Leveraging Constraint Logic Programming for Neural Network Guided Program Synthesis

by

Lisa Zhang

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Leveraging Constraint Logic Programming for Neural Network Guided Program Synthesis

Lisa Zhang
Master of Science
Graduate Department of Computer Science
University of Toronto
2018

Synthesizing programs using example input/outputs is a classic problem in artificial intelligence. We present a method for solving such problems by tightly integrating a neural network with a constraint logic programming system called miniKanren. Internally, miniKanren searches for a program that satisfies the recursive constraints imposed by the provided examples. We present a Recurrent Neural Network (RNN) model and a Gated Graph Neural Network (GGNN) model, both of which use these constraints as input to score candidate programs. We show evidence that using our method to guide miniKanren's search is a promising approach to solving PBE problems.

Contents

Chapter 1

Introduction

Program synthesis is an important area of artifical intelligence that has captured the imagination of many computer scientists. Computer scientists love to automate, and what would be more provocative than the idea of a programmer automating programming? Programming by Example (PBE) is one way to formulate program synthesis problems, where example input/output pairs are used to specify a target program.

In some sense, all of supervised learning can be considered program synthesis: "training data" is essentially another term for input/output pairs, and are used to learn continuous weights that parameterize some function in a hypothesis space. Combined with a suitable optimization method like gradient descent, continuous analogs to program synthesis have been successful in solving problems ranging from image classification to natural language translation to playing Go.

However, supervised learning does not subsume the goal of traditional program synthesis, which is to generate a discrete program in a specific programming language or a Domain Specific Language (DSL) consistent with the input/output pairs. In other words, the goal is to synthesize code that both machines and humans can read, understand, and manipulate. The interpretability of programs as code means that we can use the synthesized results in other ways: we can compare code, we can optimize code, we can compile code into other programming languages, and we can prove the correctness of code. The manipulability of code makes program synthesis continue to be relevant today.

PBE problems have recently caught the attention of machine learning researchers. However, current state of the art techniques continue to be dominated by symbolic techniques developed by the programming languages community. These methods use rule-based, exhaustive search, often manually optimized by human experts using domain specific knowledge. While these simple sounding techniques work surprisingly well, success is limited to small programs. These search strategies tend not to scale well to larger programs, and hand-crafted optimizations are not always generally applicable.

Recent works by the machine learning community explore a variety of statistical methods to improve performance on a variety of PBE problems. These contributions fall under three general categories: differentiable programming, direct synthesis, and neural guided search. Differentiable programming involves building a differentiable interpreter, then backpropagating to find a latent program [19, 22, 13]. While appealing in theory, differentiable techniques perform poorly compared to symbolic techniques in practice [12]. Direct synthesis aims to synthesize the program as a tree or sequence, and has been successfully applied to string manipulation problems [10, 21]. While string manipulation is an important

and practical application, techniques developed in this domain do not handle recursion, and therefore do not provide Turing Completeness. In neural guided search, the machine learning portion guides a search technique developed by the programming languages community. The neural model identifies promising portions of the search space to explore.

This work falls under neural guided search, but takes the integration with a symbolic system even further: we use the internal problem representations of the symbolic search as input to the neural network. The symbolic system used is called miniKanren[1] [8]. We choose miniKanren for its potential to synthesize recursive programs in dynamically typed languages [7]. This flexibility means that miniKanren can encode many synthesis problems that are difficult or impossible to encode using other systems. Internally, miniKanren searches for a program that satisfies the recursive constraints (usually called "goals") imposed by the input/output examples. Our model uses these internal constraints to score candidate programs and guide miniKanren's search.

This idea of using the constraint representation is equivalent to learning the specific flavour of constraint satisfaction miniKanren uses for program synthesis. That is, the goal of the neural model is to learn which candiate partial program has the most promise, or the most probability of fully satisfying the constraints imposed by input/output examples. The symbolic solver follows the output of the neural model to explore the search space.

The idea of neural guided search using constraints is promising for several reasons. First, symbolic systems have historically performed better than continuous systems, and neural guidance can help navigate exponentially large search spaces, leveraging progress made in both communities. Second, symbolic systems exploit the compositionality of synthesis problems: miniKanren's constraints select portions of the input/output examples relevant to a subproblem. This is akin to having a symbolic attention mechanism: using attentional methods is explored in works like [10, 21], and having a symbolic attention mechanism means having fewer components that require training. Lastly, it is difficult for neural techniques to synthesize programs larger than those seen in training [21]. Guiding a search and using constraints both alleviate this problem. We present some evidence that our approach is able to generalize to programs larger than those seen in training.

The novelty of this work is two-fold. First, we contribute the idea of working with the intermediate representation of the search system, thus building a machine learning framework that solves a slightly different problem in order to guide a symbolic search. Second, we explore a Gated Graph Neural Network (GGNN) architecture [18] to score constraints. The use of GNN to represent syntactical structures also appeared in concurrent works like [2], which used graph neural networks to represent programs, and [24], which used graph neural networks to solve boolean satisfiability (SAT) problems. Though our ideas are similar to those two works, the details of the implementation are different. With these new ideas, we show evidence that our method has the potential to generalize to bigger problems.

We test our approach to solve PBE problems in a small subset of Lisp. We present two sets of experiments. First, we compare variations of our neural guided models against non-neural guided versions on auto-generated PBE problems held out from training. We compare against two miniKanren search strategies that do not use a neural guide, and three neural-guided models. Second, we test the generalizability of our approaches on three families of synthesis problems. In this second set of experiments we additionally compare against state-of-the-art systems λ^2, Escher, and Myth. We show that our neural-guided approach using constraints can synthesize problems faster, and has better potential

[1] The name "Kanren" comes from the Japanese word for "relation".

to generalize to bigger problems.

Acknowledgements This work was done jointly with Gregory Rosenblatt, Ethan Fetaya, William Byrd, Renjie Liao, Raquel Urtasun and Richard Zemel. In particular, Gregory Rosenblatt and William Byrd are contributers to miniKanren and made many technical changes to miniKanren in anticipation of this work. While their technical contributions are not discussed in this document, they were crucial in the completion of this work. Additionally, the machine learning portion of this work would not have been possible without the contributions of Ethan Fetaya, Renjie Liao, Raquel Urtasun and Richard Zemel.

Chapter 2

Related Work

While there is a recent surge in interest in program synthesis, programming by example (PBE) problems have a long history dating to the 1970's [25, 6]. This section surveys recent related work.

2.1 Symbolic Methods

Along the lines of early works in program synthesis, the programming languages community developed search techniques that enumerate potential programs, with pruning strategies based on types, consistency, and logical reasoning to improve the eearch. While such enumerative search techniques may seem simple, they can achieve impressive results. Different methods differ in the domain specific language (DSL) used, the search technique, whether type information is required and whether the search is performed top-down (synthesizing the root of the program tree first, as in λ^2 [11], Myth [20], and miniKanren) or bottom-up (synthesizing the leaves of the program tree first, as in Escher [1]).

