Program Synthesis Through Learning the Input-Output Behavior of Commands

SIHYUNG LEE1, SEUNG YEOB NAM2, (Senior Member, IEEE), AND JIYEON KIM3

1School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, South Korea
2Department of Information and Communication Engineering, Yeungnam University, Gyeongsan 38541, South Korea
3Department of Computer Engineering, Daegu University, Gyeongsan 38453, South Korea

Corresponding author: Sihyung Lee (sihyunglee@knu.ac.kr)

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) by the Ministry of Education under Grant NRF-2021R1A6A1A03039493.

ABSTRACT
Program synthesis writes programs on behalf of humans, increasing software development productivity. Existing systems select a proper sequence of commands by assembling them in various ways and analyzing their meanings, which is accurate and fast when a detailed specification is given for each available command, including the syntax and semantics. However, preparing this specification is burdensome and must be repeated to add or modify available commands. We propose a synthesis system that requires moderate groundwork to achieve sufficient accuracy and speed. The proposed system receives the syntax of the available commands and learns their meanings independently by writing programs and observing their input-output behavior. Using the learned knowledge, the system selects a likely sequence of commands and gradually revises them to converge to a target program. We validated the system by synthesizing 1,000 integer-manipulation programs out of 245,410 possible programs. The system synthesized most of the programs within 1,000 revisions, which is 80% faster than the state-of-the-art system based on machine-learned input-output patterns. In addition, greater than 10% of the programs were synthesized within several revisions. We believe that the proposed system provides a basis for synthesis systems based on learning input-output behavior.

INDEX TERMS Input-output behavior, machine learning, neural networks, program synthesis, supervised learning.

I. INTRODUCTION
Program synthesis constructs a program as requested by an end user. It receives requirements that indicate what to do (typically input-output examples or natural languages) and transforms them into program commands that indicate how to do. If mature, program synthesis can greatly reduce the costs of software development. State-of-the-art systems can generate several lines of nontrivial code and have various present and potential applications. Spreadsheet systems automatically produce macros (e.g., macros that format phone numbers) based on input-output examples [1]. Integrated development environments (e.g., NetBeans and Visual Studio) generate graphical user interface (GUI) programs according to the end user’s choice of items, such as objects, events, and conditions [2]. Intelligent tutoring systems guide students in place of human tutors to write parts of a program where they experience difficulties [3].

Considerable research has focused on accelerating synthesis process because synthesis systems explore various programs to identify a target program, and it is expensive to consider all such possibilities. One method statically analyzes code blocks to eliminate those that cannot generate the desired outcomes [3] or to remove all but one block if multiple blocks generate the same outcome [4]. Another method builds probabilistic models that indicate which programs are more likely to appear, based on a corpus of representative programs. Using the models, the method explores programs in decreasing order of probability [5]. Overall, the existing methods speed up the process but require a detailed specification of the commands for synthesis, including the syntax (a range of commands) and semantics (their functionality and inverse functions), which takes time and effort even for experienced users.
Other methods only demand the syntax and automatically discover the semantics through machine learning. One method learns correlation between code blocks and their natural-language descriptions [6]. The descriptions must be comprehensive and longer than the corresponding codes; thus, it is a burden to prepare such training data. Furthermore, to request synthesis, an end user must describe the requirements in the same level of detail. Another method simplifies the descriptions to a few input-output examples [7]. However, the method learns the meaning of each command without regard to the context (the preceding and following commands), even though its role often depends on the context.

Considering the existing studies, we aim to design a system that does not require substantial effort in describing requirements and preparing data while achieving a performance level comparable to existing systems. Given the syntax, the proposed system learns semantics independently by composing programs and observing their outcomes for various inputs. This learning considers the context—it learns the meaning of individual commands and their combined effects. To request synthesis, an end user needs to describe a few input-output examples. The system then refers to the learned knowledge to select the appropriate commands and arrange them in proper order.

We summarize our contributions as follows.

- We propose a system that synthesizes programs at a pace equivalent to that of existing systems yet is not complicated to configure and use. Existing systems require a detailed specification of semantics or requirements [3]–[6], [8], [9] and the preparation of considerable training data [5], [6]. The proposed system generates data and learns semantics independently, and its requirements can be specified with a few input-output examples.

- We propose a syntax model and machine-learning model that distinguish the role of different commands with high accuracy and low learning cost. We also propose methods that employ the models to quickly identify the proper commands, compose a candidate program, and progressively revise the program to best match requirements.

- We evaluated the proposed system by requesting a synthesis of 1,000 integer-manipulation programs out of 245,410 possible programs. The system composed nearly 13% of the programs within 10 revisions and most programs within 1,000 revisions. We also showed that the system reduced the number of revisions by 80% compared to the state-of-the-art system based on learning input-output behavior.

The remainder of the paper is organized as follows. Section II compares the proposed system with existing systems in more detail. Next, Section III presents the architecture of the proposed system, and Section IV explains the learning methods. Section V evaluates the system in comparison with existing systems. Finally, Section VI concludes the paper with future research directions. Throughout the paper, we use the terms and symbols listed in Table 1.

**TABLE 1.** Terms and symbols (in the order they appear in the text).

| Term            | Description                                                                 |
|-----------------|------------------------------------------------------------------------------|
| Input #1        | Function to synthesize, requested by end user (requirements)                  |
| Input #2        | Commands to use for synthesis (syntax of programming language)                |
| I/O example     | Input/output example                                                         |
| e_i             | Expression i in Input #2                                                     |
| OR_i, OR_i.selection | OR node i and its selected option in tree representation of Input #2     |
| p_x             | Probability of option x being correct choice at OR node                      |
| empty, true, false | Constants for encoding empty slots and true/false values in I/O examples   |
| S_{prog}        | Set of possible programs, as described in Input #2                           |
| S_{train}, S_{params}, S_{test} | Training set, development set, and test set for machine learning |

An initial version of this paper appeared in the proceedings of Korea Computer Congress (KCC) [10]. This initial version outlined how to model a set of programs and how machine learning can help differentiate the programs using a single neuron model. Based on this initial work, we added more detailed explanations and algorithms on how to build a program model, how to learn commands, and how to guide the search of programs using learned knowledge, as shown in Sections III and IV. We also conducted an evaluation of the proposed system in comparison with baseline and existing methods, as shown in Section V.

**II. RELATED WORK**

A program-synthesis system needs to know the valid syntax of a programming language. Using this syntax, the system explores the space of possible programs and discovers the program requested by an end user. The search space is often large; thus, many previous studies have proposed efficient methods to guide the search.

