AP: Artificial Programming

Rishabh Singh[1] and Pushmeet Kohli[2]

1 **Cognition Group, Microsoft Research, Redmond, WA, USA**
risin@Microsoft.com
2 **Cognition Group, Microsoft Research, Redmond, WA, USA**
pkohli@microsoft.com

Abstract

The ability to automatically discover a program consistent with a given user intent (specification) is the holy grail of Computer Science. While significant progress has been made on the so-called problem of Program Synthesis, a number of challenges remain; particularly for the case of synthesizing richer and larger programs. This is in large part due to the difficulty of search over the space of programs. In this paper, we argue that the above-mentioned challenge can be tackled by learning synthesizers automatically from a large amount of training data. We present a first step in this direction by describing our novel synthesis approach based on two neural architectures for tackling the two key challenges of *Learning to understand partial input-output specifications* and *Learning to search programs*. The first neural architecture called the *Spec Encoder* computes a continuous representation of the specification, whereas the second neural architecture called the *Program Generator* incrementally constructs programs in a hypothesis space that is conditioned by the specification vector. The key idea of the approach is to train these architectures using a large set of (ϕ, P) pairs, where P denotes a program sampled from the DSL L and ϕ denotes the corresponding specification satisfied by P. We demonstrate the effectiveness of our approach on two preliminary instantiations. The first instantiation, called Neural FlashFill [29], corresponds to the domain of string manipulation programs similar to that of FlashFill [13, 14]. The second domain considers string transformation programs consisting of composition of API functions. We show that a neural system is able to perform quite well in learning a large majority of programs from few input-output examples. We believe this new approach will not only dramatically expand the applicability and effectiveness of Program Synthesis, but also would lead to the coming together of the Program Synthesis and Machine Learning research disciplines.

1998 ACM Subject Classification D.1.2 [Programming Techniques] Automatic Programming, I.2.2 [Artificial Intelligence] Program Synthesis, I.5.1 [Artificial Intelligence]: Neural Nets

Keywords and phrases Neural Program Synthesis, Neural Programming, Neural FlashFill

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.16

1 Introduction

The impact of computing in shaping the modern world cannot be overstated. This success is, in large part, due to the development of ever more revolutionary and natural ways of specifying complex computations that machines need to perform to accomplish tasks. Despite these successes, programming remains a complex task – one which requires a long time to master. Computer Scientists have long worked on the problem of program synthesis *ie* the task of automatically discovering a program that is consistent with a given user intent (specification) [38, 6, 22]. The impact of Program Synthesis is not just limited to democratizing programming, but as will see below, it allows us to program computers to accomplish tasks that were not possible earlier.

Considering the field of machine learning from the perspective of Program Synthesis, the specification takes the form of input-output examples (training data), and programs are restricted to certain restricted languages. For instance, linear regression involves synthesizing programs that involve a single linear expression, neural networks involve synthesizing programs that are composed of sequence of tensor operations, and decision trees involve synthesizing programs composed of nested if-then conditions. The availability of large scale training data and compute, along with the development of approaches to search-over the afore-mentioned restricted families of programs have allowed machine learning based systems to be extremely successful in many domains. This is particularly true in the case of perceptual tasks such as image [23] and speech understanding [18], where machine learning has led to dramatic recent breakthroughs in the form of development of systems whose abilities go beyond humans themselves [16, 40]. While the programs learnt using machine learning approaches have been very effective, they are restricted by limited domain specific languages they employ. Moreover, the programs learnt by such techniques are hard to interpret, verify, and correct.

Program Synthesis has also seen a substantial amount of research in the Programming Languages community. While earlier approaches to tackle this problem were mostly based on deductive reasoning [25, 26, 27], several approaches based on inductive reasoning have been recently proposed. These new approaches have exploited advances in computational power, algorithmic advances in constraint-solving, and application-specific domain insights [1]. These approaches can be broadly divided into four categories based on the search strategy they employ: (i) enumerative [39], (ii) stochastic [35], (iii) constraint-based [36, 37], and (iv) version-space algebra based [14, 30]. The enumerative approaches enumerate programs in a structured hypothesis space and employ smart pruning techniques to avoid searching a large space. The stochastic techniques use a cost function to induce a probability distribution over the space of programs conditioned on the specification, which is used to sample the desired program. The constraint-based techniques encode the search problem in low-level SAT/SMT constraints and solve them using off-the-shelf constraint solvers. Finally, the Version-space algebra based techniques learn programs in specialized DSLs to perform an efficient divide-and-conquer based search.

