IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 7, 2021, accepted April 8, 2021, date of publication May 11, 2021, date of current version May 19, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3079351

Neural-Guided Inductive Synthesis of Functional Programs on List Manipulation by Offline Supervised Learning

YUHONG WANG AND XIN LI
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai 200062, China
Corresponding author: Xin Li (xinli@sei.ecnu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61802126, in part by the Ministry of Science and Technology of China under Grant 2018YFC0830400, and in part by the Shanghai Pujiang Program under Grant 17PJ1402200.

ABSTRACT Synthesizing intended programs from user-specified input-output examples, also known as Programming by Examples (PBE), is a challenging problem in program synthesis, and has been applied to a wide range of domains. A key challenge in PBE is to efficiently discover a user-intended program in the search space that can be exponentially large. In this work, we propose a method for automatic synthesis of functional programs on list manipulation, by using offline-trained Recurrent Neural Network (RNN) models to guide the program search. We adopt miniKanren, an embedded domain-specific language for flexible relational programming, as an underlying top-down deductive search engine of candidate programs that are consistent with input-output examples. Our approach targets an easy and effective integration of deep learning techniques in making better PBE systems and combines two technical ideas on generating diverse training dataset and designing rich feature embeddings of probable subproblems for synthesis generated by deductive search. The offline-learned model is then used in PBE to guide the top-down deductive search with specific strategies. To practically manipulate data structures of lists, our method synthesizes functional programs with popular higher-order combinators including `map`, `foldl` and `foldr`. We have implemented our method and evaluated it with challenging program synthesis tasks on list manipulation. The experiments show promising results on the performance of our method compared to related state-of-the-art inductive synthesizers.

INDEX TERMS Programming by examples, functional programs, miniKanren, relational programming, deep learning, deductive search.

I. INTRODUCTION

Program synthesis aims to automatically synthesize a program that satisfies high-level specifications. Programming by Examples (PBE), also known as Inductive Synthesis, is to generate an intended program in a domain-specific language (DSL) from a given set of input-output examples. The research on PBE is appealing since example-based specifications are readily available to even non-expert users, and other synthesis problems from logic specifications can be transformed to PBE [1]. It has been applied to a wide range of application domains including end-user programming [2], data wrangling [3], program repair [4]–[6], etc. FlashFill, as a feature of Microsoft Excel 2013 and later, is a well-known example of end-user programming on string manipulation, and it can automatically fill your data in spreadsheet tables.

The last few years has seen a surge of interests in research on PBE. A series of novel techniques have been proposed to accelerate the search methodology to deal with the issue of scalability [7]–[10]. Experience shows that the search algorithms in PBE can be accelerated by leveraging domain-specific heuristics, such as structural probability defined over program patterns [7], [8], optimization techniques driven by common features of input-output examples [9], [10], etc. Although much progress has been made, there remain a number of challenges in practical applications of PBE [11]. The scalability of PBE is still a key challenge among many others, to meet the performance demands of

The associate editor coordinating the review of this manuscript and approving it for publication was Bo Pu.

synthesis queries on large-scale programs or interactive programs in real time. Another key challenge concerns how to generate intended programs that correctly generalizes to unseen inputs, provided with a very limited set of possibly ambiguous input-output examples.

The development of deep learning techniques has given new ideas to address the aforementioned challenges. Deep learning techniques are good at learning high-dimensional complex features from data samples, and are promising to make PBE even more tractable and accurate. The emerging approach to neural-guided synthesis leverages neural network models, which help predict the most probable subproblems for synthesis reduced by the deductive search [12]–[14]. In a deductive search, the original synthesis task is recursively decomposed as smaller subproblems for synthesizing small sub-expressions of the intended program. Although PBE often assumes a small number of input-output examples, samples used in training the statistic model is the large set of subproblems for synthesis produced in the deductive search. Despite of benefiting from both correctness of symbolic search and tactics of deep learning, the approach is essentially data-driven and requires a large number of training dataset for the sake of predictive accuracy. As statistical models, they also cannot guarantee that the synthesized programs are indeed user-desired, especially given very few user-specified examples.

In this work, we present a neural-guided method for automatic synthesis of functional programs on list manipulation, where Recurrent Neural Network (RNN) models are trained offline and then used to guide the program search in online evaluation. Functional programming has attracted much attention in recent years due to its safety, modularity, simplicity and other features. It becomes a trend to support lambda expressions in contemporary version of programming languages such as Java8 and C++11 onwards. In addition, functional languages are increasingly used in the development of blockchain platforms [15]. We adopt miniKanren [16], an embedded domain-specific language for flexible relational programming, as an underlying top-down deductive search engine, which helps find candidate programs that are consistent with example-based specifications. The subproblems produced in the deductive search contain partially-unfolded programs and recursive relational constraints to be satisfied.

Our approach focuses on an easy and effective integration of deep learning techniques in making better PBE systems for functional programming. Our work is inspired by the offline training approach for string manipulation explored in [13], and built upon the neural-guided synthesis system for functional programs in [12]. Our work mainly extends [12] in the following three ways.

First, we present an offline training counterpart for the training method based on reinforcement learning in [12]. In [12], the authors view the training phase as reinforcement learning, where the neural network models are regarded as an environment and path selections during the deductive search is treated as actions, which are taken by the search engine. A variety of reinforcement learning techniques are used for training, including curriculum learning, scheduled sampling [17], etc. In contrast, offline training approach used in [13] treats the deductive search process as *Markovian*. That is, subproblems produced during the search process are independent. They don't rely on the original synthesis problem from which they were deduced. Partly motivated by this observation, we study the effect of offline supervised learning on neural-guided inductive synthesis of functional programs. Besides, we experiment with simple policies to make a good quality dataset for offline training.

Next, we present several feature embedding schemes based on partial programs and relational constraints that are involved in the subproblems for synthesis. In [12], only relational constraints are used as the embedding features. By empirical study, we found that a simple adoption of offline training techniques would not lead to a satisfactory generalization accuracy. To further enhance the generalization performance of the neural-guided method, we study and experiment with various feature embeddings on samples of subproblems for synthesis

Last but not least, we target automatic synthesis of functional programs on list manipulation with three widely-used higher-order combinators including `map`, `foldl` and `foldr`, to practically manipulate recursive data structures of lists. This extension with combinators addresses the issue of poor reusability in [12] that can only synthesize non-recursive programs. As shown in Table 1 with examples on synthesis of `dropLast` taken from [12], the task is to synthesize a program that drops the last element of input lists. The column labelled "*IO Examples*" gives input-output lists of increasing length. The column labelled "*Synthesized Programs*" gives the synthesized non-recursive programs with increasing size, correspondingly. As mentioned in [12], the optimal steps of synthesis is linear in the size of synthesized program, and it would cost more time and space to synthesize from longer example lists.

TABLE 1. Examples on synthesis of `dropLast` in [12].

IO Examples	Synthesized Programs
`{(1 #f),(1)}`	`(lambda (cons (car (var ())) (quote ())))`
`{(1 #f a),(1 #f)}`	`(lambda (cons (car (var ())) (cons (car (cdr (var ()))) (quote ()))))`
`{(1 #f a b),(1 #f a)}`	`(lambda (cons (car (var ())) (cons (car (cdr (var ()))) (cons (car (cdr (cdr (var ())))) (quote ())))))`

Figure 1 shows the overall neural-guided synthesis framework of our method, where miniKanren [16] is the central top-down deductive search engine. The system consists in two phases of offline training and online testing or evaluation. The offline training phase includes offline generating samples of subproblems that are grouped as so-called *state trees*, and training the neural network models by supervised learning, and the testing phase uses miniKanren [16] for deductive

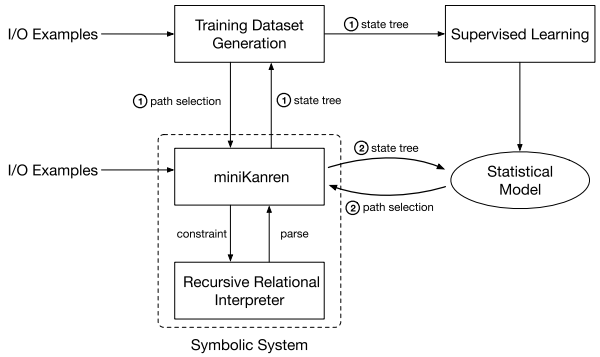


FIGURE 1. The overall framework of neural-guided synthesis by offline supervised learning. ① indicates the first phase of offline training and ② indicates the second phase of online testing. Unlabeled modules are used in both training process and evaluation process.

search of intended programs and the learned statistical model to effectively guide the symbolic search.

We have implemented our method and evaluate it with challenging program synthesis tasks on list manipulation. The experiments show promising results on the performance of our method compared to state-of-the-art inductive synthesizers.