Certain studies have focused on a particular domain and adjusted the search methods to the domain. Mefteh et al. [2] generated GUI codes by having an end user describe GUI-specific items, such as a list of events, conditions that trigger the events, and corresponding actions for the events. Each item is then mapped to predefined codes. FlashFill [1] generates string-manipulation codes that fulfill input/output (I/O) examples supplied by an end user. It compares input and output strings, identifies the locations where they differ, and uses the locations as index parameters in synthesized codes. In summary, domain-specific methods quickly generate the requested codes, but they are not generally applicable to other domains. We aim to devise methods that can be adjusted to more than one domain.
Other studies propose methods applicable to multiple domains. The methods generally explore programs in the order of length (i.e., from short programs to long programs) [8]. For example, if the syntax allows two functions \( f(), g() \), the methods initially investigate \( f() \) and \( g() \), and then uses the functions to assemble \( f(g()) \) and \( g(f()) \). The studies propose various algorithms to speed up the methods. SIMPL [3] statically analyzes the assembled programs and prunes those that can never produce the requested outcomes. For example, if \( 0 < x < 10 \), then \( x/10 \) cannot become an integer; thus, it is pruned when an integer outcome is required. TRANSIT [4] keeps one program and prunes the others if they produce the same outcomes. CEGIS [9] leverages a theory solver (e.g., SAT and SMT solvers) to quickly confirm that a program fulfills the requirements. In summary, the existing methods enumerate programs in particular orders to substantially improve the search speed. However, the methods must know the semantics of the commands (e.g., the meaning of each command, its inverse function, and how each function is decomposed), which requires a significant amount of time and effort to specify for experts and nonexperts alike. We build a system that requires the syntax but not the semantics; it learns the semantics independently by writing various programs and observing their behaviors.

A few studies have adopted machine learning. AlphaCode [6] converts the requirements (in a natural language) into a program (in a programming language), similarly to a language-translation system. It prepares training data from the web (e.g., GitHub [11] and Codeforces [12]) by selecting code with well-written descriptions. The system then learns the mapping from descriptions to code, as one remembers a solution program for a problem description. To obtain good results from AlphaCode, an end user must describe requirements in detail, more than twice as long as the target program. The proposed system requires a few I/O examples, which are easier for nonexpert users to learn and describe [1].

Both EUPHONY [5] and DeepCoder [7] take different learning approaches that are similar to the proposed system. EUPHONY learns \( p(C_i | C_j) \), the probability of command \( C_i \) appearing after commands \( C_j \), based on the representative programs selected by the authors. The system then enumerates the programs in the order of the learned probabilities. The probabilities reflect the average behavior among the training data; thus, the system quickly synthesizes programs near the average, but it does not perform as well for other programs. DeepCoder learns \( p(C_i | R) \), the probability of command \( C_i \) appearing anywhere in a program, given the requirements \( R \) from an end user. Therefore, it adjusts the probabilities to a target program, even if the program is not similar to the average. In addition, the training data comprise a diverse range of programs generated from the valid syntax, rather than a few representative programs. The proposed system shares advantages of the two existing systems. It learns \( p(C_i | C_j, R) \) and assembles programs by considering (i) the relationship with subsequent commands and (ii) the requirements by an end user. It also generates a diverse range of programs to learn the probabilities. In Section V.C, we evaluate the proposed system compared to DeepCoder because it is the most relevant and convenient and produces decent results.

### III. PROGRAM SYNTHESIS SYSTEM ARCHITECTURE

#### A. OVERVIEW OF PROGRAM SYNTHESIS SYSTEM

In this subsection, we outline the components of the proposed program-synthesis system. We then describe the details of each component in the following subsections.

Fig. 1 presents the input and output of the proposed system. It receives two inputs and generates one output. Input #1 specifies the program that an end user requests the system to synthesize. Input #2 provides the system with the range of commands that the system can utilize when synthesizing a program. The system then generates a program that operates according to Input #1 using the commands in Input #2. Fig. 1 shows an example of the two inputs and corresponding output. Through Input #1, an end user requests a program that counts even numbers in input data. For Input #2, the user supplies Python commands that manipulate integer values, such as \{sum, any\}. According to these inputs, the system synthesizes the program “output = sum( 1 for i in input if i%2==0 )”.
in input if i%2==0 ).” We post a demonstration of the synthesis process at https://youtu.be/HSDnCNaT0sA.

The two inputs typically change at different rates and have different authors. Input #1 changes per request, whereas Input #2, once specified, is reused to synthesize multiple programs. This is because the syntax of programming languages (Input #2) does not change as often as the function to synthesize (Input #1). For the same reason, Input #2 is often described by an expert. Input #1, however, is specified by various end users who aim to synthesize programs, expert and nonexpert users alike. Therefore, it is desirable to offer a convenient method to describe Input #1 to assist nonexpert users.

We explain the details of Inputs #1 and #2 in Sections III.B and III.C, respectively. In Section III.D, we describe how the system uses these inputs to synthesize a program.

B. INPUT #1: FUNCTION TO SYNTHESIZE
Input #1 of the proposed system specifies the program function to synthesize. This function is described with a set of I/O examples (i.e., inputs to the program and the corresponding execution results). For instance, Input #1 in Fig. 1 comprises two I/O examples, one in each row. The first row indicates that given input \( \{5,2,-1,3,8\} \), the program produces output \( \{2\} \), and the second row indicates that given input \( \{1,9,2,4,6\} \), the program produces output \( \{3\} \). The two I/O examples as a whole indicate that the program counts even numbers in input data. With such examples, the proposed system synthesizes a program that satisfies all examples.

There exist other methods to specify Input #1, such as logical expressions [13] and natural languages [2]. For example, the function in Fig. 1 can be described as “output={even}, where even \( \subseteq \) input and \( \forall e\in\text{even} \), \( e\mod2==0 \).” These methods can produce precise descriptions of Input #1, but the proposed system adopt I/O examples for the following reasons. Logical expressions and natural languages require a substantial amount of code to correctly understand their meanings [6], [14]. Validating whether a program fulfills such descriptions is also not trivial and can lead to errors. In contrast, a description by I/O examples is straightforward to parse, understand, and validate. Moreover, it does not require users’ prior experience and is relatively easy to learn [1], whereas logical expressions require more experience to fully employ [15]. For these reasons, many previous studies have also adopted I/O examples [1], [3], [7]. We plan to try other methods in future work.

C. INPUT #2: COMMANDS TO USE IN SYNTHESIZED PROGRAMS
Input #2 of the proposed system specifies the range of commands to use when the system composes a program. This range is described by grouping tokens1 with either “AND”

---

1A token in this paper refers to a single element of a programming language, such as a keyword or operator. We may also refer to a set of elements as a token if they are used together in a series.

(when tokens must be used together) or “OR” (when one token needs to be selected among many). For example, Input #2 in Fig. 1 indicates that “sum” or “any” needs to be selected, and tokens “for i in input if” are used together in either case. This representation resembles the propositional formulas composed of “AND” and “OR” [16].

