Programming by Examples: PL meets ML

Sumit Gulwani[1] and Prateek Jain[2]

[1] Microsoft Corporation, Redmond, USA
sumitg@microsoft.com
[2] Microsoft Research, Bangalore, India
prajain@microsoft.com

Abstract. Programming by Examples (PBE) involves synthesizing intended programs in an underlying domain-specific language from example-based specifications. PBE systems are already revolutionizing the application domain of data wrangling and are set to significantly impact several other domains including code refactoring.

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. We leverage features of the program structure as well of the outputs generated by the program on test inputs. (iii) User interaction models to facilitate usability and debuggability. We leverage active-learning techniques based on clustering inputs and synthesizing multiple programs.

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. This can not only lead to better heuristics, but can also enable easier development, maintenance, and even personalization of a PBE system.

1 Introduction

Program Synthesis is the task of synthesizing a program that satisfies a given specification [1]. The traditional view of program synthesis has been to synthesize programs from logical specifications that relate the inputs and outputs of the program. 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 [2]. PBE has emerged as a favorable paradigm for two reasons: (i) the example-based specification in PBE makes it more tractable than general program synthesis. (ii) Example-based specifications are much easier for the users to provide in many scenarios.

2 Applications

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

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 a delightful experience for the masses.

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 [3]. 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 FlashExtract technology also ships in Azure OMS (Operations Management Suite), where it enables extraction of custom fields from log files.

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" [4]. PBE can also facilitate more sophisticated string transformations that require lookup into other tables [5]. PBE is also a very natural fit for automating transformations of other data types such as numbers [6] and dates [7].

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 [8], or for re-pivoting the underlying hierarchical data that has been locked into a two-dimensional tabular format [9]. 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 [10].

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 [11].

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 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 [11].

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.

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

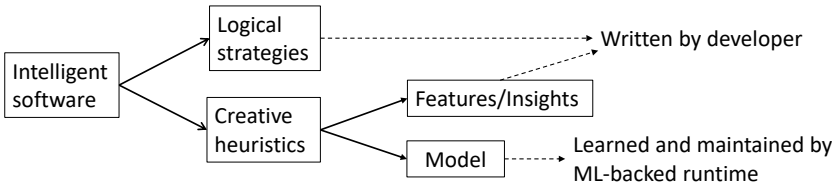

Fig. 1. A proposal for development of intelligent software that facilitates increased developer productivity and increased software intelligence.

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 [12], stack augmented neural controller [13], or even neural networks augmented with basic arithmetic and logic operations [14]. 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 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-software 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 developer's productivity, (ii) Increase in system's 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 1 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 since 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 fluid. Depending on the exact problem setting as well as the domain, the developer may want to decide which part of the system should follow deterministic logical reasoning and which part should be based on data-driven techniques.

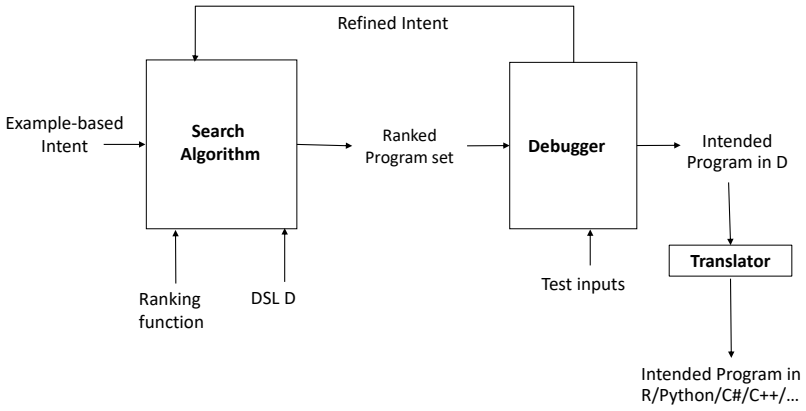

Fig. 2. Programming-by-Examples Architecture. The search algorithm, parameterized by a domain-specific language (DSL) and a ranking function, synthesizes a ranked set of programs from the underlying DSL that are consistent with the examples provided by the user. The debugging component, which leverages 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.