- Consider synthesis of non-recursive functional programs. The offline training approach performs comparatively well as the training approach by reinforcement learning, and often leads us to shorter synthesis steps and better synthesis time. In addition, the two training approaches would bring us with varying samples of subproblems complementing each other on training dataset. For instance, we found to be able to tackle with some synthesis problems by offline training that are failed to be synthesized as reported in [12].
- For synthesis of functional programs with higher-order combinators, we compare our method to the state-of-the-art inductive synthesizers on recursive functional programs, including λ^2 [18], Myth [19] and Escher [20]. Experimental results show that, our method outperforms Escher and Myth given the same number of input-output examples and resources, and generally performs comparatively well as λ^2 (specifically, worse than it on two tasks and better than it on one task).

Besides, when restricted to synthesis of non-recursive functional programs, our method outperforms the aforementioned three inductive synthesizers on some tasks like `dropLast` and `bringToFront` with which all these methods perform poorly. It came to the similar observation by empirical study in [12]. Our method can be possibly used as a supplement to other methods for particularly synthesizing non-recursive sub-expressions.

Organization: The rest of the paper is organized as follows: Section II defines our target synthesis problem to be addressed. Section III presents the details of our neural-guided synthesis method, including offline generation of training datasets, supervised learning based on enriched feature embeddings, and design of interpreter for higher-order combinators. Section IV reports our implementation and experimental results on evaluation of our method on tasks of list manipulation. Section V discusses related work and Section VI concludes with remarks on future work.

II. BACKGROUND AND PROBLEM FORMULATION

In this section, we formally state our target problem of neural-guided program synthesis from examples based on the symbolic search engine miniKanren [16].

A. PROGRAMMING BY EXAMPLES

Our task is to synthesize an intended program P in the underlying domain-specific language (DSL) $\mathcal{L}$ from a given set of input-output examples $\mathcal{X} = \{(i_1, o_1), \ldots, (i_k, o_k)\}$. The example-based specifications are often further labelled with the user-intended program P_ι to be synthesized from $\mathcal{X}$, considering the underlying domain-specific problems to be addressed. Such a PBE problem can be denoted by $(\mathcal{X}, P_\iota)$. The synthesized program is supposed to satisfy the example-based specifications, i.e., $P(i) = o$ for each $(i, o) \in \mathcal{X}$, and also expected to generalize well to unseen inputs of the tasks and correctly generates intended outputs.

Correspondingly, the synthesis algorithms are measured from the two aspects of correctness and generalization ability. Further, given a PBE problem $(\mathcal{X}, P_\iota)$, we call the result program $P \in \mathcal{L}$ of synthesis an *exact match* (E-match for short) if P is the same as P_ι, and a *general match* (G-match for short) if P satisfies all the given examples but P is different from P_ι. As aforementioned, symbolic approaches usually ensure the correctness of synthesized programs by construction, whereas statistical approaches have to perform post-hoc filtering for the guarantee. Besides, when large-scale programs need to be synthesized or synthesis queries need to be answered in real time, it is also crucial to consider the time consumed by the synthesis algorithms.

Following [12], we synthesize functional programs in a small subset of Lisp as our target DSL that contains basic constants, primitive operators, etc. To synthesize recursive programs, we further extend the original DSL with a set of higher-order combinators including `map`, `foldl` and `foldr`. Let $n \in \{0, 1, \ldots, 9\}$ be the set of decimal numbers, and let a be the meta-variable that ranges over a finite set of symbols. The symbol " . " denotes a pair operator. The program syntax is given below:

$$
\begin{aligned}
d\ (\textit{datum}) &::= \texttt{()} \mid a \mid n \mid (d\ .\ d) \\
v\ (\textit{variable name}) &::= \texttt{()} \mid (s\ .\ v) \\
e\ (\textit{expression}) &::= (\texttt{var}\ v) \mid (\texttt{quote}\ e) \\
&\quad \mid (\texttt{lambda}\ e) \mid (\texttt{app}\ e\ e) \\
&\quad \mid (\texttt{car}\ e) \mid (\texttt{cdr}\ e) \\
&\quad \mid (\texttt{cons}\ e\ e) \mid (\texttt{map}\ e\ e) \\
&\quad \mid (\texttt{foldl}\ e\ d\ e) \mid (\texttt{foldr}\ e\ d\ e)
\end{aligned}
$$

Here `cons`, `car`, `cdr` denote the primitive operators for making concatenation, head of list, tail of list, respectively. The operator `lambda` declares a lambda expression, and `app` stands for function application. The operator `quote` returns the object without evaluating it, and `()` is used for showing an empty list as standard in Lisp. The variables are encoded with de Bruijn levels declared by `var` for nameless lambda expressions. These levels are encoded as Peano numerals, e.g., `var ()` denotes level 0 and `var (s . ())` denotes level 1. For readability, we would directly use λ-expressions instead of Lisp notations, and use $(x_1\ x_2\ \cdots\ x_n)$ to represent a list $(x_1\ .\ (\cdots\ .\ (x_n\ .\ ())))$ when necessary.

B. SYMBOLIC SEARCH ENGINE

As aforementioned, we adopt miniKanren [16] as the underlying symbolic search engine for synthesis of candidate programs. It is an embedded domain specific language for relational programming, supporting a variety of language idioms and techniques for writing flexible relational programs. Relational programming belongs to the logic programming paradigm except that any logical expressions only contain pure relational goals. The core language of miniKanren [16] extends Scheme with three basic logical operators ==, `conde`, and `fresh`, and one interface operator `run` between miniKanren [16] and the host language, where == unifies two terms, `conde` behaves like logical disjunction, and `fresh` works like a lambda binder introducing lexically-scoped logical variables.

In relational programming, each function `func` can be realized as an equivalent relation `funco`, such that (`funco` i o) holds iff `func`$(i) = o$ for any input-output pair (i, o) in the problem domain. A relation can take unknown logic variables as arguments denoted by underlined letters. Thanks to the flexibility of relational programming, the underlying solver try to produce answers for each relation even when all arguments are free logic variables. Let `eval` be the language interpreter such that given any program P, `eval`$(P, i) = o$ for each input-output pairs. By defining and implementing a relational counterpart `evalo` of the interpreter in miniKanren [16], answering the query (`evalo` $\underline{p}$ i o) amounts to solving the PBE problem, where the unknown variable p is underlined denoting the program to be synthesized, and i and o denote the given input-outputs values, respectively [12].

A relation (possibly containing unknown logic variables) is also called *constraint* denoted by φ. The symbolic search of miniKanren [16] unfolds a constraint by replacing logical variables with possible constraints at a time, until no logical variables can be further expanded. The underlying solver of miniKanren [16] uses a complete interleaving search strategy. If a goal could not be answered, miniKanren [16] will search in a unfinite amount of time. Therefore, assuming a bound on the steps of constraint derivation when necessary.

Consider the PBE problem $(\{((x\ y\ a), (x\ y))\}, P_t)$ as a running example, where P_t is given below:

```
(lambda (cons (car (var ()))
 (cons (car (cdr (var ()))) (quote ()))))
```

The function is to drop the last element of a given input list of length 3. Figure 2 illustrates the expansion of the constraint (`evalo` $\underline{p}$ $(x\ y\ a)$ $(x\ y)$). The expansion process of constraints induces a sequence of state trees with one intermediate state tree given in the figure, and we also room in the first step of expansion at the upper part of the figure. A state tree is formed by leaf nodes and internal nodes. Each leaf node is labelled with a relation unfolded at the current step as well as a set of constraints to be jointly satisfied by expanding the relation. Each internal node is an unfolded leaf node, or either of a conjunction or disjunction node. Given a leaf node, the corresponding partial program can be read-out by following the path from the root to this leaf node. For some unfolded leaf nodes in the figure, we show the partial program with constraints attached to them. The node highlighted in grey is one that cannot be further unfolded.

C. NEURAL-GUIDED PROGRAM SEARCH

We denote by $\mathcal{S}$ a state tree, and by P a partial program candidate. A leaf node in the state tree can be represented as a triplet (R, P, φ) where R is the current unfolded relation, P is the corresponding partial program, and φ is the attached constraints. We denote by $Path_{(R,P,\varphi)}$ a path in the state tree leading from the root to an underlying leaf node. Thus a state tree can be represented as a finite set of leaf nodes, i.e., $\mathcal{S} = \bigcup_j Path_{(R_j,P_j,\varphi_j)}$. The goal of neural-guided program search is to predict the most probable leaf node (equivalently, the corresponding relation or path) to be selected and unfolded next. In order to match the labelled program P_t, given the current state tree:

$$\underset{j}{argmax}\ q(Path_{(R_j,P_j,\varphi_j)} \mid \mathcal{S})$$

Here q denotes the predicted distribution by the statistical model. To train the neural network, the set of state trees will be embedded and fed to the network. Then the cross-entropy loss [21] is used to minimize the distance between the predicted distribution q and the real distribution q_{true}, given below:

$$H(q, q_{true}) = \sum_j - q_{true}(P_j) log(q(P_j))$$

Here note that q_{true} assigns 1 to the leaf node exactly corresponding to the good path that matches the labelled program, and 0 to other leaf nodes.