We now detail the format of Input #2. We group tokens often used together into a unit called an expression. Then, we assign a unique ID to refer to when the same tokens are needed in other descriptions. The definition of an expression begins with “e[0-9]+:;,” where “(0-9)+” is a regular expression for a number ID. Table 2 presents an example of Input #2 with three expressions, e0 (lines 03–04), e1 (lines 06–11), and e2 (lines 13–16). Based on these expressions, the system can compose the programs, as shown in Table 3.

---

Table 2.

|TABLE 2. Example of input #2 with python commands for integer manipulation.|
|---|
|00 | `# 'input' refers to input data `|
|01 | `# 'output' refers to execution result `|
|02 | `03 | `e0:` |
|04 | `output = e1` |
|05 | `06 | `e1:` |
|07 | `07 | `sum( e2 for i in input if e3 )` |
|08 | `08 | `------` |
|09 | `09 | `any( e3 for i in input )` |
|10 | `10 | `[ e4 for i in input if e3 ]` |
|11 | `11 | `------` |
|12 | `12 | `e2:` |
|13 | `13 | `14 | `i` |
|14 | `14 | `15 | `------` |
|15 | `15 | `16 | `1` |
|16 | `16 | `17 | `------` |
|17 | `17 | `18 | `# The definitions of e0 and e4 are omitted` |

---

Table 3.

|TABLE 3. Range of programs that proposed system can explore, given table 2 as input #2.|
|---|
|ID | Synthesized Program and Its Function |
|---|---|
|1 | `output = sum( i for i in input if e3 )` |
| | `# Add all input that meets condition e3` |
|2 | `output = sum( i for i in input if e3 )` |
| | `# Count all input that fulfills e3` |
|3 | `output = any( e3 for i in input )` |
| | `# Decide whether any input fulfills e3` |
|4 | `output = [ e4 for i in input if e3 ]` |
| | `# Select input that meets e3 and apply function e4` |

---

We read Input #2, starting from e0. In Table 2, e0 is “output = e1,” indicating that the program output is determined by commands in e1. If we can choose one from multiple options, the options are separated by “------,” which means “OR.” In Table 2, e1 can be one of three options, (i) the addition of some or all input data (line 07), (ii) the confirmation of whether input data satisfy a given condition (line 09), and (iii) a selection of elements in input data (line 11). The three options (i) to (iii), within their definitions, include other
expressions, \( e_2, e_3, \) and \( e_4 \). Among these expressions, \( e_2 \) can be either “\( i \)” (line 14) or “\( 1 \)” (line 16). Similarly, \( e_3 \) and \( e_4 \) can be defined with multiple options.

The number of \( OR \) options in Input \#2 determines the size of possible programs that the proposed system can explore. Assume that the system is given the definitions in Table 2 as Input \#2 and that each of \( e_3 \) and \( e_4 \) has one fixed option from which to choose. Then, the system can explore the four different programs in Table 3. Among these programs, if the system selects line 07 for \( e_1 \), line 16 for \( e_2 \), and condition “\( i\%2==0 \)” for \( e_3 \), then it generates the program “\( \text{output} = \text{sum( } 1 \text{ for } i \text{ in input if } i\%2==0 \)”, which counts even numbers in input data, as presented in Fig. 1.

Input \#2 plays two important roles in the proposed system. First, it allows the system to learn the syntax of the programming language of user’s choice. Second, it establishes the boundary of possible programs to explore. Especially when we synthesize programs in a particular domain, we specify commands commonly used in the domain. As a result, the system focuses on these commands and explores likely programs quickly and accurately. For example, when we synthesize text-analysis programs, it helps to include string-manipulation commands [1], [8]. In this paper, we demonstrate the proposed methods with commands that process integer input because they are often used in many applications [7]. These commands include finding min/max, adding numbers, sorting/reversing numbers, selecting a subset (e.g., choosing even numbers), and applying functions (e.g., obtaining absolute values). The evaluation (Section V) presents more details of the commands and the range of possible programs. The proposed methods can be applied to other domains by specifying different command sets (e.g., string manipulation and bitwise operations).

D. PROGRAM MODEL AND SYNTHESIS METHOD

In Sections III.B and III.C, we explain the details of Inputs \#1 and \#2. This section presents how to explore the range of possible programs, as described in Input \#2, and identify a program that complies with Input \#1.

1) CONSTRUCTION OF PROGRAM MODEL

We model Input \#2 as a tree. This tree preserves the meaning of Input \#2 and allows navigation through the range of possible programs. We build the tree as follows. We create one tree for each expression. A node in the tree represents a token in the corresponding expression. If multiple tokens must be used together in a series, the corresponding nodes become children of an \( AND \) node. Otherwise, if one token needs to be chosen among multiple tokens, the corresponding nodes become children of an \( OR \) node.

Fig. 2(a) presents tree models from the expressions in Table 2. In the model for \( e_0 \), two tokens “\( \text{output} = \)” and “\( e_1 \)” are under the same \( AND \) node, meaning that they must be used together in a series. In the model of \( e_1 \), three \( AND \)ed options are under the same \( OR \) node, so one option needs to be chosen among multiple tokens, the corresponding nodes become children of an \( OR \) node. Otherwise, if one token needs to be chosen among multiple tokens, the corresponding nodes become children of an \( OR \) node.

At GitHub (https://github.com/sihyunglee26/Program-Machine-2022/), we post samples of inputs \#1-2 for three different application domains, integer manipulation, string manipulation, and bitwise operations. We also post codes for building a program model from input \#2 (Section III.D).
TABLE 4. Algorithm for navigating through different programs and identifying a target program.

| Line | Description |
|------|-------------|
| 00   | # amber: keyword, green: variable |
| 01   | # OR: list: List of all OR nodes in the tree model |
| 02   | # OR: One OR node in the tree model |
| 03   | # OR: selection: Index of the currently selected child of the OR node. This index is initialized to 0 for all OR nodes |
| 04   | # OR: size: Number of options (children) of OR node |
| 05   | 0 |
| 06   | 0 |
| 07   | 0 |
| 08   | 0 |
| 09   | # Navigate the tree to find the program that compiles with Input #1 |
| 10   | function find_program(): |
| 11   | program = program corresponding to the initial selections in the tree model |
| 12   | while program != None: |
| 13   | if program complies with Input #1: |
| 14   | return code |
| 15   | program = get_next_distinct_program() |
| 16   | return None |
| 17   | 20 # Continue to the right column |
| 21   | # Get the next distinct program from tree model |
| 22   | function get_next_distinct_program(): |
| 23   | program = move_to_next_program() |
| 24   | while there exists ineffective OR nodes whose selection > 1: |
| 25   | selection++ |
| 26   | # Skip duplicate programs |
| 27   | program = move_to_next_program() |
| 28   | return program |

TABLE 5. Application of algorithm in Table 4 on the model in Fig. 2(b).