The method λ^2 [11] is most similar to miniKanren. Unlike miniKanren, λ^2 is a specialized PBE tool that specializes in numeric, statically-typed inputs and outputs. The tool searches a type-constrained space of candidate programs that may contain unknowns. The creators of λ^2 call these candidate programs "templates". These "templates" are refined using the input/output examples, producing new synthesis subproblems. The tool also includes higher-order function combinators like `map`, `fold`, and `filter` to be able to synthesize recursive programs.

Escher [1] builds programs bottom-up using a search with two alternating phases. A forward phase builds new simple program fragments by extending existing fragments. A conditional inference phase joins existing fragments with conditional statements in order to satisfy multiple examples that cannot be covered by a single simple fragment. While synthesizing recursive programs, Escher is built as an active learner, and relies on the presence of an oracle to supply outputs for new inputs that it chooses.

Myth [20] searches for the smallest program satisfying a set of examples. Its search strategy is an iterative deepening backwards search. The iterative deepening parameters at each step limit various aspects of the size of programs considered, and are gradually increased in subsequent steps. Generated program fragments from one step are cached by type and size, and may be used during later steps. To synthesize recursive functions, Myth requires input/output examples to be trace complete, which means that every example of a recursive function call must be accompanied by examples for every recursive sub-call involved in computing the output of that example.

These three methods all use type information in one form or another, and all focus on functional DSLs. The goal of all three systems is to be able to synthesize recursive functions, as a step towards general computation.

2.2 Statistical Methods

Contributions by the machine learning community have grown in the last few years. Interestingly, while PBE problems can be thought of as a meta-learning and a few-shot learning problem, few works explore this relationship. Like in meta-learning (or learning to learn), the goal is to build a learner capable of solving many related problems. Like in few-shot learning, PBE problems usually have very few input/output examples. However, typical few-shot learning problems focus on supervised classification. Here, the desired output is more complex and structured, and often the goal is to learn an explicit program in a particular DSL, as opposed to a latent one. This section summarizes the recent contributions by the machine learning community.

2.2.1 Direct Synthesis

One use case where neural methods have been successful is FlashFill [14] and its descendents [21, 10, 5]. These techniques are trained on string manipulation tasks and are successfully applied to spreadsheet completion problems. The string manipulation is a domain that direct synthesis, where a neural model directly synthesizes a program as a sequence or a tree, performs reasonably well. An example FlashFill problem is shown in Table 2.2.1. While practical, the DSL is greatly limited, so methods developed for this domain may not scale to the larger problem of synthesizing programs in a Turing Complete DSL, especially those involving recursion.

The original FlashFill [14] technique uses a combination of search and carefully crafted heuristics. Later works that build upon [14] are increasingly more sophisticated. For example, [21] uses an RNN encoder for the input/output pairs, and a "Recursive-Reverse-Recursive Neural Network" (R3NN) on the parse tree of the partial program to determine which part of the tree to expand. The DSL grammar is then used to expand that portion of the tree, synthesizing the desired program in the form of a tree.

More recently, RobustFill [10] uses bi-directional LSTMs with attention, with each output sequence LSTM attending to the input sequence, and the program sequence LSTM attending to both input and output sequences. With multiple input/output pairs, the program hidden states are pooled at each time step. Despite using a (less structured) sequence instead of a (more structured) tree, RobustFill achieved much better results (92% vs 38%) in the FlashFill benchmark.

Input		Output
Nancy	FreeHafer	Nancy F.
Andrew	Cencici	Andrew C.
Jan	Kotas	Jan K.
Mariya	Sergienko	??

Table 2.1: Example FlashFill Problem

2.2.2 Differentiable Programming

Differentiable programming involves building a differentiable interpreter, then backpropagating through the interpreter to learn a program. This program is usually latent, encoded in terms of continuous

parameters instead of discrete sequences or trees. It is difficult to extract a symbolic program from these continuous parameters. However, if the goal is to infer correct outputs for new inputs, a discrete program is not necessary.

Work in differentiable programming began with the Neural Turing Machine [13], a neural architecture that augments neural networks with external memory and attention, emulating a von Neumann architecture in an end-to-end differentiable manner. Neural Programmer [19] similarly builds a neural networks architecture augmented using basic arithmetic and logic operations. The architecture can then use these operations on different segments of data in more complex ways, creating complex programs compositionally. The Neural Programmer-Interpreter [22] is similarly compositional, with three separate learnable components that incldue a recurrent core, a key-value memory of program embeddings, and a task-dependent encoder. It is trained on full program traces.

The RobustFill [10] work also explores using a latent program instead of an explicit one. They find that when synthesizing an explicit program succeeds, the program generalizes better. That is, the explicit program is more likely to predict correct outputs for the entire set of unseen inputs in each problem, whereas the latent program is more likely to make correct predictions per input.

While end-to-end differentiable approaches are attractive to machine learning researchers, [12] showed that differentiable interpreter based program induction still performs worse compared to discrete search-based techniques.

2.2.3 Neural Guided Search

A recent line of work in machine learning incorporates methods in both communities, and uses statistical techniques to guide the discrete search to make the best out of both worlds. DeepCoder [3] falls under this approach. It uses a neural network to encode the input/output examples, and uses the latent encoding to predict which functions are likely to appear in the program. DeepCoder is trained in the domain of programming competition problems, a domain where an unordered list of functions to apply can be a very useful signal for synthesis.

More recently, [16] uses an LSTM to guide the symbolic search system PROSE (Microsoft Program Synthesis using Examples). The search uses a "branch and bound" technique. The neural model learns the choices that best maximize the bounding function h that was hand-generated in [14] and used for FlashFill problems.

Chapter 3

Background

This section discusses the fundamental ideas used in this paper in more depth.

3.1 Relational Program Synthesis

The symbolic system miniKanren is actually a constraint logic programming language. This section describes how one can equip miniKanren with a relational interpreter **evalo** to solve PBE problems.

In the relational programming paragdigm, programmers write *relations* instead of functions. A relation defines a set of ordered tuples. For example, the greater than relation $>$ is a binary relation, and all tuples (a, b) where $a > b$ belong to this relation. The language miniKanren supports queries where part or all inputs to a relation can be unknown. For example, a query for $x > 0$ uses the definition of the relation $>$ to find all corresponding values of x where the tuple $(x, 0)$ is a member of the relation. This style of programming is also possible in logic programming languages such as Prolog.

All functions F have an equivalant relation **R**: we can define **R** such that if $\text{F}(\texttt{x}) = \texttt{y}$, then $(\texttt{x}, \texttt{y})$ belongs to the relation **R**.

