Imitation Learning for Code Generation via Recurrent State Space Embeddings

Marcus Gomez Nate Gruver Michelle Lam Rohun Saxena Lucy Wang

1. Introduction

In their 1988 paper, Rich and Waters define the Programmer's Apprentice as an "intelligent computer program that functions like a human support team," providing guidance in requirements, design, and implementation of programs [21]. The Programmer's Apprentice is differentiated from a simple code auto-completion agent in its ability to understand software design considerations and engage in helpful dialogue with the software engineer [23].

As we considered the problem of the programmer's apprentice, we came to agree that interfaces are among the most important structural elements of code. In fact, in considering our own coding habits, we realized the most helpful assistant would be one that can comprehend design through the structure of an interface and assist in the completion of the code through a divide-and-conquer approach.

In this work, we focus solely on the goal of implementation, proposing a novel framework for code generation that, given a sketch of helper function interfaces, suggests both use of primitive language operations and structuring of programs through decomposition.

2. Related Work

2.1. Related neural program synthesis tasks

Neural program synthesis focuses on generating explicit programs that meet a particular semantic specification. A large body of work in program synthesis has focused on program generation given a limited set of input and output examples. This task formulation enables the synthesis and evaluation of a variety of programs as long as they accomplish the same functionality (avoiding the problem of program aliasing), but must overcome the problem of generalization to input-output pairs outside of the training set. Recent work has discovered that incorporating the notion of recursion into neural program synthesis architectures significantly improves their generalizability while enabling formal provable guarantees that a generated program can generalize perfectly [3]. Other work has layered reinforcement learning on top of supervised techniques to reward the generation of semantically correct programs rather than solely rely on similarity to a single target program (which may incorrectly penalize differing semantically correct programs) [2]. Given the success of these approaches for the input-output pair task formulation, we similarly take advantage of structural hierarchies in code and apply RL rather than supervised approaches to generate semantically correct programs.

Meanwhile, neural program induction aims to design and train neural architectures that can mimic the behavior of a desired program. The neural programmer interpreter (NPI) is a neural program induction algorithm that consists of a task-agnostic core that decides what subprograms or primitives to invoke and includes domain-specific encoders designed to operate in diverse coding environments [20]. A benefit of this approach is that a single NPI can learn from a small number of examples and leverage previously-learned programs to complete new tasks. Recent work in robot learning has built upon this technique to take task demonstrations, recursively decompose them into sub-tasks, and feed the decomposition into an NPI [26]. Founded upon the notion of meta-learning to learn how to instantiate neural programs through recursive decomposition of a task, this method has demonstrated strong generalization on hierarchical tasks. While we choose to focus on neural program synthesis to explicitly generate compilable, interpretable code, we take inspiration from this method whereby a controller delegates the generation of each subfunction by feeding in a subset of the task specification.

2.2. Program synthesis from structured input

Furthermore, prior work achieves promising results in the domain of program synthesis by adding structure to the input and output. For the related program translation task in which code is translated to another language, existing work has utilized the parse trees of input and output programs to develop tree-to-tree translation models and achieve state-of-the-art performance [6]. In program synthesis, recent approaches have utilized syntax checkers or neural syntax models to condition on the syntactical correctness of a program and significantly prune the large search space of potential programs [2]. Prior work has also experimented with domain-specific structural input constraints by generating code representations of cards and their functionality in

the Magic the Gathering game [16]. Other work adds structure to the code representation itself via a canonical traversal of the output AST (abstract syntax tree) and constrains the prediction problem to the identification of this traversal [7]. Meanwhile, other work utilizes a custom DSL (domain-specific language) to predict the AST and evaluates effectiveness in a novel way by computing coverage (number of words in the natural language description mapped to actual code), mapping (likelihood of NL to code mapping correctness), and structure (similarity in structure between the code and the NL description) [9]. Additionally, a recent approach has proposed a two-step process of first generating a program sketch from minimal amounts of structured code information, and then, generating code from the program sketch. This enables separation between high-level semantics and program specific idiosyncrasies [18].