| # of calls to move_to_next_program() | OR#1. selection | OR#2. selection | Action in get_next_distinct_program() |
|--------------------------------------|-----------------|-----------------|--------------------------------------|
| 1                                    | 0               | 0               | Return “output = [ e4 for i in input if e3 ]” |
| 2                                    | 1               | 0               | Return “output = any( e3 for i in input )” |
| 3                                    | 2               | 0               | Return “output = sum( i for i in input if e3 )” |
| 4                                    | 0               | 1               | Skip “output = [ e4 for i in input if e3 ]” since OR#2 is ineffective and its selection ≥ 1. (We explain “ineffective” in the following paragraph.) |
| 5                                    | 1               | 1               | Skip “output = any( e3 for i in input )” since OR#2 is ineffective and its selection ≥ 1. |
| 6                                    | 2               | 1               | Return “output = sum( i for i in input if e3 )” |

When tree models are ready for all expressions in Input #2, we combine the models to create a single tree. This single tree is used to navigate through possible programs in the next step. To build the tree, we begin from the model of e₀ and replace each expression with its corresponding model. Fig. 2(b) illustrates the combined model that results from the models in Fig. 2(a). OR#1 is where e₁ is replaced with its model, and OR#2 is where e₂ is replaced with its model.

2) NAVIGATION THROUGH POSSIBLE PROGRAMS

With a combined tree model of Input #2, we generate each possible program and decide whether the program complies with Input #1. Table 4 presents a pseudo code for this search algorithm. The search begins by calling the function find_program(). It uses a while loop to examine one program in each iteration (lines 14–17) and terminates either when it discovers a target program (lines 15–16) or when it has no more program to explore (line 18). To confirm that a program complies with Input #1, we run the program with input data as described in Input #1, and observe whether it produces the same output as Input #1.

We produce different programs by changing the options selected in the OR nodes of the tree. The function move_to_next_program() changes the selected options such that it explores all possible combinations of options (lines 31–41). Before the first call of the function, the list OR_list stores all OR nodes, and the options under each OR node have different indices, starting from 0, where option 0 is initially selected (i.e., OR.selection=0). Each call to the function shifts the selection of the next OR node in the OR_list to the next option (lines 32–40). This process is similar to adding 1 to a multidigit number, where a digit past its maximum returns to 0 and +1 is carried to the next digit.

Table 5 illustrates the calls to move_to_next_program() on the model in Fig. 2(b). We assume that the OR_list contains two OR nodes [OR#1, OR#2] and that the options under these nodes are assigned distinct indices, 0–2 for OR#1 and 0–1 for OR#2, as indicated in Fig. 2(b). Each call to the function

3Multiple programs can satisfy Input #1, but discovering all of them consumes a substantial amount of time. Therefore, we aim to find one such program quickly, so that an end user further modifies or optimizes it if necessary.

4If Input #1 comprises multiple I/O examples, the test succeeds when the program produces the correct outcome for all examples.
selects a different combination of options, from 00 to 21, creating $3 \times 2^6 = 6$ different combinations.

Note that different combinations of options do not necessarily lead to different programs. For example, Calls #1 and #4 in Table 5 correspond to the same program “output = [ e4 for i in input if e3 ].” Similarly, Calls #2 and #5 correspond to the same program. This is because certain OR nodes do not influence the resulting program and thus changing their selections only leads to a duplicate program. We call these nodes ineffective. In Fig. 2(b), OR#2 affects the resulting programs only when OR#1.selection=2; otherwise, when OR#1.selection=0 or 1, OR#2 is not part of the resulting programs and becomes ineffective. When OR#2 is ineffective, increasing its selections ≥ 1 creates duplicate programs (e.g., Calls #4 and #5 are duplicates of Calls #1 and #2, respectively). We identify such ineffective nodes using the while loop in function get_next_distinct_program() (lines 24–27 in Table 4) and skip duplicate programs, examining only distinct programs.

IV. EFFICIENT SEARCH THROUGH LEARNING COMMANDS

A. OVERVIEW OF THE LEARNING METHOD

This section presents the methods to reduce the time needed to identify a target program. The search algorithm in Section III.D.2 iterates through a set of possible programs $S_{prog}$, as described in Input #2, until it encounters one that conforms to Input #1. The target program can be anywhere in $S_{prog}$; thus, it can be found after the algorithm iterates over the first half of $S_{prog}$ on average. Therefore, as $S_{prog}$ becomes large, the algorithm requires numerous iterations.

To speed up the search, we rearrange programs in $S_{prog}$, such that more suitable programs are examined near the beginning of the iterations. To this end, the proposed system learns the meaning of different options in Input #2 before synthesis begins. It composes various programs by altering selections and observes their characteristics in the program outcomes. When Input #1 is given as I/O examples to request a synthesis, the system consults the learned characteristics to determine preferred options that are likely to produce the given outputs.

We illustrate the proposed method, assuming that Input #2 corresponds to the tree model in Fig. 2(b). In the model, OR#1 has three options 0–2 with different characteristics: (i) option 0 selects elements among the input data, so the output length varies between 0 and the input length, and output is a subset of the input; (ii) option 1 decides whether the input fulfills a condition, so the output is either True or False; (iii) option 2 adds elements in the input data, so the output is a single number and a sum of input elements. The system learns these distinct characteristics. When a request comes with Input #1, the system calculates the likelihood of the options conforming to Input #1 and iterates the options in the order of likelihood. For example, Input #1 states that given the input \{1,9,2,4,6\}, the program produces output \{True\}. Option / has the highest probability of producing such output; thus, the search algorithm tries option / in the first iteration and identifies a target program without examining the other unlikely options.

B. TRAINING PHASE: STAGES AND FEATURES OF LEARNING

We perform the learning in each OR node of the tree model, OR#i. The learning objective is to differentiate the options under OR#i according to their input-output behavior. In this way, when Input #1 is given as I/O examples, we can judge which option is more suitable than the other options. The learning proceeds in two stages. First, we construct the training sets, and second, we train each OR node.

In the first stage, we prepare a training set for each option $x$ under OR#i. This set consists of I/O examples of programs in which option $x$ is selected. We generate the training set by (i) iterating different programs using the tree model (Section III.D.2), (ii) running the programs with random inputs, and (iii) observing the outputs. These inputs and outputs comprise the I/O examples of the selected options. For example, given the model in Fig. 2(b), (i) we select option 0 for OR#1 to compose the first program “output = [ e4 for i in input if e3 ].” Then (ii) we run the program with input \{1,9,2,4,6\} and (iii) observe output \{2,4,6\}. The input and output together become one I/O example “\{1,9,2,4,6\} → \{2,4,6\}” which is added to the training set for the currently selected option 0 of OR#1.

We continue to run the same program with different inputs until we collect sufficient training examples to distinguish the option from the others. We move to the next program to produce I/O examples for the other options. These I/O examples are added to the training sets for the respective options. As a result, each option has its own training set of I/O examples.