Policy for State Tree Expansion: Due to the learning ability of neural networks, the quality of training datasets, and the quality of input-output examples for training the statistical model, the path with the highest score predicted by the neural network may not be the right path to be expanded next. To achieve a better generalization accuracy, instead of expanding

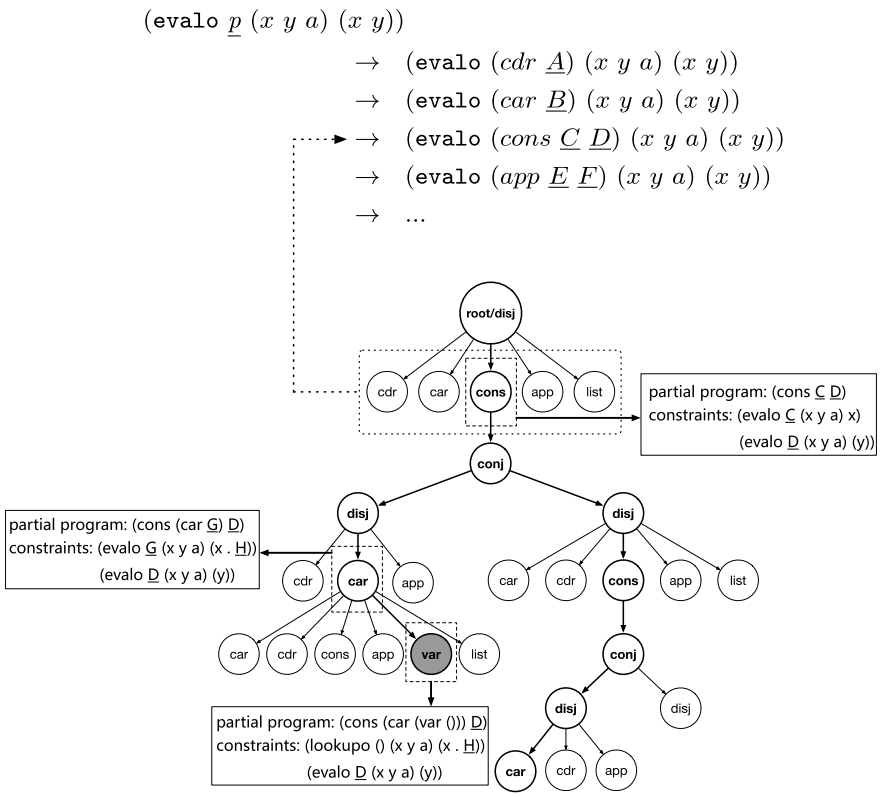


FIGURE 2. An intermediate expanded state tree of (evalo p (x y a) (x y)). The node in grey color will not be further expanded for carrying constraints without logical variables possibly expanded along the current path.

the path P with the highest predicted score, [13] also expands any path P' whose predicted score is close to the best one within a given threshold δ, i.e., $q(P)-q(P') \leq \delta$. In this work, inspired by the search strategy in [13], we expand a sequence of paths $P_0, \ldots, P_k$ with topmost predicted scores, satisfying that $P_i \geq P_{i+1}$ and $P_i - P_{i+1} \leq \delta$ for each i. Further, for the sake of synthesis performance, we limit the number k of paths to be expanded each time.

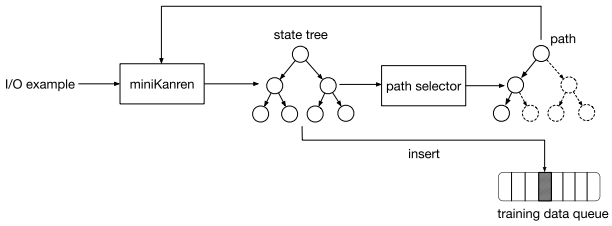


FIGURE 3. The overall process of offline generation of training dataset.

III. THE PROPOSED METHOD FOR NEURAL-GUIDED INDUCTIVE SYNTHESIS

In this section, we present a method for neural-guided synthesis of recursive functional programs, including offline generation of training dataset, supervised learning based on richer features, and design and implementation of recursive relational interpreters for supporting higher-order combinators.

A. OFFLINE GENERATION OF TRAINING DATASET

PBE assumes a very small number of input-output examples and sometime just one input-output pair. The training dataset of neural-guided search is the set of state trees produced by unfolding the recursive constraints. Recall that given the current state tree, the learning target of neural networks is to predict the most probable path to be selected and unfolded next. Figure 3 gives the overall process of the offline generation of training datasets. The "path selector" module iteratively takes as input a state tree and decides which leaf node to be unfolded next, and the path selection choice is fed back to miniKanren [16] for further tree expansion. All the deduced state trees are collected and stored in a data structure called training data queue. Since the neural-guided approach is data-driven, it is crucial to make a high quality dataset that contains diverse samples of state trees. However, the entire search space of state trees is huge and exponentially growing during tree expansion. Therefore, for a satisfactory performance of supervised learning, one needs a tree expansion strategy to trade-off the size and diversity of generated datasets.

Our method is inspired by the reinforcement learning technique of scheduled sampling [22] as explored in [12] and greatly simplified. We explore a simple policy called

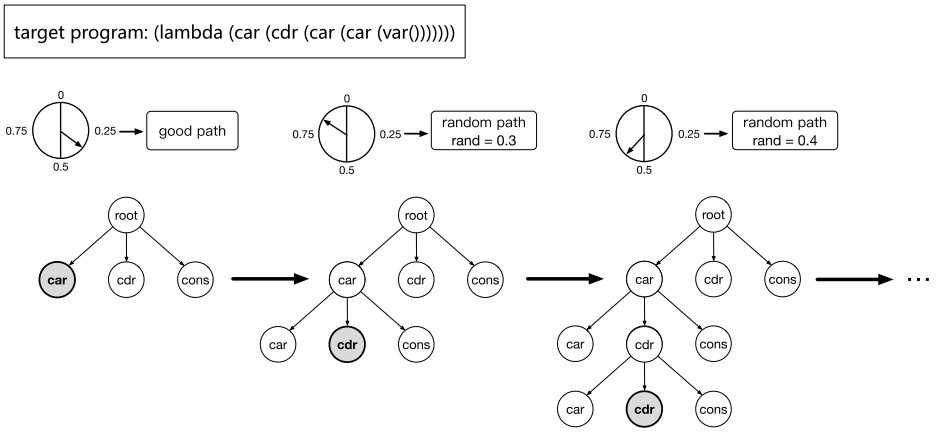


FIGURE 4. Illustrating roulette policy of expanding the state trees. The nodes highlighted in grey color selected to be unfolded at each expansion step. For ease of presentation, it is assumed that there are only three nodes per layer.

roulette policy that non-deterministically chooses the *good path* exactly matching the labelled program or any *random path* in the state tree. The expansion stops when the given step limit is reached. This policy is like dividing a roulette wheel into two sections and each of them corresponds to one policy. Each time one of the two sections is selected at random. Below is how we randomly select a path candidate. Assume there are N leaf nodes and the probability of taking each path is equal to $\frac{1}{N}$. A random seed $rand \in [0, 1]$ is generated which indicates the probability of taking the i^{th} path (numbered from left to right in the tree) with $rand \simeq \frac{i}{N}$. Thus i is taken as the ceiling of $\lceil N \times rand \rceil$.

Figure 4 illustrates the roulette policy with the following target program P_t to be synthesized:

```
(lambda (car (cdr (car (car (var ()))))))
```

Here the section $S_0 = [0, 0.5)$ indicates selecting the good path and the section $S_1 = [0.5, 1]$ indicates selecting a random path. During the tree expansion, $rand_R \in S_0$ is for the first state tree and a good-path is taken for the next step. For the second state tree, $rand_R \in S_1$ and a random path is taken. In this case, another random seed $rand = 0.3$ is generated and thus the second leaf node is selected to be unfolded in the next step. It goes for the third state tree similarly.

In addition to applying the roulette policy for dataset generation, we also generate two datasets on expanded state trees by always following the good-path and by always taking a random path at each step, respectively. The tree kinds of datasets are mixed for training the neural networks in experiments. We refer to Section IV for more details.