While there has been a significant progress in synthesizing richer and larger programs, these synthesis approaches suffer from a number of challenges. The first and most daunting challenge is the search problem for searching over the large space of programs. Modern approaches try to overcome this problem by carefully designing the DSL [14]. Not only this approach restricts the expressivity of the language but it is also extremely time consuming as the DSL designer needs to encode several domain-specific heuristics in terms of DSLs, pruning strategy, cost function etc. Second, these synthesis algorithms do not learn from previously solved tasks, i.e. they do not evolve and get better over time. Finally, these algorithms are typically designed to handle one form of specification (such as input-output examples or partial programs). It is difficult to handle multi-modal specifications such as combination of input-output examples, natural language description, partial programs, and program invariants all together as one combined specification.

In this paper, we argue that the above-mentioned challenges can be tackled by learning synthesizers automatically from a large amount of training data. We believe this new approach will dramatically expand the applicability and effectiveness of Program Synthesis and has the potential to have a massive impact on the development of complex intelligent software systems of the future. We present a first step in this direction by describing a novel synthesis approach based on two neural architectures for tackling the two key challenges of *Learning to understand specifications* and *Learning to search programs*. The first neural architecture called

the *Spec Encoder* computes a continuous representation of the specification, whereas the second neural architecture called the *Program Generator* incrementally constructs programs in a hypothesis space that is conditioned by the specification encoding vector. The key idea of the approach is to train these architectures using a large set of (ϕ, P) pairs, where P denotes a program sampled from the DSL L and ϕ denotes the corresponding specification satisfied by P.

We present two preliminary instantiations of this approach. The first instantiation, Neural FlashFill [29], corresponds to the domain of string manipulation programs similar to that of FlashFill [13, 14]. The second domain considers string transformation programs consisting of composition of API functions [4]. The specification mechanisms in both of these domains is input-output string examples and we develop an R3NN (Reverse-Recursive-Reverse Neural Network) to incrementally generate program trees in the DSL that is conditioned on the input-output examples. We show that even with preliminary encodings, the neural system is able to perform quite well in learning a large majority of programs from few input-output examples. We finally conclude with some challenges and exciting future directions.

Related work on Learning to Program. There have some recent proposals to use neural network based encodings for synthesizing programs. Neural RAM [24] constructs an end-to-end differentiable model representing the sequential compositions of a given set of modules (gates), and learns a controller defining the compositions of modules to obtain a program (circuit) that is consistent with a given set of examples. DeepCoder [2] embeds input-output examples of integers to learn a distribution over likely functions that would be useful for the task, and uses off-the-shelf synthesis techniques such as enumerative and Sketch [36] to learn the program. Terpret [9] and Forth [34] are probabilistic programming languages that allow programmers to write high-level programs with holes, which are then completed using a gradient-descent based search. While these systems have shown a lot of promise, they typically learn simple short programs, need a lot of compute resources per synthesis task, and do not learn how to perform efficient search.

There has been a number of recent proposals of neural architectures to perform *program induction*, where the goal is to learn a network that can learn the functional behavior of a program. These architectures are inspired from computation modules (Turing Machines, RAM, GPU) [12, 24, 21, 33, 28] or common data structures such as stacks [20]. The key idea in these approaches is to represent the operations in a differentiable form, which allows for efficient end-to-end training of a neural controller. However, unlike our approach that learns interpretable programs, these approaches learn only the program behavior. There is also an exciting line of work on learning probabilistic models of code from *big code* [32, 5, 17], which are used for different applications such as variable and method name inference, code-completion, generating method summaries etc.

