Programming by Examples: PL meets ML

Sumit Gulwani [a], Prateek Jain [b]

[a] *Microsoft Corporation, Redmond, USA*
[b] *Microsoft Research, Bangalore, India*

Abstract. Programming by Examples (PBE) involves synthesizing intended programs in an underlying domain-specific programming language (DSL) from example-based specifications. This new frontier in AI enables computer users, 99% of whom are non-programmers, to create scripts to automate repetitive tasks. PBE can provide 10-100x productivity increase for data scientists, business users, and developers for various task domains like string/number/date transformations, structured table extraction from log files/web pages/PDF/semi-structured spreadsheets, transforming JSON from one format to another, repetitive text editing, repetitive code refactoring and formatting. PBE capabilities can be surfaced using GUI-based tools, code editors, or notebooks, and the code can be synthesized in various target languages like Java or even PySpark to facilitate efficient execution on big data.

There are three key components in a PBE system. (i) A search algorithm that can efficiently search for programs that are consistent with the examples provided by the user. We leverage a divide-and-conquer based deductive search paradigm that inductively reduces the problem of synthesizing a program expression of a certain kind that satisfies a given specification into sub-problems that refer to sub-expressions or sub-specifications. (ii) Program ranking techniques to pick an intended program from among the many that satisfy the examples provided by the user. (iii) User interaction models to facilitate usability and debuggability.

Each of these PBE components leverage both symbolic reasoning and heuristics. We make the case for synthesizing these heuristics from training data using appropriate machine learning methods. In particular, we use neural-guided heuristics to resolve any resulting non-determinism in the search process. Similarly, our ML-based ranking techniques, which leverage features of program structure and program outputs, are often able to select an intended program from among the many that satisfy the examples. Finally, Our active-learning-based user interaction models, which leverage clustering of input data and semantic differences between multiple synthesized programs, facilitate a bot-like conversation with the user to aid usability and debuggability. That is our algorithms that deeply integrate neural techniques with symbolic computation can not only lead to better heuristics, but can also enable easier development, maintenance, and even personalization of a PBE system.

Keywords. Program synthesis, Programming by Examples, Search algorithm, Program ranking, Active learning, Data wrangling

[0] This is an extended version of the article with the same title that appeared in the proceedings for APLAS 2017 as an invited-talk contribution and was published by Springer [13]. This revision includes more than 5 pages of new content, which includes new figures and references along with expansion and better phrasing of some earlier content.

1. Introduction

Program Synthesis is the task of synthesizing a program that satisfies a given specification [15]. The traditional view of program synthesis has been to synthesize programs from logical specifications that relate the inputs and outputs of the program. A typical academic exercise in program synthesis is to synthesize complicated algorithms such as sorting algorithms [43], graph algorithms [18], and bitvector algorithms [19]. For instance, the logical specification for a sorting algorithm would state that the sorting algorithm takes as input an array $A[1 :: n]$ and outputs another array $B[1 :: n]$ s.t. B is a permutation of A, and B is sorted, i.e.,

$$\forall 1 \leq i < n : B[i] \leq B[i+1] \ \wedge$$
$$\exists \sigma, \text{ a permutation of } 1 \ldots n \text{ such that } \forall 1 \leq i < n : B[i] = A[\sigma(i)]$$

Programming by Examples (PBE) is a sub-field of program synthesis, where the specification consists of input-output examples, or more generally, output properties over given input states [11]. PBE has emerged as a favorable paradigm for two key reasons: (i) the example-based specification in PBE makes PBE more tractable than general program synthesis because it involves reasoning over concrete program states (Section 4 discusses the underlying search techniques). As a result, we can synthesize more complicated and larger programs than what was possible earlier, and we can do that very efficiently and often in real-time to facilitate usability. (ii) Example-based specifications are much easier for the users to provide in many scenarios. This not only increases the usability of synthesis technologies for developers, but also broadens the applicability to end users. This is highly significant since 99% of computer users do not know programming and would find it extremely difficult to write down logical specifications.

The advantages of PBE also present some unique challenges. First, examples are highly ambiguous form of user's intent. There are too many programs that match a small number of examples. Requiring the user to provide a large number of examples to narrow down the ambiguity affects the usability of such systems. We discuss (in Section 5) how program ranking techniques can be designed to guess an intended program from among the many that satisfy a few representative user-provided examples. Secondly, we need technologies that can help the user identify representative inputs on which to provide examples. We discuss this in Section 6.

This article is organized as follows. Section 2 discusses some key applications of PBE. In Section 3, we provide our perspectives on how PBE differs from Machine Learning (ML), both of which aim to learn from examples. We make the case that instead of thinking about ML and PBE as alternatives, ML can actually be used to create better PBE systems. The next few sections discuss opportunities for such an integration for each of the three key components of PBE: search algorithm (Section 4), ranking (Section 5), and interactivity (Section 6). Section 7 presents some directions for future work.

2. Applications

The two killer applications for programming by examples today are in the space of data transformations/wrangling and code transformations.

Figure 1. Consider the task of extracting a structured table (shown on the right side) from the custom text file (shown on the left side). This would typically require writing a complicated parsing script involving regular expressions. In contrast, the FlashExtract PBE technology [22] allows automation of such tasks from few examples. This figure illustrates a user experience around this PBE technology. Once the user highlights few examples (often one or two) of a field (using a color unique to that field), FlashExtract synthesizes a program and executes it to extract the other instances and arranges them in a new column in the output table.

2.1. Data wrangling

Data Wrangling refers to the process of transforming the data from its raw format to a more structured format that is amenable to analysis and visualization. It is estimated that data scientists spend 80% of their time wrangling data. Data is locked up into documents of various types such as text/log files, semi-structured spreadsheets, webpages, JSON/XML, and PDF documents. These documents offer their creators great flexibility in storing and organizing hierarchical data by combining presentation/formatting with the underlying data. However, this makes it extremely hard to extract the underlying data for several tasks such as processing, querying, altering the presentation view, or transforming data to another storage format. PBE can make data wrangling easier and faster.

