Integrating Reinforcement Learning Agents for Error Correction in Abstract Syntax Trees

Peter Lu, Sophia Sanchez, Yousef Hindy, Maurice Chiang, Michael Smith
Computer Science Department
Stanford University
353 Serra Mall, Stanford, CA 94305
{peterlu6, sophials, yhindy, mauricec, msmith11}@stanford.edu

Abstract

Automated code synthesis is one of the holy grails of computer science. Despite the promise of code synthesis, many attempts to solve the problem fail largely due to the enormous state space of all programs and the sparsity of the reward space for partially functioning generated code. We introduce a neural architecture that allows for error-correction in abstract syntax trees (ASTs). To do so, we model the process of writing a program as a type of game, where the action space is limited to a series of non-differentiable operations on ASTs. The network, acting as an agent equipped with a prosthetic integrated development environment (IDE), is trained by meta-reinforcement learning algorithms that allow the agent to narrow the search space and find programs more efficiently than brute force search. By making syntactically correct adjustments to an AST, the agent receives feedback from the IDE and adjusts its policy accordingly. To solve the issue of sparse and infrequent rewards, we propose a stack trace evaluator running a modified Smith-Waterman algorithm against the ground truth stack trace. As a proof of concept, we propose that this architecture be tested on simple programs with limited arguments and outputs like sorting, merging, and concatenating lists.

1. Introduction

Program synthesis involves software that is able to help humans write and debug software, which in turn would greatly speed up development time. It can be thought of as an intelligent prosthesis that extends the programmer through interaction and suggestion. The present work aims to address one critical component of code synthesis: error correction. We choose to approach this by using abstract syntax trees (AST), which are data structures meant to capture the syntactical formation of programs. When ASTs are correctly formed, the compiler's code generator creates an intermediate representation that can be executed by the machine. In this paper, our model takes in as input partially complete or incorrect ASTs alongside an embedding of desired input/output pairs and then performs imagination based planning in a highly restricted action space to output the well-formed and semantically relevant target AST.

Our approach is novel because it enables us to capture additional complexity in program structure by restricting the action space significantly. Importantly, this approach still maintains the flexibility to represent all possible ASTs. This is done by modularizing algorithms into multiple possible controllers, each trained separately, representing different kinds of strategies that the model can take to imagine and/or approach a problem. While the proposal is theoretical in nature, this constrained architecture can serve as a robust platform from which to make significant progress in the domain of error correction.

2. Related Work

Before the rise of machine learning, researchers attempted to generate handmade heuristics that would limit the size of the search space. Recently, however, with the improvement of artificially intelligent systems, there has been a rise in neural methods to solve the problem. Dan Abolafia et. al. of Google Brain presented a novel technique [1] that uses a RNN equipped with a priority queue of programs to solve simple problems like reversing a list, removing characters from a string, etc. Their approach is limited to the minimalist programming language BrainFuck (BF) [14], which only has 8 operations, and the results indicate that the system needs to be equipped with a less-sparse and more informative reward function.

Apart from building programs from I/O pairs, other attempts at program synthesis include Bhatia and Singhs contribution of a neural architecture that uses an RNN to generate repair feedback on student coding assignments.[3] Similarly, Microsoft Research proposed a method using RNNs to fill in incomplete code-snippets by leveraging several networks that perform specific functions like tree parsing or determining context.[2]

Many of these approaches work well on specifically tailored problems, but struggle to generalize beyond the training examples it was given. One of the most exciting recent developments has been the introduction of recursion as a base strategy for the neural network. Cai et. al [4] present a network that is able to generalize to novel inputs as a result of built in methods of recursion. For example, while other baseline models go from attaining high accuracy on sorting problems with short arrays to low accuracy on longer arrays, the recursive method is easily able to attain 100% accuracy regardless of the length of the array. This algorithm is able to perform such a task by dividing the problem into smaller pieces, dramatically reducing the domain of each neural network component, and as a result, making the overall systems behavior consistently predictable.

