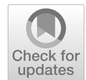
SN Computer Science (2022) 3:218
https://doi.org/10.1007/s42979-022-01090-4



ORIGINAL RESEARCH

Flexible Example-Based Program Synthesis on Tree-Structured Function Compositions

Bálint Mucsányi[1] · Bálint Gyarmathy[1] · Ádám Czapp[1] · Balázs Pintér[1]

Received: 17 May 2021 / Accepted: 11 March 2022 / Published online: 10 April 2022
© The Author(s) 2022

Abstract

We introduce a flexible program synthesis system whose task is to predict function compositions that transform given inputs to their corresponding given outputs. We process input lists in a sequential manner, allowing the system to generalize to a wide range of input lengths. We separate the operator and the operand in the lambda functions of the used higher order functions to achieve significantly wider numeric parameter ranges compared to the previous works. The evaluations show that this approach is competitive with state-of-the-art systems, while it is much more flexible in terms of the input length, the parameters of the lambda functions, and the integer range of the inputs and outputs. We extend the system to handle tree-structured function compositions by introducing two additional functions (`zip_with`, copy) and the ability to represent unfinished function compositions during the synthesis process. The extended system achieves state-of-the-art results while synthesizing complex function compositions with multiple forks. We believe that flexibility in these aspects is an important step towards solving real-world problems with example-based program synthesis.

Keywords Program synthesis · Programming by examples · Beam search · Recurrent neural network · Gated recurrent unit · Tree-structured composition

Introduction

Program synthesis aims to generate a program expressed in a formal language that meets some constraints posed by the user, where the constraints are not necessarily provided in a formal fashion. Some researchers consider this broad problem to be "the holy grail of Computer Science" [11], and understandably so: in the future, systems capable of tackling such tasks may provide explainable solutions to problems that are either difficult to algorithmize or even deemed unsolvable today.

There are two main branches of program synthesis [11]. *Deductive* program synthesis aims to produce a demonstrably correct program that conforms to a formal, rule-based specification that connects the possible inputs of the program with their outputs [1]. Such specifications, however, can prove to be even harder to provide than solving the problem at hand. In the case of *inductive* program synthesis, the desired program's expected operation is demonstrated with examples, or a textual representation [6].

Programming by Examples (PbE) is a demonstrational approach to program synthesis to specify the desired behavior of a program. The examples are composed of one or more inputs and their associated expected output [9]. Because they do not require formal specifications, PbE systems could be suitable for applications which target end-users without formal computer science training. A good example of this is Flash Fill in Microsoft Excel, which synthesizes and applies programs that transform strings based on examples provided by the user [10].

This article is part of the topical collection "Pattern Recognition Applications and Methods" guest edited by Ana Fred, Maria De Marsico and Gabriella Sanniti di Baja.

✉ Bálint Mucsányi
jlv5ae@inf.elte.hu

Bálint Gyarmathy
mzobld@inf.elte.hu

Ádám Czapp
plg78q@inf.elte.hu

Balázs Pintér
pinter@inf.elte.hu

[1] Faculty of Informatics, ELTE Eötvös Loránd University, Pázmány Péter sétány 1/C, Budapest 1117, Hungary

The two main targets of PbE systems are string transformations [10, 14, 17, 19] and list manipulations [2, 8, 24]. We address PbE systems that target list manipulations in this paper.

To deal with the combinatorial complexity of the search in the space of programs that possibly satisfy the provided specification, early program synthesis systems used theorem-proving algorithms and carefully hand-crafted heuristics to prune the search space [18, 21]. Utilizing the advances of deep learning, the field of program synthesis experienced great breakthroughs regarding both accuracy and speed [2]. One of the most eminent approaches in integrating machine learning algorithms into the synthesis process was to augment popular heuristics used in the search with the predictions of such algorithms [14, 17].

The seminal work in the field is DeepCoder [2], which serves as a baseline for several more recent papers [8, 14, 17, 24]. PCCoder [24] greatly advanced the performance of the synthesis process on the Domain-Specific Language (DSL) defined by DeepCoder, reducing the time needed by the search by orders of magnitude while achieving remarkably better results.

In spite of these great advances, there have not yet been numerous examples of successful applications of example-based program synthesis systems in real-world environments.

We think that the causes are mainly limitations of these systems, such as (i) the limitation of static or upper bounded input vector sizes, (ii) the agglutination of tokens of the used formal language, such as handling operators and their integer parameters jointly in lambda expressions of higher order functions [e.g., $(+1)$ and $(*\ 2)$], (iii) the limited integer parameter ranges of the lambda operators, and (iv) the limited integer ranges of inputs, intermediate values, and outputs.

As a consequence of (ii), the number of lambda functions required is a product of the number of supported lambda operators and their possible parameters. For example, a separate lambda function is required for each $(+1)$, $(+2)$, ...$(+n)$. Thus, the number of possible function combinations to be used in systems following this strategy is greatly reduced. Poor generalization performance to input lengths beyond a constant maximum length L is the effect of (i).

In this paper, we would like to resolve these limitations. We implement a DSL with similar functions to the ones used by DeepCoder using a function-composition-based system, in which we treat lambda operators and their parameters separately. This enables us to reduce the number of lambda functions required from the product of the number of lambda operators and their parameters to the sum of them (considering the parameters as nullary functions), and so broaden the range of possible lambda expressions: we expand the range of the allowed numerical values from $[-1,\ 4]$ to $[-8,\ 8]$.

We also extend the range of the possible integers in the outputs and intermediate values fourfold from $[-256,\ 256]$ to $[-1024,\ 1024]$. These ranges might be extended further if needed as we do not embed the integers so we do not impose restrictions on the range of inputs and outputs.

Similarly to DeepCoder and PCCoder, we use a deep neural network to assist our search algorithm. The neural network accepts input–output pairs of any length and predicts the next function to incorporate into the composition that solves the problem. Thus, the network acts as a heuristic for our search algorithm based on beam search, which uses predefined, optimized beam sizes on each level.

The main limitation of our initial system [12] is that it synthesizes function compositions where functions take only a single list as an input in addition to their fixed parameters, and the predicted next function is applied to the list output of the previous function; hence, the generated compositions are flat chains of functions.

We lift this limitation by extending our system to synthesize tree-structured function compositions. We introduce two additional functions called `zip_with` and `copy`; these allow forks in the composition. We also introduce *state tuples*, which are tuples that contain unmerged branches of function compositions. The state tuples make representing unevaluable compositions possible and allow the system to build each branch of the composition separately.

The `zip_with` function can merge two of these branches into a fork, while the `copy` function copies one of the branches, so it can be used at multiple places in the composition tree. At the end of a successful program synthesis process, the branches are all merged into one, producing the solution to the posed problem in the form of a function composition.

Our contributions are:

- We introduce a recurrent neural network architecture that generalizes well to different input lengths.
- We treat the operators in lambda functions separately from their parameters. This allows us to significantly extend the range of their parameters.
- Our architecture does not pose artificial limits on the range of integers acceptable as inputs, intermediate results, or outputs. We extend the range of intermediate results and outputs fourfold compared to previous works.
- We propose an extension to the system which can generate tree-structured compositions and obtains state-of-the-art results while synthesizing complex compositions (i.e., compositions with multiple forks).

The contributions serve to increase the flexibility of the method to take a step towards real-world tasks. We named our method *FlexCoder*, and refer to its presented extension as *the extended FlexCoder*.

Related Work

As the seminal paper in the field, DeepCoder serves as a baseline for several systems in neural program synthesis. Its neural network predicts which of their supported functions are present in the program, and therefore helps guide the search algorithm. However, their network is not used step by step throughout the search, only at the beginning of it.

To tackle the combinatorial nature of predicting all functions present in the solution program, PCCoder implemented a step-wise search which uses the current state at each step to predict only the next statement of the program, including both the function (operator) and parameters (operands). They use Complete Anytime Beam Search (CAB) [23] and cut the runtime by two orders of magnitude compared to DeepCoder.

As the neural networks used by DeepCoder and PCCoder do not process the input sequentially, they can only handle inputs with maximum length L (or shorter, owing to the use of padding), which is a possible shortcoming that has been mentioned in the DeepCoder paper. The default maximum length is $L = 20$ for both systems. FlexCoder solves this problem using GRU [5] layers to process the inputs, so it can work with a large range of input sizes without harming the trainability of the neural network.

Both DeepCoder and PCCoder embed the integers in the input–output lists, narrowing down the range of possible integer values considerably. FlexCoder does not embed the elements of the lists; its integer ranges are wider and more extensible.

We separate the lambda function parameters of our higher order functions into operators and numeric parameters to significantly widen the operand range compared to DeepCoder and PCCoder. This approach resolves the bound nature of their parameter functions, where they only have a few predefined functions with the given operator and operand (eg. $(+1)$, $(*\ 2)$).

Similarly to PCCoder, FlexCoder also uses the neural network in each step of the search process; thus, the network's task is to predict only the next function in the solving composition, given an input state and an output state.