Extraction: A first step in a data wrangling pipeline is often that of ingesting or extracting tabular data from semi-structured formats such as text/log files, web pages, and XML/JSON documents. These documents offer their creators great flexibility in storing and organizing hierarchical data by combining presentation/formatting with the underlying data. However, this makes it extremely hard to extract the relevant data. The FlashExtract PBE technology allows extracting structured (tabular or hierarchical) data out of semi-structured documents from examples [22]. For each field in the output data schema, the user provides positive/negative instances of that field and FlashExtract generates a program to extract all instances of that field. The FlashExtract technology ships as the ConvertFrom-String cmdlet in Powershell in Windows 10, wherein the user provides examples of the strings to be extracted by inserting tags around them in test. The FlashEx-

1	Email	Column 2
2	Nancy.FreeHafer@fourthcoffee.com	freehafer, nancy
3	Andrew.Cencici@northwindtraders.com	cencici, andrew
4	Jan.Kotas@litwareinc.com	kotas, jan
5	Mariya.Sergienko@gradicdesigninstitute.com	sergienko, mariya
6	Steven.Thorpe@northwindtraders.com	thorpe, steven
7	Michael.Neipper@northwindtraders.com	neipper, michael
8	Robert.Zare@northwindtraders.com	zare, robert
9	Laura.Giussani@adventure-works.com	giussani, laura
10	Anne.HL@northwindtraders.com	hl, anne
11	Alexander.David@contoso.com	david, alexander
12	Kim.Shane@northwindtraders.com	shane, kim
13	Manish.Chopra@northwindtraders.com	chopra, manish
14	Gerwald.Oberleitner@northwindtraders.com	oberleitner, gerwald
15	Amr.Zaki@northwindtraders.com	zaki, amr
16	Yvonne.McKay@northwindtraders.com	mckay, yvonne
17	Amanda.Pinto@northwindtraders.com	pinto, amanda

Figure 2. Consider the collection of email addresses in the first column. Suppose the user wants to extract last name and first name and format them as illustrated in the second column. The Flash Fill feature [10] in Excel 2013 (and onwards) allows automation of such repetitive string transformations from few examples. In this case, once the user performs one instance of the desired transformation (row 2, column 2) and proceeds to transforming another instance (row 3, column 2), Flash Fill learns a program concat(ToLower(substr(v,WordToken,12), conststr(", "), ToLower(substr(v,WordToken,1))) that extracts the first two words in input string v (first column), converts them to lowercase, and concatenates them separated by a comma and space.

	A	**B**	**C**	**D**	**E**	...	**R**
1		value	year	value	year		Comments
2	Albania	1,000	1950	930	1981		FRA 1
3	Austria	3,139	1951	3,177	1955		FRA 3
4	Belgium	541	1947	601	1950		
5	Bulgaria	2,964	1947	3,259	1958		FRA 1
6	Czech ...	2,416	1950	2,503	1960		NC

...

(a) **Input:** Semi-structured spreadsheet

	A	**B**	**C**	**D**
1	Albania	1,000	1950	FRA 1
2	Albania	930	1981	FRA 1
...				
5	Austria	3,139	1951	FRA 3
6	Austria	3,177	1955	FRA 3
...				
9	Belgium	541	1947	
10	Belgium	601	1950	
...				

(b) **Output:** Relational table

Figure 3. Consider the task of extracting a relational table (b) from the semi-structured spreadsheet (a). The FlashRelate technology [4] allows automation of such tasks from few examples. In this scenario, once the user provides a couple of examples of tuples in the output table (for instance, the highlighted ones), FlashRelate synthesizes a script and executes that script to extract other similar tuples from the input spreadsheet.

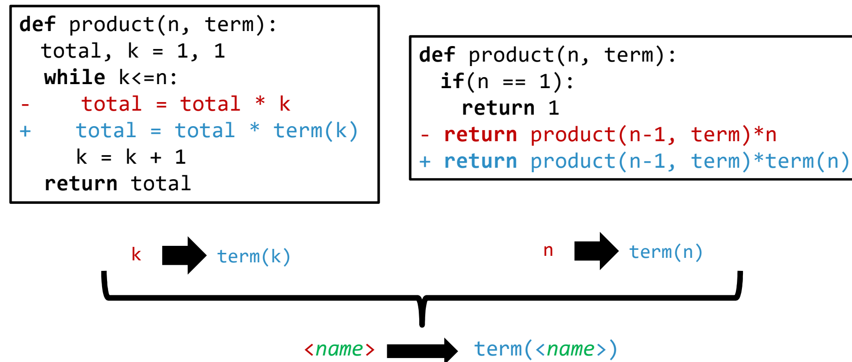

Figure 4. This figure shows two incorrect student attempts to a programming problem with a similar kind of fault, wherein the student missed applying the `term` function. The incorrect statement in each attempt is colored red while the teacher's correction is shown in blue. The Refazer tool [33] can generalize such similar teacher corrections to a more general rule that can be applied to automatically fix other students' attempts with a similar fault.

tract technology also ships in Azure OMS (Operations Management Suite), where it enables extraction of custom fields from log files. Figure 1 illustrates use of this technology to extract structured tabular data from a text file with custom format.

Transformation: The Flash Fill feature, released in Excel 2013 and beyond, is a PBE technology for automating syntactic string transformations, such as converting "FirstName LastName" into "LastName, FirstName" [10]. Figure 2 provides an illustration of the Flash Fill feature. PBE can also facilitate more sophisticated string transformations that require lookup into other tables [36]. PBE is also a very natural fit for automating transformations of other data types such as numbers [37] and dates [39].

Formatting: Another useful application of PBE is in the space of formatting data tables. This can be useful to convert semi-structured tables found commonly in spreadsheets into proper relational tables [4], or for re-pivoting the underlying hierarchical data that has been locked into a two-dimensional tabular format [16]. Figure 3 provides illustrates use of a PBE technology for performing example-based formatting. PBE can also be useful in automating repetitive formatting in a PowerPoint slide deck such as converting all red colored text into green, or switching the direction of all horizontal arrows [31].

2.2. Code Transformations

There are several situations where repetitive code transformations need to be performed and examples can be used to automate this tedious task.

A standard scenario is that of general code refactoring. As software evolves, developers edit program source code to add features, fix bugs, or refactor it for readability, modularity, or performance improvements. For instance, to apply an API update, a developer needs to locate all references to the old API and consistently replace them with the new API. Examples can be used to infer such edits from a few examples [33].

Another important scenario is that of *application migration*—whether it is about moving from on-prem to the cloud, or from one framework to another, or simply moving

```
using System;
public class Program {
  public static int[] Puzzle(int[] a) {

    int [] b= new int[a.Length];
    for (int i = 0; i < a.Length; i++)
      {
      b[a.Length-i]=a[i-1];
    }
    return b;          i = 1

                         i <= a.Length
    }
}
```

```
using System;
public class Program {
  public static int[] Puzzle(int[] a) {
    int front, back, temp;
    front = 0;
    back = a.Length-1;
    temp = a[back];
    while (front > back)
    { a[back] = a[front];
      a[front] = temp;
      ++back;                 front <= back
      ++front;
      temp = a[back];
    }
    return a;
  }          --back
}
```

Figure 5. This figure shows two incorrect student attempts to the problem of reversing an array. The AutoGrader tool [40] can find small edits to an incorrect attempt (shown in red) that transforms the program into a version that satisfies a given reference set of test cases.

from an old version of a framework to a newer version to keep up with the march of technology. A significant effort is spent in performing repetitive edits to the underlying application code. In particular, for database migration, it is estimated that up to 40% of the developer effort can be spent in performing repetitive code changes in the application code.

