learn a correct program, rather than the number of examples a user may actually give in practice, which could be many more. This is because a user does not know which examples will be useful for the system (i.e. a representative sample covering all "corner cases") and while they may make the effort to give many more examples, they may only encounter the relevant or useful examples when the system fails to work on some future email. This uncertainty of not knowing which examples to give makes the difference between requiring one or more examples very important for the user experience and product quality.

In terms of performance, there was an average of 2.9x slowdown in the disjunctive case, which is expected since disjunctive programs perform more computations as they consider multiple alternative branches. However, most test instances in the disjunctive case completed in under half a second and all completed in under 2 seconds.

Related Work

There has been significant work on improving the robustness of program synthesis systems, ranging from approaches to improve the ranking between candidate programs to expanding on the underlying DSL structures.

Ranking

Examples are a severe under-specification of the user's intent in many useful task domains. PBE systems address this challenge by leveraging a ranking scheme to select between different programs that are consistent with the examples provided by the user. The ranking can be a function of the program structure or additional test inputs. We extend this line of work by making ranking dynamic, i.e., a function of each new input, and incorporating it into the execution semantics of programs on that input. This lifts the restriction of ranking decisions being limited to the synthesis stage and permits their incorporation in the program semantics in order to make use of the additional information available at execution time.

Ranking based on program structure Many PBE systems leverage a ranking scheme over program structures to pick the highest ranked program from the underlying DSL that satisfies the examples. The ranking can either be performed in a phase subsequent to the one that identifies the many programs that are consistent with the examples (Singh and Gulwani 2015), or it can be in-built as part of the search process (Menon et al. 2013; Balog et al. 2017). In this work, we show how to leverage these existing ranking methodologies and build over them to solve a harder problem of dealing with new forms of inputs that have not been encountered in the synthesis phase.

Ranking based on additional inputs In some PBE settings, the synthesizer often has access to some additional inputs on which the intended program is supposed to be executed, such as in a spreadsheet column transformation where the whole input column is available in addition to the few input-output examples manually provided by the user. In these cases the PBE system can use semi-supervised learning approaches to utilise the additional inputs to improve ranking. Singh showed how to leverage additional inputs to guess a reduction in the search space with the goal to both speed up synthesis and rank programs better (Singh 2016). Ellis and Gulwani observed that the additional inputs can be used to re-rank programs based on similarity of the outputs produced by those programs to the outputs produced on the training/example inputs provided by the user (Ellis and Gulwani 2017). Our approach also leverages additional inputs, but addresses the settings where additional inputs are not available at learning time and may only be encountered on some future execution of the program. Furthermore, our disjunctive technique allows using different programs (disjuncts) for different inputs, since instead of forcing a premature selection of a program at the synthesis stage, we hold onto that decision until needed (when the actual execution input becomes available).

Synthesis using richer structures

Another line of related work deals with learning over richer (conditional) program representations or latent (neural) program representations, which require many examples for effective learning. Our work has aspects related to both approaches in that it learns disjunctive program structures whose execution semantics depends on a separate feature-based scoring mechanism. This allows us to learn robust transformations from very few examples.

Conditional Program Synthesis PBE has been applied to learning programs that behave correctly over rich input spaces or types. One strategy is to extend the underlying DSL that provides the syntactic bias to learn more expressive conditional tasks with explicit conditional constructs (Kini and Gulwani 2015). Another approach has been to learn decision trees over multiple DSL expressions, each of which correctly handles a different sub-class of the inputs (Alur, Radhakrishna, and Udupa 2017). While these approaches expand the scope and expressivity of PBE systems to conditional tasks, learning the right predicate or a decision tree often requires many examples. In contrast to both these lines of work, we learn multiple DSL expressions (disjuncts), each of which handles the same set of input-output examples. This allows us to learn robust programs from a much smaller set of examples. We also use a feature-based scoring criterion that compares the execution results of the disjuncts on a new input to make the choice between different disjuncts, as opposed to learning an independent predicate or decision tree over the input states only.