Feng et al. [8] provide an example of successfully using function compositions to represent synthesized programs. Their conflict-driven learning-based method can learn a knowledge base consisting of lemmas to gradually decrease the program space to be searched. They outperform a reimplementation of DeepCoder using the same Domain-Specific Language (DSL).

The work of Kalyan et al. [14] introduces real-world input–output examples for their neural-guided deductive search, combining heuristics (symbolic approach) with neural networks (statistical approach) in the synthesizing process. Their ranking function serves the same role as our network in their approach to synthesize string transformations.

Ellis et al. [7] use a Sequential Monte Carlo method to explore the program spaces their context-free grammars define. Their set of partial programs denoted by *pp* allows them to represent incomplete programs and was the main motivation of our use of *state tuples*.

Methods

In this section, we introduce our approach to neural-guided program synthesis based on beam search with optimized beam sizes for every level. Each step of the beam search is guided by the predictions of the neural network. The DSL of function compositions is built of well-known (possibly higher order) functions from functional programming. The two main parts of FlexCoder are the *beam search algorithm*, and the *neural network*. Another important part is the *context-free grammar* which defines the DSL.

Figure 1 shows an overview of FlexCoder. In each step, an input–output pair is passed to the neural network, which predicts the next function of the sequential composition. This predicted function is applied to the input of the current iteration, producing the new input for the next iteration of the algorithm. This is continued until either a solution is found or the iteration limit is reached.

Figure 2 illustrates how the extended FlexCoder constructs a solution to PbE problems. It uses the same iterative approach as FlexCoder, but the basis of the synthesis process is different and two new functions are introduced.

Example Generation and Grammar

We represent the function compositions using context-free grammars (CFGs) [4]. The clear-cut structure makes the grammar easily extensible with new functions and more parameters. We implemented the context-free grammar using the Natural Language Toolkit [3]. The whole grammar with short descriptions of each function can be seen in Fig. 3 for FlexCoder, and in Fig. 4 for the extended FlexCoder.

The numeric parameters are taken from the $[-8,\ 8]$ interval in both DSLs. This range is larger than the one used by PCCoder as they use a range of $[-1,\ 4]$. The elements in the intermediate values and output lists are taken from the $[-1024,\ 1024]$ range ($[-256,\ 256]$ in PCCoder), while the input lists are from the $[-256,\ 256]$ range. The input range is the same as for PCCoder for the sake of comparability in our evaluations, it could easily be increased to the $[-1024, 1024]$ range for example.

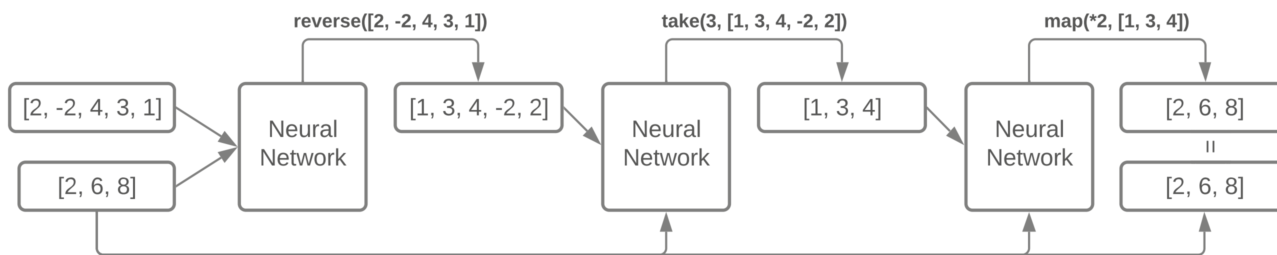


Fig. 1 The process of program synthesis. The task is to find a program that conforms to a specification. FlexCoder synthesizes a function composition which transforms the input list into the output list or output integer value. The input and output serve as the specification. In this case, the input is [2, – 2, 4, 3, 1] and the output is [2, 6, 8]. The figure depicts how at each step of the synthesis process, the neural network's predicted function is applied to the input, and then, the result is fed back into the neural network in the next step. The *reverse* function returns a new list that contains the same elements as the input list but in reversed order. The *map* function applies the lambda expression consisting of the given operator and numeric parameter (in this example '$*$' and '2', respectively) to every element of its input list, resulting in a new list. The *take* function takes some elements (in this case '3') from the front of the input list and returns these in a new list. After executing these functions in the right order on the input list, we can see that the current input list equals the output list, which means we found a solution: $map(*\,2,\ take(3,\ reverse(input)))$

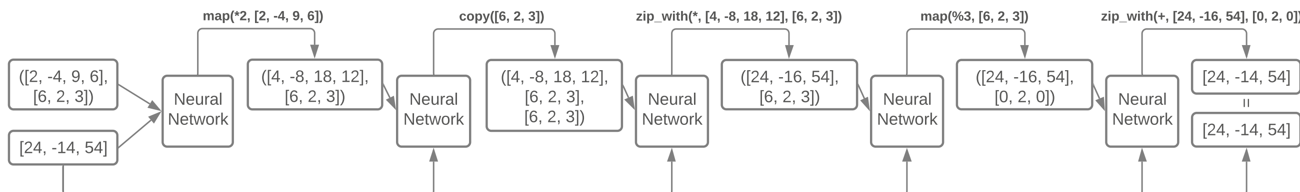


Fig. 2 The iterative construction of a function composition in the extended FlexCoder. An important difference between this example and the one in Fig. 1 is having two input lists instead of one and using two new functions (*copy* and *zip_with*). During the synthesis process, a *state tuple* is updated in each step, which consists of unmerged branches of function compositions. It contains the provided inputs in the beginning. The *zip_with* function takes two input lists and creates a new one whose *i*-th element is the result of the given operator applied to the input lists' *i*-th elements. The *copy* function is only used internally in the synthesis process. This copies one branch of the state tuple to save it before applying additional functions to it. At the end of the process, the $zip_with(+,\ zip_with(*,\ map(*\,2,\ [2, -4,\ 9,\ 6]),\ map(\%3,\ [6,\ 2,\ 3])))$ function composition is constructed, which is a solution to the posed problem

We did not include the *search* function in the DSL of extended FlexCoder, because we realized that it is not a useful component of the compositions. *Search* can only be applied to function compositions as the last transformation, where the range of possible integer values in the lists is $[-1024,\ 1024]$. In contrast, the *search* function can only be parameterized from the $[-8,\ 8]$ range, which is a small fraction, $\frac{17}{2049}$th of the range of the outputs. Consequently, the list contains the number *search* is looking for only in a very small fraction of the cases.

The *copy* function is not present in the extended FlexCoder's DSL, as it is a helper function used exclusively during the synthesis process, so it is not present in the synthesized compositions. It copies branches of function compositions built from the functions in the DSL. Regarding the extended FlexCoder, we refer to the functions present in the DSL as *regular functions*, and we use the term *synthesis function* when we also include *copy*. In FlexCoder, the two terms have the same meaning, and can be used interchangeably.

The lambda functions are divided into two categories based on their return type: Boolean lambda functions used by *filter* and numeric lambda functions used by *map* in FlexCoder. In the extended FlexCoder, there are also binary lambda functions (i.e., operators like +, –, *, etc.) which are used by the *zip_with* function. We define rules for the lambda functions to avoid errors or identity functions, such as dividing by 0, or the $(+0)$ and the $(*\ 1)$ functions. In the extended FlexCoder, we also remove $(/(-1))$, as it is equivalent to $(*\ (-1))$.

The CFGs are used to generate the functions with every possible parameterization, which are then combined into compositions. In FlexCoder, each composition is generated iteratively by choosing a function and sequentially adding it to the already generated composition. This process continues until the needed number of compositions are generated.

As the extended FlexCoder introduces *zip_with*, the compositions are no longer sequential (an example of a tree-structured composition can be seen in Fig. 5). The system can generate compositions where the programs expect