Yet another interesting scenario is in the space of feedback generation for programming assignments in programming courses. For large classes such as massive open online courses (MOOCs), manually providing feedback to different students is an unfeasible burden on the teaching staff. We observe that student submissions that exhibit the same fault often need similar fixes. The PBE technology can be used to learn the common fixes from corrections made by teachers on few assignments, and then infer application of these fixes to the remaining assignments, forming basis for automatic feedback [33]. Figure 4 illustrates such a use case. Another possibility is to search for a set of small edits to the student's incorrect attempt to make it pass a reference set of test cases, as illustrated in Figure 5.

3. PL meets ML

It is interesting to compare PBE with Machine learning (ML) since both involve example-based training and prediction on new unseen data. PBE learns from very few examples, while ML typically requires large amount of training data. The models generated by PBE are human-readable (in fact, editable programs) unlike many black-box models produced by ML. PBE generates small scripts that are supposed to work with perfect precision on any new valid input, while ML can generate sophisticated models that can achieve high, but not necessarily perfect, precision on new varied inputs. Hence, given their complementary strengths, we believe that PBE is better suited for relatively simple well-defined tasks, while ML is better suited for sophisticated and fuzzy tasks.

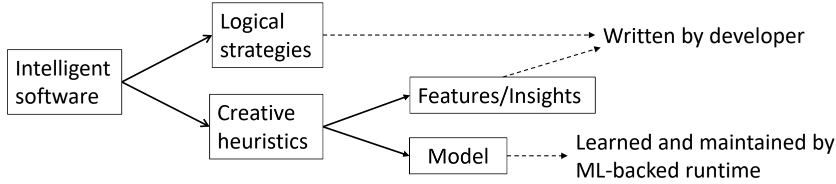

Figure 6. A proposal for development of intelligent software that facilitates increased developer productivity and increased software intelligence.

Recently, *neural program induction* has been proposed as a fully ML-based alternative to PBE. These techniques develop new neural architectures that learn how to generate 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 external memory tape [9], stack augmented neural controller [20], or even neural networks augmented with basic arithmetic and logic operations [27]. These approaches typically involve developing a continuous representation of the atomic operations of the network, and then using end-to-end training of a neural controller or reinforcement learning to learn the program behavior. While this is impressive, these techniques aren't a good fit for the PBE task domains of relatively simple well-defined tasks. This is because these techniques don't generate an interpretable model of the learned program, and typically require large computational resources and several thousands of input-output examples per synthesis task. We believe that a big opportunity awaits in carefully combining ML-based data-driven techniques with Programming Languages (PL)-based logical reasoning approaches to improve a standard PBE system as opposed to replacing it.

3.1. A perspective on PL meets ML

AI software often contains two intermingled parts: logical strategies + creative heuristics. Heuristics are difficult to author, debug, and maintain. Heuristics can be decomposed into two parts: insights/features + model/scoring function over those features. We propose that an AI developer refactors their intelligent code into logical strategies and declarative features while ML techniques are used to evolve an ideal model or scoring function over those insights with continued feedback from usage of the intelligent software. This has two advantages: (i) Increase in developers productivity, (ii) Increase in systems intelligence because of better heuristics and those that can adapt differently to different workloads or unpredictable environments (a statically fixed heuristic cannot achieve this).

Figure 6 illustrates this proposed modular construction of intelligent software. Developing an ML model in this framework (where the developer authors logical strategies and declarative insights) poses several interesting open questions as traditional ML techniques are not well-equipped to handle such declarative and symbolic frameworks. Moreover, even the boundary between declarative insights and ML-based models may be

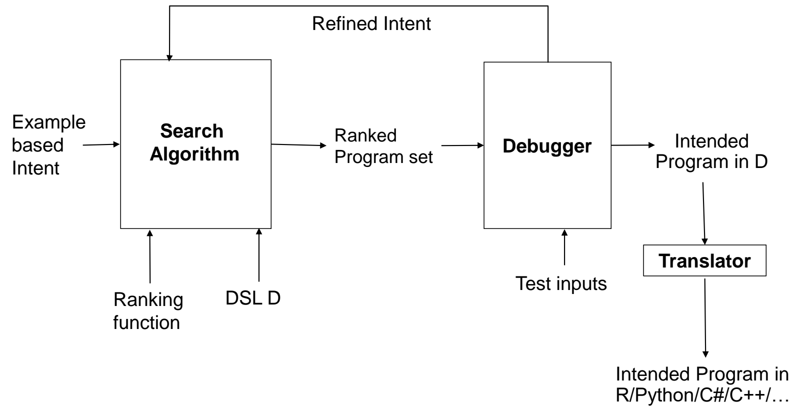

Figure 7. Programming-by-Examples Architecture. The search algorithm, parameterized by a domain-specific language (DSL) and a ranking function, synthesizes a ranked set of programs from the underlying DSL that are consistent with the examples provided by the user. The debugging component, which leverages additional test inputs, interacts with the user to refine the specification and the synthesis process is repeated. Once an intended program has been synthesized, it can be translated to a target language using standard syntax-directed translation.

fluid. Depending on the exact problem setting as well as the domain, the developer might want to decide which part of the system should follow deterministic logical reasoning and which part should be based on data-driven techniques.

3.2. Using ML to improve PBE

There are three key components in a PBE engine: search algorithm, ranking strategy, and user interaction models. Each of these components leverage various forms of heuristics. ML can be used to learn these heuristics, thereby improving the effectiveness and maintainability of the various PBE components. In particular, ML can be used to speed up the search process by predicting the success likelihood of various paths in the huge search space [21]. It can be used to learn a better ranking function [26]. It can be used to cluster test data and associate confidence measure over the outputs generated by the synthesized program to drive an effective active learning session with the user for debuggability [28].

4. Search Algorithm

Figure 7 shows the architecture of a PBE system. The most involved technical component is the search algorithm, which we discuss in this section. Section 4.1 and 4.2 describe the two key ingredients that form the foundation for designing this search algorithm. These ingredients are based on deterministic logical reasoning. Section 4.3 then discusses and speculates how machine learning can further help exploit the traditional PL-driven logical reasoning to obtain an even more efficient, real-time search algorithm for PBE.

String Expression E:=concat(E_1, E_2) | substr(E, P_1, P_2) | conststr(*String*)
Position P:= *Integer* | pos(x, R_1, R_2, k)

Figure 8. An example Domain Specific Language (DSL).substr, concat are operators to manipulate the string and conststr represents a constant string. pos operator identifies position of a particular pattern in the input x. *String* is any constant string and *Integer* is an arbitrary integer that can be negative as well.

