國立臺灣大學電資學院電信工程學研究所

碩士論文

Graduate Institute of Communication Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

探討程式執行對程式合成的影響

Investigating the Impact of Program Execution on Neural Program Synthesis

洪正維

Cheng-Wei Hung

指導教授: 孫紹華 博士

Advisor: Shao-Hua Sun Ph.D.

中華民國 113 年 7 月

July, 2024

doi:10.6342/NTU202401358

Acknowledgements

這篇論文中提到的兩個 project 都是在孫紹華教授的指導下，並與李威緒和夏良語一起討論和實作完成的，也感謝台大機器人學習實驗室全體同學給予了十分有用的討論與建議，使這些項目變得更加完善。謝謝陳彥均研究員在 NeurIPS 時指導我論文寫作的技巧，這些技巧同樣也有在碩士論文派上用場。

真的非常感謝我的指導教授孫紹華老師，以及 RLL 實驗室溫暖的大家。因為有你們，我的碩士兩年時光過得非常充實又快樂，和大家相處的每一天都充滿感恩。

研究生活中可能會遇到太多的困難，有各種原因都可能會導致身心靈不太健康而無法順利畢業。但是好禮佳在，我何其幸運地，遇到了正直善良、認真負責、互相尊重、關懷包容感恩、氣氛還和樂融融的大家，讓努力的過程中完全不孤單，看到你們努力的樣子也讓我更加督促自己（健康地）。如果沒有你們一路上的幫忙，真的無法想像有能順利畢業的今天。

我會珍惜與你們相遇的這段緣份，大家的前途應該本來就都非常光明，只由衷地希望大家未來也都能保持健康和快樂。

doi:10.6342/NTU202401358

doi:10.6342/NTU202401358

摘要

設計和實現能夠模擬機器行為的程式是人工智慧研究的核心任務之一。傳統的自然語言處理（NLP）方法在生成可讀性高的人類文本方面表現優異，然而在程式執行的準確性和一致性方面通常存在顯著差距。相較之下，神經程式合成（NPS）方法能夠通過直接從規範或示例生成可執行的程式，從而更深入地理解和操作機器行為。傳統自然語言處理（NLP）與神經程式合成（NPS）之間的主要差異在於，後者能夠生成可執行的程式碼，這些程式碼可以執行並檢查其行為。這種能力不僅有助於驗證合成的解決方案，還為從執行軌跡中學習開闢了新的途徑。

在本論文中，我們探索神經程式合成（NPS）與程式執行之間的共生關係，突顯程式執行如何成為合成程式的核心要素。本論文包含兩個主要項目：第一個項目探討學習程式執行或合成是否有助於學習另一個方面。通過分析這兩個方面之間的相互作用，我們研究神經網路如何有效地理解並生成可執行程式碼。第二個項目專注於提升從多樣化示範影片中合成程式行為的效率和有效性，同時提出指標驗證其在實際情境中的應用性。透過結構分析、前例分析和實驗驗證，本論文旨在深化對「程式執行對神經程式合成影響」的理解，並探討程式表示和行為之間的複雜性，以促進創造更安全、更強大的人工智慧系統。

關鍵字：機器學習、程式合成

doi:10.6342/NTU202401358

doi:10.6342/NTU202401358

Abstract

Understanding and synthesizing programs that represent machine behavior is a fundamental aspect of artificial intelligence research. Traditional approaches to natural language processing (NLP) have excelled in generating human-readable text but have largely remained detached from program execution. Conversely, Neural Program Synthesis (NPS) offers a paradigm shift by generating executable programs directly from specifications or examples, thereby enabling a deeper understanding and manipulation of machine behavior. The key difference between traditional NLP and NPS lies in the ability of the latter to generate executable code, which can be executed to examine its behavior. This capability not only facilitates the verification of synthesized solutions but also opens avenues for learning from execution traces.

In this thesis, we explore the symbiotic relationship between neural program synthesis (NPS) and program execution, highlighting how program execution serves as a fundamental aspect aiding in the synthesis of programs. This thesis comprises two main

doi:10.6342/NTU202401358

projects. The first project investigates whether learning program execution or synthesis aids in learning the other. By analyzing the interplay between these two aspects, we aim to uncover insights into how neural networks can effectively comprehend and generate executable code.

The second project focuses on learning to synthesize programs from entire execution traces. We aim to enhance the efficiency and effectiveness of neural program synthesis from diverse demonstration videos and propose realistic metrics to examine whether the method can be applied in more practical scenarios.

Through a combination of structural analysis, empirical studies, and experimental validation, this thesis seeks to advance our understanding of the impact of program execution on neural program synthesis. By delving into the intricacies between program representation and behavior, we aim to pave the way for more robust and intelligent AI systems.