There have been many promising innovations in the space of program synthesis and error correction, but the problem of the programmer's apprentice remains. We aim to apply the best aspects of these recent advances into a cohesive and robust programmer's apprentice to produce accurate code synthesis even on long code samples and complex input-output patterns.

3. Methods

In this section, we outline the general plan for our model, detailing the necessary components and how they fit together. In order to narrow the scope of the problem for the purposes of model design and testing, we limit the number of arguments taken as input to a function to two, and assume that both are lists of arbitrary length. Similarly, we define the output as a single list. By restricting the inputs in this way, we can still represent key computing tasks such as searching and sorting, without handling the difficulties associated with an unrestricted input.

3.1. Model

Our model consists of several RL agents working in unison. At a high level, we have an Imagination Based Planner (IBP) [10] "manager" that acts as the main controller of the overall agent. It accepts partially complete or incorrect ASTs along with an embedding of desired input/output (I/O) pairs. We slightly modify the implementation of the IBP in the original paper by introducing the idea of multiple possible controllers, which is explained in Section 3.3. To evaluate the chosen decisions, we fit our model with a prosthetic Python interpreter that can run the new AST. The stack trace generated is then compared with the ground truth to generate a reward for the agents. The components are explained in detail below.

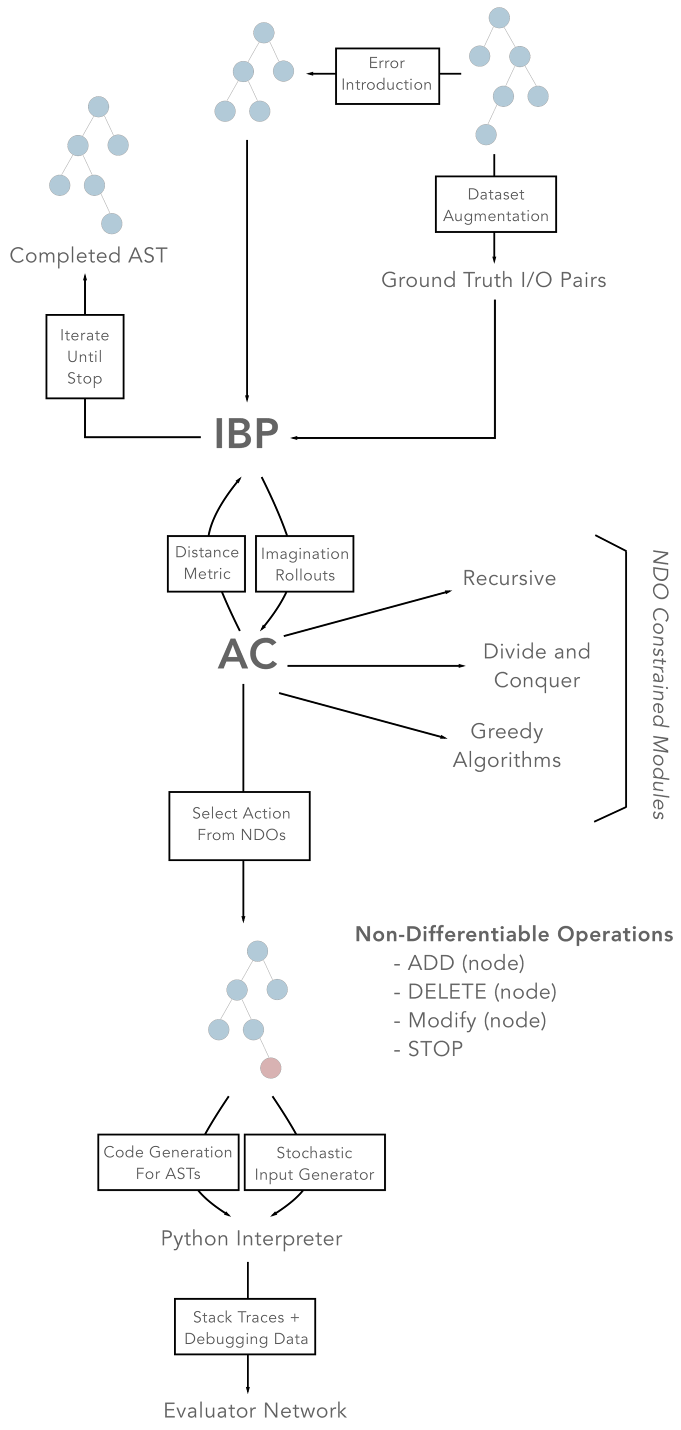

Figure 1. Overall model architecture

3.2. Data input

We have an existing dataset of valid Python ASTs [12] that is outlined in Section 4. From here, we perform data augmentation by generating ASTs with removed or modified nodes. To simulate having nearly correct code, we will only modify nodes that are leaves or parents of leaves. In addition, we will narrow our scope by filtering out ASTs that do not accept lists as input. To do so, we will use `codegen` [6] to create Python source code for each correct tree. Then, we will feed in two lists of integers into each code sample, and cross-check the resulting stack trace against Python documentation for fatal runtime errors. If there are none, the tree is added as a valid AST to the dataset.