4.1. Domain-specific Language

A key idea in program synthesis is to restrict the search space to an underlying domain-specific language (DSL) [1,12]. The DSL should be expressive enough to represent a wide variety of tasks in the underlying task domain, but also restricted enough to allow efficient search. We have designed many functional domain-specific languages for this purpose, each of which is characterized by a set of operators and a syntactic restriction on how those operators can be composed with each other (as opposed to allowing all possible type-safe composition of those operators) [11]. A DSL is typically specified as a context-free grammar that consists of one or more production rules for each non-terminal. The right hand side of a production rule can be either another non-terminal or an explicit set of program expressions or a program operator applied to some non-terminals.

For illustration, we present an extremely simple string manipulation grammar in Figure 8; this DSL is a heavily stripped down version of Flash Fill DSL [10]. The language has two key operators for string manipulations: a) substr operator which takes as input a string x, and two position expressions P_1 and P_2 that evaluate to positions/indices within the string x, and returns the substring between those positions, b) concat which concatenates the given expressions. The choice for position expression P includes the pos(x,R_1,R_2,k) operator, which returns the kth position within the string x such that (some suffix of) the left side of that position matches with regular expression R_1 and (some prefix of) the right side of that position matches with regular expression R_2.

For example, program given by,
concat(substr(Input, ε, “ ”, 1), substr(Input, “ ”, ε, -1), conststr(“@cs.colorado.edu”))
maps input “evan chang” into “evanchang@cs.colorado.edu”. Note that we overloaded concat operator to allow for more than 2 operands.

4.2. Deductive Search Methodology

A simple search strategy is to enumerate all programs in order of increasing size [1] by doing a bottom-up enumeration of the grammar. This can be done by maintaining a graph of reachable values starting from the input state in the user-provided example. This simply requires access to the executable semantics of the operators in the DSL. Bottom-up enumeration is very effective for small grammar fragments since executing operators forward is very fast. Some techniques have been proposed to increase the scalability of enumerative search: (i) divide and conquer that decomposes the problem of finding programs that satisfy all examples to that of finding programs, each of which satisfies some subset, and then combining those programs using conditional predicates [2]. (ii) operator-specific lifting functions that can compute the output set from input sets more efficiently than point-wise computation. Lifting functions are essentially the forward transformer for an operator [30].

Unfortunately, bottom-up enumeration does not scale to large grammars because there are often too many constants to start out with. Our search methodology combines

bottom-up enumeration with a novel top-down enumeration of the grammar. The top-down enumeration is goal-directed and requires pushing the specification across an operator using its inverse semantics. This is performed using *witness functions* that translate the specification for a program expression of the kind $F(e_1, e_2)$ to specifications for what the sub-expressions e_1 and e_2 should be. The bottom-up search first enumerates smaller sub-expressions before enumerating larger expressions. In contrast, the top-down search first fixes the top-part of an expression and then searches for its sub-expressions.

The overall top-down strategy is essentially a divide-and-conquer methodology that recursively reduces the problem of synthesizing a program expression e of a certain kind and that satisfies a certain specification ψ to simpler sub-problems (where the search is either over sub-expressions of e or over sub-specifications of ψ), followed by appropriately combining those results. The reduction logic for reducing a synthesis problem to simpler synthesis problems depends on the nature of the involved expression e and the inductive specification ψ. If e is a non-terminal in the grammar, then the sub-problems correspond to exploring the various production rules corresponding to e. If e is an operator application $F(e_1, e_2)$, then the sub-problems correspond to exploring multiple sub-goals for each parameter of that operator. As is usually the case with search algorithms, most of these explorations fail. PBE systems achieve real-time efficiency in practice by leveraging heuristics to predict which explorations are more likely to succeed and then either only explore those or explore them preferentially over others.

Machine learning techniques can be used to learn such heuristics in an effective manner. Below, we provide more details on one such method for a guided search in the deductive strategy [21].

4.3. ML-based Search Algorithm

A key ingredient of the top-down search methodology mentioned above is grammar enumeration where while searching for a program expression e of the non-terminal kind, we enumerate all the production rules corresponding to e to obtain a new set of search problems and recursively solve each one of them. The goal of this work [21] was to determine the best production rules that we should explore while ignoring certain production rules that are unlikely to provide a desired program. Now, it might seem a bit outlandish to claim that we can determine the correct production rule to explore before even exploring it!

However, many times the provided input-output specification itself provides clues to make such a decision accurately. For example, in the context of the DSL mentioned in Figure 8, lets consider an example where the input is “evan” and the desired output is “evan@cs.colorado.edu”. In this case, even before exploring the productions rules, it is fairly clear that we should apply the concat operator instead of substr operator; a correct program is concat(Input, conststr(“@cs.colorado.edu”)). Similarly, if our input is “xinyu feng” and the desired output is “xinyu” then it is clear that we should apply the substr operator; a correct program is substr(Input, 1, pos(Input, Alphanumeric, “ ”, 1)).

But, exploiting the structure in input-output examples along with production rules is quite challenging as these are non-homogeneous structures without a natural vector space representation. Building upon recent advances in natural language processing, our ML based approach uses a version of neural networks to exploit the structure in input-output examples to estimate the set of best possible production rules to explore. For-

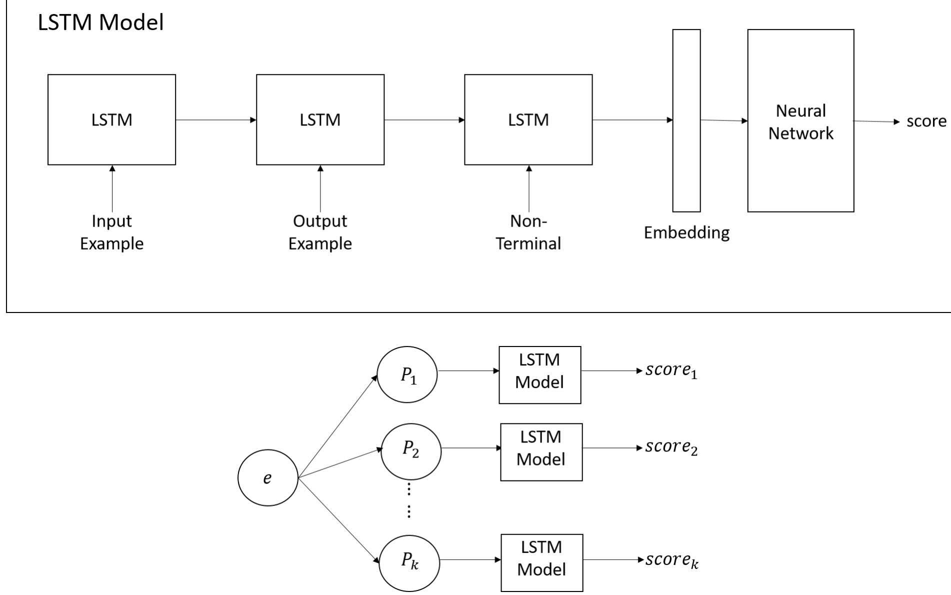

Figure 9. LSTM based model for computing score for the candidate set of production rules $P_1, \ldots, P_k$ during the grammar expansion process. The top figure shows details of the ML model used to compute score for a candidate production rule when placed in the context of the given input-output examples.

mally, given the input-output examples represented by ψ, and a set of candidate production rules $P_1, P_2, \ldots, P_k$ whose LHS is our current non-terminal e we compute a score $s_i = score(\psi, P_i)$ for each candidate rule P_i. This score reflects the probability of synthesis of a desired program if we select rule P_i for the given input-output examples ψ. Note that input-output example specification ψ changes during the search process as we decompose the problem into smaller sub-problems; hence for recursive grammars, we need to compute the scores every time we wish to explore a production rule.