B. SUPERVISED LEARNING BASED ON ENRICHED FEATURE EMBEDDING

The key insight for supervised learning trained with a set of state trees is that, the (recursive) constraints attached to each leaf node generated during tree expansion are independent subproblems of PBEs. A sample in the learning task is a state tree that is formed by a set of paths. Recall that each path is represented as a triplet (R, P, φ). To train the network, we have to learn a proper embedding of the characteristics. Following [12], we use a network model containing two phases. The first phase is a RNN model formed by a 2-layer bi-directional LSTMs [23] to learn an embedding of paths. The next phase uses an MLP (Multilayer Perceptron) with activation function ReLU [24] as the as scoring function to predict a score for each path. The two phases are trained in the same architecture of neural networks as illustrated in Figure 5. In [12], only constraints φ are used as the embedding features. In this work, we further extend the model with the following enriched-feature embeddings.

1) PARTIAL PROGRAM EMBEDDING

In addition to the embedding of constraints φ in the model, we encode the pair of (P, φ) by concatenating the embedded vector of P (also trained by RNN) to the embedding of each constraint in φ.

2) TREE CONTEXT EMBEDDING

Although each constraint can be seen as a PBE subproblem independent of the original problem, the state tree contains other information that would be useful for further guiding the tree expansion. The process of program synthesis is a sequence of unfolded state trees $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_n, \ldots$, satisfying that $\mathcal{S}_j$ is subsumed in $\mathcal{S}_{j+1}$ for each $j \geq 0$. Therefore, the current state tree $\mathcal{S}_n$ to be expanded next implicitly contains the selection choices made in all the previous expansion steps. For instance, the expansion may keep going in a wrong way other than the good path until $\mathcal{S}_n$ is reached. We found such an information is useful to shorten the synthesis steps. Further we encode the state tree as the tree context into the learning task. Generally, one could use an attention model to learn how much focus should be given to each expansion path. That is, an attentive network is essentially a weighted aggregation of the information contained in different paths.

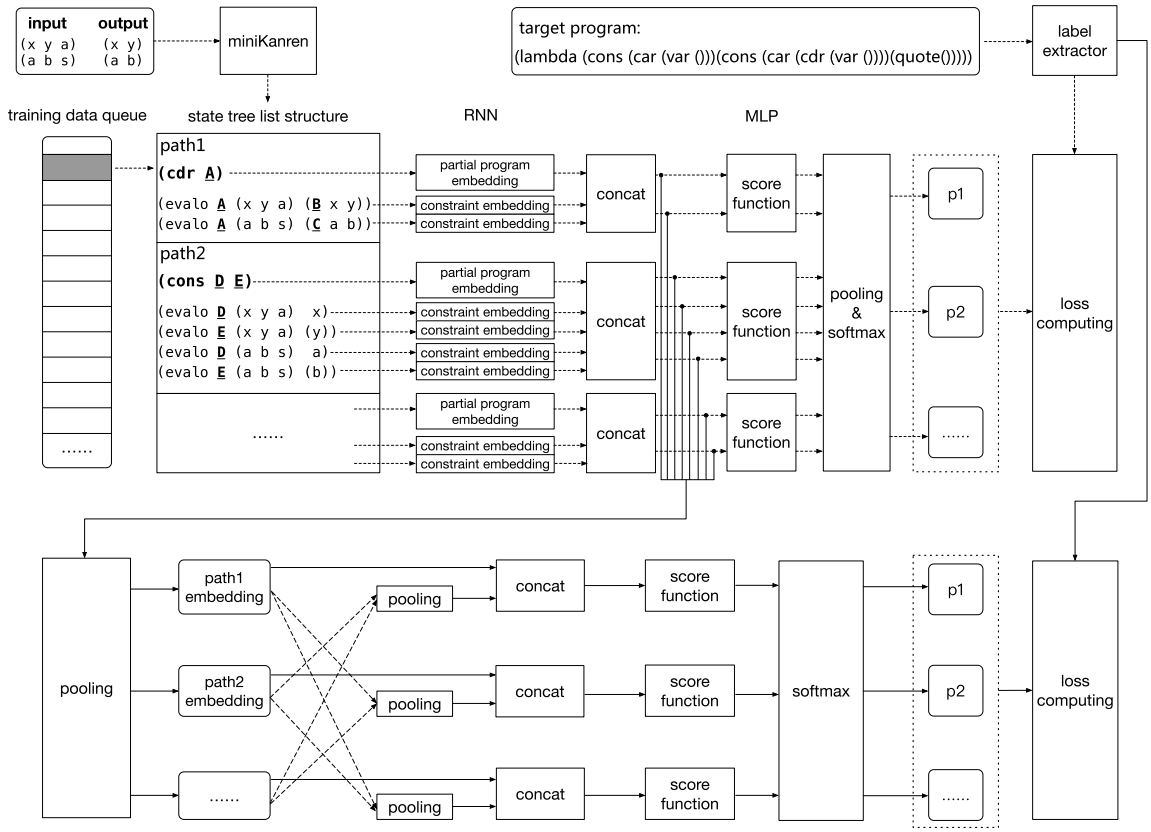


FIGURE 5. The architecture of neural networks with enriched-feature embeddings.

Here we found that a simple mean pooling of path contexts suffice for our training datasets. As shown in the lower part in Figure 5, for each path in the tree, we apply a mean pooling of the embeddings of other paths and concatenate the resulting vector to the original path embedding.

After offline training, the learned network is used to help predict the most probable path to be expanded. During the testing phase, the reduced state trees can be encoded with the embedding of program tokens learned in the training phase, and fed to the neural network model for prediction.

C. RECURSIVE RELATIONAL INTERPRETER FOR HIGHER-ORDER COMBINATORS

This section describes the design and implementation of the recursive relational interpreter for higher-order combinators used in our target DSL. We limit our focus to three popular combinators `map`, `foldl` and `foldr`. Their equivalent relation definitions `mapo`, `foldlo` and `foldro` are given in Figure 6.

The function application `map func` $(x_1 \cdots x_n)$ applies the argument function `func` to $x_1, \ldots, x_n$, respectively, and returns the list `(func` x_1`)` $\cdots$ `(func` x_n`))`. The relation `mapo` takes as arguments some function `func`, the inputs `env`, and the outputs `value`. As aforementioned, `conde` behaves like disjunction, and relations within each disjunct are conjuncted together. Thus the expression grouped by `conde` can be regarded as disjunctive normal form logically. Here there are two disjuncts grouped by `conde`. The first disjunct corresponds to the base case when the input is an empty list and the output is thus an empty list too. The second disjunct corresponds to the inductive step, when `func` is applied to the head of input list and `mapo` is recursively applied to the tail of input list.

The function `foldl` is another widely-used combinator in functional programs. The function application `foldl func acc` $(x_1 \cdots x_n)$ recursively processes the list, combines the results, and returns the value `func (... (func (func acc` x_1`)` x_2`)...)` x_n. In particular, it returns *acc* if the third argument is an empty list. Consider its relation definition `foldlo` in Figure 6. Similarly, there are also two disjuncts grouped by `conde` which correspond to the base case and inductive step of the `foldl` function definition, respectively.

Consider a function `reverse` that reverses a list of elements. It can be defined by using `foldl` as follows:

```
λx.(foldl (λy.λz.(cons y z)) () x)
```

Here we simplify the notations for readability. Based on the function defined in our target DSL, the following is the explicit form in terms of nameless lambda expressions

```
(lambda (foldl (cons (var ())
  (var (s. ())))) () (var ()))
```

```
define-relation (mapo func env value)
  (conde
  /* case 1: if env is an empty list */
    ((== `() env) (== `() value))
  /* case 2: inductive step otherwise*/
    ((fresh (va vd vc vb)
      (== `(,va . ,vd) env)
      (evalo func `(,va . `()) vc)
      (mapo func vd vb)
      (== `(,vc . ,vb) value)
    ))
  )

define-relation (foldlo func acc env value)
  (conde
  /* case 1: if env is an empty list */
    ((== `() env) (== acc value))
  /* case 2: inductive step otherwise */
    ((fresh (va vd vb vc)
      (== `(,va . ,vd) env)
      (== `(,acc . `()) vb)
      (foldlo func vc vd value)
      (evalo func `(,va . ,vb) vc)
    ))
  )

define-relation (foldro func acc env value)
  (conde
  /* case 1: if env is an empty list */
    ((== `() env) (== acc value))
  /* case 2: inductive step otherwise */
    ((fresh (va vd vb vc)
      (== `(,va . ,vd) env)
      (== `(,vb . `()) vc)
      (foldro func acc vd vb)
      (evalo func `(,va . ,vc) value)
    ))
  )
```

FIGURE 6. Relational interpreter for higher-order combinators.