In the second stage, we train each OR node using the I/O examples prepared in the first stage. Fig. 3 presents a training model and its input and output. We assume that we train the model for OR#i with $n$ options, 0 to $n-1$. We aim to train the model such that when the trained model receives a set of I/O examples, it produces a vector of $n$ probabilities $p_0$ to $p_{n-1}$, one for each option. Probability $p_x$ represents the chance of option $x$ being the correct choice given the I/O examples, and the sum of $p_0$ to $p_{n-1}$ equals 1. In the testing phase, the probabilities determine the order to choose the options, so they serve as labels for $n$-class classification. Training the model in this manner requires I/O examples in the training data to be labeled with the probability vector. We label each I/O example $e$ as follows.

If $e$ belongs to the training set for option $x$ of OR#i, we set $p_x = 1$ and all other probabilities $p_j = 0$, where $j \neq x$.

---

5 We assume that we specify Input #2 once and reuse it for multiple synthesis tasks. Thus, the system learns commands in Input #2 and applies the knowledge to synthesize various programs.

6 The training model can be various machine-learning models, and we demonstrate the use of neural networks in the evaluation (Section V).
FIGURE 3. Overview of learning to differentiate the options under each OR node.

FIGURE 4. Overview of program synthesis using trained machine-learning models.

This way of labelling is called one-hot encoding [17]. Using these labeled I/O examples, we train the model for \( OR^{#i} \). As a result, when the trained model receives I/O examples similar to \( e \) in the testing phase, \( p_x \) is likely to be larger than the others; thus, option \( x \) is preferred.

We illustrate the second stage with two I/O examples \( e_0 \) and \( e_1 \), which belong to the training sets for options 0 and 1 of \( OR^{#i} \), respectively. We label \( e_0 \) with probabilities \([1, 0]\) and \( e_1 \) with probabilities \([0, 1]\). Using these labeled examples, we train the model for \( OR^{#i} \). In the testing phase, if the trained model receives I/O examples similar to \( e_0 \), then \( p_0 \) is likely to be large, and option 0 is preferred. Likewise, if the model receives I/O examples similar to \( e_1 \), then \( p_1 \) is likely to be large and option 1 is preferred.

To summarize, we collect the training data by running various programs specified in Input #2 and observing the input-output behavior. Then, we label the data with the probabilities based on the options that compose the programs. All described stages can be automated (i.e., data generation, labeling, and training); therefore, we can train the model with as many data as needed without much human involvement or external data sources. This learning method is similar to how humans learn programming. To grasp the meaning of a new command, a learner composes several programs by tweaking the options in the command, runs the programs with various inputs, and observes the outcomes. Consequently, the learner gains a better understanding of the command.

C. TESTING PHASE: PROGRAM SYNTHESIS WITH LEARNED KNOWLEDGE

We use the trained models in Section IV.B to speed up the program synthesis. We perform the same synthesis algorithm in Table 4 and adjust the orders of options and \( OR \) nodes as follows.

Fig. 4 presents four steps to apply the trained models. First, when a synthesis request arrives with Input #1 as I/O examples, we pass them to the trained model of each \( OR \) node and obtain a vector of probabilities. If an \( OR \) node has \( n \) options, then we have a vector of \( n \) probabilities \([p_0, p_1, \ldots, p_{n-1}]\). Second, for each \( OR \) node, we adjust the order of options in decreasing order of probability so that options with greater probabilities are selected in earlier iterations. Third, we adjust the order of \( OR \) nodes in \( OR_{list} \), the list of all \( OR \) nodes in the tree, in decreasing order of entropy. The entropy of an \( OR \) node with \( n \) options is computed as \(-\sum p_j \log_2(p_j)\) for all \( j \) in 0 to \( n-1 \) [18]. The effect of this ordering is that when we encounter a program that does not fulfill the requirements in Input #1, we try altering uncertain parts of the program in the next iteration (i.e., altering options under \( OR \) nodes with high entropy), while definite parts remain unchanged. Finally, with \( OR \) nodes and their options re-ordered, we run the synthesis algorithm in Table 4 to identify a target program.

We illustrate the four steps on the model in Fig. 2(b). First, we receive a set of I/O examples as Input #1 and pass them to the trained models to obtain the following probabilities.
TABLE 6. Application of algorithm in table 2 on the model in fig. 2(b) with OR nodes and options reordered according to machine-learning models.

| # of calls to move_to_next_program() | OR#1 selection | OR#2 selection | Action in get_next_distinct_program() |
|-------------------------------------|----------------|----------------|---------------------------------------|
| 1                                   | 2              | 0              | Return “output = sum( i for i in input if e3)” |
| 2                                   | 2              | 1              | Return “output = sum( i for i in input if e3)” |
| 3                                   | 1              | 0              | Return “output = any( e3 for i in input )” |
| 4                                   | 1              | 1              | Skip, as OR#2 is ineffective and its selection ≥1. |
| 5                                   | 0              | 0              | Return “output = [ e4 for i in input if e3 ]” |
| 6                                   | 0              | 1              | Skip, as OR#2 is ineffective and its selection ≥1. |

[0.01, 0.02, 0.97] for OR#1, where option 2 is almost certainly the right selection, and [0.55, 0.45] for OR#2, where option 0 is slightly more likely than option 1.

Second, according to the probabilities, we adjust the order of options (i.e., options 2→1→0 for OR#1, and options 0→1 for OR#2). Third, according to the entropies, we adjust the order of OR nodes. The entropies are approximately 0.222 and 0.993 for OR#1 and OR#2, respectively, so the OR_list becomes [OR#2, OR#1]. Finally, we run the synthesis algorithm to iterate through different programs. Table 6 presents the order of iteration. The second and third columns reveal that (i) the options are chosen in order of probability for OR#1 (options 2→1→0) and OR#2 (options 0→1), respectively, and that (ii) changes in OR#2 precede those in OR#1 (i.e., OR#2.selection wraps around before OR#1.selection changes). In Iteration #1, we choose the options with the highest probabilities (i.e., option 2 for OR#1 and option 0 for OR#2). If the resulting program does not fulfill the requirements in Input #1, then we continue to the next program. As OR#2 precedes OR#1 in the OR_list, meaning that we are more uncertain about the selection in OR#2, we change the selection of OR#2 to produce Program #2. Assuming that the machine-learning models are correctly trained, the target program is likely to be found within the first two iterations (gray cells). This is because the initial choice for OR#1, option 2, is almost certainly the right choice with a 97% chance, and thus it does not need to change. If we do not use machine learning and reordered selections accordingly, it would take more iterations to encounter the two most-likely programs (Table 5).

V. EXPERIMENT AND ANALYSIS

Based on the models and methods from Sections III and IV, we developed and demonstrated a system to synthesize programs. Section VA describes the details of the implementation and experimental setup. Section VB evaluates the system’s performance concerning the baseline, and Section VC compares the proposed system with previous work.