This pruning will allow us to generate ground truth I/O pairs by randomly generating lists of varying sizes and feeding them into the remaining programs. The corresponding outputs will then be saved along with the inputs. We then feed this information into the manager of the IBP in the form of a tuple (`incorrect_AST`, `input1`, `input2`, `output`), where `incorrect_AST` is fed in after going through a subtree embedding network as in Piech. et al [11] and `input1`, `input2`, and `output` are Python lists represented as tensors.

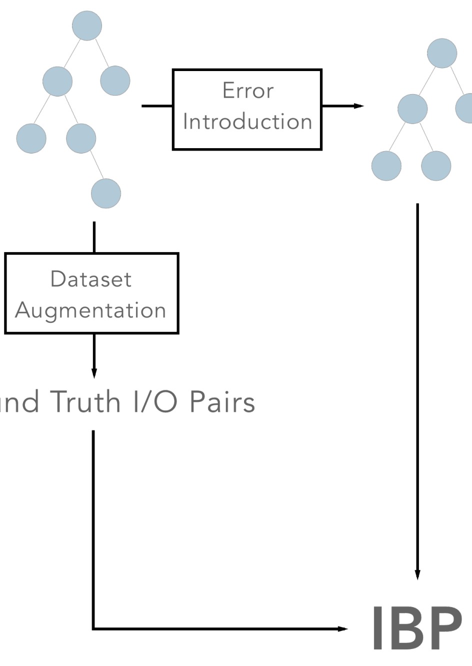

Figure 2. Data input closeup

3.3. Imagination Based Planning

On each iteration i, the IBP either executes an action in the environment by directly modifying the current AST or imagines what would happen if it took a certain action by evaluating an internal model of its environment, represented here as the prosthetic Python interpreter. In this section, we explain the different components of the IBP in detail. For simplicity, we follow the notation of the original paper [10], with actions in the real environment indexed by j and imagined actions indexed by k. The model has access to information about both external states (states s_j, actions a_j, rewards r_j) and internal states (states $\hat{s}_{j,k}$, actions $\hat{a}_{j,k}$, rewards $\hat{r}_{j,k}$), which are stored by iteration in d_i. The model is initialized with s_0 being the given incorrect AST.

Manager The history of all actions $h_i = (d_0, \dots, d_i)$ is passed to the manager $\pi^M : \mathcal{H} \to \mathcal{U} \times \mathcal{C}$, which takes in a history h and returns a route $u \in \mathcal{U}$, which denotes whether or not the model will act or imagine. It is important to note that when deciding to imagine, the manager also decides which state to imagine from, allowing it to replay previous episodes and make rational decisions to explore alternate actions if the current tree edit being imagined leads to a fatal runtime error, infinite loop, or a large regression in stack trace performance. It also returns $c \in \mathcal{C}$, which decides which action controller will be needed for the step at hand. $\mathcal{C}$ is the set of all action controller types that are explained below.