Suppose one applies the reverse function to a list `(x y a)`. We step through the changes in the values of each variable during the interpretation in Table 2. To synthesize a function reverse, one simply call (`evalo` p `env value`) given input-output examples wrapped in `env` and `value`.

TABLE 2. The changes in the values of each variable when interpreting (`evalo reverse (x y a) value`).

Step	env	acc	va	vb	vc	vd
1	(x y a)	()	x	()	(x)	(y a)
2	(y a)	(x)	y	(x)	(y x)	(a)
3	(a)	(y x)	a	(y x)	(a y x)	()
4	()	(a y x)				

The function `foldr` is similar to `foldl` expect that it processes elements in the list from right to left. As given in Figure 6, The relation `foldro` can be defined similarly.

IV. EXPERIMENTS AND RESULTS

We have implemented the proposed method for neural-guided synthesis with relational interpreter in Scheme code and other modules in Python code. We rely on the tool made by Lisa *et al.* in [12] for building the extended relational interpreter. It provides transparent data structures for expanded state trees in miniKanren [16] and enables us to realize the neural-guided synthesis.

In the following, our major goal is to evaluate the proposed synthesis method on generalization accuracy and performance, when applied to unseen inputs for synthesizing non-recursive and recursive functional programs, respectively. We deploy two types of platforms for experiments: (i) Server machine equipped with Intel(R) Core(TM) 3.60GHz i7-9700K processor, GeForce RTX 2070 and 32GB RAM; (ii) Local machine equipped with Intel(R) 3.10GHz Core(TM) i7-5557U processor and 16GB RAM.

A. SYNTHESIS OF NON-RECURSIVE PROGRAMS

1) EXPERIMENTAL CONFIGURATION

We are concerned with list manipulation tasks and mainly use benchmarks from [12] with the small subset of Lisp codes as the synthesis targets. The benchmarks include two kinds of input-output examples called EasyLisp and Cons, respectively. Both benchmarks have 1000 training examples and 100 testing examples. Each example consists of 5 pairs of input-output lists. As aforementioned, the training dataset fed to the neural network is the set of state trees generated during unfolding the recursive constraints. In order to enhance the diversity of generated training datasets, some input-output examples are randomly chosen to be repeated in the training set on purpose. That is, the input-output examples is given as a multiset. The same input-output example would generate varying training dataset. Besides, the three policies introduced in III-A are used to generate datasets separately on the training input-output examples, which jointly result in around 25000-30000 of training samples of state trees. For offline training, the size of each embedded vector for constraints, partial programs and tree contexts is 128. Adam optimizer [25] with weight-decay is used and learning rate decays 10% each 5 epochs. The cross-entropy loss is adopted to minimize the learned predictive distribution and the labelled distribution.

2) GENERALIZATION ACCURACY

To evaluate the generalization ability of our method, we conduct two kinds of synthesis tasks. One applies the tool to unseen inputs of similar kinds, another applies the tool to input lists of unseen lengths, with experimental results summarized in Table 3 and Table 4, respectively. Here we consider two possible completed synthesis results of E-match and G-match when constraints contain no logic variables and could not be further expanded. Recall that both E-match and G-match return synthesized programs correctly satisfying all example-based specifications, whereas the synthesized

TABLE 3. Results on generalization accuracy when applied to unseen input lists of similar length, where M_φ refers to the neural network model based on embeddings of constraints, $M_{P,\varphi}$ refers to the model based on embeddings of both partial programs and constraints, and $M_{P,\varphi,cxt}$ refers to the model based on embeddings of partial programs, constraints and tree contexts.

Benchmarks	EasyLisp			Cons		
Models	M_φ	$M_{P,\varphi}$	$M_{P,\varphi,cxt}$	M_φ	$M_{P,\varphi}$	$M_{P,\varphi,cxt}$
Solved Rate	85%	95%	**96%**	87%	**99%**	98%
E-match	85%	94%	96%	86%	97%	96%
G-match	0%	1%	0%	1%	2%	2%
Average Steps	7.87	6.9	**6.84**	6.95	6.32	**6.22**
Time (server)	40.3s	**35.9s**	36.5s	36.7s	33.6s	**33.3s**
Time (local)	**399s**	835s	840s	**309s**	714s	661s

TABLE 4. Generalization accuracy with increasing the length of input lists, where "–" at the upper right corner means that all results belong to G-match, and "–" at the bottom left corner means that all results belong to E-match.

Models	Repeat(N)	DropLast(N)	BringToFront(N)	
M_φ	3	3	-	E-match
$M_{P,\varphi}$	7	2	2	
$M_{P,\varphi,cxt}$	30+	4	3	
M_φ	30+	4	4	G-match
$M_{P,\varphi}$	**40+**	5	**8**	
$M_{P,\varphi,cxt}$	-	**5**	5	

program exactly match the target program for E-match and is not the same as the target program for G-match.

Table 3 presents the generalization accuracy of our tool. It is measured by the number of programs that the model is able to correctly synthesize as given in the row labelled "Solved Rate". The columns labelled "M_φ", "$M_{P,\varphi}$" and "$M_{P,\varphi,cxt}$" show the impact of different embedding strategies of neural network model on the results of synthesis. To measure the performance of synthesis, we set the maximum state tree expansion steps for each input-output example as twice the optimal synthesis steps (i.e., the expansion steps by always taking the good path). Table 3 shows that, the neural network models $M_{P,\varphi}$ and $M_{P,\varphi,cxt}$ obviously outperform the model M_φ based on embeddings of constraints only used in [12] with offline training. It suggests that using enriched-feature embeddings would possibly lead to a higher predictive accuracy of the neural network. The generalization accuracy of using the neural network models $M_{P,\varphi}$ and $M_{P,\varphi,cxt}$ are pretty much the same considering the solved rate. Yet the synthesis based on the model $M_{P,\varphi,cxt}$ often need fewer synthesis steps in average, as shown in the row labelled "Average Steps". The total time in seconds spent by using each model for synthesizing all testing examples is given in the row labelled "Time". It takes around half a minute to synthesize the testing examples on the server.

Table 4 presents generalization results when applying the synthesis system to unseen input lists of increasing length. We adopt three benchmarks used in [12] including three programs *Repeat*(N), *DropLast*(N) and *BringToFront*(N), respectively, where N is the variable for the length of lists. For the three programs, *Repeat*(N) repeats a token N times, *DropLast*(N) discards the last element of list and *BringToFront*(N) brings the last element of list to the front of list. In the training dataset, the input-output examples include the target programs *Repeat*(N) and *DropLast*(N) tailored for the input list of length $N = 2$ and $N = 3$, respectively. The training dataset does not include any examples labelled with the program *BringToFront*(N). The maximum synthesized steps is set to be 200 steps, and the timeout is set to be 10 minutes. The numbers in the table are the largest length N of lists that our tool could handle and synthesize a program using different models. The larger N indicates a better generalization ability of the model. Although we could not directly compare with the tool in [12] that is not open-sourced, our method shows obviously better generalization ability on the example *Repeat*(N) for which $N = 20+$ is reported in [12]. Table 4 shows that, the model $M_{P,\varphi,cxt}$ based on enriched embeddings of partial programs, constraints and tree contexts demonstrates the best generalization ability in general. Here we only adopt a simple encoding of tree contexts by mean pooling. Advanced embedding methods of tree contexts would possibly further enhance the model's generalization ability.

B. SYNTHESIS OF PROGRAMS WITH COMBINATORS

1) EXPERIMENTAL CONFIGURATION

In this experiment, we evaluate the effectiveness of our method for synthesizing functional programs with higher-order combinators, to more practically manipulates the recursive data structure of lists. As shown in Table 5, we choose several non-trivial textbook examples for list manipulation as our synthesis tasks. We are interested in the generalization ability of our method for synthesizing functional programs with combinators when applied to input lists of increasing length. To this end, we train our neural network model with input-output examples containing lists of short length and test whether our tool can successfully synthesize programs on input-output examples of lists with longer length. Specifically, each example-based specification contains 5 pairs of input-output lists that consist of random numerical numbers generated by running the random seed generator. For each task in Table 5, we prepare 20 example-based specifications for lists of length 1, 2 and 3 each, and then use these 420 example-based specifications in total to generate the training samples offline by applying the proposed three policies. The generation process is similar to the case of non-recursive program synthesis. The offline

TABLE 5. Program synthesis tasks on List manipulation.