2.3. Program synthesis from natural language

One of the more challenging problems in the program generation domain is transforming natural language descriptions directly to code. Previous work has used grammar structures in natural language to generate abstract syntax trees [27]. Additionally, there have been efforts to translate text to specific scripting applications: for example, encoder-decoder networks with attention have been used to generate shell script from natural language [15], and a two-stage latent attention model has been used to generate IFTT programs from English descriptions [4]. More recently, work has been done to generate SQL queries from natural language, leveraging an reinforcement learning technique to learn a policy from pointer networks applied to the natural language sequence and data frame header [28]. Other approaches have used a combination of natural language and input-output examples (and unit tests) in effort to generate more accurate programs [19] [8].

In addition to code synthesis from natural language, the reverse problem been explored: given a code input, generate a natural language output description. Multiple works have generated datasets that map code to language descriptions [13] [11]. Translation approaches have included using LSTMs to convert StackOverflow code snippet answers to their corresponding English questions, as well as using neural machine translation architectures to generate git diffs from commit messages [12][11].

At a high level, translation attempts involving natural language and code have generally tackled simple code snippets: simple functions, short queries, or even single lines of code. To our knowledge, there is sparse precedent for translation between natural language descriptions and complex programs with multitudes of interacting functions.

3. Problem Formulation

```
// Main function that creates a HashMap of first names
// mapped to phone numbers given a list of full
// names and phone numbers
def main(names: Array[String], numbers:Array[String]){
    first_names = get_firstnames(names)
    clean_numbers = get_numbers(numbers)
    address_book = create_mapping(first_names, clean_numbers)
    return address_book
} -> HashMap[String, Int] //return type

// Takes a list of full names and returns a list
// of the first name space separated
def get_firstnames(names: Array[String]){
    remove_punctuation(names)
    //rest filled in by the library and NL intent
} -> Array[String]

//applies a series of functions to list of strings
// returning the cleaned numbers
def get_numbers(numbers: Array[String]){
    numbers = remove_punctuation(numbers)
    numbers = add_area_code(numbers, "650")
    numbers = add_country_code(numbers, "1")
    numbers = toInt(numbers)
} -> Array[Int]

//Removes all punctuation from a list of strings
def remove_punctuation(numbers: Array[String]){
    //Filled by NL intent and library of modules
}

//Adds default area code passed in to each number in list
// if length is less than 10
def add_area_code(numbers: Array[String], default: String){
    //Filled by NL intent and library of modules
} -> Array[String]

//Adds default country code passed in to each number in list
def add_country_code(numbers: Array[String], default: String){
    //Filled by NL intent and library of modules
} -> Array[String]

//Converts every entry in list from a String to an Int
def toInt(numbers: Array[String]){
    //Filled by NL intent and library of modules
} -> Array[Int]

//make a map with keys as first list and values as second list
def create_mapping(first: Array[String], numbers: Array[Int]){
    //standard function that will be in library of modules
} ->  HashMap[String, Int]
```

Figure 1. Example of the header file that would be provided by the programmer.

In considering the Programmer's Apprentice problem, we sought to design an agent that could integrate seamlessly into a programmer's work flow. We deviated away from canonical designs that require copious interaction between the human and the agent [23], in favor of a design that allows the Apprentice to remain unobtrusive. In our framework, the human provides a design architecture as a