We envision that the manager would be implemented as a many-to-one LSTM [9] that is trained in a meta-reinforcement learning framework [15] so that we can process the information keeping temporal relationships in mind.

Action Controller Next, the route is passed to the corresponding action controller. First, we will describe the generic action controller and then we will explain the details that differentiate them. The controller $\pi^C : \mathcal{S} \times \mathcal{H} \to \mathcal{A}$ takes in the current (or imagined) state along with a history of what has happened and returns the appropriate action to take in the situation. To limit the search space of the problem and ease training, we limit the action space to be four non-differentiable operations:

1. ADD (node)
2. DELETE (node)
3. MODIFY (node, new value)
4. STOP

The different action controllers are meant to represent the different kinds of programming strategies one can use when tackling a problem. For example, we have devised models for recursion, divide and conquer (D&C), and greedy algorithm modules. Each module will essentially be a Differentiable Neural Computer (DNC) [7] that is equipped with external memory and also trained in a similar fashion to the manager using a meta-reinforcement learning paradigm [15].

Each module, except for the recursive model, will be created by hand-picking a set of problems and training each module on a representative dataset for each algorithm. At this stage, we are not aware of any automated way to do this, so this step may require extensive human tagging of programs. For the recursive model, we will follow a similar architecture and training methodology to [4], using a controller that picks from a certain set of functions that allow for recursion.

Imagination Next, the imagination $\mathcal{I} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S} \times \mathcal{R}$ maps the proposed action and current (or imagined) state to what the next state would be and the reward gained. In our case, the imagination will consist of the prosthetic Python interpreter accompanied by our distance metric explained in Section 3.4. In particular, the prosthetic will generate a stack trace and output that will be compared to the ground truth. The reward $r \in \mathcal{R}$ will be propagated back through the network during training. The advantage of such an imagination module is that during training, the network will be able to explore certain paths without totally destroying the AST and making repair intractable. Instead, it will have opportunities to make mistakes and see the outcomes in a controlled manner.

Unlike the original paper [10], we forgo the memory module μ. We do so because each module already has its own implementation of a memory. In particular, we intend for the different kinds of action controllers to have different memories, as they will need to act in differing manners.

In summary, the managers role is to categorize the current AST tree into the appropriate problem class (recursion, divide and conquer, greedy algorithm, etc) and direct the appropriate submodule to interact with the tree. At test time, the weights of the network will be fixed and the network will adapt to changing inputs through modulating activity at each time step, in the same philosophy as meta reinforcement learning [15]. This structure relies heavily on the cogency of the submodules and evaluation network to provide meaningful reward signals.

By providing separate memory banks for each submodule, they will be able to maintain unique understandings of each type of algorithm, and can serve as useful subcontained interfaces for the IBP to treat as blackboxes. By creating a hierarchy of blackbox abstraction of action space (NDOs and algorithmic classification), we build on Dawn Song's insights into NDOs. We also enable both the submodules and the IBP to reap the benefits of a highly restricted action space, which stays in the realm of valid trees and provides a useful reward in the form of evaluation scores.