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. It can be used to learn a better ranking function, allowing users to provide fewer examples and thus increasing usability. 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.

4 Search Algorithm

Figure 2 shows the architecture of a PBE system. The most involved technical component is the search algorithm, which we discuss in this section. Sections 4.1 and 4.2 describe the two key PL ingredients that form the foundation for designing this search algorithm that is based on logical/symbolic reasoning. Section 4.3 then discusses and speculates how ML can further build over the traditional PL-style logical reasoning to obtain an even more efficient, real-time search algorithm for PBE.

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

Fig. 3. An example domain-specific language. substr, concat are string operators and conststr represents a constant string constructor. 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) [15, 16]. 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) [2]. 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 a non-terminal, an explicit set of program expressions, or a program operator applied to some non-terminals or expressions.

For illustration, we present an extremely simple string manipulation grammar in Figure 3; this DSL is a heavily stripped down version of the Flash Fill DSL [4]. 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 k^{th} 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, the following program maps input "evan chang" into "evanchang@cs.colorado.edu".
concat(substr(Input, ϵ, " ", 1), substr(Input, " ", ϵ, -1), conststr("@cs.colorado.edu"))
Note that we overload 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 by doing a bottom-up enumeration of the grammar [15]. 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 method 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 [17]. (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 [18].

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. In the next subsection, we provide more details on one such investigation related to predicting the choice over exploring multiple production rules for a grammar non-terminal. In particular, we describe our ML-based problem formulation, our training data collection process as well as some preliminary results.

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 investigation is 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 3, 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. Formally, 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 [19], our method embeds input-output examples in a vector space using a popular neural network technique called LSTM (Long Short-Term Memory) [20]. 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 4 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 [4] that includes operators over rich 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.

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

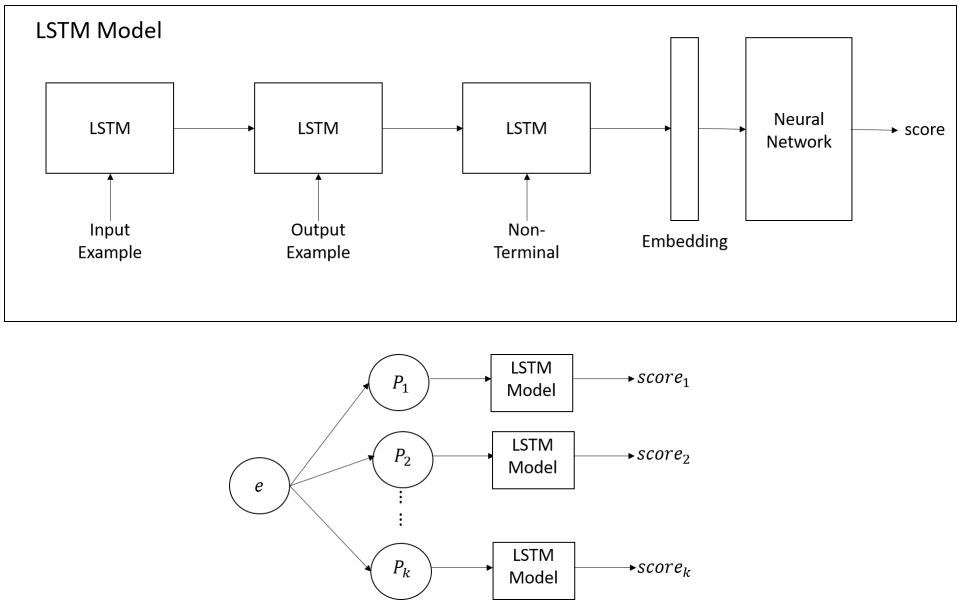

Fig. 4. LSTM-based model for computing score for the candidate set of production rules $P_1, \ldots, P_k$ during the grammar expansion process. The top figure shows details of the ML model used to compute the score for a candidate production rule when placed in the context of the given input-output examples.

symbol (say expr in Figure 3) 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 3). 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, i.e., only 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 correct production rule in 92% of the instances. Unfortunately, selecting a 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 τ to 0.1, we can increase chances of selection of a correct 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 8x over naïve exploration for many challenging test benchmarks.