Keywords: Neural Program Synthesis, Machine Learning

doi:10.6342/NTU202401358

Contents

doi:10.6342/NTU202401358

doi:10.6342/NTU202401358

doi:10.6342/NTU202401358

doi:10.6342/NTU202401358

List of Figures

doi:10.6342/NTU202401358

doi:10.6342/NTU202401358

List of Tables

doi:10.6342/NTU202401358

doi:10.6342/NTU202401358

Denotation

NPS	Neural Program Synthesis
NPE	Neural Program Execution
DSL	Domain Specific Language
NLP	Natural Language Processing
LLM	Large Language Model
CV	Computer Vision
PBE	Programming By Example
AI	Artificial Intelligence
API	Application Programming Interface
I/O	Input-Output
CNN	Convolutional Neural Networks
RNN	Recurrent Neural Networks

doi:10.6342/NTU202401358

LSTM	Long Short-Term Memory
RL	Reinforcement Learning
PRL	Programmatic Reinforcement Learning
LearkyReLU	Leaky Rectified Linear Unit
ReLU	Rectified Linear Unit
D2P	demo2program

doi:10.6342/NTU202401358

Chapter 1 Introduction

1.1 Background

1.1.1 Recent Advance in Deep Learning

Deep learning is a pivotal technique within the field of Machine Learning, demonstrating remarkable success across various domains such as Computer Vision (CV), Natural Language Processing (NLP) [1, 17], Speech Recognition and Synthesis, Robotics [11], Gaming [21], Autonomous Vehicles [25], and Recommendation Systems [4]. Deep learning utilizes neural networks with multiple layers like Convolutional Neural Networks (CNNs) [16], Recurrent Neural Networks (RNNs), attention mechanisms, and transformers as models that are designed to capture complex relationships between input data and predicted outcomes.

Deep learning models are optimized through algorithms that adjust the weights and biases of the network, often involving millions of parameters. The optimization process involves learning a loss function that minimizes the discrepancy between the predicted output and the ground truth. This learning process requires extensive annotated data to accurately map the input to the desired output."

doi:10.6342/NTU202401358

1.1.2 Teach Neural Network to Learn Rules

On the other hand, rule-based systems are built on only a limited number of explicit rules to address various situations. Prior works [30] have demonstrated that these models struggle to generalize well to new scenarios, such as adding more bits in digit addition [30], or increasing string lengths in string transformation tasks [7]. Given the constraints of limited resources, it is neither realistic nor sample-efficient to exhaustively ”remember” the mapping of every data point to its desired output. Moreover, pure hand-engineered rule-based systems often face challenges in extension and are susceptible to noise [7]. Consequently, researchers have been exploring effective neural methods to learn rules.

1.1.3 Programming By Example

A program consists of a set of instructions designed to guide a computer in performing a specific task or solving a particular problem. These instructions dictate the actions the computer should take under various conditions, making the program a typical rule-based system. Programming By Example (PBE) is a fundamental aspect of learning programs in the Artificial Intelligence (AI). Engineers provide the system with several instances of inputs and their corresponding desired outputs to guide the inference of the program’s underlying behavior. The machine’s objective is to fulfill these user-provided specifications or examples.

1.1.4 Neural Program Induction and Synthesis

Two common approaches aimed at enabling machines to infer or create programs from data are (1) Neural Programs Induction (2) Neural Program Synthesis. The former fo-

doi:10.6342/NTU202401358

cuses on training neural networks to infer underlying rules or algorithms from input-output examples, generating models that mimic a program's behavior and directly transform input data to the desired output as specified. Meanwhile, the latter focuses on training neural networks to infer the underlying rules or algorithms in the program, which are symbols or languages predefined by programmers, generating models that output desired programs. Humans can parse or compile these programs, execute them on the input data, and obtain the output to verify whether these outputs meet the desired output in the specification or examples. This thesis primarily investigates neural program synthesis.

1.1.5 The Execution Property of Program

A key difference between neural program synthesis and other applications of deep neural networks, such as Natural Language Processing (NLP), is that neural program synthesis generates 'explicit and executable' programs. A common approach in deep neural networks is to follow a supervised procedure to train a model that minimizes the difference between the predicted output and the ground truth. In addition to this, approaches in neural program synthesis can leverage additional signals, such as compilation and execution results of unit tests. This capability allows neural program synthesis to consider aliasing programs [1] [3] , verify the grammar and execution of unit tests for predicted programs before final submission [12], and even repair defective or partially correct programs [5, 10, 27, 28]. In summary, neural program synthesis can utilize program execution to ensure that the generated programs are both syntactically correct and semantically meaningful.