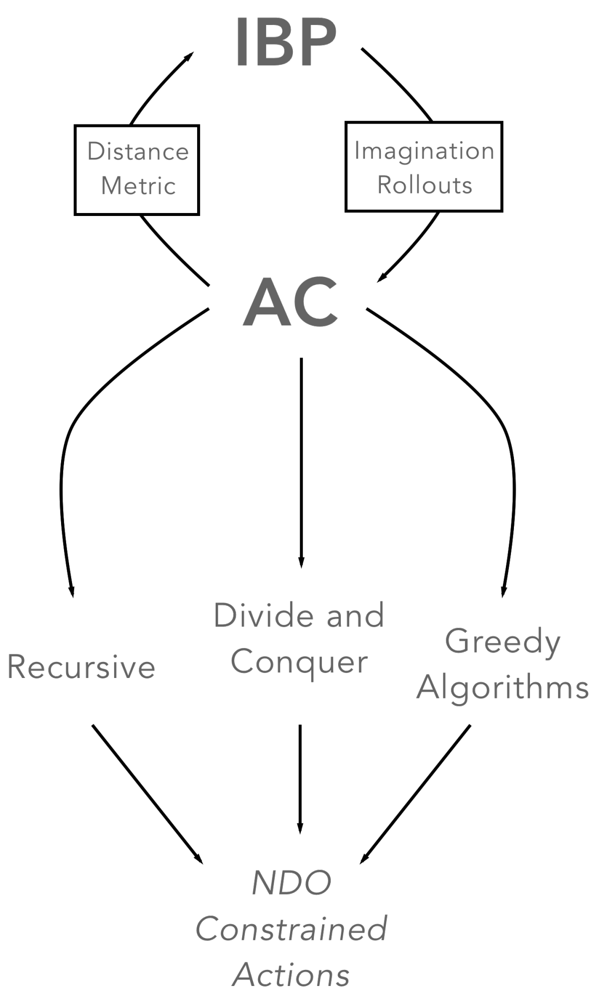

Figure 3. Action controller closeup

3.4. Evaluation

For the action controller to select the optimum tree modification action to take, the imagined AST tree must be evaluated. Creating valuable intermediate reward signals is a huge challenge in program synthesis, due to the sparsity of the input-output space, and difficulty deriving signals from syntactically broken code. We remedy this by evaluating the stack trace output from code matching our AST, to ensure that a reward signal can always be derived, even if the code has runtime errors.

This can be done using the `codegen` package [6] to produce source code from the AST, and a stochastic input generator to produce a variety of input examples across the constrained input space (two arguments, each must be list of numbers). This source code and input examples can be fed into a standard Python interpreter, and the resulting stack trace and debugging information can be fed into an evaluation network to determine a metric of success for the particular partial AST.

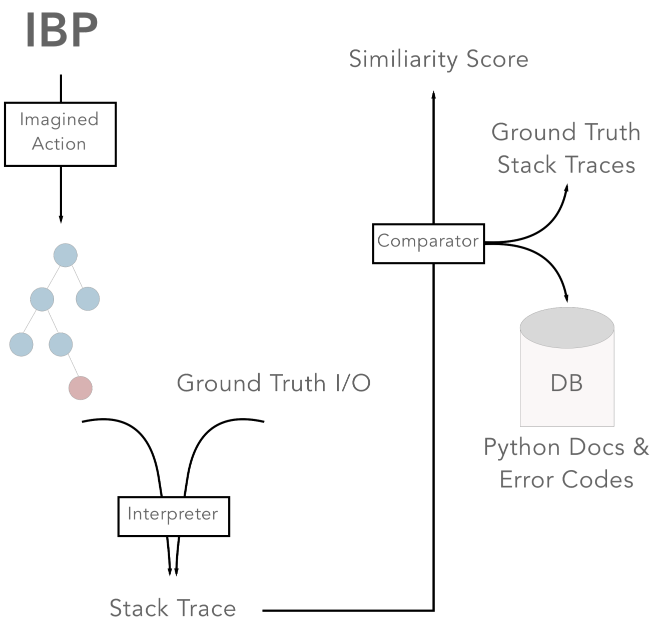

Figure 4. Evaluation module closeup

The evaluation network will take the text containing the stack trace, send it through a parser, removing uninformative information such as function names, and compare the two stacks using a global alignment schema [8]. The network will also match error codes found against a database of Python documentation through regular expressions, enabling it to determine when the current search path has hit a dead-end and should terminate. Specifically, the best global alignment between the given stack trace and the ground truth stack trace is defined as the maximum match (subsequence in the given stack trace that can be matched to the ground truth stack trace), allowing for gaps. This approach utilizes a dynamic programming algorithm and a cost function with three primary considerations:

1. The importance of a symbol in the stack trace
2. The proximity of the query symbol to the top of the stack trace. Symbols closer to the top of the trace tend to have greater importance.
3. The extent of the gap between the symbol in the given stack trace and the ground truth stack trace.