In particular, an interpter $\text{EVAL}(\texttt{expr}, \texttt{env}) = \texttt{output}$ has a relational equivalent $\textbf{evalo}(\texttt{expr}, \texttt{env}, \texttt{output})$. We can use **evalo** to solve PBE tasks. If we use **p** to denote the (unknown) desired program, `i` an example input (known), and `o` the corresponding output (known), then we need **p** to satisfy **evalo**((*app* **p** `i`), ϕ, `o`) where *app* denotes function application (using Lisp syntax) and ϕ the empty environment. Here, **evalo**((*app* **p** `i`), ϕ, `o`) is equivalent to **evalo**(**p**, (`i`), `o`), and imposes constraints on **p**. We use the term "relation" and "constraint" interchangably. In the miniKanren literature, our notion of "constraint" is typically referred to as a "goal". Here, we refer to **p** as a logic variable, an unknown whose value is to be determined. The terminology explicitly distinguishes between logic variables in the relational regime and variables of functions.

It is important to note that the constraint **evalo** is recursive: it is defined in terms of other constraints (possibly another **evalo**). Thus, uses of **evalo** can be unfolded via replacement with its definition in terms of other constraints. For example, in a Lisp interpreter, a program **p** can be a constant, a function call, a variable lookup, or some other expression. These possibilities are revealed as the clauses of a disjunction that replaces **evalo**.

Figure 3.1 shows an example unfolding of **evalo**(**p**, (`i`), `o`). First, miniKanren replaces **p**, the only logic variable present, with a disjunction of its possible expansions. Then, each resulting constraint is

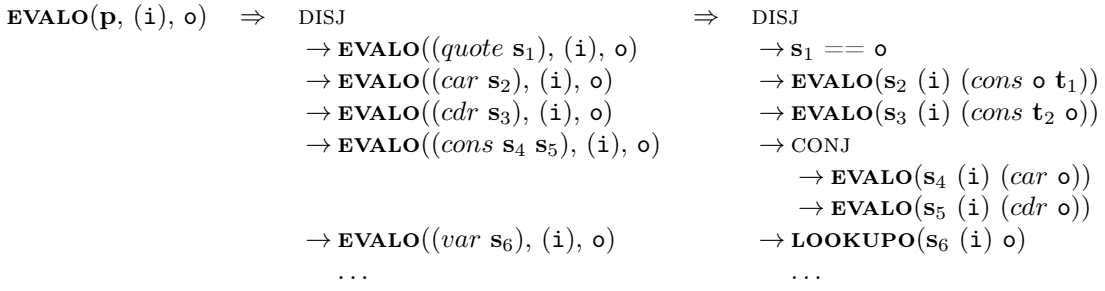

Figure 3.1: Expansions and reduction of **EVALO**(**p**, (i), o)

further reduced until no longer possible without choosing another logic variable $\mathbf{s}_j$ to expand.

3.2 Constraint Representation for PBE

To specify **p** by multiple input/output examples as is typical, we use a conjunction of **EVALO** constraints over the logic variable **p** and the individual input/output pairs. Naturally, the constraints can be represented as a tree, where internal nodes are conjunctions and disjunctions, and leaf nodes represent constraints. As constraints are unfolded, they are replaced by corresponding subtrees, as in Figure 3.1.

Search entails iteratively unfolding EVALO and other constraints. This unfolding happens gradually since a fully unfolded constraint tree is usually infinite. It is this unfolding of constraints that defines the search process. As we unfold more nodes, branches of the constraint tree constrain **p** to be more specific. If at some point we find a fully specified **p** that satisfies all the relevant constraints, then **p** is a solution to the PBE problem. We refer to a partial specification of **p** as a "candidate" partial program.

In a standard implementation of miniKanren, the constraint tree is implicitly represented using suspended computations and program continuations. For this work, a language contributor built a transparent version of miniKanren to make the constraint tree explicit.

The default search process used by miniKanren is a biased interleaving search. Here, "interleaving" means that the search alternates between clauses of a disjunction. The alternation is not uniform random, but is "biased" towards branches that have more of their constraints already satisfied. This type of search is *complete*, meaning that if a solution exists, then miniKanren is guaranteed to find the solution in some fininte (but possibly large) time.

For this work, we make one further transformation to the constraint tree representation, as in Table 3.2. We discard global tree structure, and associate subtrees with their corresponding candidate partial program **p**. We will use the subtree of constraints associated with each candidate to score candidates. This way, we only concern ourselves with constraints associated with each candidate, thus being able to score candidates and think in terms of candidates.

Some of the constraints in Table 3.2, specifically those for the candidate program (*quote* $\mathbf{s}_5$), are obviously not satisfiable. In such cases, miniKanren will remove the candidate partial program and its associated constraints from consideration, so the machine learning guide will not need to score obviously bad candidates.

Table 3.1: Constraints Associated with Candidates for Examples {1 → (1 1 1), a → (a a a)}

Candidate: (*car* $\mathbf{s}_1$)
CONJ
→ **EVALO**($\mathbf{s}_1$ (1) (*cons* (1 1 1) $\mathbf{t}_1$))
→ **EVALO**($\mathbf{s}_1$ (a) (*cons* (a a a) $\mathbf{t}_1$))
Candidate: (*cdr* $\mathbf{s}_2$)
CONJ
→ **EVALO**($\mathbf{s}_2$ (1) (*cons* $\mathbf{t}_2$ (1 1 1)))
→ **EVALO**($\mathbf{s}_2$ (a) (*cons* $\mathbf{t}_2$ (a a a)))
Candidate: (*cons* $\mathbf{s}_3$ $\mathbf{s}_4$)
CONJ
→ **EVALO**($\mathbf{s}_3$ (1) 1)
→ **EVALO**($\mathbf{s}_4$ (1) (1 1))
→ **EVALO**($\mathbf{s}_3$ (a) a)
→ **EVALO**($\mathbf{s}_4$ (a) (a a))
Candidate: (*quote* $\mathbf{s}_5$)
CONJ
→ $\mathbf{s}_5$ == (1 1 1)
→ $\mathbf{s}_5$ == (a a a)

3.3 Long-Short Term Memory

Our method involves scoring each candidate by scoring the constraints associated with them. One way to represent constraints is as sequences. In this section, we review LSTMs.

A Recurrent Neural Network (RNN) takes as input a sequence of variable length. Each token x_t in the sequence updates a vector embedding h_{t-1} of the sequence so far.

$$h_t = f(x_t, h_{t-1})$$

This traditional RNN has several issues. One issue is the difficulty in learning long-term dependencies. A single hidden state vector h_t must summarize all information up until x_t. Further, at every time step t, the same function f is used to update h_t. This restriction is not only limiting, but also causes the model to be difficult to train. The gradient of f is multiplied a large number of times, leading to issues of gradient vanishing or explosion. These issues motivated the creation of the Long-Short Term Memory (LSTM) network [15].

LSTM networks introduce memory cells, which adds adds several gating mechanisms: an **input gate** i_t that determines how much the input x_t should affect the hidden state h_t, a **forget gate** f_t that determines how much the previous hidden state h_{t-1} should be decayed (or removed), and an **output gate** o_t that combines the input and previous hidden state.