For learning the scoring model, similar to [6], our method embeds input-output examples in a vector space using a popular neural network technique called LSTM (Long Short-Term Memory) [17]. The embedding of a given input-output specification essentially captures its critical features, e.g., if input is a substring of output or if output is a substring of input etc. We then match this embedding against an embedding of the production rule P_i to generate a joint embedding of (ψ, P_i) pair. We then learn a neural network based function to map this joint embedding to the final score. Now for prediction, given scores $s_1, s_2, \ldots, s_k$, we select branches with top most scores with large enough margin, i.e., we select rules $P_{i_1}, \ldots, P_{i_\ell}$ for exploration where $s_{i_1} \geq s_{i_2} \cdots \geq s_{i_\ell}$ and $s_{i_\ell} - s_{i_{\ell+1}} \geq \tau$; $\tau > 0$ is a threshold parameter that we discuss later.

See Figure 9 for an overview of our LSTM based model and the entire pipeline.

To test our technique, we applied it to a much more expressive version of the Flash Fill DSL [10] that includes operators over richer data types such as numbers and dates. For training and testing our technique, we collected 375 benchmarks from real-world customer scenarios. Each benchmark consists of a set of input strings and their corresponding outputs. We selected 300 benchmarks for training and remaining 75 for testing.

Metric	**PROSE**	**DC**	**RF**	**NGDS**
Accuracy (% of 73)	67.12	32.88	16.44	**68.49**
Speed-up (× PROSE)	1.00	1.51	0.26	**1.67**

Table 1. (Table 1 of [21]) Accuracy and average speed-up of NGDS vs. baseline methods. Accuracies are computed on a test set of 73 tasks. *Speed-up* of a method is the geometric mean of it's per-task speed-up (ratio of synthesis time of PROSE and of the method) when restricted to a subset of tasks with PROSE's synthesis time is ≥ 0.5 sec.

For each training benchmark, we generated top 1000 programs using existing top-down enumerative approach and logged relevant information for our grammar enumeration. For example, when we want to expand certain grammar symbol (say expr in Figure 8) with the goal of mapping given inputs to required outputs, we log all the relevant production rules P_i, $\forall i$ (i.e., rules in Line 1 of Figure 8). We also log the score s_i of the top program that is generated by applying production rule P_i. That is, each training instance is (ψ, P_i, s_i) for a given node with input-output examples ψ. We use standard DNN tools to train the model for grammar enumeration. That is, whenever we need to decide on which production rule to select for expansion, we compute score for each possible rule P_i and select the rules whose scores are higher than the remaining rules by a margin of τ.

Threshold τ is an interesting knob that helps decide between exploration vs exploitation. That is, smaller τ implies that we trust our ML model completely and select the best choice presented by the model. On the other hand, larger τ forces system to be more conservative and use ML model sparingly when it is highly confident. For example, on the 75 test benchmarks, setting $\tau = 0$ i.e. selecting ML model's predicted production rule for every grammar expansion decision, we select the best production rule 92% of the instances. Unfortunately, selecting wrong production rule 8% of the times might lead to synthesis of a relatively poor program or in worst case, no program. However, by increasing $\tau = 0.1$ we can increase our chances of selection of the best production rule to 99%. Although in this case, for nearly 50% instances the ML model does not differentiate between production rules, i.e., the predicted scores are all within $\tau = 0.1$ length interval. Hence, we enumerate all the rules in about 50% of the grammar expansion instances and are able to prune production rules in only 50% cases. Nonetheless, this itself leads to impressive computation time improvement of up to 12x over naïve exploration for many challenging test benchmarks. Table 1 presents average speed-up obtained by our method (NGDS) over the naïve exploration technique used by the PROSE system as well as two of the existing deep learning based techniques: RobustFill [6] and DeepCoder [3]. RobustFill (RF) does not leverage the deductive search structure and instead tries to synthesize programs end-to-end using deep learning. As seen by the table, the accuracy of such a system is not very good and in fact, even the overall computation cost is also significantly worse than NGDS. DeepCoder (DC) is a technique that imposes a static priority over operators to be explored during deductive search. So unlike NGDS, DeepCoder does not change the priority list over operators with each step's input-output pair.

5. Ranking

Examples are a severe under-specification of the user's intent in many useful task domains. As a result, several programs in an underlying DSL are consistent with a given

set of training examples, but are unintended, i.e., they would produce an undesired output on some test inputs. Usability concerns further necessitate that we learn an intended program from as few examples as possible.

PBE systems address this challenge by leveraging a ranking scheme to select between different programs consistent with the examples provided by the user. Ideally, we want to bias the ranking of programs so that *natural* programs are ranked higher. While the notion of *naturalness* of programs is highly subjective, still in practice, one can see certain succinct patterns associated with natural programs that one can try to capture via real-world training datasets.

The ranking can either be performed in a phase subsequent to the one that identifies the many programs that are consistent with the examples [38], or it can be in-built as part of the search process [25,3]. Furthermore, the ranking can be a function of the program structure or additional test inputs.

5.1. Ranking based on Program Structure

A basic ranking scheme can be specified by defining a preference order over program expressions based on their features. Two general principles that are useful across various domains are: prefer small expressions (inspired by the classic notion of Kolmogorov complexity) and prefer expressions with fewer constants (to force generalization). For specific DSLs, more specific preferences or features can be defined based on the operators that occur in the DSL.

5.2. Ranking based on test inputs

The likelihood of a program being the intended one not only depends on the structure of that program, but also on features of the input data on which that program will be executed and the output data produced by executing that program. In some PBE settings, the synthesizer often has access to some additional test inputs on which the intended program is supposed to be executed. Singh showed how to leverage these additional test inputs to guess a reduction in the search space with the goal to speed up synthesis and rank programs better [35]. Ellis and Gulwani observed that the additional test inputs can be used to re-rank programs based on how similar are the outputs produced by those programs on the test inputs to the outputs in the training/example inputs provided by the user [7].