A. EXPERIMENTAL SETUP AND IMPLEMENTATION

1) SPECIFICATION OF AVAILABLE COMMANDS

We aimed to synthesize programs that process integers because these functions are used in many applications, and we can directly compare results with related work [7]. To this end, we supplied Python commands that manipulate integers as Input #2 of the synthesis system. We also allowed multiple lines of the commands. In particular, we gradually added commands and increased the possible number of lines until the evaluation subjects (i.e., the proposed system and related work) exhibited differences in performance. As a result, Input #2 included the commands in Table 7 and allowed one to four lines, corresponding to 245,410 possible programs. Then, we built a tree model of Input #2, as described in Section III.D.1. The tree model comprises 162 OR nodes.

2) DATA GENERATION AND MACHINE LEARNING

Upon building the tree model of available programs, we prepared data for machine learning (i.e., I/O examples), as described in Section IV.B. In particular, we generated 1,000

TABLE 7. Summary of commands that manipulate integers and sample programs with the commands.

| ID | Command |
|----|---------|
| 1  | sum(... for i in input) |
| 2  | any(... for i in input) |
| 3  | [ i for i in input if ... ] |
| 4  | [ ... for i in input ] |
| 5  | sorted(input, reverse=-) |
| 6  | input[:i-1] |
| 7  | max(input), min(input) |
| 8  | input[0], input[-1] |
| 9  | The first or last input element |

| ID | Sample Program |
|----|----------------|
| 1  | output=sorted(input, reverse=True) |
| 2  | m1 = [ i for i in input if i%2==0] output = max(m1) |
| 3  | m1 = [abs(i) for i in input] m2 = [ i for i in m1 if i%2] output = sum( i for i in m2) |

63516
I/O examples for each distinct program. We randomly split them into three sets, (i) $S_{\text{train}}$ with 800 examples for training machine-learning models, (ii) $S_{\text{params}}$ with 100 examples for estimating parameters, and (iii) $S_{\text{test}}$ with the remaining 100 examples for testing the models. To create an I/O example, we generated five random integers between $-10$ and 10 (e.g., $\{5, 2, -1, 3, 8\}$), used the integers as input to a program, and observed the output. As a result, we prepared at least 800 training data to characterize each choice under OR nodes.

Using the training data $S_{\text{train}}$, the synthesis system learned the commands under each OR node. This learning requires a machine-learning model (Fig. 3), and we used a feedforward neural network [19], as depicted in Fig. 5. The network receives a set of I/O examples as input and passes the examples through one hidden layer and an output layer, producing a vector of probabilities. We assume that each I/O example in $x$ contains up to 10 integers, where the first five elements $i_1$ to $i_5$ are input to a program and the last five elements $o_1$ to $o_5$ are the execution output. There are cases where fewer than 10 elements exist, or some elements are not integers, and we address them as follows. First, when input (or output) has $n < 5$ elements, we pad the empty slots $i_{n+1}$ to $i_5$ (or $o_{n+1}$ to $o_5$) with an integer $c_{\text{empty}}$. For example, the I/O example $\{1, 9, 2, 4, 6\} \rightarrow \{2, 4, 6\}$ is encoded as $\{1, 9, 2, 4, 6, 2, 4, 6, c_{\text{empty}}, c_{\text{empty}}\}$. $c_{\text{empty}}$ is set to be far smaller than the nonempty elements so that empty slots appear differently from nonempty elements. Second, when an I/O example is an outcome of any command (command #2 in Table 7), it contains non-integer Boolean outputs (i.e., True and False). We encode these values as integers $c_{\text{true}}$ and $c_{\text{false}}$, respectively, and they are also chosen to be different from normal integer elements. In this experiment, we used $c_{\text{empty}} = -100$, $c_{\text{true}} = 100$, and $c_{\text{false}} = 101$, as normal integer elements stayed within $-50$ to 50, given the input range from $-10$ to 10 and the commands in Table 7.

The hidden layer in 2 includes 20 units, and each unit performs linear regression and ReLU [20]. It is followed by the output layer in 3 with $n$ units, and $n$ is the number of options under the OR node for which we train the network. The $n$ output units perform linear regression and Softmax to produce a vector of $n$ probabilities in 4.

The number of hidden units affects classification accuracy and efficiency, and the optimal number can change as we alter the command set (Input #2). Therefore, we provide guidelines on how to decide the number. We decided the number of units in the hidden layer by (i) gradually adding units, (ii) training the network with $S_{\text{train}}$, and (iii) measuring the classification accuracy with $S_{\text{params}}$. We measured the accuracy as the proportion of I/O examples in $S_{\text{params}}$ correctly classified into their respective options. We considered an I/O example to be correctly classified if it is labeled with option $x$ and the network predicts $x$ to have the highest probability. Fig. 6 displays the effect of different unit numbers. On the horizontal axis, $\{u_1, u_2, \ldots, u_L\}$ indicates that the network has
L hidden layers and that layer j has u_j units. For example, (5) indicates one hidden layer with 5 units, and (20, 5) indicates two hidden layers with 20 and 5 units, respectively. The vertical axis depicts the distribution of the classification accuracy over all OR nodes. In general, as the number of units increased, so did the accuracy. When 20 or more units were used, the median accuracy approached 97%, with the minimum above 70%. In practice, when most OR nodes have an accuracy greater than 70%, the synthesis system discovers a target program within a few iterations and does not incur much delay. This is because we typically specify a set of I/O examples as one synthesis request (Input #1) rather than specifying a single I/O example; as far as the majority of the examples are correctly classified, the system selects the correct options. We demonstrate that this is the case in Section V.B. Hence, we chose to use one hidden layer with 20 units (i.e., (20) in Fig. 6).

We trained the neural networks using machine-learning algorithms and parameters summarized in Table 8. We implemented the algorithms using Python, TensorFlow [21], and NumPy [22] and performed experiments on a machine with a 3.4GHz CPU (Intel Core i7-6700) and 16GB RAM. Training the networks for 162 OR nodes took approximately 1.2 hours. Synthesizing a program (i.e., discovering a target program using the trained networks) took less than 5 seconds most of the time and no longer than 2 minutes.

**TABLE 8. Summary of algorithms and parameters used for machine learning.**

| ID | Description                                      |
|----|--------------------------------------------------|
| 1  | # of epochs used for training = 500              |
| 2  | Adam optimization with learning rate a = 0.08    |
| 3  | Batch normalization with momentum = 0.99. The momentum was used to compute moving means and variances. |
| 4  | L2 normalization with regularization constant λ = 0.1. No dropout was used. |

**B. PROGRAM SYNTHESIS AND PERFORMANCE EVALUATION**

Based on the implementation described in Section V.A, we synthesized various programs and measured the duration required. In particular, we measured the number of programs that the synthesis system explores until it discovers a target program (i.e., the number of iterations in the navigation algorithm in lines 14–17 of Table 4). We measured this number because it does not depend on particular implementations and environments, whereas runtime (execution time) does depend [23].