2 Overview of Neural Program Synthesis

The overview of our approach is shown in Figure 1. We assume that the hypothesis space of programs is given in the form of grammar similar to that of SyGuS [1] and the goal is to find a derivation (program) from the grammar that is consistent with the specification provided by a user. There are two key neural modules: (i) Specification Encoder (Learning to understand specification) and (ii) Program Generator (Learning to search over programs). The specification encoder generates a continuous vector representation of specification (potentially in multiple formats). The program generator takes the specification vector and the DSL L

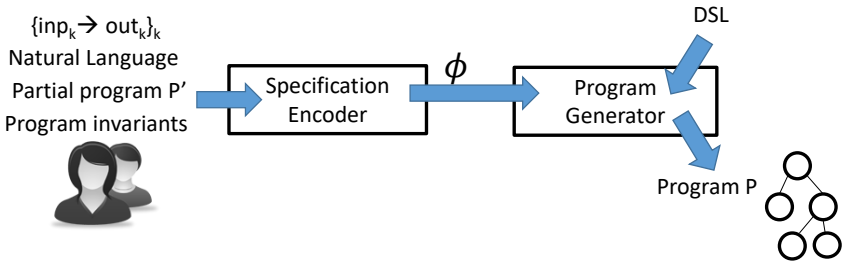


Figure 1 An overview of the neural program synthesis approach consisting of two components. The specification encoder generates a distributed representation of the specification, which is then used to condition the program generator module that incrementally generates programs in a DSL.

as inputs, and generates a program $P \in L$ that conforms to the specification. We briefly describe the two modules and different challenges associated with them.

2.1 Neural Architectures

Specification Encoder: The challenge of designing a specification encoder is that it needs to handle multiple forms of specifications, but at the same time it also needs to maintain a differentiable representation that can be efficiently learnt. Some possible specification modalities we aim to support in our architecture include input-output examples, natural language descriptions, partial programs, and program invariants. The challenge in encoding input-output examples is that the set of examples can be of variable size and each example can be of different length. The encoding should also be invariant to the order of examples. Moreover, the inputs and outputs can be of different types such as strings, integers, arrays, etc., which adds another challenge to the encoder. For encoding partial programs, the encoder needs to take into account the tree structure of parse tree of the program compared to the simpler sequence structure of base types.

Program Generator: The program generator component needs to generate a program from the DSL that is consistent with the specification vector obtained from the specification encoder. One approach is to model program semantics in the continuous domain and have the neural network perform optimization over that space. Since programs are typically discrete in nature, i.e. a small change in inputs (or in programs) can lead to big changes in output, embedding continuous representation of program semantics is challenging. Another approach can be to instead learn a controller that selects different choices from the hypothesis space (DSL) to construct a program. This controller needs to encode partial program trees and learn how to incrementally expand the trees. Depending on the complexity of the DSL, it may additionally need to encode program states, memory, and stack information.

2.2 Training the Neural Architectures

One of the reasons for success of neural networks recently has been the availability of large amounts of training data. While for domains such as natural language processing and computer vision, acquiring good labeled data is a challenging task, it is relatively easier for our domain of program synthesis. Given a DSL, we use a program sampler to sample million of programs from the language and order them using different metrics such as program size,

operator usage etc. Given a sampled program, we can then design a rule-based approach to generate corresponding specifications that are consistent with the sampled program.

However, there are also several challenges in generating the training data. First, sampling programs from a context-free grammar is relatively straightforward, but sampling programs from a more complex stateful grammar is more complex as the sampler needs to ensure that the program is well-formed, e.g. there are no variable usage before define or array out-of-bound errors. Second, the generation of consistent specification for a program can also be challenging as the sampler needs to ensure that the inputs satisfy the pre-conditions of the sampled programs. For generating natural language descriptions, one challenge is that there might be multiple different ways to specify the functionality.

Given the training data of input-output examples together with their corresponding programs, the specification encoder and the program generator can be trained in an end-to-end supervised manner using the backpropagation techniques. The goal of training is to learn best parameters for the two neural architectures that result in learning consistent programs for as many training points as possible. There are also several choices for cost function for optimizing the training loss. One choice can be to ensure that the learnt program is syntactically similar to that of training program. This cost function is easy to optimize since we can use the complete supervision for each individual component of the program. However, this cost function also penalizes many good programs that are syntactically different but semantically equivalent with respect to the training program. This is especially true in case of inductive specifications such as input-output examples, where there might be multiple consistent programs. The cost function can be enhanced to optimize over the outputs of the learnt programs instead of their syntactic structure, but this makes the optimization algorithm harder since we no more have intermediate supervisory signal, and we can only check for program correctness after constructing the complete program.

2.3 Synthesizing programs from the learnt architectures

After learning the neural architectures, we can use them to learn programs given some specification. The specification encoder encodes the specification using the learnt parameters and the specification vector is fed to the program generator to construct a consistent program. Another interesting property of the program synthesis domain unlike other domains such as NLP and vision is that we can execute the learnt programs and check if they are correct with respect to the specification at test time, which allows us multiple choices during program generation. We can either generate the 1-best program using the program generator, or we can use the learnt distributions over the grammar expansions to instead sample and generate multiple programs until finding one that is consistent with the given specification. However, checking for correctness might not be feasible for all types of specifications such as Natural Language specifications.