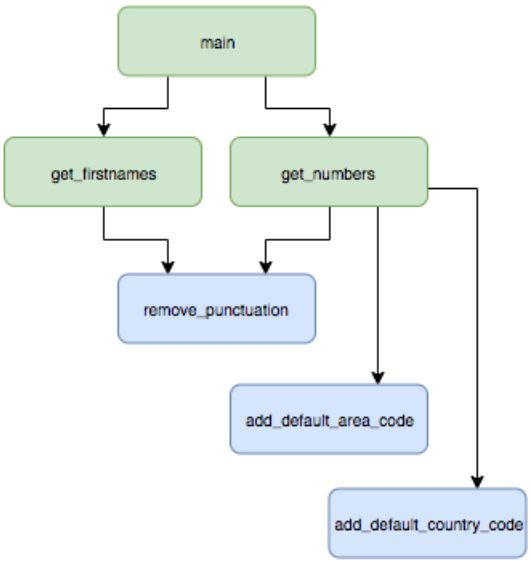

Figure 2. Example of a dependency DAG that would be passed in as an input. Leaf nodes (blue) will not call any user defined functions.

header file (a format familiar to any programmer), and the Assistant automatically fills in the rest.

Formally, we define the problem as follows. As input, the architecture takes in a header / prototype file, containing the types of every function defined by the user, and corresponding natural language descriptions of each function (see Figure 1). As additional input, we define what we call a dependency graph, which is a DAG; each node in the graph represents one of the functions, and a directed edge from function f_1 to function f_2 indicates that f_1 may call the function f_2 in execution (see Figure 2.* Lastly, the code that has been written for all of the children of a given function f will be passed in as input as well.

More formally, let the primitive types of the DSL be contained in the set T. Define the DAG as some graph $G = (F, E)$, where F are the user defined functions and E are the aforementioned between them. We assume that every node in the DAG can have at most K_C children, that the DAG has a single root, and that the longest path from the root to a child is at most length K_P, where K_C and K_P are chosen based on the dataset.

Then, each function can be represented as the tuple $f = (t_i, t_o, NL, C_f)$, where $t_i, t_o \in T$ are the input and output types, $NL \in \mathbb{R}^{N \times M}$, where N is the size of the natural language embedding (here, we opt to use GLoVE vectors), and M is the maximum number of allowed tokens in the natural language description (i.e. each column corresponds to a word in the NL sequence), and C_f is a $K_C \times L_E$ vector where L_E is the length of the flattened vector embedding of helper function code. Each row in C_f consists of the embedding of a helper function, f', in order of a left-most traversal of the dependency graph, G, at the node corresponding to f.

As output, the architecture will generate code for each function defined by the user in the header file. It will do so by imitating the expert code examples. Learning takes place through a mixture of adversarial training with provided inputs and reinforcement learning using reward augmentation.

4. Model Architecture

In order to leverage reinforcement learning techniques to train our code-creation network, we frame the problem as an MDP using the features described above.

4.1. MDP description

Our MDP "environment", E, has state space, S, and action space, A, where

$$S = G \times TS \times NL \times C_f \times I \times F_{\text{terminal}}$$

and

$$A = P \cup \{f_1, ..., f_{K_C}\} \cup \{a_{\text{terminate}}\}$$

where $TS = (t_i, t_o)$ and G, t_i, t_o, NL, and C_f are as above. The only new elements are I, the code that has been written so far in the implementation of the current function and F_{terminal}, a flag bit that denotes the function is intended to be a complete implementation. S thus encapsulates the semantic context of a software engineer choosing the contents of function body word by word. A is thus the set of all primitive operations P and the helper functions, $\{f_1, ..., f_{K_C}\}$, that can be called from the current function given the provided interface and $a_{\text{terminate}}$ is a special action that denotes the end of a function implementation. As in practice the state is of variable length, we employ a complex embedding scheme that yields state vectors of fixed length. Actions coming from a fixed number of options can be represented as simple one-hot vectors.

4.2. State Embedding

Formally, given that the agent is generating code for some function f, the state is comprised as follows

1. $TS = (t_i, t_o)$ (the input and output types of f, comprising the type signature)
2. NL (the embedded natural language description of f)
3. C_f (a tuple of ASTs)
4. G (the dependency graph of the user-defined functions)
5. I (the code written so far for the function f)
6. F_{terminal} (the function termination flag)

*Note that it is thus a requirement that there be no recursive function calls.