$$
\begin{aligned}
S &\rightarrow \mathit{ARRAY_FUNCTION}\\
S &\rightarrow \mathit{NUMERIC_FUNCTION}\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{sort}(\mathit{array})\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{take}(\mathit{POS}, \mathit{array})\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{drop}(\mathit{POS}, \mathit{array})\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{reverse}(\mathit{array})\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{map}(\mathit{NUM_LAMBDA}, \mathit{array})\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{filter}(\mathit{BOOL_LAMBDA}, \mathit{array})\\
\mathit{NUMERIC_FUNCTION} &\rightarrow \mathit{max}(\mathit{array}) \mid \mathit{min}(\mathit{array})\\
\mathit{NUMERIC_FUNCTION} &\rightarrow \mathit{sum}(\mathit{array}) \mid \mathit{count}(\mathit{array})\\
\mathit{NUMERIC_FUNCTION} &\rightarrow \mathit{search}(\mathit{NUM}, \mathit{array})\\
\mathit{NUM} &\rightarrow \mathit{NEG} \mid 0 \mid \mathit{POS}\\
\mathit{NEG} &\rightarrow -8 \mid -7 \mid \ldots \mid -2 \mid -1\\
\mathit{POS} &\rightarrow 1 \mid \mathit{GREATER_THAN_ONE}\\
\mathit{GREATER_THAN_ONE} &\rightarrow 2 \mid 3 \mid \ldots \mid 8\\
\mathit{BOOL_LAMBDA} &\rightarrow \mathit{BOOL_OPERATOR}\ \mathit{NUM}\\
\mathit{BOOL_LAMBDA} &\rightarrow \mathit{MOD}\\
\mathit{BOOL_OPERATOR} &\rightarrow == \mid < \mid >\\
\mathit{MOD} &\rightarrow \%\ \mathit{GREATER_THAN_ONE} == 0\\
\mathit{NUM_LAMBDA} &\rightarrow *\ \mathit{MUL_NUM}\\
\mathit{NUM_LAMBDA} &\rightarrow /\ \mathit{DIV_NUM}\\
\mathit{NUM_LAMBDA} &\rightarrow +\ \mathit{POS}\\
\mathit{NUM_LAMBDA} &\rightarrow -\ \mathit{POS} \mid \%\ \mathit{GREATER_THAN_ONE}\\
\mathit{MUL_NUM} &\rightarrow \mathit{NEG} \mid 0 \mid \mathit{GREATER_THAN_ONE}\\
\mathit{DIV_NUM} &\rightarrow \mathit{NEG} \mid \mathit{GREATER_THAN_ONE}\\
\mathit{array} &\rightarrow \mathit{list} \mid \mathit{ARRAY_FUNCTION}
\end{aligned}
$$

Fig. 3 The grammar used in FlexCoder in CFG notation. This grammar is used to generate all functions with every possible parameterizations. These are then combined to generate function compositions. The functions do not modify the input array; they return new arrays. The *sort* function sorts the elements of *array* in ascending order. *Take* keeps, *drop* discards the first POS elements of *array*. The *reverse* function reverses the elements of the input. *Map* and *filter* are the two higher order functions used in this grammar. *Map* applies the NUM_LAMBDA function on each element of its parameter. *Filter* only keeps the elements of *array* for which the BOOL_LAMBDA function returns true. The *min* and *max* functions return the smallest and the largest element of *array*. The *sum* function returns the sum of, *count* returns the number of elements of *array*. *Search* returns the index of the element of *array* which is equal to NUM. In the generated examples, we only accept cases where the NUM is present in *array*

$$
\begin{aligned}
S &\rightarrow \mathit{ARRAY_FUNCTION}\\
S &\rightarrow \mathit{NUMERIC_FUNCTION}\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{sort}(\mathit{array})\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{take}(\mathit{POS}, \mathit{array})\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{drop}(\mathit{POS}, \mathit{array})\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{reverse}(\mathit{array})\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{map}(\mathit{NUM_LAMBDA_UNARY}, \mathit{array})\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{filter}(\mathit{BOOL_LAMBDA_UNARY}, \mathit{array})\\
\mathit{ARRAY_FUNCTION} &\rightarrow \mathit{zip_with}(\mathit{NUM_LAMBDA_BINARY}, \mathit{array}, \mathit{array})\\
\mathit{NUMERIC_FUNCTION} &\rightarrow \mathit{max}(\mathit{array}) \mid \mathit{min}(\mathit{array})\\
\mathit{NUMERIC_FUNCTION} &\rightarrow \mathit{sum}(\mathit{array}) \mid \mathit{length}(\mathit{array})\\
\mathit{NUM} &\rightarrow \mathit{NEG} \mid 0 \mid \mathit{POS}\\
\mathit{LESS_THAN_MINUS_ONE} &\rightarrow -8 \mid -7 \mid \ldots \mid -2\\
\mathit{NEG} &\rightarrow \mathit{LESS_THAN_MINUS_ONE} \mid -1\\
\mathit{POS} &\rightarrow 1 \mid \mathit{GREATER_THAN_ONE}\\
\mathit{GREATER_THAN_ONE} &\rightarrow 2 \mid 3 \mid \ldots \mid 8\\
\mathit{BOOL_LAMBDA_UNARY} &\rightarrow \mathit{BOOL_UNARY_OPERATOR}\ \mathit{NUM}\\
\mathit{BOOL_LAMBDA_UNARY} &\rightarrow \mathit{MOD}\\
\mathit{BOOL_UNARY_OPERATOR} &\rightarrow == \mid < \mid >\\
\mathit{MOD} &\rightarrow \%\ \mathit{GREATER_THAN_ONE} == 0\\
\mathit{NUM_LAMBDA_UNARY} &\rightarrow *\ \mathit{MUL_NUM}\\
\mathit{NUM_LAMBDA_UNARY} &\rightarrow /\ \mathit{DIV_NUM}\\
\mathit{NUM_LAMBDA_UNARY} &\rightarrow +\ \mathit{POS}\\
\mathit{NUM_LAMBDA_UNARY} &\rightarrow -\ \mathit{POS} \mid \%\ \mathit{GREATER_THAN_ONE}\\
\mathit{NUM_LAMBDA_BINARY} &\rightarrow * \mid + \mid - \mid \mathit{max} \mid \mathit{min}\\
\mathit{MUL_NUM} &\rightarrow \mathit{NEG} \mid 0 \mid \mathit{GREATER_THAN_ONE}\\
\mathit{DIV_NUM} &\rightarrow \mathit{LESS_THAN_MINUS_ONE}\\
\mathit{DIV_NUM} &\rightarrow \mathit{GREATER_THAN_ONE}
\end{aligned}
$$

Fig. 4 The grammar used in the extended FlexCoder. In this grammar, we introduce a new function to be used in the function compositions: the higher order *zip_with*. This function accepts two input lists and returns a new one whose *i*-th element is the result of the given operator applied to the input lists' *i*-th elements. The *search* function is taken out from the set of used functions, as the function only searches for elements in the [– 8, 8] range, which does not scale well with the extension of input, intermediate, and output integer ranges. The *copy* function is not present in our DSL, as it is used solely in the synthesis process, and copies branches of function compositions built from the functions in the DSL. The *count* function is renamed to *length* to better capture its functionality. The remaining differences between this and Fig. 3 are only minor optimizations, such as disallowing the lambda expression (/(– 1)), as it is equivalent to (*(– 1))

a given number of *inputs*, and contain a given number of *forks*. This also determines the number of *branches* and the number of *copies*. The number of branches is always one more than the number of forks, because each fork (i.e., each *zip_with*) connects exactly two branches, and all branches have to be connected to the tree eventually. The number of copies is the difference between the number of inputs (i.e., the initial branches), and the number of branches (i.e., the total number of branches required in the composition).

The generation process is based on dynamic programming. First, each possible composition of length one is generated.[1] After generating some compositions of length $(n-1)$, the compositions of length n are generated by sampling a function which serves the role of the root function, and choosing subcompositions from the previous levels that are used as children of the root function, where the length of the resulting composition must be equal to n.

The number of branches (and thus, forks) in the compositions has to be less than or equal to the number requested by the user on the intermediate levels. On the last level (i.e., for the final compositions, when the length of the currently generated compositions match the length desired by the user), the number of branches has to be strictly equal to the

[1] The length of a function composition is the number of functions it contains.

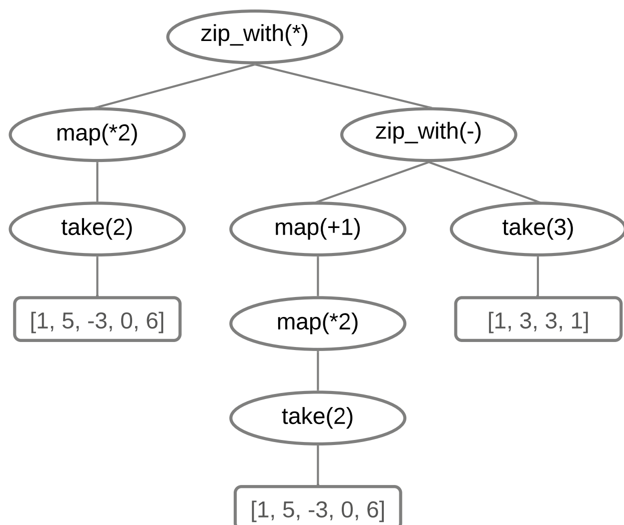


Fig. 5 A tree-structured function composition from the DSL of the extended FlexCoder. In the synthesis process, the two *map*($*$ 2, *take*(3, [1, 5, −3, 0, 6])) subcompositions have to be built only once, and then, the *copy* function can be used to copy this subcomposition once. The composition has two (unique) inputs, three branches, two forks, and one copy

requested number, as the generated compositions have to satisfy the user-provided parameters.

If the number of inputs is less than the number of branches, the system then enumerates all the possible structures of compositions that conform to the parameters of the generation (e.g., the number of inputs and branches), and generates compositions with the needed number of copies, using the previously generated copyless compositions as building blocks.