We compared the number of iterations for different subsets of the proposed methods to validate their effectiveness. This validation helps understand whether the methods do improve search speed and why. The subsets are as follows:

- (R) navigation through programs in Random order without using learned knowledge,
- (P) ordering of options according to predicted Probability,
- (E) ordering of OR nodes according to the Entropy, and
- (PE) combination of P and E.

The subset R is the basic search algorithm in Section III, P and E apply one of the machine-learning techniques in Section IV, and PE applies all proposed methods.

Using the four subsets of methods, we synthesized 1,000 randomly selected programs among the 245,410 possible programs. For each program, we used 100 I/O examples in $S_{test}$ as Input #1. In particular, we split the examples into 20 equal-sized sets so that each set contains 5 examples. Then, we used each set as one synthesis request, measured the number of iterations, and computed the average over all sets.

Fig. 7. presents the number of iterations for the four different methods. The vertical axis lists the methods, and the horizontal axis indicates the number of iterations in the log scale. On average, the random search (R) has taken more than 10,000 iterations to find a target program. As we used machine learning techniques (P and E), the number was reduced by an order of magnitude, and when we used their combination (PE), the number went further down to several tens to hundreds. This means that the techniques substantially improve the search speed, and they complement each other—the system selects the most probable options first (P), and when selections are not right, it alters the most uncertain part (E) to the next probable option (P) in the following iterations.

As P and E depend on machine learning, we further analyzed how the machine-learning model in Fig. 5 learns to differentiate commands. We present the intuition behind how the model works using (i) a simplified version of the model, one unit of linear regression, and (ii) a binary classification scenario, where an OR node chooses one from the following two options.

**FIGURE 7. Number of iterations required to synthesize a program for different subsets of methods.**

Option 0: `output = sum(i for i in input)`
# Add all inputs
Option 1: `output = [i for i in input if i%2==0]`
# Select even inputs

$$\text{Distribution of \# of Iterations}$$

| Navigation Methods |
|--------------------|
| Distribution       |
| min | max |
| --- | --- |
| 1  | 25% |
| 2  | 75% |
Given an I/O example \([i_1, i_2, i_3, i_4, i_5, o_1, o_2, o_3, o_4, o_5]\), the linear-regression unit computes the equation \(w_1 i_1 + w_2 i_2 + \ldots + w_5 i_5 + w_6 o_1 + w_7 o_2 + \ldots + w_{10} o_5\), where \(w_1\) to \(w_{10}\) are parameters to learn. If the computed value is larger than a threshold, then the OR node prefers option 0 over 1. Otherwise, option 1 is preferred. Table 9 presents a sample of learned parameters \(w_1\) to \(w_{10}\) and four I/O examples. Examples #1–2 are of option 0; thus, output \(o_1\) represents the sum of inputs \(i_1\) to \(i_5\), and the remaining slots \(o_2\) to \(o_5\) are filled with \(c_{\text{empty}} = -100\). Similarly, Examples #3–4 are of option 1, so outputs represent even numbers among inputs. The last column lists the values computed by the equation. The values are greater for option 0 (\(>300\)) than option 1 (\(<100\)), meaning that they help differentiate the two options. This is because the equation captures their differences. Option 0 produces a single sum, so \(o_1\) tends to be large and \(o_2\) to \(o_5\) are small, whereas option 1 can produce more outputs, so the outputs have opposite characteristics. Therefore, positive \(w_6\) and negative \(w_7\) to \(w_9\) tend to increase the equation for option 0 and decrease it for option 1.

As another example, assume that an OR node chooses one from the following two options.

**Option 0:** output = sorted(input, reverse = False)
# sort in increasing order

**Option 1:** output = sorted(input, reverse = True)
# sort in decreasing order

Option 0 sorts the inputs in increasing order, so \(o_1 < o_2 < o_3 < o_4 < o_5\), whereas option 1 sorts the inputs in decreasing order, so \(o_1 > o_2 > o_3 > o_4 > o_5\). Therefore, if parameters \(w_6\) to \(w_{10}\) are learned such that \(w_6 < w_7 < w_8 < w_9 < w_{10}\), the equation produces larger values for option 0 than 1.

To summarize, we demonstrated that linear-regression units capture differences among commands. This experiment used more than one such unit; therefore, we were able to differentiate a more diverse range of commands.

### C. COMPARISON WITH DEEPCODER

In Section V.B, we compared four different subsets of the proposed methods. In this section, we compared the full version of the proposed methods (PE in Section V.B) with DeepCoder [7], a state-of-the-art system that synthesizes programs based on machine-learned I/O patterns. In particular, we aimed to determine which system synthesizes requested programs more quickly and efficiently.

For the proposed system, we used the same experimental settings as in Section V.A. For the existing system, we used mostly the same settings, adjusting a few to the peculiarities of the system, as follows. First, we used the same set of available commands (Input #2) (i.e., we allowed one to four lines of the integer-manipulation commands in Table 7). Second, we trained the existing system about the available programs using the I/O examples in \(S_{\text{train}}\), but the system used a different neural network, as summarized in Table 10. This network takes I/O examples as input and produces how likely the expected program is to include each single-line command (e.g., “\(c_{\text{empty}} = \sum(1 \text{ for } i \text{ in input if } i \% 2 == 0)\)”). The output does not indicate in which line in the program the command appears. Lastly, we synthesized 1,000 randomly selected programs using the I/O examples in \(S_{\text{test}}\) and measured the number of different programs the system navigated until it discovered the expected program. The system begins by navigating programs of length 1 and gradually increases the length. For each length, the system tries commands in the order of predicted probabilities.

Fig. 8(a) presents histograms for the number of iterations of the two systems. The horizontal axis indicates the number of I/O examples.
of iterations, and the vertical axis represents the proportion of programs. The proposed system found target programs within the first 10 iterations 13.2% of the time, whereas the same occurred in the existing system for <4% of the time. The proposed system required an average of 87.5% fewer iterations than the existing system. We also present the execution times of the two systems in Fig. 8(b). The horizontal axis indicates the execution time, and the vertical axis represents the cumulative percentage of programs. The proposed system found most of the target programs within 120 seconds, whereas the existing system has taken up to 1000 seconds. Overall, the execution times were roughly proportional to the numbers of iterations.

![Histogram of number of iterations](image)

![Cumulative distribution of execution time](image)

**FIGURE 8.** Number of iterations and execution time required to synthesize a program for the proposed and existing systems.