3 Neural FlashFill and NACIO

We present a preliminary instantiation of our framework on two domains – Neural FlashFill and Nacio. The Neural FlashFill system learns regular expression based string transformations in a DSL similar to that of FlashFill given a set of examples. Nacio learns string transformation programs that involves composition of API functions. We use the same neural architectures for Specification Encoder and the Program Generator for both the systems.

String e	:=	$\texttt{Concat}(f_1, \cdots, f_n)$
Substring f	:=	$\texttt{ConstStr}(s)$
	|	$\texttt{SubStr}(v, p_l, p_r)$
Position p	:=	$(r, k, \texttt{Dir})$ | $\texttt{ConstPos}(k)$
Direction $\texttt{Dir}$	:=	$\texttt{Start}$ | $\texttt{End}$
Regex r	:=	s | T_1 $\cdots$ | T_n

(a)

	Input v	Output
1	William Henry Charles	Charles, W.
2	Michael Johnson	Johnson, M.
3	Barack Rogers	Rogers, B.
4	Martha D. Saunders	Saunders, M.
5	Peter T Gates	Gates, P.

(b)

$\texttt{Concat}(f_1, \texttt{ConstStr(", ")}, f_2, \texttt{ConstStr(".")})$,
where $f_1 \equiv \texttt{SubStr}(v, (\text{“ ”}, -1, \texttt{End}), \texttt{ConstPos}(-1))$ and $f_2 \equiv \texttt{SubStr}(v, \texttt{ConstPos}(0), \texttt{ConstPos}(1))$

(c)

Figure 2 (a) The regular expression based string transformation DSL, (b) an example task in the DSL for transforming names to last names followed by first name initial, and (c) an example program in the DSL for this task.

3.1 Domain-specific Language

The domain-specific language for Neural FlashFill is shown in Figure 2(a). The top-level expression is a concatenation of a sequence of substring expressions, where each substring expression is either a constant string s or a substring of an input string between two positions p_l and p_r (denoting the left and right indicies in the input string). A position expression can either be a constant index or a token match expression $(r, k, \texttt{Dir})$, which denotes the $\texttt{Start}$ or $\texttt{End}$ of the $k^{\texttt{th}}$ match of token r in input string v. A regex token r can either be a constant string s or one of 8 predefined regular expressions such as alphabets, alphanumeric, capital etc. An example benchmark is shown in Figure 2(b) and a possible DSL program for the transformation is: $\texttt{Concat}(f_1, \texttt{ConstStr(", ")}, f_2, \texttt{ConstStr(".")})$, where $f_1 \equiv \texttt{SubStr}(v, (\text{“ ”}, -1, \texttt{End}), \texttt{ConstPos}(-1))$ and $f_2 \equiv \texttt{SubStr}(v, \texttt{ConstPos}(0), \texttt{ConstPos}(1))$. The program concatenates: (i) substring between the end of last whitespace and end of string, (ii) constant string “, ”, (iii) first character of input string, and (iv) constant string “.”.

The top-level expression in the NACIO DSL (Figure 3(a)) is similar to that of the FlashFill DSL consisting of concatenation of a sequence of string expressions. The main difference is that a substring expression can now also be an API expression belonging to one of the three classes of APIs: lookup APIs, regex APIs, and transform APIs, and the DSL allows for both composition and nesting of APIs. The lookup APIs such as `GetCity`, `GetState` etc. comprise a dictionary of a list of strings and perform a lookup on an input string. The regex APIs such as `GetFirstNum`, `GetLastWord`, etc. search for certain regular expression patterns in the input string and return the matched string. Finally, the transform APIs such as `GetStateFromCity` transform strings from one dictionary to another dictionary. In total, the DSL consists of 107 APIs (84 regex, 14 lookup, 9 transform). An example NACIO task shown in Figure 3(b) can be performed by the DSL program $\texttt{GetAirportCode}(\texttt{GetCity}(v))$.