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hc}h_{(t1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t1)} + b_{ho}) \\
c_t &= f_t c_{(t1)} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

The use of these gating mechanism not only resolves gradient flow issues, but also allows the model to learn longer-term dependencies. Further, RNNs with LSTM memory cells have been shown to be able to encode complex structure, even though the input is taken as a simple sequence.

3.4 Gated Graph Neural Network

Another way to represent constraints is as a graph. Each constraint is a tree, but different constraints can reference the same unknown logic variable. We can represent these interconnected constraint trees using a Gated Graph Neural Network (GGNN) [18], which this section summarizes.

A graph neural network is a neural network whose architecture is defined according to a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ on which we wish to perform inference. We denote the nodes of the graph as $\mathcal{V} = 1, 2, \ldots, N$ and directed edges as $(v', v) \in \mathcal{E}$, Each node has an initial node embedding $h_v^{(0)} = e_v \in \mathbb{R}^D$, which can be initialized to be random, or to some embedding provided to the network. These embeddings are updated via a recurrence that follows the edges in $\mathcal{E}$, as follows.

Algorithm 1 Gated Graph Neural Network Recurrence

Require: Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
1: **for all** $(v', v) \in \mathcal{E}$ **do**
2: $\quad m_{v' \to v} = \text{MESSAGE}(h_v^{(t)})$ ▷ Compute Messages
3: **end for**
4: **for all** $v \in \mathcal{V}$ **do**
5: $\quad m_v \to \text{MERGE}(m_{\cdot \to v})$ ▷ Merge Messages
6: $\quad h_v^{(t+1)} \to \text{GRU}(m_v, h_v^{(t)})$ ▷ Update Embedding
7: **end for**

We use the notation $m_{\cdot \to v}$ to denote all the messages $m_{v' \to v}$ where $(v', v) \in \mathcal{E}$. In this case, MESSAGE is a feed-forward neural network, MERGE can be a pooling operation like MAX, MIN, and MEAN, and GRU is a gated recurrent unit.

In this case, the messages are sent asynchronously: all message sending and merging occurs prior to any embedding updates. This need not be the case; finding message schedules so that fewer steps of the recurrence is required is an active area of research. Further, a GGNN can be augmented with node and edge types, so that the MESSAGE function is dependent on the edge types, and the MERGE and GRU functions are dependent on the vertex types.

Once the propagations have completed for a pre-determined number of steps T, we can use the final node embedding $h_v^{(T)}$ as features to perform inference on the node. In general, different forms of inference

include node or edge selection, classification, and scoring. In our case, we are interested in scoring a subset of the nodes as follows:

$$a_v = \textsc{score}(h_v^{(T)}),$$

where the output function SCORE is parameterized using a neural network.

Chapter 4

Neural Network Guided Synthesis

This section presents our neural guided synthesis model, which scores candidate programs using its corresponding constraints, then uses the scores to guide the search for a valid program consistent with the input/output examples.

4.1 Problem Setup

The interaction between the machine learning agent and the symbolic system miniKanren is summarized in Figure 4.1.

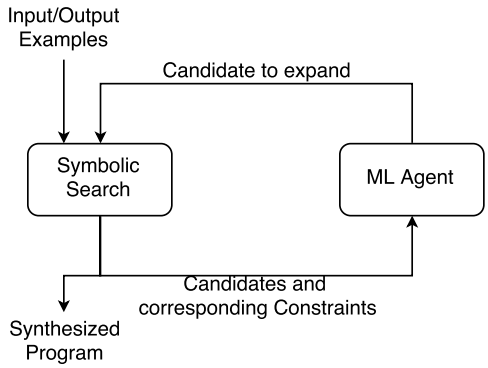

Figure 4.1: Interaction between ML Agent and Symbolic System

This problem setup is similar to that of Reinforcement Learning. A Machine Learning agent makes discrete choices amongst possible actions (candidate programs to be expanded) given the current state (set of candidates and their constraint trees). The symbolic system acts as the non-differentiable environment, which produces new states (expanded set of candidates and their constraint trees) given an existing state and an action. In this setup, the goal would be to solve the synthesis problem in as few steps as possible.

We can certainly treat this as a reinforcement learning problem. With an appropriate reward signal (for example, a reward of -1 for each step taken), we can train the Machine Learning agent using policy gradient [26], Q-learning, or more advanced reinforcement learning algorithms.

Algorithm 2 Neural Guided Program Synthesis

Input: $E \in \mathcal{D}^{2n}$ ▷ Input/output examples
Output: $p \in \mathcal{P}$, p fully specified
Initialize $C_0 \leftarrow \text{START}(E)$ ▷ Initial constraint tree
Initialize $t \leftarrow 0$
repeat
    $a_t \leftarrow \text{CHOOSE}(C_t)$ ▷ Choose action
    $C_{t+1} \leftarrow \text{STEP}(C_t, a_t)$ ▷ Get next state, or the synthesized program p
    $t \leftarrow t + 1$
until C_t is not None or $t \geq \text{MAXSTEPS}$

However, unlike most reinforcement learning problems, we can obtain the ground truth "best" action at each state. Specifically, the ground truth action is to choose the candidate program that matches with the actual program. The ground truth action is not necessarily the only "correct" action: there can be multiple different programs that are semantically equivalent to each other, known as program aliasing. For this work, we ignore the problem of aliasing and use the ground truth program, and treat learning the ML Agent as a supervised learning problem. Still, our training process borrows many ideas from reinforcement learning.

We now summarize the inference process more formally. We will use the following notation: let $\mathcal{L}$ be the Domain Specific Language (DSL) of the target program to be synthesized, $\mathcal{D}$ the set of possible data values in $\mathcal{L}$, and *prog* the collection of allowable fully and partially specified programs in $\mathcal{L}$ (partially specified programs will contain logic variables whose values are to be determined). For a Programming by Example (PBE) problem, the synthesis problem is defined in terms of n input and output example pairs

$$E = \{(i_1, o_1), (i_2, o_2), \ldots, (i_n, o_n)\} \in \mathcal{D}^{2n},$$

where $i_j, o_j \in \mathcal{D}$. At each step, our states are a collection of pairs $S = \{(p_1, c_1), \ldots, (p_l, c_l)\}$ with $p_i \in \mathcal{P}$ and $c_i \in \mathcal{C}$. Let $\mathcal{S}$ denote all possible such states. Our space of actions for a given state will be $A_S = \{p_1, \ldots, p_l\}, p_i \in S, p_i \in \mathcal{P}$, which corresponds to the partial programs that we can choose to expand.

The symbolic system has an implementation of $\text{START} : \mathcal{D}^{2n} \rightarrow \mathcal{S}$, which produces a state that encodes the PBE problem. The symbolic system also contains an implementation of $\text{STEP} : \mathcal{S}, \mathcal{P} \rightarrow \mathcal{S}$, which takes a constraint tree and a candidate partial program to expand, and performs a step of the search at the chosen candidate. If the candidate is a fully specified program that is consistent with the input/output examples, then STEP will return a null state, signifying that the search is complete. Both START and STEP are assumed to be discrete, non-differentiable processes.