5 Ranking

Examples are a severe under-specification of the user's intent in many useful task domains. As a result, there are often many programs in an underlying DSL that are consistent with a given set of training examples, but are unintended, i.e., they would produce an undesired output on some other test input. Usability concerns further necessitate that we are able to 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. We capture the notion of *naturalness* of programs by performing training on real-world 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 [21], or it can be in-built as part of the search process [22, 23]. 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 [24]. 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 [25].

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. 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 Figure 5 confirm 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. 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 may 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 that is composed of expressions $\{e_1, \ldots, e_m\}$. Now, using the previously mentioned classes of features, we embed each of our sub-expression in a d-dimensional vector space. Formally, $\phi(e_i) \in \mathbb{R}^d$ is a d-dimensional embedding of sub-expression e_i and the program P itself is represented as a weighted combination of these sub-expressions: $\phi(P) = \sum_i w_i \phi(e_i)$,

No. I/O	Random	Shortest	ML-Ranker
1	14.2%	2.4%	69.2%
2	53.7%	77.5%	91.4%
3	77.0%	88.0%	96.5%
4	86.6%	94.4%	98.6%

No. I/O	Random	Shortest	ML-Ranker
1	19.1%	9.3%	70.6%
2	57.1%	74.0%	88.1%
3	77.3%	86.8%	94.8%
4	86.8%	92.8%	97.8%

Fig. 5. Ranking: Left table compares precision@1 accuracy for various methods when supplied different number of input-output example pairs while right table compares precision@5 accuracy. Our ML-ranker provides significantly higher accuracy and estimates correct program for 69% test benchmarks using just one input-output example.

where $w_i \geq 0$ are positive weights that are also learned depending on the type of sub-expression e_i. That is, $\phi(P)$ is a d-dimensional vector representation of program P.

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 j-th coordinate of θ and $\phi(P)$ respectively.

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 described in Section 4.3, we learn our ranking function using a collection of important benchmarks from real-world customer scenarios. We select about 75 benchmarks for training and test our system on remaining 300 benchmarks. We evaluate performance of our ranker using precision@k metric, which 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.

The tables in Figure 5 compare accuracy (measured in precision@k) of our method with two baselines: a) random ranker, which at each node selects a random sub-expression, b) shortest program, which selects programs with the smallest number of operators. Note that with 1 input-output example, our method is 50% more accurate than the two baselines. Naturally with 4 examples, baselines' performance also improves as there are very few programs that satisfy 4 examples, i.e., with 4 input-output examples searching for "any" consistent program is enough. However, as mentioned earlier, in many settings providing even 4 examples is going to burden the user significantly and hence renders the solution

impractical. Moreover, with bigger and more powerful grammars, even 4 examples might lead to several thousands of consistent programs, thus necessitating a data driven ranker of the sub-expressions/programs.

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 [26]. 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 [27]. 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* [28]. 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 columns. Below, we discuss one such investigation related to clustering of strings in a column.

6.1 Clustering of Strings

We propose an agglomerative-hierarchical-clustering-based method for clustering a collection of strings. Intuitively, we want to cluster strings such that each cluster can be represented by a specific but natural description. For example,

given strings $\{1990, 1995, 210\text{BC}, 450\text{BC}\}$, we want to find the two clusters that can be described by the regular expressions Digit^4 and $\text{Digit}^3 \cdot \text{BC}$.