This is done as follows. First, the system selects an extensible copyless composition from the previously generated compositions. A composition is extensible if its number of branches and number of contained functions allow it to be extended further by copies to create a composition with the length, number of branches, and number of inputs specified by the user. Then, the system determines its leaves where subcompositions (or inputs, as they can also be copied) and their copies should be placed. The subcompositions are also selected from the set of previously generated compositions. A composition can contain multiple copied branches.

The number of compositions of each length is controlled by a hyperparameter called *branching factor*. This determines the number of new compositions that are created from previously generated compositions by adding a new function as the root. It is the exponential base of the function providing the number of generated compositions of length *n*.

As the functions in the grammar have a different number of possible parameterizations, we have to ensure that each type of function occurs approximately the same number of times in the generated dataset. To tackle this problem, the compositions are generated by a two-step sampling process from the 151 possible functions in FlexCoder, and from the 138 possible functions in the extended FlexCoder.

In the first step, a function type is chosen uniformly at random (e.g. *map*), then in the second step, a fully parameterized function is chosen uniformly at random out of the possible parameterizations of the picked function type (e.g., *map(* 2, array)*). A notable exception to this in the extended FlexCoder is the generation of compositions of length one: each possible function is enumerated in that case to supply an exhaustive base for the compositions.

We use several filters on the generated functions to remove or fix redundant functions and suboptimal parameterizations. This is needed to create a well-trainable problem for the neural network and to reduce the ambiguity of the dataset. The optimizations happen during the generation of the compositions.

We distinguish between *structural optimizations* that can be optimized regardless of the input(s) of the composition, and *example-based optimizations* that can only be done for the exact input(s) provided for the composition at hand. Unless otherwise stated, the showcased optimization techniques are used in both FlexCoder and the extended FlexCoder in the same way.

Structural Optimizations

The first filter merges functions inside the composition, if possible:

- map(+ 1, map(+ 2, [1, 2])) → map(+ 3, [1, 2])
- take(3, take(4, [– 4, 2, 9, 3, 0])) → take(3, [– 4, 2, 9, 3, 0])
- drop(1, drop(2, [4, 5, – 4, – 1, 0])) → drop(3, [4, 5, – 4, – 1, 0]).

The next two filters are only present in the extended FlexCoder.

The second filter switches the order of *take*, *drop*, and *map*, if needed:

- drop(3, map($*$ 2, [2, – 3, 5, 10])) → map($*$ 2, drop(3, [2, – 3, 5, 10])).

This is a minor efficiency optimization in evaluating the compositions. The functions do not have to be next to each other in this case, all cases are covered where this alteration can mean an improvement.

The third filter eliminates redundant applications of *sort* or *reverse*

- reverse(reverse(take(3, [3, 0, 5]))) → take(3, [3, 0, 5])

– sort(map(∗ 3, sort([4, – 3, 4]))) → map(∗ 3, sort([4, – 3, 4])).

This technique can also be applied to functions that are not adjacent, provided that the intermediate functions are not order-altering.

Example-Based Optimizations

The first filter optimizes the parameters of functions

– filter(> 2, [1, 6, 7]) → filter(> 5, [1, 6, 7]).

In the presented example, (> *i*) ($i \in \{2, 3, 4, 5\}$) would all mean semantically different compositions, but these all produce the same output for the provided input list. Having all of these in the generated dataset would induce ambiguity.

The second filter removes the identity functions on the concrete input–output pairs:

– sum(filter(> 5, [6, 7, 8])) → sum([6, 7, 8]).

Identity functions cannot surface during *structural optimizations*, as these are prohibited by the grammars. An example for this is *map*(+0, [2, 3, 5]).

The third filter handles compositions resulting in empty lists:

– filter(< 1, [2, 3, 4]) → [ ]
– drop(4, [1, 2, 3]) → [ ].

In this case, FlexCoder simply discards the example. The extended FlexCoder, however, traverses the tree-structured function composition in a postorder way and optimizes the parameters of functions that return an empty list for the concrete inputs provided. Consequently, this optimization is not an equivalent transformation.

The fourth filter removes the examples which contain integers outside the allowed range of $[-1024, 1024]$:

– map(∗ 8, [1, 2, 255]) → [8, 16, 2040].

Beam Search

As previously mentioned, we formulate the problem of program synthesis as searching for an optimal function composition. We build compositions applying one *synthesis function* at a time. It is important to remark that the proposed *copy* function in the extended FlexCoder makes it possible to incorporate several functions at once into the composition being built.

The main difference between FlexCoder and its extension is the states used in the search algorithm, and consequently, the predictions of the neural network. FlexCoder's synthesized programs are sequential; thus, there is no need to represent partial programs. The states in this case are single sequential function compositions.

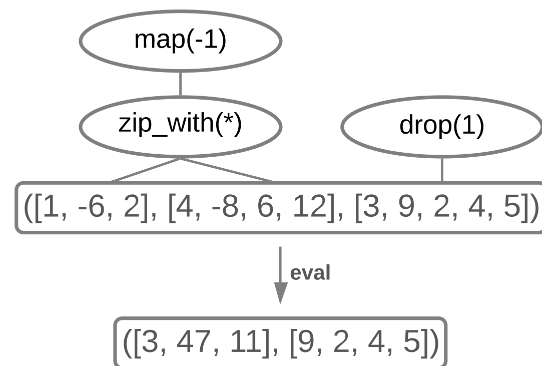


Fig. 6 This figure shows how *state tuples* can facilitate the representation of unfinished programs in the form of function compositions. During the synthesis process, only the evaluated form is stored, as the search algorithm keeps track of the functions applied at each step. Therefore, the figure merely serves as a visualization of the partial programs (branches) the *state tuples* represent

In the extended FlexCoder, we use *state tuples* to represent the states. State tuples contain branches of the function composition which are not yet merged together, so a state tuple represents a whole unmerged composition. Such a composition cannot be evaluated as a whole, but their branches can be and are evaluated. In fact, the state tuple contains only the results of the evaluation of these branches, because beam search keeps track of the concrete functions each branch is composed of (Fig. 6). A state tuple becomes a finished program when all the branches are merged, that is, in contains a single branch which is the tree of the whole function composition.

The branches in a state tuple are built up separately from more and more functions. The *copy* function can be used to create new branches by copying existing ones in the state tuple. The maximum length of the state tuple is 3, so we can have at most 3 branches at once.

The beam search (Fig. 7) is implemented based on Complete Anytime Beam Search (CAB), which involves the introduction of a time limit in the search.

The algorithm builds a directed tree of node objects in both systems. Each node has three fields: a function, the result of that function applied to the output of its parent node, and its rank. The parent node is also stored to recursively gather the solution once it has been found. In the extended FlexCoder, an indices field is added to indicate which branches in the *state tuple* the chosen transformation should be applied on.

For each node of the tree, the neural network uses the current state of the program input stored in the parent node

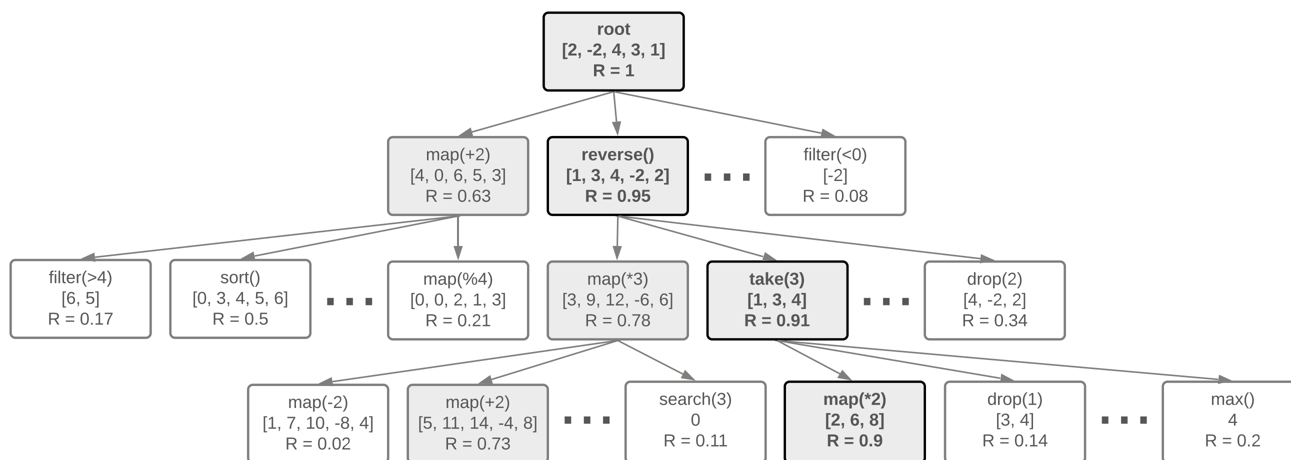


Fig. 7 A successful beam search in FlexCoder, in which we are looking for a function composition that produces the output sequence [2, 6, 8] when given the input $[2, -2, 4, 3, 1]$. Each node has three fields: a function, the result of that function applied to the output of its parent node, and its rank. The gray rectangles are the nodes considered; these are selected based on their rank marked by R. The highlighted and bolded ones show the result $\text{take}(3, \text{map}(*\,2, \text{reverse}(\text{input})))$

and the program's expected output to determine the ranks of the next possible functions of the synthesized composition. If there are multiple input–output examples, we pass them in a batch to the network, then we take the geometric mean of the predictions.