The machine learning agent implements $\text{CHOOSE} : \mathcal{S} \rightarrow \mathcal{P}$, which chooses an action (candidate partial program) given a state. This is a continuous model trainable by backpropagation. Our contribution involves using the constraint $\mathcal{C}$ as input to the machine learning agent. We experiment with two possible parameterizations of CHOOSE, outlined in the next sections.

4.2 Recurrent Neural Network Model

One way to parameterize the machine learning agent is using an RNN operating on constraints as sequences. We use an RNN with bi-directional LSTM units [15] to score constraints associated with

		LSTM		Score		Pool and Softmax
Candidate: (*car* $\mathbf{s}_1$)						
CONJ						
→ **EVALO**($\mathbf{s}_1$ (1) (*cons* (1 1 1) $\mathbf{t}_1$)	⇒	Embed	⇒	Score	⇒	0.09
→ **EVALO**($\mathbf{s}_1$ (a) (*cons* (a a a) $\mathbf{t}_1$))	⇒	Embed	⇒	Score	⇒	
Candidate: (*cdr* $\mathbf{s}_2$)						
CONJ						
→ **EVALO**($\mathbf{s}_2$ (1) (*cons* $\mathbf{t}_2$ (1 1 1)))	⇒	Embed	⇒	Score	⇒	0.10
→ **EVALO**($\mathbf{s}_2$ (a) (*cons* $\mathbf{t}_2$ (a a a)))	⇒	Embed	⇒	Score	⇒	
Candidate: (*cons* $\mathbf{s}_3$ $\mathbf{s}_4$)						
CONJ						
→ **EVALO**($\mathbf{s}_3$ (1) 1)	⇒	Embed	⇒	Score	⇒	
→ **EVALO**($\mathbf{s}_4$ (1) (1 1))	⇒	Embed	⇒	Score	⇒	0.72
→ **EVALO**($\mathbf{s}_3$ (a) a)	⇒	Embed	⇒	Score	⇒	
→ **EVALO**($\mathbf{s}_4$ (a) (a a))	⇒	Embed	⇒	Score	⇒	
Candidate: ...	⇒	Embed	⇒	Score	⇒	...

Figure 4.2: Recurent Neural Network (RNN) model for scoring candidates. Each constraint is embedded and scored separately, then pooled per candidate. The softmax probability scores determine which candidate to expand.

candidates. A summary of the model is presented in Figure 4.2

First, each constraint is separately tokenized and embedded. The tokenization process removes identifying information of logic variables, and treats all logic variables as the same token. This is necessary because there can be hundreds of distinct logic variables used during the course of a synthesis problem. Since each constraint is embedded and scored separately, the logic variable identity is of limited usefulness.

We learn separate RNN weights for each constraint type (**EVALO**, **LOOKUPO**, etc), obtaining embeddings for each constraint based on its type. Note that the particular set of constraint types differs depending on the DSL, so we need to learn different RNN's for each DSL.

Now, we use a MLP as a scoring function. Each constraint embedding is scored independently. The scores are then pooled together using a combination of average-pooling and min-pooling, depending on the structure of the constraint tree: we use average-pooling over conjunctions in the constraint tree, and min-pooling over disjunctions in the constraint tree. Conjunctions are more frequently seen in the constraint tree, and we find that using max-pooling hinders effective gradient flow.

Finally, we softmax over the scores for each candidate program, then make a discrete choice to predict the optimal candidate to expand in the following step. During training, this discrete choice is made by sampling from a multinomial distribution parameterized by the softmax probabilities. During test, we choose the candidate program with the highest score.

4.3 Graph Neural Network Model

The RNN model has the advantage of simplicity and performance. However, we may lose considerable information by removing the identity of logic variables. Two constraints associated with a logic variable may independently be satisfiable, but may be obviously unsatisfiable together. We have seen an example

in Table 3.2. We hypothesize that identifying information about logic variables is important for more difficult synthesis problems.

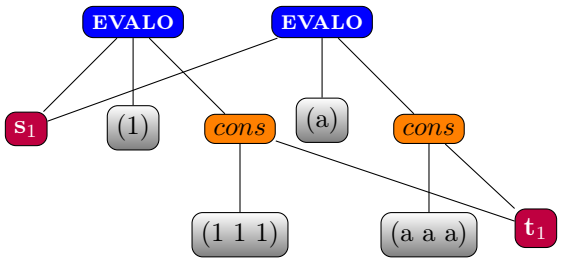

Figure 4.3: Graph Neural Network Representation of Constraints

To that end, we use a Gated Graph Neural Network (GGNN) model operating on each constraint. Since each constraint can be structured as a tree, we use a synchronous message schedule, much like in R3NN [21]. We propagate messages first from the leaves of the tree upwards towards the root, then from the root of the tree down to the leaves. Each node updates its own embedding prior to sending messages further up/down the tree. Crucially, logic variables appear as leaf nodes in each constraint tree. If two constraints reference the same logic variable, their representative constraint trees will send and receive messages to and from the same logic variable node. (Note that while each constraint is a tree, the set of all constraints, or the union of all constraint trees, will not be a tree. There can be cycles since different constraint trees can reference the same logic variable.)

The nodes types of the GGNN follow the tokens in the constraint tree, for example **EVALO**, *cons*, and others. Each node type has its own aggregation function and Gated Recurrent Unit weights. Further, the edge types will also follow the node type of the parent node. Most node types will have asymmetric children, so the edge type will also depend on the position of the child. Like the RNN, the set of node types differs depending on the DSL, so we need to learn different GNN models for each DSL.

We present the details of the GNN model in the following subsections.

4.3.1 Initialization

Each node is given an initial embedding. All non-leaf nodes are given zero initializations, since these embedding values will always be updated before being used to compute outgoing messages. For leaf nodes, both constants and logic variables have initial embeddings e_{label} that are learned as parameters of the model.

$$h_v = \begin{cases} e_{label} & \text{if } v \text{ is a leaf (constant or logic variable)} \\ \mathbf{0} & \text{otherwise} \end{cases}$$

4.3.2 Upward Pass

Upward passes are ordered leaf-to-root, so that a node receives all messages from its children and updates its embedding before sending a message to its parents. Since a non-leaf node always has a fixed number of children, the MERGE function is parameterized as an MLP with a fixed size input. The upward pass is summarized in Algorithm 3.