To partition a given set of strings into such natural clusters, we learn regular expressions as cluster descriptions, using program synthesis over a DSL that describes regular expressions [27]. Our clustering algorithm first randomly samples a few strings and then generates candidate regular expressions by synthesizing regular expressions that describe various pairs of strings. We define a measure to compute the cost of describing a string using a candidate regular expression. Using this cost measure, we apply standard complete-linkage agglomerative hierarchical clustering to obtain compact clusters with low cost of describing the contained strings with a regular expression.

For example, given strings from a dataset containing postal codes such as: {99518, 61021-9150, 2645, K0K 2C0, 61604-5004, S7K7K9,...}, our system identifies clusters described by regular expressions such as:

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

Note that these regular expressions not only capture the key clusters such as Digit^5 etc, but they also expose certain anomalies such as S7K7K9. We evaluated our system over real-world datasets, and used a Normalized Mutual Information (NMI) metric, which is a standard clustering metric, to measure the accuracy of our system in its ability to identify the expected clusters.

We observe that, given enough computation time, our system is able to achieve nearly optimal NMI of ≈ 1.0. Moreover, by using appropriate sampling and synthesizing regular expressions, we can speed up the computation by a factor of 2 despite recovering clusters with over 0.95 NMI. We refer the interested readers to [27] for more details.

7 Future Directions

Applications: 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 would be an effective means for personalizing robots for a household.

Multi-model intent specification: While this article has focused on leveraging examples as specification of intent, certain classes of tasks are best described using natural language such as spreadsheet queries [29] and smartphone scripts [30]. 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 the user to express intent using combination of various means [31] such as examples, demonstrations, natural language, keywords, and sketches [32].

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 or log files, or splitting a column into multiple columns [18]. While providing examples is already much more convenient than authoring one-off scripts, having the system guess the intent without any examples can power novel user experiences.

Adaptive Synthesis: Another interesting future direction is to build systems that learn user preferences based on past user interactions across different program synthesis sessions. For instance, the underlying ranking can be dynamically updated. This can pave the way for personalization and learning across users within an organization or within the cloud.

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 [33–36]. For instance, language features can be developed that allow the developer to express non-determinism with some default resolution strategies that can then automatically become 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 opportunities to impact how we program and think about interacting with computer systems in general.

8 Conclusion

PBE is set to revolutionize the programming experience for both developers and end users. It can provide a 10-100x productivity increase for developers in some task domains, and also enable computer users, 99% of whom are non-programmers, to create small scripts to automate repetitive tasks. In fact, several studies show that data scientists spend 80% time wrangling data while developers spend up to 40% time refactoring code in a typical application migration scenario. Hence, data wrangling and code refactoring seem to be two killer applications for PBE today where PBE-based systems stand to significantly improve productivity of data scientists as well as developers.

Building a usable and practical PBE system is challenging and can leverage insights from both PL (for symbolic reasoning) and ML (for heuristics). 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 language 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 to prefer certain choices over others during both grammar enumeration 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 ranker is a function of various symbolic features of a program such as size, number of constants, and use of a certain combination of operators. It 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 combine these features so that performance of the ranker is good over real-world customer scenarios.

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, clustering of inputs and outputs to identify discrepancies, and navigation through a large program set represented succinctly as a grammar.

All the above-mentioned directions highlight opportunities to design novel techniques that carefully combine symbolic reasoning and declarative insights with novel ML models to solve the various technical challenges associated with a PBE system. 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.

Acknowledgments

We thank Nagarajan Natarajan, Naren Datha, Danny Simmons, Abhishek Mohta, Alex Polozov, and Daniel Perelman for their participation in the ongoing unpublished work that we have described in this article related to using ML techniques for search and ranking. We thank the entire PROSE team [3] that has been developing and productizing the PBE technology inside many Microsoft products while also inspiring relevant research problem definitions. We thank Sriram Rajamani, Rishabh Singh, and Joseph Sirosh for stimulating discussions related to the topics discussed in this article.