Tasks	Input-output Examples	Description
reverse	`(1 2)` → `(2 1)`	Reverse a list
getLast	`(1 2)` → `2`	Get the last element
addZero	`(1 2)` → `(1 2 0)`	Append 0 to the end
stutter	`(1 2)` → `(1 1 2 2)`	Duplicate each element
getHead	`((1 2) (3 4))` → `(1 3)`	Get the first element
dropLast	`(1 2)` → `(1)`	Drop the last element
bringTofront	`(1 2)` → `(2 1)`	Bring the last element to the front of list

generation jointly produces about $18000 - 20000$ state trees as the training dataset for supervised learning. For testing, we prepare for each task one example-based specification for lists of length ranging from 1 to 20 each. That is, there are 20 specifications for each task in total. We iteratively test our synthesis method on specifications from input-output lists of length 1 to that of length 20, and repeat the testing 10 times for each task and report the longest length that can be successfully synthesized. All of the following experiments are conducted on the local machine.

2) GENERALIZATION ACCURACY

As discussed in Section II-C, the path with the highest score predicted by the neural networks may not be the right path to be expanded. Thus we would also like to evaluate the impact of expansion strategies controlled by the threshold δ through our experiments. Recall that δ measures the difference among the predicted scores of paths taken to be expanded. That is, the larger δ, the more paths to be expanded. To further evaluate the impact of feature embeddings used by the statistical model, we conduct the aforementioned experiments for program synthesis based on feature embeddings $M_{P,\varphi}$ and $M_{P,\varphi,cxt}$, with their experimental results given in Table 6 and 7, respectively.

Table 6 and 7 show that, the generalization ability of our method generally becomes betters with increasing the value of δ, which is consistent with theoretical expectations. Our synthesis method based on both model embeddings $M_{P,\varphi}$ and $M_{P,\varphi,cxt}$ performs equally well on tasks of `getLast`, `addZero` and `getHead`, and delivers fair results on the task of `stutter`. For the remaining tasks of `reverse`, `dropLast` and `bringTofront`, the generalization performance shown so far is not very satisfactory. By further analysis, we found that in some cases, many paths of the state tree have close scores predicted by the neural networks and none of them are very high, and the path suggested by the model is not the right path to be expanded. In this situation, we would still possibly miss the good path even when the expansion strategy is used given a large δ.

TABLE 6. Results on generalization accuracy with increasing the length of input lists given the model embedding $M_{P,\varphi}$ and multiple tree expansion policies controlled by the parameter δ.

Task	$M_{P,\varphi}$			
	$\delta = 0$	$\delta = 0.1$	$\delta = 0.3$	$\delta = 0.5$
reverse	2	**5**	**5**	**5**
getLast	**20**	**20**	**20**	**20**
addZero	19	**20**	**20**	**20**
stutter	4	8	**9**	**9**
getHead	**20**	**20**	**20**	**20**
dropLast	**4**	3	**4**	**4**
bringTofront	**3**	2	**3**	**3**

TABLE 7. Results on generalization accuracy with increasing the length of input lists given the model embedding $M_{P,\varphi,cxt}$ and multiple tree expansion policies controlled by the parameter δ.

Task	$M_{P,\varphi,cxt}$			
	$\delta = 0$	$\delta = 0.1$	$\delta = 0.3$	$\delta = 0.5$
reverse	**4**	**4**	**4**	**4**
getLast	**20**	**20**	**20**	**20**
addZero	**20**	**20**	**20**	**20**
stutter	9	9	10	**12**
getHead	**20**	**20**	**20**	**20**
dropLast	**4**	**4**	**4**	**4**
bringTofront	2	2	2	**3**

TABLE 8. The comparison of our method with some state-of-the-art program synthesis systems on generalization performance. Here - means a task can not be synthesized in 10 minutes or need more input-output examples.

Task	$M_{P,\varphi}$	$M_{P,\varphi,cxt}$	Myth [19]	λ^2 [18]	Escher [20]
reverse	5	4	**20**	**20**	1
getLast	**20**	**20**	-	**20**	-
addZero	**20**	**20**	**20**	**20**	**20**
stutter	9	12	-	**20**	-
getHead	**20**	**20**	-	**20**	-
dropLast	**4**	**4**	1	**4**	1
bringTofront	**3**	**3**	2	2	2

We also evaluate and compare our method with other state-of-the-art program synthesis tools. The results of this experiment is given in Table 8. The columns labelled "$M_{P,\varphi}$" and "$M_{P,\varphi,cxt}$" show respectively the best results of our synthesis method according to the experimental results in Table 6 and 7. The columns labelled "Myth", "λ^2" and "Escher" show respectively the generalization performance of these three tools, when applied to the same random example-based specifications for each task as our method uses for testing. It shows that, both our synthesis method and λ^2 performs comparatively well by successfully synthesizing all the tasks within the given source limit. Specifically, our method delivers slightly better performance than λ^2 on the task of `bringTofront`, whereas λ^2 show excellent results on the tasks of `reverse` and `stutter`. The tool Escher sometime fails due to the need of more input-output examples. The tool Myth sometime fails because it could not find a feasible solution within the iteration bound. Note that, all methods perform poorly on tasks of `dropLast` and `bringTofront`.

C. RESULT ANALYSIS

1) OFFLINE TRAINING VS ONLINE TRAINING

Table 9 lists several target programs that are failed to be synthesized by some models within the given limit of synthesis steps. It indicates that online training and offline training complement each other on the diversity of generated training datasets. There is a tradeoff between accuracy and generalization on the size of training datasets. We would like to remark that, our models M_{φ}, $M_{P,\varphi}$ and $M_{P,\varphi,cxt}$ can all successfully synthesize the program on the first row in Table 9. For the programs on the second and third row, if the limit of synthesis steps is further increased to 200, then the models $M_{P,\varphi}$ and $M_{P,\varphi,cxt}$ can synthesize these programs. $M_{P,\varphi}$ takes 18 steps with returning an E-match result, and $M_{P,\varphi,cxt}$ takes 13 steps with returning an G-match result for the first case and takes 18 steps with returning an E-match result for the second case.

TABLE 9. The example programs failed to be synthesized within the given limit of synthesis steps.

Models	Programs	The limit of synthesis steps
[12]	`(lambda (cons (cons (var ()) (var ())) (cons (var ()) (car (cdr (var ()))))))`	200
$M_{P,\varphi}$	`(lambda (car (car (car (cdr (car (cdr (car (var ()))))))))))`	twice the optimal steps
$M_{P,\varphi,cxt}$	`(lambda (cons (cdr (var ())) (car (cdr (var ()))))))` `(lambda (car (cdr (cdr (cdr (cdr (car (cdr (var ()))))))))))`	twice the optimal steps

Thus the synthesis method with offline supervised learning is competitive and promising compared to the synthesis method based on reinforcement learning for training in [12].

2) E-MATCH VS G-MATCH

Sometimes the synthesis succeeds, but returns the G-match results. Table 10 gives three examples to compare the target program to be synthesized with the returned G-match results. In the first example, the user-intended target program is not optimal and the G-match result is correct and much shorter. For the second example, the G-match result is semantically equal to the E-match result yet much longer. For the last example, the G-match result is different from the target program. We leave it as our future work on how to optimize the synthesized programs.

TABLE 10. Comparison between the G-match and E-match results.

Synthesized programs	Type
Example 1	
`(lambda (cons (car (var ())) (cdr (var ())))))`	E-match
`(lambda (var()))`	G-match
Example 2	
`(lambda (cdr (var())))`	E-match
`(lambda (cons (car (cdr (var ()))) (cdr (cdr (var ())))))`	G-match
Example 3	
`(lambda (car (car (cdr (car (var ()))))))`	E-match
`(lambda (cdr (car (cdr (car (var ()))))))`	G-match

3) ON SYNTHESIZING PROGRAMS WITH HIGHER-ORDER COMBINATORS

As shown in Table 8, there is still room for improvement on synthesizing `reverse` and `stutter` in our approach. By further inspection of their synthesis process, we found that the neural network model could not distinguish some sub-problems for synthesis that have similar feature embeddings, and therefore make a wrong prediction during the deductive search. For instance, we found that the first tree expansion step is wrong when synthesizing the task `reverse` from input-output examples of length 5. The three paths with the highest probability in the first step of state tree expansion is given in Table 11.

As shown in Table 11, the predicated score of the good path is far below the score of the selected path. Therefore, if we want to include the good path for expansion, we need to set a very large value of δ, which would cause almost all paths to be selected for expansion and make the neural-guided synthesis approach meaningless. In this case, we can see that the feature embeddings of `(lambda (foldr A B C))` and `(lambda (foldl F G H))` are quite similar, which makes the neural network model confused.

TABLE 11. The three paths with the highest probability in the first step of state tree expansion when synthesizing the task `reverse`: `(lambda (foldl (cons (var ()) (var (s. ()))) () (var ())))`. Here capital letters indicate unknown logic variables.