Algorithm 3 Upward Pass

Require: $C_1, \ldots, C_k$ constraint trees with ordered edges
1: **for all** node $v \in C_1 \cap C_2 \cdots \cap C_k$, in topologically sorted order **do**
2: **if** $\textsc{children}(v) \neq$ **then**
3: **for all** $v' \in \textsc{children}(v)$ **do**
4: $m_{v' \to v} = \textsc{message}_{\textsc{type}(v),\textsc{position}(v,v'),\textsc{up}}(h_{v'})$ ▷ Compute Messages
5: **end for**
6: $m_v \to \textsc{merge}_{\textsc{type}(v)}(m_{\cdot \to v})$ ▷ Merge Messages
7: $h_v \to \textsc{gru}_(m_v, h_v)$ ▷ Update Embedding
8: **end if**
9: **end for**

4.3.3 Downward Pass

Downward passes are ordered root-to-leaf, so that a node receives all messages from its parents and updates its embedding before sending a message to its children. Since the constraint tree is (almost) a tree, nodes will typically have one parent, so no merge function is required. The only exceptions are constants and logic variables. Constant embeddings should never change, so no updates are required. Logic variables represent unbound structures, and the same logic variable can appear multiple times in the constraint tree, so they can have multiple parents. Messages from different parents need to be merged together. Since different logic variables will have a different number of parents, average pooling is used as the merge function. The downward pass is summarized in Algorithm 4.

Algorithm 4 Downward Pass

Require: $C_1, \ldots, C_k$ constraint trees with ordered edges
1: **for all** node $v \in C_1 \cap C_2 \cdots \cap C_k$, in reverse topologically sorted order **do** ▷ DownwardPass
2: **if** $\textsc{children}(v) \neq$ **then**
3: **for all** $v' \in \textsc{children}(v)$ **do**
4: $m_{v \to v'} = \textsc{message}_{\textsc{type}(v),\textsc{position}(v,v'),\textsc{down}}(h_v)$ ▷ Compute Messages
5: **if** $\textsc{type}(v') \notin \{\textsc{lvar}, \textsc{constant}\}$ **then**
6: $h_{v'} \to \textsc{gru}_{\textsc{type}(v')}(m_{v \to v'}, h_{v'})$ ▷ Update Embedding for non-leaf nodes
7: **end if**
8: **end for**
9: **end if**
10: **if** $\textsc{type}(v) = \textsc{lvar}$ **then**
11: $m_v \to \textsc{merge}_{\textsc{var}}(m_{\cdot \to v})$ ▷ Merge Messages
12: $h_v \to \textsc{gru}_{\textsc{var}}(m_v, h_v)$ ▷ Update Embedding
13: **end if**
14: **end for**

4.3.4 Scoring

After a pre-defined number of upward and downward passes, we take the root embedding of each constraint as the final constraint embedding. Note that the final pass in the GNN should always be an upward pass, since the previous downward pass update to the logic variable embeddings needs to be reflected in the embeddings of the root nodes, which never change during an upward pass.

As in the RNN model, we score each individual constraint embedding using a multi-layer perceptron (MLP). Again, we pool together constraint scores belonging to the same candidate program using a combination of average-pooling and min-pooling, depending on the structure of the constraint tree. We

again use average-pooling over conjunctions in the constraint tree, and min-pooling over disjunctions in the constraint tree.

Finally, as in the RNN, we softmax over the scores for each candidate program, then make a discrete choice to predict the optimal candidate to expand.

4.4 Problem Generation

For a particular DSL $\mathcal{L}$ with a relational interpreter **evalo**, we programmatically generate training problems by first generating a desired program p consistent with $\mathcal{L}$. Specifically, we run the relational interpreter **evalo** "backwards" to generate arbitrary programs using the constraint logic programming language miniKanren. We then generate input/output examples (`i`, `o`) that satisfy **evalo**((*app* **p** `i`), ϕ, `o`), again using miniKanren. If input/output expressions contain constants (which they usually do), we choose random constants to ensure that a variety of constants appear in training.

4.5 Training

The ML agent is trained using supervised learning loss, specifically the cross-entropy loss on the leaf choice. We use the ADAM optimizer [17], with weight decay for regularization.

We use curriculum learning, beginning with training problems with shorter target programs and thus fewer synthesis steps, then gradually allowing larger ones. We generate training states by using the current model parameters to make action choices at least some of the time. We use scheduled sampling [4] with a linear schedule, to increase exploration and reduce teacher-forcing as training progresses. We use prioritized experience replay [23], storing explored states into a replay buffer and sampling mini-batches of states from this buffer. Experience replay is useful for reducing correlation in a minibatch, since nearby states in the same problem tend to be highly correlated. Priority sampling, re-sampling states with higher loss in a previous iteration with higher probability, has been shown to improve sample efficiency and improve the speed of training. To prevent an exploring agent from becoming "stuck", we stop episodes when the agent chooses the wrong candidate some number of consecutive times.

Importantly, we choose to expand two candidates per step during training, instead of the single candidate as described earlier. That is, we sample two candidates using the softmax probabilities, and miniKanren expands both candidates in the following step. During test time, we still expand one candidate per step, and use a greedy policy that always expands the highest scoring state. We find that expanding two candidates during training allows a better balance of exploration / exploitation during training, leads to faster training, and helps reduce cascading errors at test time.

Chapter 5

Experiments

Following the programming languages community, we focus on list construction as a natural starting point towards expressive computation. We use a small subset of Lisp as our target DSL $\mathcal{L}$. This subset consists of *cons*, *car*, *cdr*, along with several constants and function application. The full grammar is shown in Figure 5.1.

```
datum (D)          ::= () | #t | #f | 0 | 1 | x | y | a | b | s | (D . D)
variable-name (V)  ::= () | (s . V)
expression (E)     ::= (var V) | (app E E) | (lambda E) | (quote D) | (cons E E)
                     | (car E) | (cdr E) | (list E ...)
```

Figure 5.1: Subset of Lisp DSL

In the rest of this section, we run two experiments. First, we test on programmatically generated synthesis problems held out from training. We compare two miniKanren search strategies that do not use a neural guide, and three neural-guided models. Then, we test the generalizability of these approaches on three families of synthesis problems. In this second set of experiments we additionally compare against state-of-the-art systems λ^2, Escher, and Myth.

The reason we defer comparisons with symbolic systems outside of the miniKanren family is that λ^2, Escher, and Myth all assume a strongly typed DSL, whereas our training and test problems are dynamically-typed, improper list construction problems. Further, these symbolic systems do not interact well with input data that are non-numeric, whereas we use symbols. Encoding symbols as numbers proved unfair: for example λ^2 takes more time to synthesize the constant function that returns 6 than the constant function that returns 1 if neither value appeared in the input.

5.1 List Manipulation Problems in Lisp

We generate programs from $\mathcal{L}$ using the method described in Section 4.4. We test on 100 problems held out from training, and report results for combinations of symbolic-only and neural guided models. Some examples of generated problems are shown in Table 5.1.