[3] https://microsoft.github.io/prose/team/

References

1. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Foundations and Trends in Programming Languages **4**(1-2) (2017) 1–119
2. Gulwani, S.: Programming by examples - and its applications in data wrangling. In: Dependable Software Systems Engineering. (2016) 137–158
3. Le, V., Gulwani, S.: FlashExtract: a framework for data extraction by examples. In: PLDI. (2014) 542–553
4. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: POPL. (2011) 317–330
5. Singh, R., Gulwani, S.: Learning semantic string transformations from examples. PVLDB **5**(8) (2012) 740–751
6. Singh, R., Gulwani, S.: Synthesizing number transformations from input-output examples. In: CAV. (2012) 634–651
7. Singh, R., Gulwani, S.: 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. (2016) 343–356
8. Barowy, D.W., Gulwani, S., Hart, T., Zorn, B.G.: 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. (2015) 218–228
9. Harris, W.R., Gulwani, S.: Spreadsheet table transformations from examples. In: PLDI. (2011) 317–328
10. Raza, M., Gulwani, S., Milic-Frayling, N.: Programming by example using least general generalizations. In: AAAI. (2014) 283–290
11. Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., Hartmann, B.: 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. (2017) 404–415
12. 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., Hassabis, D.: Hybrid computing using a neural network with dynamic external memory. Nature **538**(7626) (2016) 471–476
13. Joulin, A., Mikolov, T.: Inferring algorithmic patterns with stack-augmented recurrent nets. In: NIPS. (2015) 190–198
14. Neelakantan, A., Le, Q.V., Sutskever, I.: Neural programmer: Inducing latent programs with gradient descent. CoRR **abs/1511.04834** (2015)
15. Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Dependable Software Systems Engineering. (2015) 1–25
16. Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. Commun. ACM **55**(8) (2012) 97–105
17. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: TACAS. (2017) 319–336
18. Raza, M., Gulwani, S.: Automated data extraction using predictive program synthesis. In: AAAI. (2017) 882–890

19. Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A., Kohli, P.: Robustfill: Neural program learning under noisy I/O. In: ICML. (2017)
20. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8) (November 1997) 1735–1780
21. Singh, R., Gulwani, S.: 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. (2015) 398–414
22. Menon, A.K., Tamuz, O., Gulwani, S., Lampson, B.W., Kalai, A.: 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. (2013) 187–195
23. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: Deepcoder: Learning to write programs. In: ICLR. (2017)
24. Singh, R.: Blinkfill: Semi-supervised programming by example for syntactic string transformations. PVLDB **9**(10) (2016) 816–827
25. Ellis, K., Gulwani, S.: Learning to learn programs from examples: Going beyond program structure. In: IJCAI. (2017) 1638–1645
26. Mayer, M., Soares, G., Grechkin, M., Le, V., Marron, M., Polozov, O., Singh, R., Zorn, B.G., Gulwani, S.: User interaction models for disambiguation in programming by example. In: UIST. (2015) 291–301
27. Padhi, S., Jain, P., Perelman, D., Polozov, O., Gulwani, S., Millstein, T.: Flashprofile: Interactive synthesis of syntactic profiles. arXiv preprint arXiv:1709.05725 (2017)
28. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE. (2010) 215–224
29. Gulwani, S., Marron, M.: Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In: SIGMOD. (2014) 803–814
30. Le, V., Gulwani, S., Su, Z.: Smartsynth: synthesizing smartphone automation scripts from natural language. In: The 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys). (2013) 193–206
31. Raza, M., Gulwani, S., Milic-Frayling, N.: Compositional program synthesis from natural language and examples. In: IJCAI. (2015) 792–800
32. Solar-Lezama, A.: Program Synthesis by Sketching. PhD thesis, University of California, Berkeley (2008)
33. Simpkins, C.: 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)
34. Bielik, P., Raychev, V., Vechev, M.T.: 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. (2015) 41–50
35. Feser, J.K., Brockschmidt, M., Gaunt, A.L., Tarlow, D.: Neural functional programming. CoRR **abs/1611.01988** (2016)
36. Singh, R., Kohli, P.: AP: artificial programming. In: 2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA. (2017) 16:1–16:12