On each level of the search algorithm, we sort all possible parameterized functions in descending order based on their rank.

After generating all the child nodes, we keep the first $v_i \in \mathbb{N}$ $(i \in 1..\vartheta)$ nodes (where v_i denotes the beam size on the current level and ϑ is the depth limit) from the sorted list of the nodes and fill their result fields by evaluating them. We repeat this step until either a solution is found or the algorithm reaches the iteration limit (or optionally the time limit) provided. If a solution is found, the parent pointers are followed recursively until the root of the tree is reached to get the synthesized composition. When the depth limit ϑ is reached without finding a solution, each v_i value is doubled and the search is restarted. Since the network is called multiple times during the search, a caching method is used to save the ranks for every unique input–*output pair to speed up the process. This is possible, because the network is a pure function.

Algorithm 1: Beam search with a time limit or an iteration limit.

```
found = false;
while iteration limit or time limit is not reached and not found
 do
    depth = 0;
    nodes = [root];
    while depth < ϑ and not found do
        // beam sizes are predefined for each
           depth
        // we double the beam sizes in each
           iteration of the outer loop
        beam size = ν[depth]*2^i;
        // filter removes logically incorrect
           function calls
        // process assigns a rank to every node
        // take_best sorts output_nodes and takes
           the first beam size nodes from it
        output_nodes = process(filter(nodes));
        found = check_solution(output_nodes);
        nodes = take_best(beam size, output_nodes);
        depth += 1;
    end
end
```

Algorithm 1 uses previously optimized beam sizes for each depth. We optimized the beam sizes based on experimental runs on the validation set: we approximated the minimum beam size $\in \{v_1, v_2, \ldots, v_n\}$ on each level that contains the next function of the solving composition. This gives a higher chance to find the solution in the first or early iterations. We chose the beam size or each level that included the original solution 90% of the time during the benchmarking process. This seems to be an ideal trade-off between accuracy and speed. In the extended FlexCoder, a lower limit of 150 has also been set for beam sizes, and

the optimized values were used when they were larger than 150.

The first step of the algorithm is to remove programs that violate the range constraints of the evaluated output list mentioned in "Example Generation and Grammar" or a length constraint. The length constraint in the case of list outputs ensures that we only keep nodes where the state contains lists (or a singular list in FlexCoder) that have as many as or more elements than the original output list, as the used DSLs do not make it possible to extend lists. After this filtering step, a rank is assigned to all the remaining programs by the neural network.

The programs are also executed to check whether they satisfy the solution criteria in each step. If a solution is found, the algorithm stops and returns it. Otherwise, the first *beam_size* states with the highest rank are selected and are further transformed with a new *synthesis function* on the next iteration of the inner loop. If the inner loop finishes, the beam sizes are doubled, but the previously computed ranks are not computed again due to the caching method mentioned previously.

Neural Network

The architecture of FlexCoder's neural network is shown in Fig. 8. The input to the network is one input–output example of a single program. The outputs are 6 vectors that contain the ranks for each function, parameter, and lambda function. The extended FlexCoder also uses an output head for the indices of the state tuple the predicted function should be applied to. The rank of a parameterized function is determined by the geometric mean of the ranks of its components.

In FlexCoder, we define F as the set of *synthesis functions*, every element of which is a tuple (f_{class}, f_{arg}), where f_{class} is the function name and f_{arg} is the list of its arguments. Using this definition, the rank of a function is determined using the formula

$$R(f) = \sqrt[n+1]{R(f_{\text{class}}) * \prod_{i=1}^{n} R(f_{\text{arg}_i})}, \quad (1)$$

where $n \in \mathbb{N}$ denotes the number of parameters.

In the extended FlexCoder, the indices of the state tuple to which the function should be applied to, denoted by f_{ind} also have to be taken into consideration. Consequently, the formula is modified as follows, using the notation introduced previously:

$$R(f) = \sqrt[n+k+1]{R(f_{\text{class}}) * \prod_{i=1}^{n} R(f_{\text{arg}_i}) * \prod_{i=1}^{k} R(f_{\text{ind}_i})}, \quad (2)$$

where k denotes the number of lists in the state tuple.

Architecture

The input to the networks is a state and the expected output. In the case of FlexCoder, the state is a sequential composition's evaluated form, which is a single list. The state in the extended FlexCoder is a *state tuple* with maximal length 3. This is a tuple containing the evaluated forms of partial programs in the form of tree-structured compositions (branches).

Inputs and outputs are passed separately to two blocks of recurrent layers, which makes using variable-size input possible. These blocks each consist of two layers of GRU cells, each containing 256 neurons in both systems.

In FlexCoder, the GRU representations of the input and the output are concatenated and then given to a dense block consisting of seven layers with SELU [16] activation function, and 128, 256, 512, 1024, 512, 256, and 128 neurons in order.

The extended FlexCoder creates a GRU representation for each list in the *state tuple* separately, and then concatenates these and the output GRU representation into a single tensor. The main dense block this tensor is given to contains nine layers with SELU activation function, and 1024, 1024, 1024, 1024, 1024, 1024, 512, 256, and 128 neurons in order.

With SELU activations, we experienced a faster convergence while training the networks, due to the internal normalization these functions provide

$$\text{SELU}(x) = \lambda \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{if } x \leq 0. \end{cases} \quad (3)$$

After this block, we have an output layer predicting the probabilities of the possible next functions using sigmoid activation for each function in FlexCoder, and softmax activation in the extended FlexCoder. This layer is also fed into five smaller dense blocks. Each of these contains five layers using the SELU activation function with 128, 256, 512, 256, 128 neurons in FlexCoder, and six layers of 512, 512, 512, 512, 256, 128 neurons in the extended FlexCoder.

These smaller dense blocks produce the remaining five outputs of the model in FlexCoder, and six outputs in the extended FlexCoder. They are vectors, each representing the probabilities of parameters associated with the next parameterized function of the composition. These vectors are corresponding to (1) the bool lambda operator, (2) the numeric lambda operator, (3) the numeric argument of the bool lambda function, (4) the numeric argument of the numeric lambda function, and (5) the parameter for non-higher order functions with only one numeric argument, e.g., the value used in *take*. The extended FlexCoder uses an additional output for (6) the indices of the state tuple the predicted function should be applied to. We use the sigmoid activation function for each entry of all output vectors in FlexCoder,

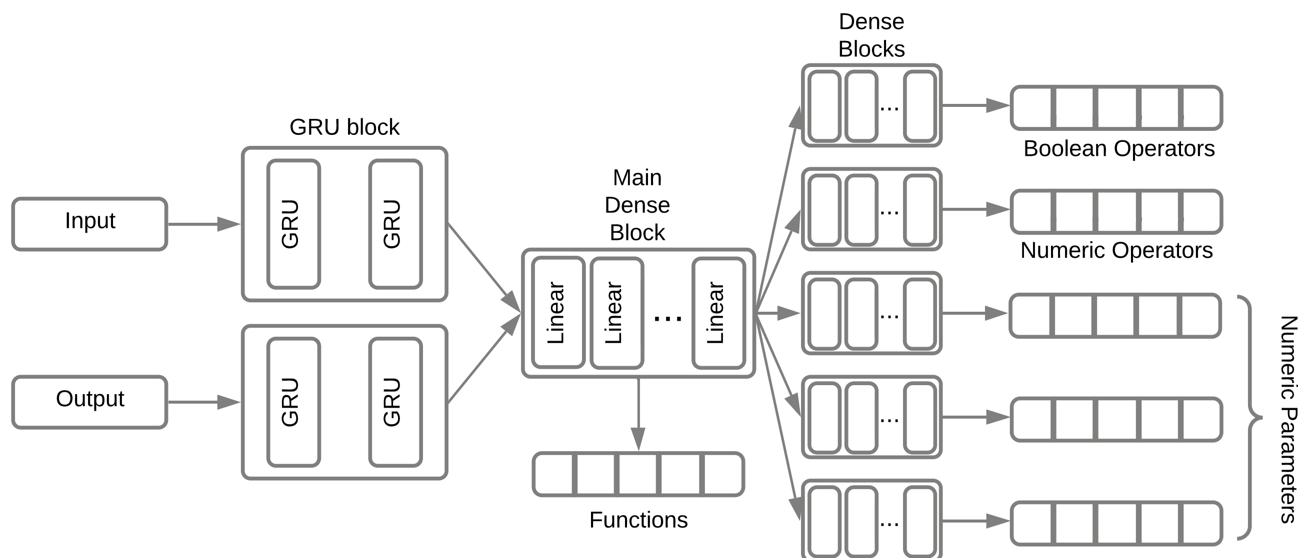


Fig. 8 The general architecture of the neural network used in FlexCoder. The compositions in this system are sequential; thus, they only have a single input. Each input–output example is passed separately to the GRU block of the network, which generates a representation that is passed through the dense block and then splits into six parts, five of which pass through another dense block. In the extended FlexCoder, the compositions can be tree-structured, allowing them to have multiple inputs. The input to the network in this case is a state tuple containing the evaluated form of multiple partial programs (branches of the tree). The architecture used in the extended FlexCoder introduces an additional output head compared to the neural network used in FlexCoder. This specifies the indices of the state tuple the predicted parameterized function is to be applied to

whereas in the extended FlexCoder, the softmax activation is used for all output vectors except the one responsible for the indices, as its task is a multilabel classification. The smallest output vector has three elements, whereas the largest one has 17. The network's loss (L) is the sum of all of the output components' loss values marked with L_i, each denoting the corresponding binary cross-entropy or cross-entropy loss functions depending on the exact output head.