We compare two variants of symbolic methods that use miniKanren. The "Naive" model uses biased-interleaving search, as described in [9]. The "+ Heuristic" model uses additional hand tuned heuristics described in [7]. The neural guided models include the RNN+Constraints guided search described in

Program: (lambda (car (car (var ()))))	
Input	Output
`((b . #t))`	`b`
`((() . b) . a)`	`()`
`((a . s) . 1)`	`a`
`(((y . 1)) . 1)`	`(y . 1)`
`((b))`	`b`
Program: (lambda (cons (car (var ())) (quote x)))	
Input	Output
`(a)`	`(a . x)`
`(#t . s)`	`(#t . x)`
`((1 . y) . y)`	`((1 . y) . x)`
`((y 1 . s) . 1)`	`((y 1 . s) . x)`
`(((x . x)) . y)`	`(((x . x)) . x)`
Program: (lambda (quote x))	
Input	Output
`y`	`x`
`()`	`x`
`#t`	`x`
`a`	`x`
`b`	`x`

Table 5.1: Example auto-generated training problems using Lisp grammar. The variables in a function are encoded using de Bruijn indices. The symbol . denotes a pair.

Section 4.2 and the GNN+Constraints guided search in Section 4.3. The RNN model uses 2-layer bi-directional LSTMs with embedding size of 128. The GNN model uses a single-layer GGNN with embedding size 64 and message size 128, and a single upward and downward pass. Further, we compare against a baseline RNN model that does not take constraints as input: instead, it computes embeddings of the input, output, and the candidate partial program using LSTM, then scores the concatenated embddings using a MLP. This baseline model also uses 2-layer bi-directional LSTMs with embedding size of 128. All models use a 2-layer neural network with ReLU activation as the scoring function. Note that none of the RNN models, including the baseline model, are attentional.

We report the percentage of problems solved within 200 steps. The maximum time the RNN-Guided search used was 11 minutes, so we limit the symbolic-only models to 30 minutes. The GNN-Guided search is significantly more computationally expensive, and the RNN baseline model (without constraints) is comparable to the RNN-Guided models (with constraints as inputs). All test experiments are run on Intel i7-6700 3.40GHz CPU with 16GB RAM. Results are shown in Table 5.2.

Table 5.2: Synthesis Results on Test Problems

Method	Percent Solved	Average Steps
Naive [9]	27%	N/A
+Heuristics [7]	82%	N/A
RNN-Guided (No Constraints)	93%	46.7
GNN-Guided + Constraints	88%	44.5
RNN-Guided + Constraints	**99%**	**37.0**

All three neural guided models performed better than symbolic methods in our tests, with the RNN + Constraints model solving all but one problem. The GNN-Guided + Constraints model was particularly

difficult to train: the training takes much longer, as it is much more difficult to batch dynamic GNN computations than RNN computations. The RNN model without constraints also performed reasonably, but took more steps on average than both RNN models. Figure 5.2 shows the number of actual vs optimal steps taken by the three models. The RNN+Constraints model is worse at smaller problems, but better overall. The RNN model without constraints typically takes fewer steps, but there are certain problems with 12 optimal steps where the model takes close to the cutoff of 200 steps.

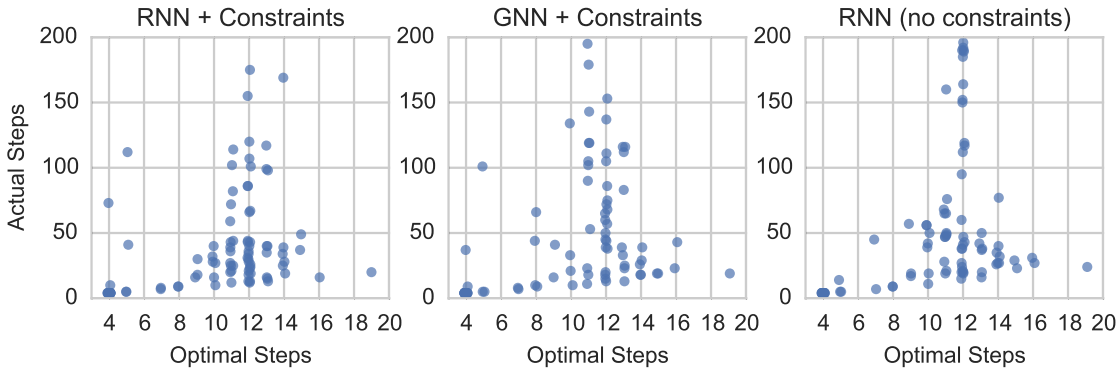

Figure 5.2: Optimal vs. Actual Steps for Solved Neural Guided Models

It is also interesting to note the problems for which the methods fail. Table 5.3 shows examples of problems on which the methods failed. The single problem that RNN + Constraints failed to solve is a fairly complex problem. The problems that the GNN + Constraints failed to solve all include a complex list accessor portion. This actually makes sense: it is conceivable for multi-layer RNNs to be better at this kind of problem compared to a single-layer GNN. The RNN without constraints also fails at complex list accessor problems.

Table 5.3: Sample Problems for which Neural Guided Synthesis Fails

Method	Problem
RNN + Constraints	(lambda (cons (cons (var ()) (var ())) (cons (var ()) (car (cdr (var ())))))))
GNN + Constraints	(lambda (car (car (cdr (cdr (car (cdr (cdr (var ())))))))))
	(lambda (car (car (car (cdr (car (car (cdr (var ())))))))))
	(lambda (car (cdr (cdr (car (car (cdr (var ()))))))))
	(lambda (car (cdr (car (cdr (cdr (car (cdr (var ())))))))))
	(lambda (cons (var ()) (car (car (car (car (var ()))))))))
RNN (No Constraints)	(lambda (car (car (cdr (cdr (cdr (cdr (cdr (cdr (var ())))))))))))
	(lambda (cdr (car (car (cdr (car (cdr (var ()))))))))
	(lambda (car (car (car (cdr (cdr (cdr (cdr (cdr (var ())))))))))))
	(lambda (cdr (cdr (cdr (car (cdr (cdr (cdr (var ())))))))))
	(lambda (cdr (car (car (car (car (car (var ())))))))))

5.2 Generalization and Comparison

In this experiment, we use the same model weights as above to demonstrate generalizability. We synthesize three families of programs of varying complexity: `Repeat(N)` which repeats a token N times, `DropLast(N)` which drops the last element in an N element list, and `BringToFront(N)` which brings the

last element to the front in an N element list. As a measure of how synthesis difficulty increases with N, `Repeat(N)` takes $4+3N$ steps, `DropLast(N)` takes $\frac{1}{2}N^2+\frac{5}{2}N+1$ steps, and `BringToFront(N)` takes $\frac{1}{2}N^2+\frac{7}{2}N+4$ steps. The largest training program takes optimally 22 steps to synthesize. The number of optimal steps in synthesis correlates linearly with program size.

We compare against state-of-the-art systems λ^2, Escher, and Myth. It is difficult to compare our models against other systems fairly, since all three state-of-the-art systems use type information. For `Repeat(N)`, `DropLast(N)` and `BringToFront(N)`, typed systems should have an advantage. Further, λ^2 assumes advanced language constructs like `fold` that other methods do not. Escher is built as an active learner, and requires an "oracle" to provide outputs for additional inputs. We forbid this functionality of Escher, and limit the number of input/output examples to five. We allow every method up to 30 minutes. Our model is further restricted to 200 steps for consistency with Section 5.1.