3.2 Specification Encoder

The specification encoder for the Neural FlashFill and NACIO encodes a set of input-output strings to a continuous vector representation. The main idea of the encoder is to first run two separate bidirectional LSTMs [19] over the input and output strings respectively, and then perform compute a cross-correlation vector between the two representations. It then

$$
\begin{array}{rcl}
\text{String } e & := & \texttt{Concat}(f_1, \cdots, f_n) \\
\text{Substring } f & := & \texttt{CStr}(s) \mid \mathcal{R}(f) \\
& \mid & \mathcal{T}(f) \mid \mathcal{L}(v) \mid v \\
\text{Lookup API } \mathcal{L} & := & \mathcal{L}_1 \mid \cdots \mid \mathcal{L}_m \\
\text{Regex API } \mathcal{R} & := & \mathcal{R}_1 \mid \cdots \mid \mathcal{R}_l \\
\text{Transform API } \mathcal{T} & := & \mathcal{T}_1 \mid \cdots \mid \mathcal{T}_k
\end{array}
$$

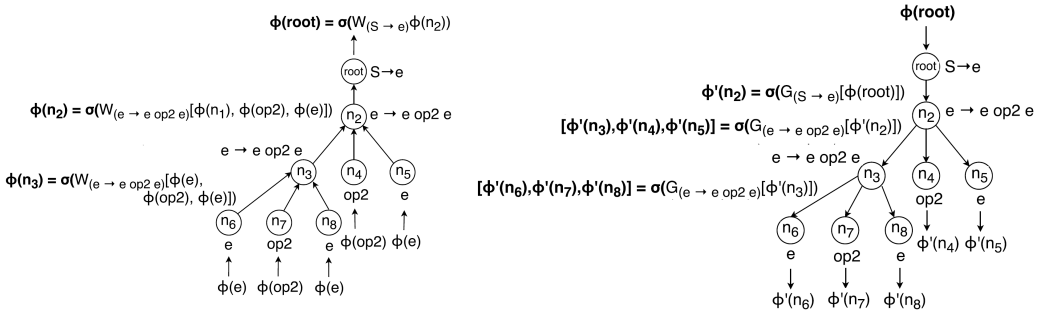
(a)

	Input v	Output
1	Los Angeles, CA	LAX
2	Boston, MA	**BOS**
3	San Francisco, CA	**SFO**
4	Chicago, IL	**ORD**
5	Detroit, MI	**DTW**

(b)

(c) `GetAirportCode(GetCity(`v`))`

Figure 3 (a) The NACIO DSL for API composition, (b) a NACIO task to obtain airport code and (c) a program in the DSL for the task.



(a) Recursive pass (b) Reverse-Recursive pass

Figure 4 (a) The initial recursive pass of the R3NN. (b) The reverse-recursive pass of the R3NN where the input is the output of the previous recursive pass.

concatenates the representations of all input-output examples to get a representation for the set of examples.

3.3 Program Generator

We develop a new R3NN (Recursive-Reverse-Recursive Neural Network) [29] to define a generation model over trees in a DSL (grammar). The R3NN model takes a partial program tree (derivation in the grammar), and decides which non-terminal node in the tree to expand and with which expansion rule, given the I/O encoding vector. The model first starts with the start symbol of the grammar and incrementally incrementally constructs derivations in the grammar until generating a tree with all terminal leaf nodes.

The R3NN model has the following 4 parameters: (i) a vector representation for each symbol in the grammar, (ii) a vector representation for each rule in the grammar, (iii) a deep neural network that takes as input the set of Right-hand side (RHS) symbols of a rule and generates a representation of the corresponding Left-hand side (LHS) symbol, and (iv) a deep neural network that as input the representation of an LHS symbol of a rule and generates a representation for each of the corresponding RHS symbols. The R3NN first assigns a vector representation to each leaf node of a partial tree, and then performs a recursive pass going up in the tree to assign a global representation to the root. It then performs a reverse-recursive pass from the root to assign a global representation to each node in the tree. Intuitively, the idea is to assign a representation to each node in the tree such that the node knows

Sample	Train	Test
1-best	60%	63%
1-sam	56%	57%
10-sam	81%	79%
50-sam	91%	89%
100-sam	94%	94%

(a)

Size	Train	Test
7	45%	37%
8	67%	53%
9	36%	28%
10	41%	33%

(b)

Sample	NeuralFF	Nacio
1-best	8%	15%
1-sam	5%	12%
10-sam	13%	24%
50-sam	21%	34%
100-sam	23%	37%

(c)

Figure 5 (a) Neural FlashFill performance on synthetic data of programs upto size 13, (b) NACIO performance on synthetic data of programs upto 3 concats, and (c) Perfomance of Neural FlashFill and NACIO on 238 real-world FlashFIll Benchmarks.

about every other node in the tree. The R3NN encoding for an example partial tree in the grammar with the recursive and reverse-recursive passes is shown in Figure 4(a) and Figure 4(b) respectively.