Thus, the loss is calculated as

$$L(Y, \hat{Y}) = \sum_{i=1}^{N} L_i(Y_i, \hat{Y}_i), \tag{4}$$

where N is 6 in FlexCoder and 7 in the extended FlexCoder.

Training

Before training, we break down the compositions into functions and turn each parameterized function into six or seven separate one-hot vectors to obtain a single label used for training the networks introduced.

Out of the generated examples, 98% is used as the training set, and the remaining 2% serves the role of the validation set. The test sets are generated on a per-experiment basis.

We use the Adam optimization algorithm [15] with the default hyperparameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. We trained the neural network on a computer with an Intel i5-7600k processor and an NVIDIA GTX 1070 GPU using a standard early stopping method with a patience of five. We trained the networks for a maximum of 30 epochs.

Experiments

The experiments for FlexCoder were run on a c2-standard-16 (Intel Cascade Lake) virtual machine on the Google Cloud Platform with 16 vCores, 64 GB RAM, and no GPU. In the experiments for the extended FlexCoder, we used a PC with an AMD Ryzen 7 3700× processor, 16 GB RAM, and no GPU. For these experiments, the neural network of FlexCoder was trained on compositions of 7 functions, and the length of the input array was between 15 and 20. We provided 1 input–output example for each program in the training process. In all of the experiments, we created the test datasets by sampling the original program space uniformly at random, with a sample size of 1000. The neural network of the extended FlexCoder was trained on compositions with a length of 6, while the other parameters remained the same.

Filtering the Datasets

Filtering the datasets using the method described in "Example Generation and Grammar" made the problem more learnable for the neural network, resulting in improvements

in the case of each output head of the network of FlexCoder (Fig. 9). In the extended FlexCoder, we kept these optimizations and introduced new ones, as well ("Example Generation and Grammar").

This filtering is used on all datasets in all experiments.

Different Recurrent Layers

In this section, we compare the effect of different recurrent layers on accuracy in FlexCoder. We ran experiments with LSTM [13], bidirectional LSTM [20], GRU, and bidirectional GRU cells in the first layer of the network.

In the first experiment, we looked at the accuracy of the different layers as the composition length increased (Fig. 10). The bidirectional layers proved to be suboptimal, as these—somewhat surprisingly—did not make the system more accurate for longer function compositions, but training and testing both took considerably more time. In the case of both bidirectional models and the LSTM model, we used a hidden state consisting of 200 neurons. For the GRU model, we increased this amount to 256, as the GRU cells require less computation. Despite the fact that the bidirectional LSTM achieves better accuracy for shorter compositions, its accuracy falls below regular LSTM and GRU cells as the composition length increases. It is also the slowest of the three in terms of execution time. The bidirectional GRU model is the second slowest, and its accuracy is the worst of the layers tested for longer compositions. Between the regular LSTM and GRU cells, GRU is preferred as it performs well in terms of both execution time and accuracy.

In the second experiment, we analyzed how FlexCoder scales based on the length of the input–output examples in terms of accuracy. We generated program input–output vectors with a length of 10–50 in increments of five for testing purposes. Figure 11 shows that FlexCoder with GRU was capable of generalizing well to longer inputs.

Similarly to the first experiment, GRU was the most accurate while being the best in terms of execution time. Both bidirectional models performed similarly, but the bidirectional LSTM was markedly slower. The GRU model performed consistently better than the LSTM-based network on longer example lengths.

Based on the results of these two experiments, we elected to use GRU as the recurrent layer of both architectures.

Accuracy and Execution Time

Tables 1 and 2 show the accuracy and the time needed to find a solution in terms of the number of input–output pairs and the composition length in FlexCoder. By increasing the number of input–output pairs the problem becomes more specific: finding a program that fits all the pairs becomes a more complex task, because the set of possible solutions narrows. Similarly, as we increase the composition length, the space of possible programs of that length also increases.

The accuracy achieved by FlexCoder is comparable to that of PCCoder with the time limit replaced by the same iteration limit as in FlexCoder. In terms of execution time, FlexCoder sometimes falls behind, but the performance of the two systems is generally similar.

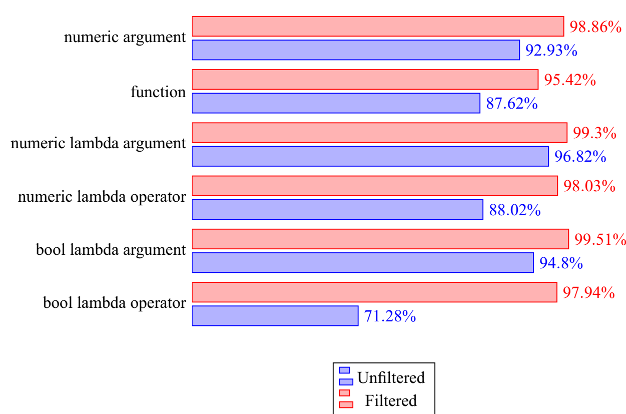


Fig. 9 The improvements in the output accuracies of the neural network used in FlexCoder after applying filtering to the training data. After filtering, the problem is more learnable. The results were measured during the training of the GRU model. Each row shows the final validation accuracy of the network's outputs at the end of training

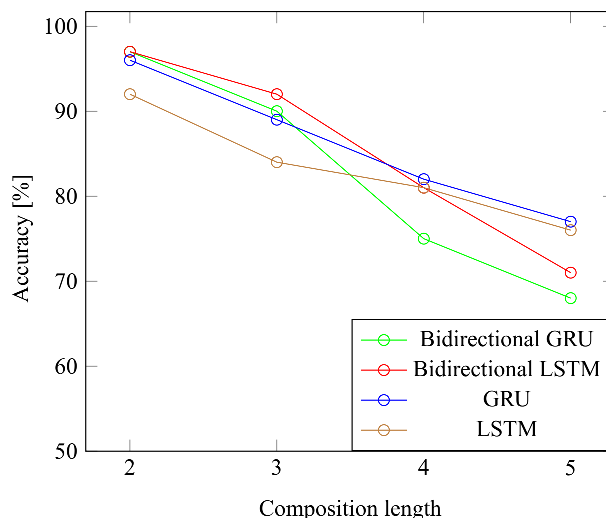


Fig. 10 The accuracy of different recurrent layers in FlexCoder, in relation to the number of functions used in the composition. Although the bidirectional LSTM performs best with shorter compositions, its performance decreases greatly as the composition length increases, and it is also the slowest. Considering speed and accuracy the GRU model is the most favorable. The extended FlexCoder also uses GRU cells in its recurrent layers

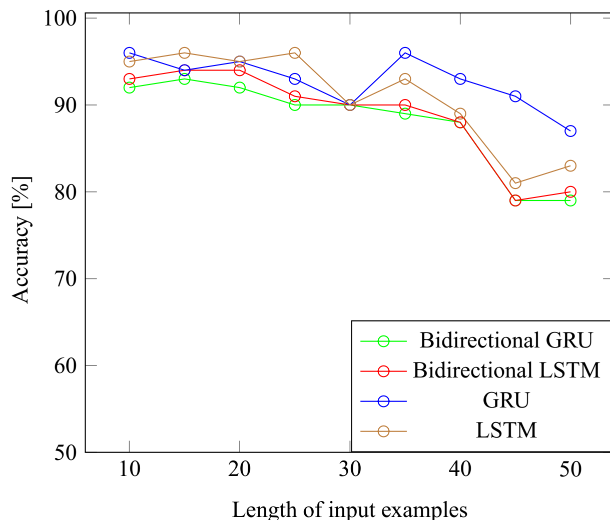


Fig. 11 The accuracy of FlexCoder using four different recurrent layers as a function of the length of the input. The datasets contained compositions of length five. GRU generalizes best to longer input lengths

Comparison of FlexCoder with PCCoder

We compare both FlexCoder and the extended FlexCoder to PCCoder, which has outperformed DeepCoder by orders of magnitude [24].

Our approach to program synthesis is quite different from the approach of PCCoder (see Sects. 1 and 3.1). We synthesize a function composition, whereas they synthesize a sequence of statements. The expressiveness of the grammars is also different: On one hand, the grammar used in FlexCoder is missing some functions like `ZipWith` or `Scanl1`. On the other hand, this grammar is much more expressive in terms of parameter values.