Note that if given the full 30 minutes, the RNN+Constraints model is able to synthesize `DropLast(7)` and `BringToFront(6)`, and the GNN+Constraints model is also able to synthesize `DropLast(7)`. However, Myth was able to solve Repeat(N) much faster than our model, taking less than 15ms per problem. Results are shown in Table 5.4. In summary, the RNN+Constraints and GNN+Constraints models are both able to solve problems much larger than those seen in training, and is competitive in its potential to generalize to larger programs.

The GNN+Constraints method in particular generalizes best compared to other neural guided models, despite its poor performance in test problems. In particular, the GNN+Constraints model actually solved `DropLast(N)` problems without any mistakes, always selecting the correct candidate partial program in each step. The RNN+Constraints method also shows better generalizability compared to the RNN without constraints. Even though the test performance of the two models were comparable in Section 5.1, the use of constraints as input improves generalizability.

Table 5.4: Generalization Results: largest N for which synthesis succeeded, and failure modes (out of **time**, out of **memory**, requires **oracle**, other **error**)

Method	Repeat(N)	DropLast(N)	BringToFront(N)
Naive [9]	6 (time)	2 (time)	- (time)
+Heuristics [7]	11 (time)	3 (time)	- (time)
RNN-Guided + Constraints	**20**+	**6** (time)	5 (time)
GNN-Guided + Constraints	**20**+	**6** (time)	**6** (time)
RNN-Guided (no constraints)	9 (time)	3 (time)	2 (time)
λ^2 [11]	4 (memory)	3 (error)	3 (error)
Escher [1]	10 (error)	1 (oracle)	- (oracle)
Myth [20]	**20**+	- (error)	- (error)

Chapter 6

Discussion, Limitations, and Future Work

6.1 Discussion

The experimental results suggest that a neural-guided search using miniKanren's internal constraints as input is a promising direction.

However, it is unclear whether the added difficulty of training a GNN model is beneficial. GNN models should allow communication between different constraints. However, in a small grammar as in Section 5.1, the communication between different constraints is not as important. Whether the RNN model will continue to outperform GNN model in larger DSLs is to be seen.

The failure case of GNN is also quite interesting. Finding an accessor function is a subproblem in its own right. This is an area where graph-based attention methods might be useful, where there are long-range dependencies across different parts of a graph. Further, finding an accessor is a domain where a symbolic system like miniKanren is unhelpful. A top-down approach like miniKanren's breaks the synthesis problem into subproblems by the outputs. A bottom-up approach would break the problem up using the inputs.

6.2 Limitations

A clear, practical limitation of the neural guided synthesis is that each "step" of synthesis is slower. For large synthesis problems, the expense of each step may be offset by taking fewer steps. This means that scalability is a major concern in building neural guided synthesis models.

Another limitation of our approach is that of program aliasing. For a set of input/output examples, there are many possible programs that are consistent with the examples. However, our supervised approach assumes that there is one correct answer. Further, because our training data is automatically generated, there is no guarantee that the programs used during training are the simplest. At test time, our model may generate programs more complex than necessary.

That we auto generate problems at all also means that there are likely domain-shift issues, where our programmtically generated problems contain unintended biases. We observed evidence of such biases. For example, we omitted `GetLast(N)`, a program that returns the last element of a size N list, from our

experimetal results. Despite this being a long accessor function, our model performed extremely well. In contrast, our model would perform poorly on `GetFirst(N)`, a program that returns the first element of a size N list. This is because our program generator first generates a target program, then chooses the simplest input/output examples consistent with that target program. Thus, during training, when synthesizing a target program like (car (var ())), we would likely observe input data like (x) instead of (x y a b x y). The neural model would learn to avoid generating simple programs when given complex inputs. Resolving this particular bias simply involves choosing input/output examples in other ways. However, having absolutely no bias in the generated problem set is extremely difficult if not impossible to achieve.

The completeness property of the biased interleaving search is also lost when using a neural-guided search strategy like ours. However, an incomplete search that is faster is likely more useful in practice. One can always run two parallel searches: a fast but incomplete search coupled with a slower complete search.

6.3 Future Work

One clear direction for future work is to synthesize recursive functions. Using the current set of weights trained on the small subset of Lisp, we can add in grammatical constructs one by one. The way that we would add recursion is to first add the Lisp grammatical constructs if and $null?$. With that in place, we can add in $foldr$, $foldl$ and map as basic building blocks for synthesizing recursive functions. With a more complex DSL, a comparison between RNN vs GNN encoding of constraints should be more clear. Further, any advantages of using constraints should also be more clear.

Longer-term future work includes using neural bottom-up attention combined with symbolic top-down "attention". Further, if synthesis of recursive functions is successful, we can speed up synthesis by distilling the neural networks model into a smaller model and optimize our code. Another direction is to explore using a different DSL, for example using another programming language other than Lisp, or use a FlashFill like DSL to perform spreadsheet string manipulation tasks.

Chapter 7

Conclusion

We presented a neural guided synthesis model where the neural guide takes as input the internal constraint encoding of the PBE problem used by miniKanren. Using a DSL consisting of a small subset of Lisp, we show promising results in the model's ability to generalize to larger problems. The ability to encode recursive problems is available in miniKanren, so learning to guide recursive program synthesis is left as future work.

Bibliography

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *International Conference on Computer Aided Verification*, pages 934–950. Springer, 2013.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[3] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *International Conference on Learning Representations*, 2017.

[4] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179, 2015.

[5] Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Deep API programmer: Learning to program with APIs. *arXiv preprint arXiv:1704.04327*, 2017.

[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] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP):8, 2017.

[8] William E. Byrd and Daniel P. Friedman. From variadic functions to variadic relations. In *Proceedings of the 2006 Scheme and Functional Programming Workshop, University of Chicago Technical Report TR-2006-06*, pages 105–117, 2006.

[9] William E. Byrd, Eric Holk, and Daniel P. Friedman. miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, pages 8–29. ACM, 2012.

[10] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural program learning under noisy I/O. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[11] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM, 2015.

[12] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.

[13] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[14] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *International Conference on Learning Representations*, 2018.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[18] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *International Conference on Learning Representations*, 2016.

[19] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *International Conference on Learning Representations*, 2016.

[20] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 619–630. ACM, 2015.

[21] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *International Conference on Learning Representations*, 2017.

[22] Scott Reed and Nando de Freitas. Neural programmer-interpreters. *International Conference on Learning Representations*, 2016.

[23] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *International Conference on Learning Representations*, 2016.

[24] Daniel Selsam, Matthew Lamm, Benedikt Bunz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *International Conference on Learning Representations*, 2018.

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

[26] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.