Neural Program Synthesis Neural program induction involves generating outputs for new inputs by using a latent program representation induced by learning some form of neural controller. Various forms of neural controllers have been proposed such as ones that have the ability to read/write to an external memory tape (Graves et al. 2016), stack augmented neural controller (Joulin and Mikolov 2015), or even neural networks augmented with basic arithmetic and logic operations (Neelakantan, Le, and Sutskever 2015). In contrast, we learn multiple program structures and our execution semantics allows selecting between them after executing all of them on a new input.

Noise handling Devlin et.al. showed how to train neural models to deal with noise such as typos in the user-provided examples (Devlin et al. 2017). Our work instead handles the rich variation that can occur in the unseen inputs on which the user intends to run the synthesized program.

Conclusion and Future Work

In this paper, we address the problem of one-shot learning for programming by example systems when representative inputs may not be available at learning time, and present a new computational model for program synthesis from examples. The key challenge with traditional example-based synthesis is to deal with the ambiguity that is inherent in the small number of examples provided by the user, as these examples may not be representative of all possible variations in the input domain. Existing synthesis systems need to learn a program from the limited training data, and are forced to make a premature decision to select one program over many others that satisfy the given examples. In contrast, we suggest postponing that decision to when needed: at program execution time (when the execution input is available) rather than training or synthesis time. Our computational model is that of a disjunction of programs, in which ranking decisions are made at program execution time as a choice between different disjuncts that is controlled by a scoring function that picks disjuncts based on the outputs they generate on the given input. The feature-based scoring function picks the program in the disjunction of programs that yields an output that is most similar to the training outputs, based on the set of features that are included in the DSL.

We have presented a generic, domain-agnostic framework in which DSL designers may add disjunctive support for appropriate components in arbitrary DSLs. Our framework is designed to save them the algorithmic and engineering effort to otherwise implement disjunctive programs and their synthesis. The DSL designer simply needs to provide a DSL, specify which language components to make disjunctive, and the features of the outputs that would be required to choose between different disjuncts. These design choices are commonly informed by empirical studies in a concrete application domain, based on investigating which components of the program the ranking usually gets wrong in the test scenarios and what properties of the outputs usually distinguish the correct results from incorrect ones. We have investigated instantiations of our framework in the concrete domains of extracting fields from web pages and extracting substrings from text strings, and reported the significant reduction in the number of examples required to learn correct programs.

Although the basic idea behind the disjunctive approach is to maintain top-ranked synthesized programs and choose between them at execution time, an important aspect of our approach is the ability to have disjunctive support at arbitrary sub-program levels in the DSL rather than only for complete programs. This is because (1) disjuncts are normally required for certain DSL operators—simulating them only at the top level would lead to a combinatorial explosion (e.g., 10 disjuncts for substring and 10 for node selection in $\mathcal{DL}_{tw}$ leads to 100 possible complete programs) which can either cause performance issues or discarding of potentially correct disjuncts. (2) More importantly, features are required on the sub-program outputs rather than the final outputs of the program in order to effectively choose between different disjuncts.

Our computational model is also related to common software engineering concepts such as exception handling and assertion checking, which are designed to improve the robustness and quality of programs. Programmers use exception handling mechanisms to take care of cases that are different or to recover gracefully from unexpected executions. They also make assertions on the output of certain operations in order to ensure that certain desirable properties hold, as it is often easier to check the results of a computation than

to prove the computation correct in all possible input scenarios. Our computational model consists of a disjunction of programs, with a feature-based scoring function to select between different disjuncts based on the outputs they generate. In this respect the various disjuncts can be seen as a collection of exception handling routines, and the features computed on the outputs of disjuncts are akin to assertions a programmer may make in certain parts of their code.

An interesting direction for future work is to research scenarios that require more sophisticated scoring functions that may leverage various features of the disjunctive programs and their outputs, perhaps using weighted combinations of feature values. Another direction to explore is generalizing the expressiveness of disjunctive programs to handle more complex conditional tasks. In our technique for synthesizing disjunctive programs we select a subset of disjuncts that satisfy all examples, but in theory we may investigate relaxations of this constraint to also consider program disjuncts that may not each satisfy all the examples. With appropriate selection mechanisms, these disjuncts may be combined in ways to guarantee that the overall disjunctive program satisfies all training examples, while at the same time addressing more expressive tasks that would otherwise require conditional operators in the DSL.