We identified three major reasons as follows. First, the existing system does not consider the order of operations; it predicts the probability of each command appearing anywhere in a program and tries various orders of commands without considering which order is better. However, different orders often produce different outcomes (e.g., sum after selection ("sum(i for i ..." → "(i for i ...") performs a different function from selection after sum). The proposed system considers such orders and chooses a more probable one. For example, Fig. 9(a) illustrates part of the tree representation for two consecutive commands. The predicted probabilities indicate that sum after selection is more likely (red options); thus, OR#3 chooses sum and OR#4 chooses selection. Second, when a program under investigation does not conform to the requested function, the existing system alternates the program in a predefined order (i.e., alters line 1 and then alters line 2). In contrast, the proposed system alters the most uncertain part of the program (e.g., if line 2 is the most uncertain, then the system alters line 2 in the next iteration, rather than changing line 1). Third, the existing system begins exploration from programs of length 1, and if a target program is not found, it increases the length by one. In contrast, the proposed system begins to search from the most probable length. Fig. 9(b) presents an example. The probabilities indicate that length 2 is more likely than length 1, so the system explores length 2 first and then length 1.

![Tree model for two commands](image)

![Tree model for 1–2 commands](image)

**FIGURE 9.** Tree representation for multiple lines of commands.

**VI. CONCLUSION**

We propose a system that synthesizes programs requested by end users. The system is given a set of commands to use and learns their meanings by observing the outcomes of various inputs. Similarly, it learns the combined effect of multiple commands. According to the learned knowledge, the system selects appropriate commands and constructs a program. Then, the system progressively revises the program through iterations until it fully fulfills the request. We evaluated the system by requesting 1,000 integer-manipulation programs out of 245,410 possible programs. The system successfully synthesized the programs in >80% fewer iterations than the baseline and the existing methods.

We plan to extend the pool of possible programs to above a million and evaluate various methods to control the learning cost at a reasonable level. Our initial results suggest that we prioritize learning critical selections that affect numerous programs (i.e., focus on learning OR nodes near the root of tree representation, rather than learning every OR node). In this way, the system quickly makes correct choices for critical parts of a program and determines the rest through a few iterations. We also plan to adapt the system to programs in domains other than integer manipulation. We learn bitwise-manipulation commands by applying the same proposed methods, since input and output are numbers. To learn string-manipulation commands, we represent characters in input-output strings as numbers, similarly to Unicode encoding, and apply the proposed methods as if we learn number patterns. The initial results are encouraging, and we intend to perform more extensive experiments.
REFERENCES

[1] S. Gulwani, “Automating string processing in spreadsheets using input-output examples,” in Proc. POPL, Austin, TX, USA, 2011, pp. 317–330, doi: 10.1145/1926385.1926423.

[2] M. Mefteh, N. Bouassida, and H. Ben-Abdallah, “Towards naturalistic programming: Mapping language-independent requirements to constrained language specifications,” Sci. Comput. Program., vol. 166, pp. 89–119, Nov. 2018, doi: 10.1016/j.scico.2018.05.006.

[3] S. So and H. Oh, “Synthesizing imperative programs from examples guided by static analysis,” in Proc. SAS, New York, NY, USA, 2017, doi: 10.1007/978-3-319-66706-5_18.

[4] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur, “TRANSIT: Specifying protocols with concolic snippets,” in Proc. PLDI, Seattle, WA, USA, 2013, pp. 287–296, doi: 10.1145/2491956.2462174.

[5] W. Lee, K. Heo, R. Alur, and M. Naik, “Accelerating search-based program synthesis using learned probabilistic models,” in Proc. PLDI, Philadelphia, PA, USA, Jun. 2018, pp. 436–449, doi: 10.1145/3192366.3192410.

[6] Y. Li et al. (2022). Competition-level code generation with Alpha-Code. DeepMind. [Online]. Available: https://alphacode.deepmind.com/ and https://arxiv.org/abs/2203.07814

[7] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, “DeepCoder: Learning to write programs,” in Proc. ICLR, Toulon, France, 2017, pp. 1–21, doi: 10.48550/arXiv.1611.01989.

[8] W. Lee, “Combining the top-down propagation and bottom-up enumeration for inductive program synthesis,” in Proc. POPL, Copenhagen, Denmark, 2021, pp. 1–28, doi: 10.1145/3434335.

[9] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen, “Counterexample guided inductive synthesis modulo theories,” in Proc. CAV, Oxford, U.K., 2018, pp. 270–288, doi: 10.1007/978-3-319-96145-3_15.

[10] S. Lee, K. Kim, B. Kim, H. Park, and D. Lee, “Program synthesis through input/output samples and pattern learning,” in Proc. KCC, Jeju, South Korea, 2021, pp. 1379–1381.

[11] GitHub: Where the World Builds Software. Accessed: May 29, 2022. [Online]. Available: https://github.com/

[12] Codeforces: Competitive Programming Contests. Accessed: May 29, 2022. [Online]. Available: https://codeforces.com/

[13] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, “Search-based program synthesis,” Commun. ACM, vol. 61, no. 12, pp. 84–93, Nov. 2018, doi: 10.1145/3208071.

[14] G. Kim, V. Duong, X. Lu, and L. Schubert, “A transition-based parser for unscoped episodic logical forms,” in Proc. IWCS, Groningen, The Netherlands, 2021, pp. 184–201, doi: 10.48550/arXiv.2103.08759.

[15] S. Lukins, A. Levicki, and J. Burg, “A tutorial program for propositional logic with human/computer interactive learning,” in Proc. SIGCSE, Covington, GA, USA, 2002, pp. 381–385, doi: 10.1145/656340.6563490.

[16] J. Groote and J. Warners, “The propositional formula checker HoeR Hugo,” J. Automated Reasoning, vol. 24, pp. 101–124, Feb. 2000, doi: 10.1023/A:1006366304347.

[17] T. Al-Shehari and R. A. Alsoawil, “An insider data leakage detection using one-hot encoding, synthetic minority oversampling and machine learning techniques,” Entropy, vol. 23, no. 10, Sep. 2021, Art. no. 1258, doi: 10.3390/entropy10101258.

[18] S. Winters-Hilt, Informatics and Machine Learning. Hoboken, NJ, USA: Wiley, 2022.

[19] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. Cambridge, MA, USA: MIT Press, 2016.

[20] K. Haras, D. Saito, and H. Shouno, “Analysis of function of rectified linear unit used in deep learning,” in Proc. IJCNN, Killarney, Ireland, Jul. 2015, pp. 1–8, doi: 10.1109/IJCNN.2015.7280578.

[21] TensorFlow: An Open Source Machine Learning Library for Research and Production. Accessed: May 29, 2022. [Online]. Available: https://www.tensorflow.org/

[22] NumPy: The Fundamental Package for Scientific Computing With Python. Accessed: May 29, 2022. [Online]. Available: https://numpy.org/

[23] M. Wolf. High-Performance Embedded Computing. Burlington, MA, USA: Morgan Kaufmann, 2014, pp. 139–200, doi: 10.1016/C2012-0-07058-5.