For instance, consider the task of extracting years from input strings of the kind shown in the table below.

Input	Output
Missing page numbers, 1993	1993
64-67, 1995	1995

The program $P1$: “Extract the last number” can perform the intended task. However, if the user provides only the first example, another reasonable program that can be synthesized is $P2$: “Extract the first number”. There is no clear way to rank $P1$ higher than $P2$ from just examining their structure. The above However, the output produced by $P1$ (on the various test inputs), namely $\{1993, 1995, \ldots, \}$ is a more meaningful set (of 4 digit numbers that are likely years) than the one produced by $P2$, namely (which manifests

greater variability). The meaningfulness or similarity of the generated output can be captured via various features such as IsYear, numeric deviation, IsPersonName, and number of characters.

5.3. ML-based Ranking Function

Typically, *natural* or intended programs tend to have subtle properties that cannot be captured by just one feature or by an arbitrary combination of the multiple features identified above; empirical results presented in Table 2 confirms this hypothesis where the accuracy of the shortest program based ranker or a random ranker is poor. Hence, we need to learn a ranking function that appropriately combines the features in order to produce the intended natural programs. In fact, learning rankers over programs/sub-expressions represents an exciting domain where insights from ML and PL can have an interesting and impactful interplay.

Below, we present one such case study where we learn a ranking function that ranks sub-expressions and programs during the search process itself [26]. We learn the ranking function using training data that is extracted from diverse real-world customer scenarios. However learning such a ranking function that can be used to rank sub-expressions during the search process itself poses certain unique challenges. For example, we need to rank various non-homogeneous sub-expressions during each step of the search process but the feedback about our ranking decisions is provided only after synthesis of the final program. Moreover, the ranking function captures the intended program only if the final program is correct, hence, a series of "correct" ranking decisions over various sub-expressions might be nullified by one incorrect ranking decision.

To solve the above set of problems, we implement a simple program embedding based approach. Consider a program P whose AST is given by $\mathscr{A}(P)$. Then the embedding of P is computed recursively where $\phi(\mathscr{A}(P)) = \sum_i w_i \phi(\mathscr{A}(P_i))$, P_i are the children of P in $\mathscr{A}(P)$. Now leaf nodes of $\mathscr{A}(P)$ are embedded in d-dimensions using a few operator-specific features. We now pose the ranking problem as: find $\theta \in \mathbb{R}^d$ s.t. $\sum_j \theta_j \phi(P_a)_j \geq \sum_j \theta_j \phi(P_b)_j$ where P_a is a "correct" program, i.e., it produces desired output on training datasets and P_b is an "incorrect" program. θ_j and $\phi(P)_j$ represents the jth coordinate of θ and $\phi(P)$ respectively.

Now recall that our goal is to ensure that all the ranking decisions in benchmark are correct, so we need to use a different metric than the standard classification metric (see [26] for more details).

For learning θ as well as weights w_i, we use training benchmarks where each benchmark consists of a set of inputs and their corresponding outputs. For each benchmark, we synthesize 1000 programs using the first input-output pair in that benchmark, treating it as an example input-output pair. We categorize a synthesized program as "correct" if it generates correct output on all the other benchmark inputs, and "incorrect" otherwise. We then embed each sub-expression and the program in d-dimensional space using hand-crafted features. Our features reflect certain key properties of the programs, e.g., length of the program etc. We then use straightforward block-coordinate descent based methods to learn θ, w_i's in an iterative fashion.

Empirical Results: similar to search experiments, we learn our ranking function using a collection of important benchmarks from real-world customer scenarios. We select about 100 benchmarks for training and test our system on the remaining 640 bench-

RANKING METHOD	ACC@1		ACC@10	
	$m=1$	$m=2$	$m=1$	$m=2$
RANDOM	0.22	0.60	0.38	0.67
(A) SHORTEST PROGRAM	0.37	0.69	0.49	0.80
(B) FEWER CONSTANTS	0.38	0.60	0.59	0.80
(A) and (B)	0.44	0.72	0.60	0.87
ML-based Ranker	0.65	0.81	**0.79**	**0.92**

Table 2. Ranking: table compares precision@1 and precision@10 accuracy for various methods when supplied different number of input-output example pairs ($m = 1, 2$). Our ML-ranker provides significantly higher accuracy and estimates correct program for 65% test benchmarks using just one input-output example.

marks. We evaluate performance of our ranker using precision k metric. That is, precision k is the fraction of test benchmarks in which at least one "correct" program lies in the top-k programs (as ranked by our ranker). We also compute precision k for different specification sizes, i.e., for different number of input-output examples being supplied.

Table 2 compares accuracy (measured in precision@k) of our method with four baselines: a) random ranker that at each node selects a random sub-expression, b) shortest program which selects programs with the smallest number of operators. c) program that selects the smallest number of constants. d) a linear combination of the shortest and smallest constants heuristics. Note that with 1 input-output example, our method is almost 50% more accurate than baselines. Naturally with 2 examples, baselines' performance also improves as there fewer programs that satisfy 2 examples.

Additionally, we can learn individual θ for each user/organization thus leading to personalized ranker. For example, our method can learn processing an input string as "European" style date-time instead of "American" style date-time.

6. Interactivity

While use of ranking in the synthesis methodology attempts to avoid selecting an unintended program, it cannot always succeed. Hence, it is important to design appropriate user interaction models for the PBE paradigm that can provide the equivalent of debugging experience in standard programming environments. There are two important goals for a user interaction model that is associated with a PBE technology [24]. First, it should provide transparency to the user about the synthesized program(s). Second, it should guide the user in resolving ambiguities in the provided specification.

In order to facilitate transparency, the synthesized program can be displayed to the user. In that context, it would be useful to have readability as an additional criterion during synthesis. The program can also be paraphrased in natural language, especially to facilitate understanding by non-programmers.

In order to resolve ambiguities, we can present multiple synthesized programs to the user and ask the user to pick between those. More interestingly, we can also leverage availability of other test input data on which the synthesized program is expected to be executed. This can be done in few different ways. A set of representative test inputs can be obtained by clustering the test inputs and picking a representative element from each cluster [28]. The user can then check the results of the synthesized program on those

representative inputs. Alternatively, clustering can also be performed on the outputs produced by the synthesized program. Yet, another approach can be to leverage *distinguishing inputs* [19]. The idea here is to synthesize multiple programs that are consistent with the examples provided by the user but differ on some test inputs. The PBE system can then ask the user to provide the intended output on one or more of these distinguishing inputs. The choice for the distinguishing input to be presented to the user can be based on its expected potential to distinguish between most of those synthesized programs.

There are many heuristic decisions in the above-mentioned interaction models that can ideally be learned using ML techniques such as what makes a program more readable, or which set of programs to present to the user, or how to cluster the input or output column. Below, we discuss one such investigation related to clustering of strings.