FlexCoder's grammar is capable of expressing 151 different functions, 130 of which can be anywhere in the sequence and 21 can only appear as the outermost function as these return a scalar value. The DSL used by PCCoder can express 105 different functions. The number of possible programs with a length of five is about 43.13 billion for FlexCoder and about 12.76 billion for PCCoder, resulting in our program space being 3.38 times larger when considering programs with a length of five.

To extend FlexCoder to tree-structured function compositions, we redesigned it to work with *state tuples* which contain branches of the composition, and introduced the *copy* and *zip_with* functions to the grammar of the extended FlexCoder. The *search* function was taken out (see "Example Generation and Grammar"). Because of these changes, the extended FlexCoder's grammar expresses 138 different parameterized functions, with 4 of them to be only used as the outermost function of a composition.

Table 1 Relation between composition length, the number of input–output pairs, and the accuracy

#I/O pairs Comp. length	1 (%)	2 (%)	3 (%)	4 (%)	5 (%)
2	97	97	97	96	97
3	99	94	92	95	93
4	97	89	86	83	85
5	96	88	81	79	78

Increasing composition length or the number of pairs almost always decreases the accuracy

Table 2 Relation between composition length, the number of input–output pairs, and the execution time in seconds

#I/O pairs Comp. length	1 (s)	2 (s)	3 (s)	4 (s)	5 (s)
2	211	255	264	277	274
3	524	1082	1239	1141	1291
4	1224	2900	3250	3923	3620
5	1520	3445	4986	5537	5530

Increasing either composition length or the number of inputs almost always also increases execution time

To make fair comparisons despite these differences, we run experiments where both FlexCoder and PCCoder run on their own dataset, and we also compare them by running them on the dataset of the other system using programs of length five in both cases. The extended FlexCoder is tested against PCCoder on compositions with varying number of forks (i.e., varying number of *zip_with* functions in the ground-truth composition). In each of the experiments, the systems were trained on their own datasets.

In the first experiment, we examine what we consider a crucial aspect of any program synthesis tool: how well it generalizes with respect to the length of the input–output lists. We trained both PCCoder and FlexCoder on input–output vectors of length 15 to 20 with program length 7. For PCCoder, we set the maximum vector length to 50. We tested the systems on input–output vectors with a length of 10–50 in increments of 5, having 5 input–output examples per program, each on their own dataset. The results are shown in Fig. 12.

In the second experiment, we compare the accuracy of FlexCoder and PCCoder in a less realistic scenario when PCCoder performs best: on the same input lengths, the systems have been trained on. In this experiment, PCCoder does not have to generalize to different input lengths. We compare the systems both on their own dataset and on the datasets of each other in Fig. 13.

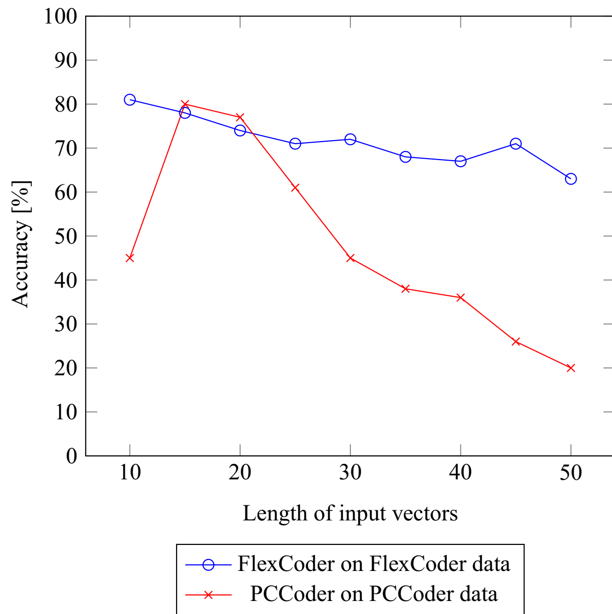


Fig. 12 The accuracy of FlexCoder and PCCoder in relation to the length of the input. Both systems were trained on input–output lists of length 15–20 with composition length 7. For PCCoder, we set the maximum list length to 50. We tested the systems on input–output lists with a length of 10–50 in increments of 5, having 5 input–output examples per program, each on their own dataset

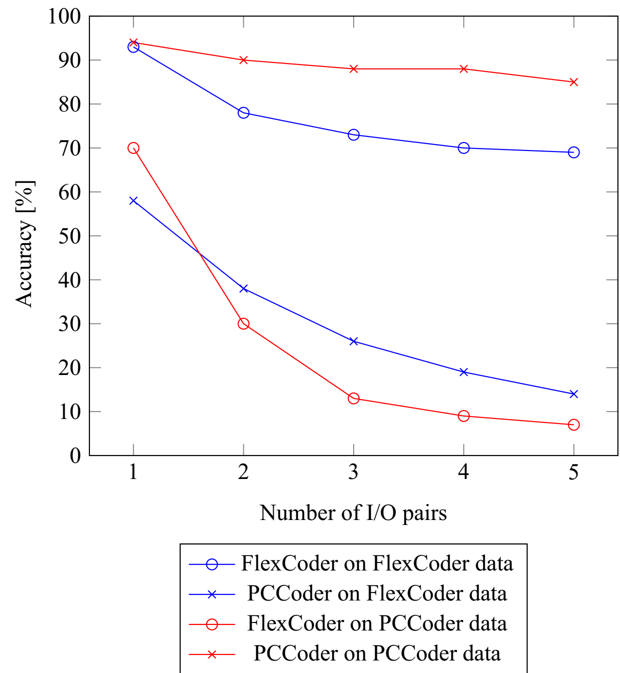


Fig. 13 The accuracy of FlexCoder and PCCoder on their own and the other's dataset. The parameters are the same as in the first experiment, except that no generalization over the input length is needed: the length of the input lists is the same as for training. The number of input–output pairs range from 1 to 5

FlexCoder defines an iteration limit or a time limit for the search algorithm, while the search used by PCCoder only has a time limit version. To make the experiment fair, we also used the time limit, and chose a timeout of 60 s like PCCoder. The use of a time limit in our search algorithm makes our system's accuracy go down by a couple of percent compared to Fig. 11, so FlexCoder could perform even better with the original iteration limit. The parameters in this experiment are the same as in the first experiment, except for the length of the input lists which is the same as for training, and the number of input–output examples which range from 1 to 5.

Comparison of the extended FlexCoder with PCCoder

The goal of these comparisons is to measure how well the extended FlexCoder and PCCoder can produce compositions which are really tree-like, that is, they have a relatively large number of forks. PCCoder is capable of producing such programs using the *zip_with* function with two previously defined variables.

For these comparisons, we generated new datasets using our grammar with varying number of *zip_with* functions in the program (i.e., varying number of forks in the composition tree). We measured how the accuracy depends on the number of forks by fixing the number of inputs to the program to 1, and varying the number of forks in the tree from 0 to 3 (Fig. 14). The dependence of the accuracy on the number of inputs to a single program was measured by fixing the number of forks to 3 and varying the number of inputs from 1 to 3 (Fig. 15).

PCCoder was retrained for every number of inputs tested, whereas the extended FlexCoder was only trained once on a dataset of compositions with two inputs and three forks (two *copies*, and three *zip_withs*), and a length of 6. On average, these compositions were broken down to 7.5 samples for the neural network. To make a fair comparison, we trained PCCoder with programs of length 8. The length of input lists was 15–20 in both cases; thus, PCCoder does not have to generalize to input lengths it was not trained on.

Discussion

FlexCoder generalized well with respect to the input length in contrast to PCCoder. PCCoder only excelled on input lengths it was trained on. PCCoder has an upper limit on the length of the input–output vectors; we set the maximum vector length of PCCoder to 50 to accommodate this experiment. Applying PCCoder to longer inputs would require retraining with a larger maximum vector length.