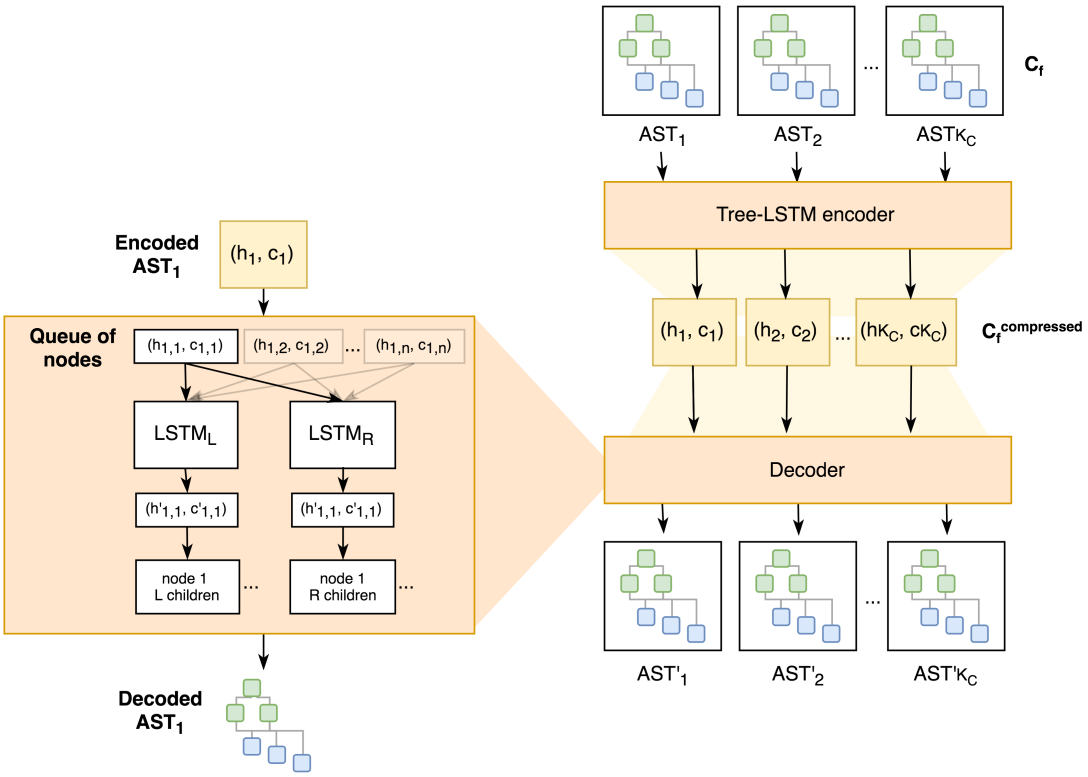

Figure 3. Summary of the proposed LSTM-based state embedding architecture for C_f.

$TS = (t_i, t_o)$ are represented as one-hot vectors of dimension $|T|$, where T is the set of all available types.

To learn a compressed fixed size state for NL (which is an ordered list of GLoVe vectors), we first train an autoencoder RNN penalized on reconstruction loss and take the output state of the encoder to be the compressed representation NL_f.

The dependency graph is represented as G_{flat}, the flattened representation of the adjacency matrix of G; importantly, we always store adjacency matrix as a matrix of size $K_C^{K_P} \times K_C^{K_P}$, and just leave entries as 0 if nodes are unused (we will, by construction, never traverse over the unused nodes).

C_f by default is variable size since the ASTs are variable size; we solve this problem by training a an autoencoder penalized on reconstruction loss; the decoded output is the binary tree representation of the original AST, converted in Left-Child Right-Sibling manner (this is done to limit the number of LSTMs needed for decoding). The encoder is a Tree-LSTM, as described in [24], and the decoder is slightly more complex; a summary of the full architecture is illustrated in Figure 3.