[1] aliasing programs: Programs consist of different tokens or representations that yield the same execution results. When using a small number of input/output examples, the chance of program aliasing increase.

doi:10.6342/NTU202401358

1.2 Research

In this section, I will introduce two research project conducted during my master's studies, detailed in Chapter 3 and Chapter 4: (1) **Bridging Neural Program Synthesis and Execution** (2) **Demo2Program+: Improve Neural Program Synthesis from Diverse Demonstration Videos and Evaluate on Demonstrations of Goal-Oriented Task**. Both projects are related to program execution. The first project investigates the contributions of program execution to neural program synthesis, while the second project explores the contributions of execution traces to neural program synthesis.

1.2.1 Bridging Neural Program Synthesis and Execution

Neural Program Execution (NPE) refers to the use of neural networks to interpret and execute computer programs. Unlike traditional program execution, which relies on a predefined set of rules and a virtual machine or processor, neural program execution involves a neural network learning to understand and perform the operations specified by the program. This approach aims to create models that can infer the underlying rules and behaviors of the program, dynamically follow the instructions, and execute them. Models often learn by training on numerous example programs and their executions, enabling the network to generalize and handle new programs.

Both Neural Program Synthesis and Neural Program Execution share the goal of leveraging neural networks to automate and enhance aspects of programming and computation. They both require an understanding of the structure and semantics of programs, as well as the ability to interpret dynamic information from the environment in which the

doi:10.6342/NTU202401358

instructions or programs are applied.

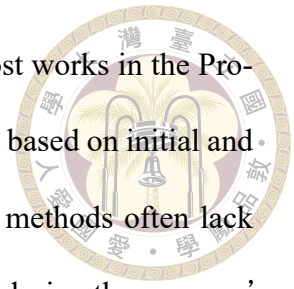
In the context of Programming By Example (PBE), the data formats differ between the two approaches. For Neural Program Synthesis, the model is provided with example input-output (I/O) pairs and is tasked with predicting the corresponding programs. Conversely, for Neural Program Execution, the model is given the programs and example inputs, and it aims to predict the desired outputs.

Building upon the shared characteristics of Neural Program Synthesis (NPS) and Neural Program Execution (NPE) mentioned above, our research aims to explore the similarities between these two approaches and investigate whether training in neural program synthesis can enhance execution. We employ two different types of encoder-decoder model structures—Long Short-Term Memory (LSTM) and Transformer—to determine if sharing model weights between NPS and NPE is beneficial. Our research focuses on the domain of string transformation, utilizing the Domain Specific Language (DSL) from Devlin et al. [7] and generating our dataset based on the methodology of Shin et al. [20].

1.2.2 Demo2Program+: Improve Neural Program Synthesis from Diverse Demonstration Videos and Evaluate on Demonstrations of Goal-Oriented Task

Understanding and interpreting decision-making logic in demonstration videos can indeed facilitate machines in collaborating with and mimicking human behavior. Due to its structured and predefined nature, a program offers an efficient approach to achieve this. Hence, prior works [6, 8, 22] have utilized neural program synthesizers to explicitly synthesize programs from demonstration videos. By utilizing demonstration videos

doi:10.6342/NTU202401358



or execution traces, this approach also addresses the limitation of most works in the Program By Examples (PBE) paradigm, which only synthesize programs based on initial and final states (I/O pairs) of the data processed by the program. Such methods often lack detailed information about the intermediate steps and decisions made during the program's execution.

However, obtaining real-world video data with precise program annotations is often challenging. Due to this difficulty, prior works [6, 8, 22] resort to generating artificial data of programs and demonstration videos, which are then split into training, validation, and testing sets. The method of creating this data involves purely random sampling of program tokens by a specific probability to form a program. Subsequently, initial states are randomly sampled and executed to obtain their traces, forming unit tests until a valid program and corresponding unit tests are found. Consequently, this human-created data is task-agnostic and often exhibits meaningless behavior. As a result, a domain gap persists between models trained on synthetic data and their application in the scenarios that users are interested in.

To this end, we utilize demonstration videos from Programmatic Reinforcement Learning (PRL) [14, 23] tasks as the target video to evaluate our method. Agents in these videos follow an optimal program policy generated by Liu et al. [14], accomplishing a strategy to fulfill the task and maximize the rewards they receive. Hence, this data more closely resembles a decision-making process compared to artificially created data. We use these videos as the model's input and calculate the rewards obtained by executing predicted programs as our evaluation metric.

Based on this new metric, we incorporate several features to investigate whether these

doi:10.6342/NTU202401358

improvements enhance our method, conducting our experiments in the Karel [18] domain:

1. **Utilizing Large Language Model (LLM):** We employ a code generation language model, essentially CodeT5 [26, 29], as it surpasses the common LSTM structure seen in previous works.

2. **Using Visual Words of Action/Perception as Model's Input:** Incorporating visual words of action/perception as input to the model [15].