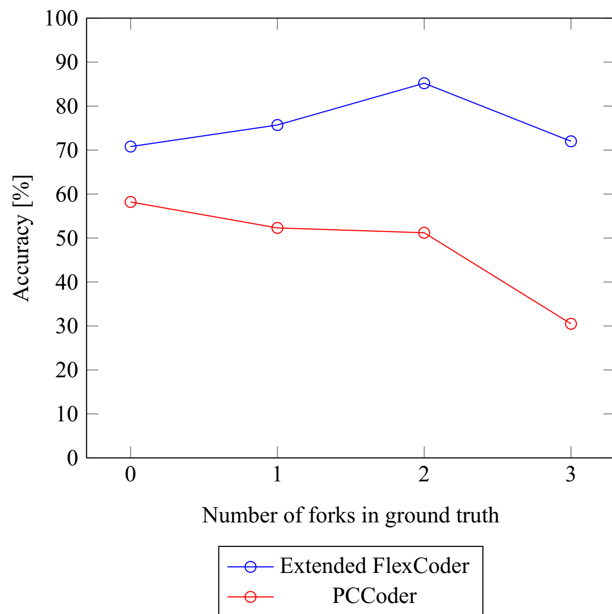


Fig. 14 The accuracy of the extended FlexCoder and PCCoder on programs which take one input as the number of forks varies from 0 to 3. The figure shows that PCCoder did not manage to perform well on compositions with many forks. In contrast, the extended FlexCoder handles *copies* and *zip_withs* much better, and solved significantly more problems

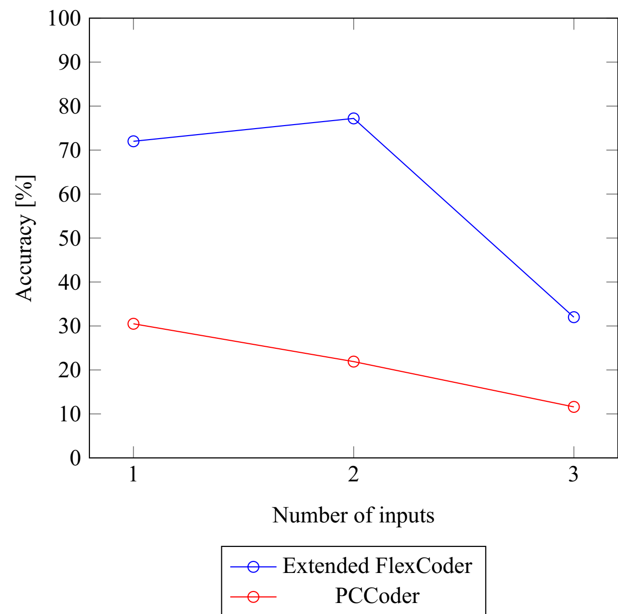


Fig. 15 The accuracy of the extended FlexCoder and PCCoder on programs whose ground-truth compositions have three forks, where the number of inputs per program varies from 1 to 3. PCCoder could not synthesize programs with multiple inputs well, even though it was retrained for each different number of inputs. Extended FlexCoder performed best on the number of inputs it was trained on, adapted well to less inputs, but did not manage to generalize well to more inputs

We also compared the two systems when the inputs are the same length for testing as for training; thus, PCCoder does not need the generalize to different input lengths. In this easier and less realistic scenario, both systems beat the other on their own dataset. Also, both systems perform notably worse on the DSL of the other system.

The comparison between the extended FlexCoder and PCCoder on datasets where the ground-truth programs contained multiple forks has shown that the extended FlexCoder could synthesize function compositions with multiple forks very well in the case of one or two inputs per program, but less well for three inputs.

In contrast, the performance of PCCoder worsened as the number of forks or the number of inputs of the programs increased. PCCoder had to be retrained separately for the different number of inputs.

To gain further insight into the performance of PCCoder, we analyzed the programs it generates. We found that it generates mostly linear programs without too many *zip_with*s. The average number of *zip_with*s in their one input, two input, and three input datasets were 0.65, 0.69, and 0.82, respectively. The extended FlexCoder was trained on programs which were generated to be more complex with three *zip_with*s.

PCCoder did not have to generalize to different input lengths in the comparison between the extended FlexCoder and PCCoder. The difference in accuracy would be much greater in favor of the extended FlexCoder in that more realistic scenario.

Conclusion

The DSL of DeepCoder is limited in terms of expressivity as stated by the authors themselves in their seminal DeepCoder paper. The main motivation of our paper is to extend it and move towards real-world applications.

We presented FlexCoder, a program synthesis system that generalizes well to different input lengths, separates lambda operators from their parameters, and increases the range of integers in the input–output pairs.

To increase the system's expressivity and allow it to take multiple inputs, we proposed an extension that can synthesize tree-structured compositions with multiple forks. This was achieved by redesigning the system to work with *state tuples* which can contain multiple unevaluated branches of the composition, and by introducing two new functions compared to FlexCoder, *copy* and *zip_with*.

In future work, the expressivity of the system could be increased further if any subcomposition that could be evaluated to an integer value (that is, compositions that have the *max*, *min*, *sum* or the *length* function as their root function)

could be used as the numerical parameter of the functions. This could, for example, allow compositions such as take(max(arr), arr).

Further experimenting with our neural network might also include changing the architecture of the used neural network, and integrating an attention-based architecture, such as variants of the Transformer [22]. Such architectures may better capture the key features of the input and output states; thus, they could function more effectively as the encoder layer of the neural network used.

FlexCoder proved to be accurate and efficient even when generalizing to input vectors with a length of 50, with much wider parameter ranges than current systems. Its extension showed promising results when synthesizing function compositions with multiple forks, greatly surpassing the results of PCCoder. We hope that the proposed systems represent a step towards the wide application of program synthesis in real-world scenarios.

Acknowledgements The authors would like to express their gratitude to Zsolt Borsi, Tibor Gregorics, and Teréz A. Várkonyi for their valuable guidance. The materials were produced as part of EFOP—3.6.3-VEKOP-16-2017-00001: Talent Management in Autonomous Vehicle Control Technologies. The Project is supported by the Hungarian Government and co-financed by the European Social Fund.

Funding Open access funding provided by Eötvös Loránd University.

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Alur R, Bodik R, Juniwal G, Martin MM, Raghothaman M, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A. Syntax-guided synthesis. IEEE, 2013.
2. Balog M, Gaunt AL, Brockschmidt M, Nowozin S, Tarlow D. Deepcoder: learning to write programs. 2016. arXiv preprint arXiv:1611.01989
3. Bird S, Klein E, Loper E. Natural language processing with Python: analyzing text with the natural language toolkit. " O'Reilly Media, Inc." 2009.
4. Chomsky N, Schützenberger MP. The algebraic theory of context-free languages. In: Studies in Logic and the Foundations of Mathematics, vol 26, Elsevier, 1959; pp. 118–61
5. Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. 2014. arXiv preprint arXiv:1406.1078
6. Desai A, Gulwani S, Hingorani V, Jain N, Karkare A, Marron M, Roy S. Program synthesis using natural language. In: Proceedings of the 38th International Conference on Software Engineering, 2016; pp. 345–56
7. Ellis K, Nye M, Pu Y, Sosa F, Tenenbaum J, Solar-Lezama A. Write, execute, assess: Program synthesis with a repl. 2019. arXiv preprint arXiv:1906.04604
8. Feng Y, Martins R, Bastani O, Dillig I. Program synthesis using conflict-driven learning. SIGPLAN Not. 2018;53(4):420–35. https://doi.org/10.1145/3296979.3192382.
9. Gulwani S. Programming by examples: applications, algorithms, and ambiguity resolution. In: Olivetti N, Tiwari A, editors. Automated reasoning. Cham: Springer International Publishing; 2016. p. 9–14.
10. Gulwani S. Automating string processing in spreadsheets using input-output examples, 2011.
11. Gulwani S, Polozov O, Singh R. Program synthesis. Foundations and Trends® in Programming Languages 2017;4(1-2):1–119. https://doi.org/10.1561/2500000010.
12. Gyarmathy B, Mucsányi B, Ádám Czapp, Szilágyi D, Pintér B. Flexcoder: Practical program synthesis with flexible input lengths and expressive lambda functions. In: Proceedings of the 10th International Conference on Pattern Recognition Applications and Methods - Volume 1: ICPRAM, INSTICC, SciTePress, 2021; pp. 386–95. https://doi.org/10.5220/0010237803860395
13. Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput. 1997;9(8):1735–80.
14. Kalyan A, Mohta A, Polozov O, Batra D, Jain P, Gulwani S. Neural-guided deductive search for real-time program synthesis from examples. 2018. arXiv preprint arXiv:1804.01186
15. Kingma DP, Ba J. Adam: a method for stochastic optimization. 2014. arXiv preprint arXiv:1412.6980
16. Klambauer G, Unterthiner T, Mayr A, Hochreiter S. Self-normalizing neural networks. In: Advances in neural information processing systems, 2017; pp. 971–980
17. Lee W, Heo K, Alur R, Naik M. Accelerating search-based program synthesis using learned probabilistic models. ACM SIGPLAN Notices. 2018;53(4):436–49.
18. Manna Z, Waldinger R. Knowledge and reasoning in program synthesis. Artif Intell. 1975;6(2):175–208.
19. Parisotto E, rahman Mohamed A, Singh R, Li L, Zhou D, Kohli P. Neuro-symbolic program synthesis. 2016. 1611.01855
20. Schuster M, Paliwal KK. Bidirectional recurrent neural networks. IEEE Trans Signal Process. 1997;45(11):2673–81.
21. Shapiro EY. Algorithmic program debugging. ACM distinguished dissertation, 1982.
22. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I. Attention is all you need. 2017. CoRR abs/1706.03762, arxiv:1706.03762
23. Zhang W. Search techniques. In: Handbook of data mining and knowledge discovery, 2002; pp. 169–184
24. Zohar A, Wolf L. Automatic program synthesis of long programs with a learned garbage collector. In: Advances in Neural Information Processing Systems, 2018; pp. 2094–2103

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.