To this goal, we employ a modified version of the Smith-Waterman algorithm [13], where the edit distance is weighted by the depth of the stack trace from which the current edit distance is derived.

Modified SW, with a modifier multiplier λ:

$$\begin{aligned}D_w(S_i, T_j) = \lambda \cdot \min\{&D_w(S_{i-1}, T_{j-1}) + w(s_i, t_j),\\ &D_w(S_{i-1}, T_j) + w(s_i, \phi),\\ &D_w(S_i, T_{j-1}) + w(\phi, t_j)\}\end{aligned}$$

$D_w(S_i, T_j)$ represents the edit distance between the first i characters of S and the first j characters of T. $w(a, b)$ is the error metric between character a and character b, and ϕ in this case represents a gap (i.e. character misalignment).

We then normalize this as in the patent to account for different stack trace lengths:

$$S_E(S, T) = \frac{\max(|S|, |T|) - D_w(S, T)}{\max(|S|, |T|)}$$

4. Dataset

The dataset consists of 150,000 parsed Parsed ASTs sourced via the Machine Learning for Programming project [12]. Programs were originally scraped from GitHub repositories and converted using the Python AST parser. All programs used had permissive and non-viral licenses, such as but not limited to MIT or Apache. Duplicate files, project forks, and programs with more than 30,000 nodes in the AST were removed.

We augment the dataset to introduce errors for training and testing. We limit the introduced errors to semantic, rather than syntactic errors. To produce these errors, a script will enumerate the set of nodes that is either a leaf of the AST or an immediate parent of a leaf. Then, a node will be randomly chosen to be deleted or have its value modified. If a parent of a leaf is deleted, its children will be deleted as well. This will primitively represent a logical block being removed, such as a for loop. Future directions could see nodes higher up in the tree being changed, producing larger scale errors in the code.

5. Training

To reduce training complexity, the submodules of the action controller will be trained separately, on restricted datasets that represent canonical examples of the domains (i.e. the recursion module will receive recursion code generation problems). The submodules are permitted a restricted action space within the NDOs described earlier. We anticipate the DNC memory structure will enable these modules to store specific patterns associated with each type of algorithmic problem. By training the submodules first, we can treat them as self-contained operations for the action controller.

Once the submodules are trained, the action controller itself will be trained, this time across all algorithmic classes of problems. The weights of the DNCs for the submodules will be frozen to maintain the structure learned in the previous learning stage, while the action controller learns to distinguish between the differing classes of problems.

During the entirety of the training phase, the evaluation module will be used as reward feedback for the DNC.

6. Testing

During testing, we simply freeze the weights of the action controller DNC as well as the submodules, and feed in the AST-I/O tuples as a timeseries. After the time series terminates, the network will generate a single node action on the AST after evaluating the imagination rollouts, and this AST will be recurrently fed back into the planner to evaluate the next step.

We look to a variety of metrics for success here to determine the utility of the model architecture. Beyond the fundamental metric of program correctness, the usefulness of the action controller is critical to the novelty and success of this architecture. We expect a well-trained action controller to determine precisely what submodule needs to run on the problem at any given timestep. Thus, imagination rollouts with less branching are representative of a more highly trained action controller than a highly branched rollout.

This can also be evaluated qualitatively on the final output by checking the length of the code compared to the ground truth AST, which should match if the appropriate optimal algorithms are applied at the correct stages of the problem.