Finally, it would be interesting to explore the applicability of the disjunctive approach to other application domains. In theory, the disjunctive approach is applicable to any PBE domain since choosing between top-ranked programs is a common challenge. Broad categories of practical domains include extraction from different data formats (e.g. extracting fields, lists or tables from PDF, JSON, XML or other formats) and transformations between different data formats (e.g. transformations between different JSON/XML formats, table to table, number/date transformations, or even code transformations such as refactoring, formatting, etc). Many of these domains involve standard operations for selection, pattern matching or filtering, which often yield numerous candidate expressions that can satisfy a small set of examples. For instance, filtering regions in a PDF document by certain properties is very similar to selecting DOM nodes in a web document. We therefore expect DSLs in such domains to also benefit significantly from disjunctive support.

Acknowledgments

We thank the anonymous reviewers for providing very valuable feedback that helped to improve this paper.

References

Alur, R.; Radhakrishna, A.; and Udupa, A. 2017. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 319–336.

Balog, M.; Gaunt, A. L.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2017. Deepcoder: Learning to write programs. In *ICLR*.

Chvatal, V. 1979. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research* 4(3):233–235.

Devlin, J.; Uesato, J.; Bhupatiraju, S.; Singh, R.; Mohamed, A.; and Kohli, P. 2017. Robustfill: Neural program learning under noisy I/O. In *ICML*.

Ellis, K., and Gulwani, S. 2017. Learning to learn programs from examples: Going beyond program structure. In *IJCAI*.

Graves, A.; Wayne, G.; Reynolds, M.; Harley, T.; Danihelka, I.; Grabska-Barwinska, A.; Colmenarejo, S. G.; Grefenstette, E.; Ramalho, T.; Agapiou, J.; Badia, A. P.; Hermann, K. M.; Zwols, Y.; Ostrovski, G.; Cain, A.; King, H.; Summerfield, C.; Blunsom, P.; Kavukcuoglu, K.; and Hassabis, D. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538(7626):471–476.

Gulwani, S.; Harris, W. R.; and Singh, R. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55(8).

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

Joulin, A., and Mikolov, T. 2015. Inferring algorithmic patterns with stack-augmented recurrent nets. In *NIPS*, 190–198.

Kini, D., and Gulwani, S. 2015. Flashnormalize: Programming by examples for text normalization. In *IJCAI*, 776–783.

Lau, T. A.; Wolfman, S. A.; Domingos, P.; and Weld, D. S. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53(1-2):111–156.

Le, V., and Gulwani, S. 2014. Flashextract: a framework for data extraction by examples. In O'Boyle, M. F. P., and Pingali, K., eds., *PLDI*, 55. ACM.

Lieberman, H., ed. 2001. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers.

Manshadi, M. H.; Gildea, D.; and Allen, J. F. 2013. Integrating programming by example and natural language programming. In desJardins, M., and Littman, M. L., eds., *AAAI*. AAAI Press.

Menon, A. K.; Tamuz, O.; Gulwani, S.; Lampson, B. W.; and Kalai, A. 2013. A Machine Learning Framework for Programming by Example. In *ICML (1)*, volume 28 of *JMLR Proceedings*, 187–195. JMLR.org.

Neelakantan, A.; Le, Q. V.; and Sutskever, I. 2015. Neural programmer: Inducing latent programs with gradient descent. *CoRR* abs/1511.04834.

Polozov, O., and Gulwani, S. 2015. FlashMeta: a framework for inductive program synthesis. In Aldrich, J., and Eugster, P., eds., *OOPSLA*, 107–126. ACM.

Raza, M., and Gulwani, S. 2017. Automated data extraction using predictive program synthesis. In Singh, S. P., and Markovitch, S., eds., *AAAI*, 882–890. AAAI Press.

Singh, R., and Gulwani, S. 2015. Predicting a correct program in programming by example. In *Computer Aided Verification (CAV)*.

Singh, R. 2016. BlinkFill: Semi-supervised Programming by Example for Syntactic String Transformations. In *PVLDB*, 816–827.