The decoding process follows the method of [5]. The decoded tree is initialized to just a root node, associated with output state of the the encoder Tree-LSTM, (h,c). A queue is maintained, and the decoder iteratively pops nodes and their state (h_n, c_n) from the queue and processes the current node to predict the existence and content of the children; the queue is initialized with the root node. When the decoder pops from the queue, it first computes

$$t_n = \arg\max \text{softmax}(W \cdot h_n)$$

where W is a learned parameter (this corresponds to predicting the token of the current node). Two LSTMs are maintained; $LSTM_L$ and $LSTM_R$; the the left and right children are then given as

$$(h', c') = LSTM_{L/R}((h, c), Bt_n)$$

where B is a learned token embedding matrix mapping AST tokens to a low dimensional space. In addition to all the normal tokens available in the given language for the AST, an EOS token is also included in the AST vocabulary; if the decoder predicts that the current node is the EOS token, it does not predict any more children. At the time that the RL training begins, the output state of the Tree-LSTM is taken to be the encoding of the AST. We take the final state of C_f, $C_f^{\text{compressed}}$, to be the ordered concatenation of all of the output states of the elements C_f passed through the Tree-LSTM, each a vector of length L_E.

I is encoded as $I^{\text{compressed}}$ with the same autoencoder architecture as C_f; the type checking constraint is done with reward shaping in 4.4.

The full state then, is represented as

$$S = [t_i; t_o; G_{\text{flat}}; NL_f; C_f^{\text{compressed}}; I^{\text{compressed}}; F_{\text{terminal}}]$$

4.3. Learning Environment

The state embedding of the previous section allows us to create a policy mapping from fixed length state vectors to discrete actions using a neural network. In order to train this policy network, however, we require a training environment that defines the transition model of the MDP as well the termination conditions. The transition distribution, T is simply

$$P(s_{t+1}|s_t, a_t) = \begin{cases} 1 \text{ if } I_{t+1} = I_t \cup a_t \text{ and } a_t \neq a_{\text{terminate}} \\ 1 \text{ if } I_{t+1} = I_t \text{ and } a_t = a_{\text{terminate}} \\ 0 \text{ otherwise} \end{cases}$$

and the termination function is

$$\text{Terminal}(s_t) = \begin{cases} \top \text{ if } (F_{\text{terminal}})_t = 1 \\ \top \text{ if } I_t \text{ is parsable but does not type check} \\ \bot \text{ otherwise} \end{cases}$$

The environment also describes a reward model–used for guiding the optimization of the imitation learning objective–and this is described in more detail in the following section.

4.4. Expert Trajectories

In order to perform imitation learning, we must also prepare expert state-action pairs from the data. In our case this is done by extracting the features of each state from the expert code base with an iterative procedure:

Algorithm 1: Extract Expert State-Action Pairs

Input : File of code with root (main) function f_{root}
Output: Expert state-action pairs

pairs $\leftarrow []$
Extract G from function calls starting in f_{root}
for do
  Parse NL from function header comments and embed
  Extract TS, C_f from the code and dependency graph and embed C_f
  for do
    Extract I, F_{terminal}, and action, a, from the code
    Embed I then concatenate together TS, G_{flat}, NL_f, $C_f^{\text{compressed}}$, $I^{\text{compressed}}$ and F_{terminal} as s
    Add (s,a) to the list of expert state-action pairs
  end
end
return pairs

Algorithm 2: GAIL with Reward Augmentation

Input : Expert trajectories $\tau_E \sim \pi_E$, initial policy and discriminator parameters θ_0, w_0
Output: Parameters of the policy and discriminator networks, θ and w

for $i = 0, 1, 2, ...$ **do**
  Sample trajectories $\tau_i \sim \pi_{\theta_i}$
  Update the discriminator parameters from w_i to w_{i+1} with the gradient

$$\hat{\mathbb{E}}_{\tau_i}[\nabla_w \log(D_w(s,a))] + \hat{\mathbb{E}}_{\tau_E}[\nabla_w \log(1 - D_w(s,a))]$$