Program candidates	Predicted score	Remark
`(lambda (foldr A B C))`	0.99927324	*selected path*
`(lambda (map D E))`	0.00072170	
`(lambda (foldl F G H))`	0.00000487	*good path*

A similar situation occurred during the synthesis of `stutter` from input-output examples of length 9. The first 5 steps were correct, and it selected an unintended path `(lambda (foldr (car (cons _ _)) _ (var ())))` next yet `(lambda (foldr (cons (var ()) (cons _ _)) _ (var ())))` is the good path. Again, the reason is because the predicated score of the good path is far below the score of the selected path and we can see that the two paths are quite similar.

To address the problem, one approach is to improve the current feature embeddings by taking into account more context information of the program. Another approach is to combine other search heuristics, e.g., structural properties [7], cost model [18], to further narrow down the search space.

V. RELATED WORK

Program synthesis has been extensively studied in the past few decades [11], [26]. In this section, we mainly discuss prior work that is closely related to our target problem and proposed method.

Symbolic program synthesis has a long history [27], and most methods are DSL-constrained or sometime type-directed enumerative or deductive search of candidate programs [7], [28]–[31]. The search space of symbolic approaches is usually huge prohibiting the technique from scaling to non-trivial synthesis problems. Research on PBE could be dated back to the work of lisp program construction from examples [32] and regained its interests in recent years with the development of machine learning technology [33]–[37]. A popular framework for program synthesis is known as counter-example guided inductive synthesis (CEGIS) which is pioneered by Sketch [38]. There are two main approaches to applying neural networks for program synthesis. Some work designs neural network models to generate intended programs directly as a sequence generation problem. However, these approaches could not guarantee

the correctness of synthesized programs. The neural-guided synthesis takes another approach, and the learned statistic model is used to effectively shrink the symbolic search space. It has emerged as a promising approach that combines the advantages of both symbolic and statistical methods for program synthesis. To our knowledge, currently the approach is only used for synthesis of non-recursive programs in different problem domains.

A. NEURAL-GUIDED PROGRAM SYNTHESIS

In 2017, DeepCoder [14] was proposed to use neural networks to prioritize function components of DSL library trained with input-output examples labelled by program properties. It is a hybrid approach that used the trained network model to guide symbolic search in the language space. Experiments showed that Deepcoder leads to an order of magnitude speed-up over baselines of symbolic techniques and purely statistical methods using RNN. It can also generate low-level and simple programs. As the trailblazer of neural-guided program synthesis, DeepCoder inspired many prominent works.

Sumit Gulwani *et al.* [13] proposed neural-guided deductive search (NGDS) to synthesize string manipulation programs given a small number input-output examples. Similar to DeepCoder, NGDS combines the symbolic techniques and machine learning models, and treat the selection of partial program paths as subproblems of PBE that can be predicted by network models as supervised learning tasks. In particular, NGDS is concerned with real-time program synthesis problems. Experiments showed that the system can synthesize accurate programs with maximum 12 times speed-up compared to state-of-the-art synthesizers by evaluating real-world demands.

Alex Skidanov *et al.* [39] proposed an algorithm for synthesizing programs in a lisp-inspired DSL from textual specifications in natural language and a small number of input-output examples. The algorithm is efficient by using Seq2Tree model which has the similar architecture to the traditional Seq2Seq [40] model with attention mechanism [41] yet generates the AST tree structure in the part of decoder. The node of tree is scored by using tree beam search algorithms until the whole program is completed.

Lisa Zhang *et al.* [12] presented a neural-guided synthesizer for the PBE problems using miniKanren [16] as the underlying symbolic search engine. Compared to previous approaches, they tactically designed an online supervised learning process that alternated enumerative search, neural network training, and prediction by the network model for guiding the search space. By viewing the process as a reinforcement learning problem, they adopted a variety of learning techniques from the area which require nontrivial engineering efforts to implement and directly motivates this work.

Notably, Uri Alon *et al.* [42] presented very recently a neural model to learn continuous distributed vectors for snippets of source codes, inspired by the well-known work of word2vec in NLP. The learned code vectors can be used in other learning scenarios to perform a variety of downstream tasks like code search and completion.

B. RECURSIVE FUNCTIONAL PROGRAM SYNTHESIS

Most work on recursive functional program synthesis from example-based specifications is based on symbolic approach [18]–[20], [43]–[47]. Compared to non-recursive program synthesis, recursive program synthesis appears to be more challenging.

Aws Albarghouthi *et al.* [20] proposed a novel algorithm called Escher to synthesize recursive programs by exploiting a new data structure called goal graph. Escher interacts with an oracle like users by input-output examples and its synthesis process is divided into forward search and conditional inference alternately. Escher tends to need much more examples than initially given. In comparison, our system only need 5 pairs of examples to synthesize program successfully.

Peter-Michael Osera *et al.* [19] proposed an algorithm of recursive program synthesis based on proof-theoretic techniques. The synthesized functional programs are used to process algebraic datatypes. Apart from input-output examples, the type information is also exploited in order to reduce the search cost. On the shape of synthesized program, the constraints are represented by data structure which is named as *refinement trees*.

John Feser *et al.* [18] presented a recursive functional program synthesis system on manipulates recursive data structures called λ^2 that combines three technical ideas of inductive generalization, deduction and enumerative search. In this system, input-output examples are generalized to hypotheses where new input-output examples for the missing sub-expressions are inferred by deduction. The enumerative search is to find hypothesis can be realized into a program that satisfies input-output examples. The higher-order combinators like *map* and *fold* are also supported in their target DSL. By using the best-first enumerative search strategy, the synthesized program is guaranteed to be the simplest.

VI. CONCLUDING REMARKS

We propose a neural-guided method to synthesize functional programs from input-output examples by offline supervised learning. We present simple yet effective policies on generating high-quality samples of subproblems for synthesis and enriched feature embeddings for training an accurate statistical model offline. The learned model is used to guide the top-down deductive search within the symbolic search engine miniKanren [16]. We target functional programs on list manipulation and evaluate our method with synthesis tasks of both non-recursive functional programs and programs with higher-order combinators, which are implemented as primitive operators in our target DSL. Our evaluation shows that the proposed method is promising and competitive as compared to the state-of-the-art inductive synthesizers. It can be likely used as a supplement to other methods for synthesizing non-recursive sub-expressions given its generalization performance.

As our future work, we will consider the combination of generative method and our method. It is inspired by [48], which adopts Generative Adversarial Network(GAN) [49] to generate programs based on natural languages. In our case, GAN can be used to generate programs based on input-output examples and constraints. Different from pure supervised learning method, GAN is able to generate data which doesn't exist in training data, which may enhance model generalization capability.

ACKNOWLEDGMENT

The authors would like to thank Zhenjiang Hu and Jian Guo for their valuable suggestions and comments and Lisa Zhang for sharing her tools and benchmarks.

REFERENCES

[1] S. Gulwani, "Programming by examples: Applications, algorithms, and ambiguity resolution," in *Proc. 19th Int. Symp. Princ. Pract. Declarative Program.*, W. Vanhoof and B. Pientka, Eds., Namur, Belgium, Oct. 2017, p. 2, doi: 10.1145/3131851.3131853.

[2] R. Singh and S. Gulwani, "Synthesizing number transformations from input-output examples," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2012, pp. 634–651.

[3] S. Gulwani, "Programming by examples—And its applications in data wrangling," in *Dependable Software Systems Engineering* (NATO Science for Peace and Security Series-D: Information and Communication Security), J. Esparza, O. Grumberg, and S. Sickert, Eds. Amsterdam, The Netherlands: IOS Press, vol. 45, 2016, pp. 137–158, doi: 10.3233/978-1-61499-627-9-137.

[4] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2005, pp. 226–238.

[5] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 772–781.

[6] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2013, pp. 15–26.

[7] W. Lee, K. Heo, R. Alur, and M. Naik, "Accelerating search-based program synthesis using learned probabilistic models," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 436–449, Dec. 2018.

[8] R. Ji, Y. Sun, Y. Xiong, and Z. Hu, "Guiding dynamic programing via structural probability for accelerating programming by example," *Proc. ACM Program. Lang.*, vol. 4, pp. 1–29, Nov. 2020.

[9] R. Alur, P. Černỳ, and A. Radhakrishna, "Synthesis through unification," in *Proc. Int. Conf. Comput. Aided Verification*. Cham, Switzerland: Springer, 2015, pp. 163–179.

[10] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur, "TRANSIT: Specifying protocols with concolic snippets," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 287–296, Jun. 2013.

[11] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Found. Trends Program. Lang.*, vol. 4, nos. 1–2, pp. 1–119, 2017, doi: 10.1561/2500000010.

[12] L. Zhang, G. Rosenblatt, E. Fetaya, R. Liao, W. E. Byrd, M. Might, R. Urtasun, and R. S. Zemel, "Neural guided constraint logic programming for program synthesis," in *Proc. Adv. Neural Inf. Process. Syst., Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, Montréal, QC, Canada, Dec. 2018, pp. 1744–1753.