7. Discussion

We think of a Programmer's Apprentice as a software tool that can interact with a programmer and augment his/her work by capturing the design rationale behind his/her software while detecting and correcting common flaws in the program design. Our code correction algorithm attempts to approach this goal by correcting simple code that contains semantic errors. Although we focused on a subtask within the Programmer's Apprentice problem, the present work does have a few notable restrictions. We limit the number of arguments taken as input to a function to two and restrict the outputs to tasks like sorting, merging, and concatenating lists. Additionally, we only introduce semantic errors, avoiding syntactic errors – we do this to bound the complexity of the problem, and allow for better compatibility with the game engine.

Although the scope of the problem addressed here is limited – it represents measurable progress towards a larger goal. We believe that future work in this area should focus on the broader applicability of such a technology, and not be restricted to the specifications required for our algorithm. For example, in a real-world software development environment, it would be unrealistic to have such input/output restrictions, as the programs that we limit our model to account for a small subset of the possible space of real-world scenarios. To handle this increased range of inputs, a novel way to generate a representative input set is needed. Increased generalizability goes hand in hand with increased complexity – in future work, we might look to implement other heuristics to constrain the search space. However, ultimately, even with a higher degree of generalizability, targeted repairs constitute an important component strategy as a part of a more comprehensive solution.

There is also potential to add more structure to the I/O inputs by running them through an invariant recognition network, with a Prolog-logic module [5]. This will enable the planner to ingest the underlying invariants for the outputs. For example, a sorting algorithm ground truth would have an invariant for every output where the list was sorted. If the network can process this invariant, it will maintain a much stronger reward signal that accurately represents the actual intent of the input-output data.

Another area of innovation could be the way in which algorithmic modules are trained. This still requires manual curation of canonical problems for each algorithm, which is nontrivial and may not yield ideal training results. One can envision a clustering system that can identify which ASTs occupy the same algorithmic space, and dynamically create modules with varying architectures adapted to the problem rather than using a catch all DNC structure and relying on memory storage to capture learned problem details.

Integrating RL agents for error correction in ASTs is a key component of the Programmer's Apprentice problem. While much remains to be done with regards to broadening the scope of the proposed architecture, the model represents an advancement towards the ultimate goal of automated code synthesis.

References

[1] D. A. Abolafia, M. Norouzi, and Q. V. Le. Neural program synthesis with priority queue training. *CoRR*, abs/1801.03526, 2018.

[2] M. Allamanis and M. Brockschmidt. Smartpaste: Learning to adapt source code. *CoRR*, abs/1705.07867, 2017.

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

[4] J. Cai, R. Shin, and D. Song. Making neural programming architectures generalize via recursion. *CoRR*, abs/1704.06611, 2017.

[5] W. Dai, Q. Xu, Y. Yu, and Z. Zhou. Tunneling neural perception and logic reasoning through abductive learning. *CoRR*, abs/1802.01173, 2018.

[6] A. Fokau. andreif/codegen, May 2015.

[7] A. Graves and G. Wayne. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538:471476, Oct 2016.

[8] R. Gupta. System and method for matching a plurality of ordered sequences with applications to call stack analysis to identify known software problems, Nov 2010.

[9] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[10] R. Pascanu, Y. Li, O. Vinyals, N. Heess, L. Buesing, S. Racanière, D. P. Reichert, T. Weber, D. Wierstra, and P. Battaglia. Learning model-based planning from scratch. *CoRR*, abs/1707.06170, 2017.

[11] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. J. Guibas. Learning program embeddings to propagate feedback on student code. *CoRR*, abs/1505.05969, 2015.

[12] V. Raychev, P. Bielik, and M. Vechev. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731747, 2016.

[13] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[14] S. Varma. brain-lang/brainfuck, Mar 2017.

[15] J. X. Wang, Z. Kurth-Nelson, D. Kumaran, D. Tirumala, H. Soyer, J. Z. Leibo, D. Hassabis, and M. Botvinick. Prefrontal cortex as a meta-reinforcement learning system. *bioRxiv*, 2018.