  Take a policy step from θ_i to θ_{i+1}, using the TRPO rule with cost function $\log(D_{w_{i+1}}(s,a))$. Specifically, take a KL-constrained natural gradient step with

$$\hat{\mathbb{E}}_{\tau_i}[\nabla_\theta \log \pi_\theta(s|a) Q(s,a)] - \lambda \nabla_\theta H(\pi_\theta) - \lambda' \eta(\pi_\theta),$$

  where $Q(\bar{s}, \bar{a}) = \hat{\mathbb{E}}_{\tau_i}[\log(D_{w_{i+1}}(s,a))|s_0 = \bar{s}, a_0 = \bar{a}]$
end
return θ_i, w_i

4.5. Imitation Learning

As there is no obvious a-priori reward function for identifying high-quality programs, we use imitation learning to train our code generation policy. In particular, Ho and Ermon's Generative Adversarial Imitation Learning (GAIL) [10] and Li et al.'s InfoGAIL [14] is used to learn a neural network parametrized policy from the expert state-action pairs.

The formal objective of Generative Adversarial Imitation Learning can be stated as

$$\min_{\pi} \max_{D \in (0,1)^{S \times A}} \mathbb{E}_\pi[\log D(s,a)] + \mathbb{E}_{\pi_E}[\log(1 - D(s,a))] - \lambda H(\pi)$$

where D is discriminator network, π is a policy network, and π_E is the expert policy, and $H = \mathbb{E}_\pi[-\log \pi(a|s)]$ is the discounted causal entropy of the policy network. This objective is optimized using Algorithm 2,

To guide this optimization using domain knowledge, we can incorporate reward shaping into the policy update step as described in [14] by adding a term $-\lambda'\eta(\pi)$ to the policy objective. This creates a hybrid imitation-reinforcement learning approach.

Here, in the code generation context, we intend to in-

troduce reward shaping that encourages creation of type-consistent and semantically correct code. More specifically, at a given state s_t, if we are able to obtain a full parse of the current method body as currently defined in s_t then a positive reward $\delta_{\text{type check}}$ is provided in the environment if the generated AST passes type checking. If the program does not type check a negative reward $\delta_{\neg(\text{type check})}$ is given and the state becomes terminal. Furthermore, if test cases are provided for the program context, any method body that passes type checking internally and whose return type matches the declared return type can be run on test cases and a large positive reward bestowed on correct output. In some cases this can only occur at the granularity of the entire program, making it a less useful reward signal in general.

With this complex augmented objective, we can use relatively standard optimization frameworks and neural architectures such as a Wasserstein GAN discriminator model [1] and TRPO [22] with a baseline [25] and replay buffer [17] for policy optimization.

4.6. Code Generation

To generate code we maintain a queue of functions to be written and iteratively pop from the queue and run the RL agent over the queued element. The queue is initially populated with all functions that are nodes in the dependency graph with out-degree 0; upon the RL agent finishing processing the current element, the controller pushes to the queue all of the parents of the current element in G that have not been processed already. This is repeated until the entire graph has been processed.

5. Data

In order to train the model we will propose, we need data that meets two requirements. The first is that the code we are training on needs to be in a fully-functional language like Haskell or Scala. This requirement stems from the ability to be able to strongly type some basic primitive functions and allow for the recursive construction of the full program. The second requirement is that the code is well documented with quality comments above each function header explaining the intent of the function in natural language in order to be able to encode the natural language into the embedding space that the program generator will take. In particular we would like to use the Google code base which in specific does have Haskell and Scala code. It is also, for the most part, guaranteed to be code of both high quality and extensive commenting as a result of an extensive code review process. Therefore we can say with a high degree of confidence that the code from the Google code base is a good source of well commented and functional code, as a result of the stringent and clearly defined code standards they have for their engineers before they can push to production.