[13] 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 *Proc. 6th Int. Conf. Learn. Represent. (ICLR)*, Vancouver, BC, Canada, Apr./May 2018, pp. 1–18. [Online]. Available: https://openreview.net/forum?id=rywDjg-RW

[14] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to write programs," in *5th Int. Conf. Learn. Represent. (ICLR)*, Toulon, France, Apr. 2017, pp. 1–21. [Online]. Available: https://openreview.net/forum?id=ByldLrqlx

[15] S. Kfir and C. Fournier, "DAML: The contract language of distributed ledgers," *Commun. ACM*, vol. 62, no. 9, pp. 48–54, Aug. 2019.

[16] D. P. Friedman, W. E. Byrd, and O. Kiselyov, "The reasoned schemer," *The Reasoned Schemer*. Cambridge, MA, USA: MIT Press, 2005.

[17] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," 2015, *arXiv:1506.03099*. [Online]. Available: http://arxiv.org/abs/1506.03099

[18] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, D. Grove and S. Blackburn, Eds., Portland, OR, USA, Jun. 2015, pp. 229–239, doi: 10.1145/2737924.2737977.

[19] P.-M. Osera and S. Zdancewic, "Type-and-example-directed program synthesis," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 619–630, Aug. 2015.

[20] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2013, pp. 934–950.

[21] R. Y. Rubinstein, "The cross-entropy method for combinatorial and continuous optimization," *Methodol. Comput. Appl. Probab.*, vol. 1, no. 2, pp. 127–190, Sep. 1999.

[22] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," in *Proc. Adv. Neural Inf. Process. Syst., Annu. Conf. Neural Inf. Process. Syst.*, Montreal, QC, Canada, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., Dec. 2015, pp. 1171–1179. [Online]. Available: https://proceedings.neurips.cc/paper/2015/hash/e995f98d56967d946471af29d7bf99f1-Abstract.html

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.

[24] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. 14th Int. Conf. Artif. Intell. Statist. JMLR Workshop Conf.*, 2011, pp. 315–323.

[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, San Diego, CA, USA, May 2015, pp. 1–15. [Online]. Available: http://arxiv.org/abs/1412.6980

[26] S. Gulwani, "Dimensions in program synthesis," in *Proc. 12th Int. ACM SIGPLAN Symp. Princ. Pract. Declarative Program. (PPDP)*, 2010, pp. 13–24.

[27] R. J. Waldinger and R. C. T. Lee, "PROW: A step toward automatic program writing," in *Proc. 1st Int. Joint Conf. Artif. Intell.*, Washington, DC, USA, May 1969, pp. 241–252. [Online]. Available: http://ijcai.org/Proceedings/69/Papers/024.pdf

[28] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Proc. Formal Methods Comput.-Aided Design*, Oct. 2013, pp. 1–8.

[29] Y. Feng, R. Martins, O. Bastani, and I. Dillig, "Program synthesis using conflict-driven learning," in *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Philadelphia, PA, USA, Jun. 2018, pp. 420–435, doi: 10.1145/3192366.3192382.

[30] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann, "Resource-guided program synthesis," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Phoenix, AZ, USA, Jun. 2019, pp. 253–268, doi: 10.1145/3314221.3314602.

[31] N. Polikarpova and I. Sergey, "Structuring the synthesis of heap-manipulating programs," *PACMPL*, vol. 3, pp. 72:1–72:30, Jan. 2019, doi: 10.1145/3290385.

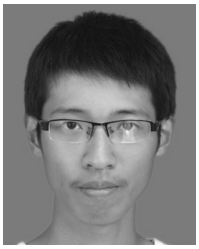
[32] P. D. Summers, "A methodology for LISP program construction from examples," *J. ACM*, vol. 24, no. 1, pp. 161–175, Jan. 1977.

[33] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," in *5th Int. Conf. Learn. Represent. (ICLR)*, Toulon, France, Apr. 2017, pp. 1–14. [Online]. Available: https://openreview.net/forum?id=rJ0JwFcex

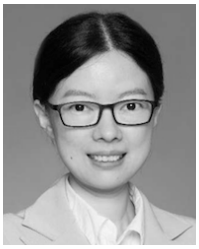
[34] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli, "RobustFill: Neural program learning under noisy I/O," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, Sydney, NSW, Australia, Aug. 2017, pp. 990–998. [Online]. Available: http://proceedings.mlr.press/v70/devlin17a.html

[35] X. Chen, C. Liu, and D. Song, "Execution-guided neural program synthesis," in *Proc. 7th Int. Conf. Learn. Represent. (ICLR)*, New Orleans, LA, USA, May 2019, pp. 1–15. [Online]. Available: https://openreview.net/forum?id=H1gfOiAqYm

[36] C. Shu and H. Zhang, "Neural programming by example," in *Proc. 31st AAAI Conf. Artif. Intell.*, San Francisco, CA, USA, Feb. 2017, pp. 1539–1545. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14482
[37] X. Chen, C. Liu, and D. Song, "Towards synthesizing complex programs from input-output examples," in *6th Int. Conf. Learn. Represent. (ICLR)*, Vancouver, BC, Canada, Apr. 2018. [Online]. Available: https://openreview.net/forum?id=Skp1ESxRZ
[38] A. Solar-Lezama, L. Tancau, R. Bodík, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Proc. 12th Int. Conf. Architectural Program. Lang. Operating Syst. (ASPLOS)*, San Jose, CA, USA, J. P. Shen and M. Martonosi, Eds., Oct. 2006, pp. 404–415, doi: 10.1145/1168857.1168907.
[39] I. Polosukhin and A. Skidanov, "Neural program search: Solving programming tasks from description and examples," in *Proc. 6th Int. Conf. Learn. Represent. (ICLR)*, Vancouver, BC, Canada, Apr./May 2018, pp. 1–11. [Online]. Available: https://openreview.net/forum?id=BJTtWDyPM
[40] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst., Annu. Conf. Neural Inf. Process. Syst.*, Montreal, QC, Canada, Dec. 2014, pp. 3104–3112. [Online]. Available: http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks
[41] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Lisbon, Portugal, Sep. 2015, pp. 1412–1421. [Online]. Available: https://www.aclweb.org/anthology/D15-1166/
[42] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *PACMPL*, vol. 3, pp. 40:1–40:29, Jan. 2019, doi: 10.1145/3290353.
[43] P. Flener, "Inductive logic program synthesis with DIALOGS," in *Proc. Int. Conf. Inductive Log. Program.* Berlin, Germany: Springer, 1996, pp. 175–198.
[44] F. Esposito, D. Malerba, and F. A. Lisi, "Induction of recursive theories in the normal ILP setting: Issues and solutions," in *Proc. Int. Conf. Inductive Log. Program.* Berlin, Germany: Springer, 2000, pp. 93–111.
[45] A. Varlaro, M. Berardi, and D. Malerba, "Learning recursive theories with the separate-and-parallel conquer strategy," in *Proc. Workshop Adv. Inductive Rule Learn. Conjunct (ECML/PKDD)*, 2004, pp. 179–193.
[46] R. Olsson, "Inductive functional programming using incremental program transformation," *Artif. Intell.*, vol. 74, no. 1, pp. 55–81, Mar. 1995.
[47] B. M. Østvold, "Inductive synthesis of recursive functional programs," *ACM SIGPLAN Notices*, vol. 32, no. 8, p. 323, 1997.
[48] Y. Zhu, Y. Zhang, H. Yang, and F. Wang, "GANCoder: An automatic natural language-to-programming language translation approach based on GAN," in *Proc. 8th CCF Int. Conf., Natural Lang. Process. Chin. Comput. (NLPCC)* (Lecture Notes in Computer Science), vol. 11839, J. Tang, M. Kan, D. Zhao, S. Li, and H. Zan, Eds., Dunhuang, China. Cham, Switzerland: Springer, 2019, pp. 529–539, doi: 10.1007/978-3-030-32236-6_48.
[49] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *Proc. Adv. Neural Inf. Process. Syst., Annu. Conf. Neural Inf. Process. Syst.*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., Montreal, QC, Canada, Dec. 2014, pp. 2672–2680. [Online]. Available: https://proceedings.neurips.cc/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html



YUHONG WANG received the B.S. degree in computer science from the Wuhan University of Technology, Wuhan, China, in 2018. He is currently the Graduate Student with the School of Software Engineering, East China Normal University. His research interest includes program synthesis.



XIN LI received the Ph.D. degree in information processing from the Japan Advanced Institute of Science and Technology, Nomi, in 2007. She is currently a Research Associate Professor with East China Normal University, Shanghai. Her research interests include formal methods and program verification, especially about the theory and practice of model checking, and interdisciplinary machine learning research.