6.1. Clustering of Strings

We propose an agglomerative hierarchical clustering based method for clustering the strings. Intuitively, we want to cluster strings together which can be represented by a specific but natural regular expression. For example, given strings $\{1990, 1995, 210\text{BC}, 450\text{BC}\}$, we want to find the two clusters represented by regular expressions Digit^4 and $\text{Digit}^3 \cdot \text{BC}$.

We find the tightest and natural regular expression representing a given set of strings using program synthesis over a regular expression specific language. [28]. Our algorithm randomly samples a few strings and then finds the most likely regular expressions by synthesizing them using pairs of strings. The most highly rates regular expressions can be thought of as cluster representatives. We then define a distance function that computes distance of a string to a regular expression. Using this distance function, we then apply standard agglomerative hierarchical clustering algorithm to obtain representative regular expressions.

For example, given strings from a dataset containing postal codes such as: {99518, 61021-9150, 2645, K0K 2C0, 61604-5004...}, our system finds clusters such as:

- Digit^5
- Digit^4
- $\text{UpperCase} \cdot \text{Digit} \cdot \text{UpperCase Digit} \cdot \text{UpperCase} \cdot \text{Digit}$
- $61\text{Digit}^3 - \text{Digit}^4$
- S7K7K9

Note that the regular expressions are able to capture the key clusters such as Digit^5 etc, but it also captures certain anomalies such as S7K7K9. We also evaluate our system over real-world datasets using Normalized Mutual Information (NMI) metric which is a standard clustering metric. We observe that if given enough computation time, our system is able to obtain nearly optimal NMI of ≈ 1.0. Moreover, by appropriately sampling and synthesizing regular expressions, we can speed up the computation by a factor of 2 despite recovering clusters with NMI of 0.95. We refer the interested readers to [28] for more details.

7. Future Directions

Applications Robotic Process Automation (RPA) can be another killer application for program synthesis. The goal in RPA is to automate high-volume rules-driven business

processes that often connect different applications. These typically require logging into IT systems and copying and pasting data across systems. For instance, consider the task of opening an invoice in a PDF format that is received as an attachment in an email, extracting various fields from the invoice, and entering them inside multiple systems. Program synthesis technologies can help synthesize such scripts from few demonstrations by the business user.

Another interesting application of program synthesis can be in the space of programming real-world robots. General-purpose programmable robots may be a common household entity in a few decades from now. Each household will have its own unique geography for the robot to navigate and a unique set of chores for the robot to perform. Example-based training could be an effective means for programming robots for personalized tasks and personalized household environments.

Performant Synthesis The synthesized scripts might need to be executed on big data. In such a scenario, it is desirable to synthesize not just a correct program that meets the intent, but one that is also efficient and hence does not waste computational resources.

Often there are many different programs to accomplish a particular task. These programs may not be semantically equivalent but they have the same behavior on the kinds of inputs they are expected to be executed on. The ranking schemes in program synthesis are generally tuned to pick any of these *correct* programs. However, some of these programs may be much more efficient than the other, and it may be desirable to pick one such efficient program. For instance, suppose the goal is to extract LastName from inputs of kind “FirstName LastName”. One correct program to accomplish such a task can operate by extracting the second word, while another correct program to accomplish the same task can operate by extracting all characters after the last space. It turns out that the latter program is much more efficient than the former since it avoids use of regular expressions.

Readable Synthesis The synthesized scripts might need to be readable and modifiable. In some scenarios, it is not important to inspect the code of the synthesized program, especially when the goal is to execute the script for a one-off task and wherein the correctness of the underlying transformation can be verified by visual inspection over small input data. Applying Flash Fill [10] on small-sized input columns is an example of such a scenario, wherein an end user may simply inspect the derived column to verify that the string transformation has been performed correctly. However, if the input on which the synthesized script is to be executed is large, or if the synthesized script needs to be executed multiple times in the future, then the user may want to inspect, and possibly even edit the code of the underlying synthesized program. In such scenarios, it is important to synthesize code that is readable. Furthermore, such a code may need to be synthesized in a specific target language desired by the user. This leads to many interesting research challenges such as leveraging idiomatic patterns and libraries that the user is familiar with, choice of variable names, and formatting of code. Another interesting concern relates to maintainability of such synthesized code. For instance, if the user provides additional examples in the future to adapt the behavior of the code on new additional inputs, then what happens to any changes that the user may have made in the old synthesized code? One interesting possibility is to ensure that the newly synthesized code is as similar to the old synthesized code as possible, which can be regarded as automation of test-driven development [29].

Synthesizing readable code in specific target languages shall allow PBE technologies to be integrated inside main-stream coding workflows such as IDEs or notebooks.

Multi-model intent specification While this article has focused on leveraging examples as specification of intent, certain class of tasks are best described using natural language such as spreadsheet queries [14] and smartphone scripts [23]. The next generation of programming experience shall be built around multi-modal specifications that are natural and easy for the user to provide. The new paradigm shall allow expressing intent using combination of various means [32] such as examples, demonstrations, natural language, keywords, and sketches [42].

Predictive Synthesis For some task domains, it is often possible to predict the user's intent without any input-output examples, i.e., from input-only examples. For instance, extracting tables from web pages, PDF documents, or log files, or splitting a column into multiple columns [30]. While providing examples is already much more convenient than authoring one-off scripts, there are scenarios where providing examples can be quite tedious overall. For instance, consider the task of extracting fields from a log file. If the number of fields is large, then providing examples for each field would be quite tedious. Having the system guess the user's intent without any examples can also power novel user experiences such as enabling question-answering on semi-structured data, wherein the system can automatically infer the underlying relational tabular structure without requiring the user to provide any examples.

Adaptive Synthesis Another interesting future direction is to build systems that learn user preferences based on past user interactions across different programming sessions. For instance, the underlying ranking can be dynamically updated. This can pave the way for personalization of PBE technologies to specific users, as well as enable learning across users in a given organization or cloud. Tasks that required more examples earlier can now be accomplished with fewer examples. In fact, this can also facilitate predictive synthesis. For instance, consider the task of parsing a custom log file or extracting a table from a web page. Initially, a user may have to provide some examples. In the future, when the same user or even a different user, is faced with the same task but on a different input of the same format, the underlying adaptive synthesis system should be able to handle the task predictively, i.e., without any examples.

PL meets ML While PL has democratized access to machine implementations of precise ideas, ML has democratized access to discovering heuristics to deal with fuzzy and noisy situations. The new AI revolution requires frameworks that can facilitate creation of AI-infused software and applications. Synergies between PL and ML can help lay the foundation for construction of such frameworks [34,5,8,41].