At this point each program will be pre-processed into the designated input architecture, which will look similar to a header file. The header-like file will demarcate a few important pieces of information. The first is a hierarchy of prototypes of all the functions that will be used by the program, specifically calling out which function calls another function*. The next important designation made in this architecture file is the parameter type and the return type of the function prototype. Finally, the architecture file will also have a natural language comment that represents the intent of the function it is describing.

The Google code will be parsed using a simple grammar specific for the input language and will output an architecture file as specified above. One of the assumptions we will make about the code produced is that any function specified by the architecture file can at most call only up to K_C other prototypes in the file but can call as many primitive functions, a safe assumption as in practice most programmers do not call too many of their own functions for cleanliness and will abstract further if there is a need to call that many helper functions. This means that we will have to clean the Google code base for any programs that contain functions which have calls to more than K_C helper functions. The point of this pre-processing is to mimic the kind of input architecture that the programmers assistant would receive from a programmer at test time.

6. Discussion

Our system addresses a fundamental problem in software engineering: given code structure and natural language context, write the remaining pieces. However, this system is still a few steps removed from what an actual software engineer might deem useful. Importantly, our architecture requires the entire code dependency graph to be known a priori. This differs from standard software engineering in a few ways. For one, the detailed structure and decomposition of all functions might not always be known before beginning to write code: frequently, as any engineer will point out, structure is updated and adapted on the fly when the original designs flaws are noted. Thus, the structure fed into our network may not be the finalized or optimal solution. Additionally, design is not necessarily done to this level of detail prior to writing the software: our network expects the entire code structure to be fleshed out up to base functions, but in practice, only a subgraph of the DAG might be available, and not all functions might even be known.

Fixing both of these problems, and more fundamentally, fixing the requirement of knowing the full DAG, is not out of reach. One might imagine an augmentation to our architecture as follows. In addition to the current available set

*This requires the program to be able to be represented as a directed acyclic graph preventing the formation of any recursive programs

of actions, the agent could also be given the option to create a new function with a specified type signature; in such a case, it might be expected to also generate a natural language description of the runtime-generated function, to allow the current system to be reapplied to the newly specified function. Further, the agent could also be given the ability to modify the DAG: edge removal and edge creation could both be added to the action set, the additional necessary constraints (i.e. the graph operations should not violate acylicity, if an edge is added f $\rightarrow$g, then f should in fact call g) could be implemented via improved reward shaping. The combination of these two features relaxes the constraint of our current system and allows for engineers to just give approximate architecture and function specifications.

The above addition also sheds light on a more promising route if the DAG can be constantly modified and updated, and new nodes can continue to be added, then, if an agent were good enough, only minimal specification may be required (e.g. only top level functions might need to be specified, and the agent could infer the remainder of the structure). The limitations on implementing this approach right now mainly revolve around search space size; the number of possible type signatures for new functions is at least $|\mathsf{T}|^{|\mathsf{T}|}$, which means the action space could correspondingly get intractably huge. Future research should investigate alternative methods for new function generation.

7. Conclusion

In this work, we propose a novel idea that aims to liberate programmers from the minutiae of implementation. In our framework, the human acts as the creative architect, providing only high level design expertise in a blueprint interface file. The computer agent acts as the builder, leveraging adversarial training and reward augmented reinforcement learning to fill in function implementations.

We believe the key contributions of our framework are the following:

- We present an end-to-end natural-language-to-code framework that generalizes to full programs with multitudes of interacting functions, rather than focusing on smaller code snippets, which have commonly been studied.

- We suggest an instantiation of the Programmer's Apprentice that is crafted for usability: our agent is non-invasive, design-conscious, and familiar to programmers, who are already accustomed to crafting header files and interfaces.

- Our work is among the first to describe applications of Generative Adversarial Imitation Learning (GAIL) to the code synthesis domain.

While the Programmer's Apprentice problem is far from solved, our proposed agent contributes progress to the problem, offering both code auto-implementation and design apprehension.