More concretely, we first retrieve the distribution representation $\phi(S(l))$ for every leaf node $l \in L$ in the tree and then perform a standard recursive bottom-to-top, RHS→LHS pass by going up the tree and applying $f_{R(n)}$ for every non-leaf node $n \in N$ on its RHS node representations. We continue this pass until we reach the root node, where $\phi(root)$ denotes the global tree representation. This global representation has lost any notion of tree position and we perform a reverse-recursive pass to pass this information to all the leaf nodes of the tree. We start this pass by providing the root node representation $\phi(root)$ as an input to the second set of deep networks $g_{R(root)}$ where $R(root)$ denotes the production rule for expanding the start symbol. This results in a representation $\phi'(c)$ for each RHS node c of $R(root)$. We iteratively apply this procedure to all non-leaf nodes c, i.e., process $\phi'(c)$ using $g_{R(c)}$ to get representations $\phi'(cc)$ for every RHS node cc of $R(c)$. At the end of this pass, we obtain a leaf representation $\phi'(l)$ for each leaf node l, which has an information path to every other node in the tree. Using the global leaf representations $\phi'(l)$, we can generate the scores for each tree expansion e as: $z_e = \phi'(e.l) \cdot \omega(e.r)$, where $e.l$ denotes the leaf node l associated with the expansion e and $e.r$ denotes the expansion rule. The expansion scores can then be used to obtain the expansion probabilities as: $\pi(e) = e^{z_e} \Big/ \sum_{e' \in E} e^{z_{e'}}$.

3.4 Training and Evaluation

The I/O encoder and R3NN models are trained end-to-end over the training set of 2 million programs sampled from the DSL. Because of training complexity of the models, the size of the programs is currently limited to 13 (number of AST nodes) for Neural FlashFill and limited to 3 concats for NACIO. We perform tests on two types of generalization: (i) Input-output generalization: we test the performance of the model on 1000 randomly sample programs that the model has seen during training but with different input-output examples (`Train`) and (ii) Program generalization: performance on 1000 randomly sampled programs that the system has not seen during training (`Test`). We also evaluate the performance of both the systems on real-world FlashFill benchmarks as well. The results are shown in Figure 5.

The Neural FlashFill system after being trained on synthetic programs of size upto 13 is able to successfully synthesize both programs (upto size 13) that it has seen during the training but with different input-output examples, and programs that it has not seen during the training. The 1-best strategy yields an accuracy of about 60% whereas it increases to 94% with 100-samples. Moreover, it is also able to learn desired programs for 55 (23%) of 238 real-world FlashFill benchmarks. The NACIO system is able to get an accuracy of about

41% on Train set and 33% on Test set. Note that the NACIO DSL allows for a much richer class of transformations using complex API functions. However, the NACIO system perform much better on the FlashFill benchmark set with the success rate of 37% with 100-samples. A majority of the unsolved FlashFill benchmarks belong to the category of programs larger than the ones the two systems are trained on.

4 Future Challenges and Other Applications

There are several exciting research challenges in scaling the neural program synthesis framework for larger programs and for synthesizing programs in richer and more sophisticated DSLs. We briefly discuss a few of these directions and also discuss other program analysis applications that can be enabled by such a framework.

Scaling to Larger programs: The current complexity of both the I/O encoder and the R3NN model limits the training capacity to only programs upto a fixed size. We would like to explore new encoders and tree generation architectures for more efficient training.

Modeling program states: The DSLs we have considered till now are functional and do not model stateful assignments. One interesting challenge in the R3NN network is to encode variable-dimensional program states and the imperative state update semantics.

Reinforcement Learning for R3NN: We currently use the supervised training signal to teach the network to generate syntactically similar programs. A key extension to this would be to allow the model to learn semantically equivalent programs (resulting in infrequent reward signal). We believe reinforcement learning techniques can be useful in this setting.