For instance, language features can be developed that allow the developer to express non-determinism with some default resolution strategies that can then automatically get smarter with usage. As opposed to traditional AI based domains such as vision, text, bioinformation, such self-improving systems present entirely different data formats and pose unique challenges that foreshadow an interesting full-fledged research area with opportunity to impact how we program and think about interacting with computer systems in general.

8. Conclusion

PBE is a new frontier in AI and is set to revolutionize the programming experience. The technology has already matured to the extent that it can provide 10-100x productivity increase in many task domains for both data scientists and developers. The two killer applications for PBE today are: data wrangling and code refactoring. Data scientists spend 80% time wrangling data while developers spend up to 40% time refactoring code in a typical application migration scenario. Another significant aspect of PBE is its potential to enable programming for the masses, given that 99% people who use computers do not know programming.

We have leveraged inspiration from both logical reasoning and machine learning to build usable and practical PBE systems. The Microsoft PROSE SDK [1] exposes generic search and ranking algorithms, allowing advanced developers to construct PBE capabilities for new task domains. This SDK has been used to build product-quality implementations of many PBE capabilities that have shipped through multiple Microsoft products across Office, Windows, SQL, and Azure.

A key challenge in PBE is to search for programs that are consistent with the examples provided by the user. On the symbolic reasoning side, our search methodology in PBE leverages two key ideas: restrict the search to a domain-specific programming language (PL) specified as a grammar, and perform a goal-directed top-down search that leverages inverse semantics of operators to decompose a goal into a choice of multiple sub-goals. However, this search can be made even more tractable by learning tactics (using ML) to prefer certain choices over others during both grammar exploration and sub-goal selection.

Another key challenge in PBE is to understand the user's intent in the face of ambiguity that is inherent in example-based specifications, and furthermore, to understand it from as few examples as possible. For this, we leverage use of a ranking function with the goal of the search now being to pick the highest ranked program that is consistent with the examples provided by the user. The ranking is a function of various symbolic features of a program such as size, number of constants, use of a certain combination of operators. The ranking is also a function of the outputs generated by the program (non-null or not, same type as the example outputs or not) and more generally the execution traces of the program on new test inputs. While various PL concepts go into defining the features of a ranking function, ML-based techniques can be used to build models over these different classes of features.

A third challenge relates to debuggability: provide transparency to the user about the synthesized program and help the user to refine the specification in an interactive loop. We have investigated user interaction models that leverage concepts from both PL and ML including active learning based on synthesis of multiple top-ranked programs (each of which is consistent with the user's specification) and leveraging their differences, clustering of inputs to identify various input classes and hence representative inputs, clustering of outputs to identify any potential discrepancies, and navigation through a large program set represented succinctly as a grammar.

The above-mentioned directions highlight opportunities to design novel techniques that combine logical reasoning based symbolic methods developed in the PL community

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

with ML methods to solve various challenges that arise in construction of efficient, robust, and usable PBE systems. We believe that the ongoing AI revolution shall further drive novel synergies between PL and ML to facilitate creation of intelligent software in general. PBE systems, and more generally program synthesis systems, that relate to real-time intent understanding are a great case study for investigating ideas in this space.

Programming has evolved from use of punched cards and low-level assembly language programming to programming with high-level languages in beautiful code editors. The next evolution will leverage advances in program synthesis techniques to take programming closer to natural human communication, wherein it will become multi-modal and will involve use of various forms of intent expression including examples and natural language. Today, examples are already present in programming in the form of test cases, and comments are nothing but natural-language-based specifications. However, these artifacts, namely test cases and comments, are today constructed after code has been written in order to test code or to document code. The next frontier will lift these artifacts to first-class citizens for the process of authoring code itself.

References

[1] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

[2] R. Alur, A. Radhakrishna, and A. Udupa. Scaling enumerative program synthesis via divide and conquer. In *TACAS*, pages 319–336, 2017.

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

[4] D. W. Barowy, S. Gulwani, T. Hart, and B. G. Zorn. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 218–228, 2015.

[5] P. Bielik, V. Raychev, and M. T. Vechev. Programming with "big code": Lessons, techniques and applications. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, pages 41–50, 2015.

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

[7] K. Ellis and S. Gulwani. Learning to learn programs from examples: Going beyond program structure. In *IJCAI*, pages 1638–1645, 2017.

[8] J. K. Feser, M. Brockschmidt, A. L. Gaunt, and D. Tarlow. Neural functional programming. *CoRR*, abs/1611.01988, 2016.

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

[10] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

[11] S. Gulwani. Programming by examples - and its applications in data wrangling. In *Dependable Software Systems Engineering*, pages 137–158. 2016.

[12] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[13] S. Gulwani and P. Jain. Programming by examples: PL meets ML. In *Programming Languages and Systems - 15th Asian Symposium APLAS, Suzhou, China*, volume 10695 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017.

[14] S. Gulwani and M. Marron. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD*, pages 803–814, 2014.

[15] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[16] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328, 2011.

[17] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[18] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, pages 36–46, 2010.

[19] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224, 2010.

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

[21] A. Kalyan, A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *International Conference on Learning Representations*, 2018.

[22] V. Le and S. Gulwani. FlashExtract: a framework for data extraction by examples. In *PLDI*, pages 542–553, 2014.

[23] V. Le, S. Gulwani, and Z. Su. Smartsynth: synthesizing smartphone automation scripts from natural language. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 193–206, 2013.

[24] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. G. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. In *UIST*, pages 291–301, 2015.

[25] A. K. Menon, O. Tamuz, S. Gulwani, B. W. Lampson, and A. Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 187–195, 2013.

[26] N. Natarajan, N. Datha, D. Simmons, S. Gulwani, and P. Jain. Learning natural programs from a few examples in real-time. In *AIStats*, 2019.

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

[28] S. Padhi, P. Jain, D. Perelman, O. Polozov, S. Gulwani, and T. D. Millstein. Flashprofile: a framework for synthesizing data profiles. *PACMPL*, 2(OOPSLA):150:1–150:28, 2018.

[29] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *PLDI*, pages 408–418, 2014.

[30] M. Raza and S. Gulwani. Automated data extraction using predictive program synthesis. In *AAAI*, pages 882–890, 2017.

[31] M. Raza, S. Gulwani, and N. Milic-Frayling. Programming by example using least general generalizations. In *AAAI*, pages 283–290, 2014.

[32] M. Raza, S. Gulwani, and N. Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI*, pages 792–800, 2015.

[33] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 404–415, 2017.

[34] C. Simpkins. Integrating reinforcement learning into a programming language. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[35] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016.

[36] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5(8):740–751, 2012.

[37] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, pages 634–651, 2012.

[38] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 398–414, 2015.

[39] R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 343–356, 2016.

[40] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.

[41] R. Singh and P. Kohli. AP: artificial programming. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, pages 16:1–16:12, 2017.

[42] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

[43] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.