8. Appendix

Algorithm 3: Generate Code from Policy and Spec

Input : Parameters of policy network, θ, and header file, h

Output: h filled in with code that fits the function descriptions

```
G = extract_dependency_graph(h)
for func ∈ bottom-up_traversal(G) do
    state = [type signature of func; flattened G, NL
      description of func; C_f for func; empty
      implementation of func; 0]
    while action ≠ F_terminal do
        action = π_θ(state)
        if action ≠ F_terminal then
            Add action to implementation of func
        end
        Update state
    end
    Add implementation in state to h
end
return h
```

Algorithm 4: End-to-end Procedure

Input : Expert code base, B, and header file, h

Output: A policy network that that is able to generate code given the input header file

```
expert_pairs ← []
for file ∈ B do
    expert_pairs += Algorithm 1 (file)
end
θ, w = Algorithm 2 (expert_pairs)
h_filled = Algorithm 3 (θ, h)
return h_filled
```

References

[1] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[2] R. Bunel, M. J. Hausknecht, J. Devlin, R. Singh, and P. Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. 2018.

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

[4] X. Chen, C. Liu, E. C. R. Shin, D. Song, and M. Chen. Latent attention for if-then program synthesis. *CoRR*, abs/1611.01867, 2016.

[5] X. Chen, C. Liu, and D. Song. Tree-to-tree neural networks for program translation. *CoRR*, abs/1802.03691, 2018.

[6] X. Chen, C. Liu, and D. X. Song. Tree-to-tree neural networks for program translation. *CoRR*, abs/1802.03691, 2018.

[7] J. Cheng, S. Reddy, V. Saraswat, and M. Lapata. Learning an executable neural semantic parser. *CoRR*, abs/1711.05066, 2017.

[8] A. Cozzie and S. T. King. Macho: Writing programs with natural language and examples. 2012.

[9] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy. Program synthesis using natural language. *CoRR*, abs/1509.00413, 2015.

[10] J. Ho and S. Ermon. Generative adversarial imitation learning. In *Advances in Neural Information Processing Systems*, pages 4565–4573, 2016.

[11] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Program synthesis from natural language using recurrent neural networks.

[12] S. Jiang, A. Armaly, and C. McMillan. Automatically generating commit messages from diffs using neural machine translation. *CoRR*, abs/1708.09492, 2017.

[13] S. Z. Kyle Richardson and J. Kuhn. The code2text challenge: Text generation in source code libraries. 2017.

[14] Y. Li, J. Song, and S. Ermon. Infogail: Interpretable imitation learning from visual demonstrations. In *Advances in Neural Information Processing Systems*, pages 3815–3825, 2017.

[15] X. V. Lin, C. Wong, D. Pang, K. Vu, L. Z. C. Xiong, and M. D. Ernst. Program synthesis from natural language using recurrent neural networks.

[16] W. Ling, E. Grefenstette, K. M. Hermann, T. Kociský, A. Senior, F. Wang, and P. Blunsom. Latent predictor networks for code generation. *CoRR*, abs/1603.06744, 2016.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[18] V. Murali, S. Chaudhuri, and C. Jermaine. Bayesian sketch learning for program synthesis. *CoRR*, abs/1703.05698, 2017.

[19] I. Polosukhin and A. Skidanov. Neural program search: Solving programming tasks from description and examples. *CoRR*, abs/1802.04335, 2018.

[20] S. E. Reed and N. de Freitas. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015.

[21] C. Rich and R. C. Waters. The programmer's apprentice: A research overview. *Computer*, 21(11):10–25, 1988.

[22] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[23] H. E. Shrobe, B. Katz, R. Davis, et al. Towards a programmer's apprentice (again). In *AAAI*, pages 4062–4066, 2015.

[24] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[25] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.

[26] D. Xu, S. Nair, Y. Zhu, J. Gao, A. Garg, L. Fei-Fei, and S. Savarese. Neural task programming: Learning to generalize across hierarchical tasks. *CoRR*, abs/1710.01813, 2017.

[27] P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696, 2017.

[28] V. Zhong, C. Xiong, and R. Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.