More sophisticated specification encoders: We have only developed a few simple specification encoders for one kind of specification mechanism, i.e. input-output examples over strings. One extension of this would be to consider other data types such as integers, arrays, dictionaries, and trees. Another important extension would be to handle multi-modal specification such as natural language descriptions, partial programs, assertions etc.

Combining Neural approaches with symbolic approaches: The neural architectures are good at recognizing patters in the specifications, but are not good at modeling complex program transformations. A combination of neural architectures with logical reasoning techniques might alleviate some of the function modeling issues.

Learning to Superoptimize: A recent approach based on reinforcement learning was proposed to guide the superoptimization of assembly code [8, 7]. We can use program synthesis for super-optimization with reference implementation as the specification.

Neural Program Repair: The neural representation of programs can also aid in program repair. Several recent approaches learn a language model over programs to perform syntax correction over programs [3, 15, 31]. Neural program synthesis techniques can be extended with distance metrics to correct semantic errors in the programs.

Neural Fuzzing: Fuzzing has proven to be an effective technique for finding security vulnerabilities in software [10]. These techniques have shown impressive results for binary-format file parsers but not for more complex input formats such as PDF, XML etc. parsers, where a grammar needs to be written to define the input formats. Neural architectures can be developed to automatically learn these grammar representations from a set of input examples [11], and the learning can further be guided using metrics such as code coverage.

5 Conclusion

The problem of program synthesis is considered to be the holy grail of Computer Science. Although there has been tremendous progress made recently, the current approaches have either limited scalability or are domain-specific. In this paper, we argued that some of these limitations can be tackled using a learning-based approach that learns to encode specifications and to generate programs from a DSL using a large amount of training data. We presented two preliminary instantiations of the neural program synthesis approach, but we believe this approach can dramatically expand the applicability and effectiveness of Program Synthesis.

References

1 Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shamwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

2 Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.

3 Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *CoRR*, abs/1603.06129, 2016.

4 Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Deep API Programmer: Learning to Program with APIs. *CoRR*, 2017.

5 Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: probabilistic model for code. In *ICML*, pages 2933–2942, 2016.

6 Alan W. Biermann. The inference of regular lisp programs from examples. *IEEE transactions on Systems, Man, and Cybernetics*, 8(8):585–600, 1978.

7 Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. Learning to superoptimize programs. *CoRR*, abs/1611.01787, 2016.

8 Rudy R. Bunel, Alban Desmaison, Pawan Kumar Mudigonda, Pushmeet Kohli, and Philip H. S. Torr. Adaptive neural compilation. In *NIPS*, pages 1444–1452, 2016.

9 Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016.

10 Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

11 Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. *CoRR*, abs/1701.07232, 2017.

12 Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014. URL: `http://arxiv.org/abs/1410.5401`.

13 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

14 Sumit Gulwani, William Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, Aug 2012.

15 Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. DeepFix: Fixing common C language errors by deep learning. In *AAAI*, 2017.

16 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *ECCV*, pages 630–645, 2016.

17 Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, 2016.

18 Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

19 Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

20 Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 190–198, 2015.

21 Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

22 John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

23 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

24 Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2015. URL: `http://arxiv.org/abs/1511.06392`.

25 Zohar Manna and Richard J. Waldinger. Synthesis: Dreams – programs. *IEEE Trans. Software Eng.*, 5(4):294–328, 1979.

26 Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

27 Zohar Manna and Richard J. Waldinger. Fundamentals of deductive program synthesis. *IEEE Trans. Software Eng.*, 18(8):674–704, 1992.

28 Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*, 2015.

29 Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *CoRR*, abs/1611.01855, 2016.

30 Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126, 2015.

31 Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for moocs. *CoRR*, abs/1607.02902, 2016.

32 Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "big code". In *POPL*, pages 111–124, 2015.

33 Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2016. URL: `https://arxiv.org/abs/1511.06279`.

34 Sebastian Riedel, Matko Bosnjak, and Tim Rocktäschel. Programming with a differentiable forth interpreter. *CoRR*, abs/1605.06640, 2016. URL: `http://arxiv.org/abs/1605.06640`.

35 Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic program optimization. *Commun. ACM*, 59(2):114–122, 2016.

36 Armando Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Dept., UC Berkeley, 2008.

37 Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

38 Phillip D. Summers. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

39 Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In *PLDI*, pages 287–296, 2013.

40 W. Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. The microsoft 2016 conversational speech recognition system. *CoRR*, abs/1609.03528, 2016.