3. **Training on Data with Better Generating Heuristic:** We generate data using Liu et al. [14]'s method that filter redundant information.

4. **Employing Action/Perception Prediction and Function/Branch Prediction as Auxiliary Objective Loss Functions:** We use action/perception prediction and function/branch prediction as our auxiliary objective loss functions.

5. **Utilizing Exedec [19] to Break Down Long Horizon Execution Traces:** We leverage Exedec [19] to break down a long-horizon execution trace into smaller pieces.

doi:10.6342/NTU202401358

doi:10.6342/NTU202401358

Chapter 2 Literature Review

The literature review chapter provides a comprehensive understanding of the existing knowledge, research trends, and advancements relevant to the subject matter of this study. In this chapter, we delve into the current state of research in the fields of **neural program synthesis**, **demo2program**, and **programmatic reinforcement learning**. Each of these areas is pivotal to the development and advancement of artificial intelligence and machine learning technologies.

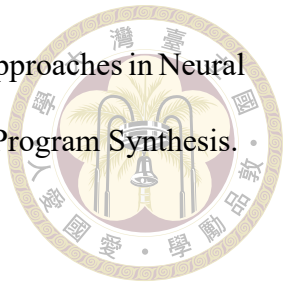
We begin with an exploration of neural program synthesis, which involves the automatic generation of computer programs using neural networks. Following this, we discuss the line of demo2program framework, which focuses on the conversion of user demonstrations into executable programs. We also examine programmatic reinforcement learning, a technique that combines programming and reinforcement learning to enable machines to learn complex tasks through interaction with their environment.

2.1 Neural Program Synthesis

Neural Program Synthesis (NPS) has emerged as a promising approach for automatically generating programs from high-level specifications. This paradigm leverages the power of neural networks to learn the mapping between input-output examples and their

doi:10.6342/NTU202401358

corresponding programs. In this section, I will introduce two types of approaches in Neural Program Synthesis: 1. Search-Based Program Synthesis 2. Seq2Seq Program Synthesis.



2.1.1 Search-Based Program Synthesis

Before the advent of neural networks for program synthesis, various search algorithms and heuristics were proposed for the program-by-example (PBE) paradigm in program synthesis. In search-based techniques, the output program is found using a process or solver designed by humans, operating within a deterministic time step. However, the cost of the search process, or the search space, increases exponentially with the token length in a program.

To address this challenge, Balog et al. [2] integrate neural network architectures with search-based techniques, rather than replacing them entirely. They predict the probability of the presence or absence of individual functions in a program to facilitate the search process. Similarly, Ellis et al. [9] also focus on learning the presence of functions. They determine whether a specific sequence of tokens is common in the dataset and, if so, add a new function to the library to represent this sequence. This augmentation of the library space speeds up the search process. Both methods cleverly integrate search techniques with neural network learning techniques, enhancing the efficiency and effectiveness of program synthesis.

2.1.2 Seq2Seq - Recurrent Neural Networks

Seq2Seq models, often utilized in natural language processing tasks, have been adapted for program synthesis. These models employ an encoder-decoder architecture to take se-

doi:10.6342/NTU202401358

quences of input-output pairs and generate the corresponding programs. One significant advantage of this approach is that programs can be executed before final submission, allowing the model to maintain a beam of several candidate programs and choose the one that satisfies the given specification.

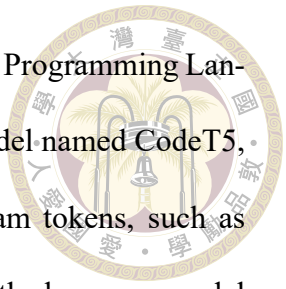
Devlin et al. [7] adapted an attentional Long Short-Term Memory (LSTM) structure and demonstrated that the Neural Program Synthesis (NPS) architecture outperforms both Neural Program Induction architectures and rule-based approaches. Additionally, their work showed that NPS is robust to noisy examples, highlighting its effectiveness in practical scenarios. Subsequently, numerous works utilizing LSTM have emerged, as LSTM is adept at encoding time series or sequences.

Bunel et al. [3] highlighted the problem of program aliasing, where different programs composed of varying tokens or representations yield the same execution results but are still considered correct. Additionally, models with LSTM structures in an end-to-end style often synthesize programs with syntactic errors, leading to failures in compilation or execution. To address these issues, they used reinforcement learning (RL) to tackle program aliasing and implemented a syntax checker to prevent syntactic errors.

2.1.3 Seq2Seq - Transformer

After Google's paper "Attention Is All You Need" [24], pre-trained transformer models for Natural Language (NL) like BERT and GPT have gained significant popularity in the machine learning research community. These models have also been shown to transfer well to common Programming Languages (PL) like Python and C++, benefiting a broad set of code-related tasks.

doi:10.6342/NTU202401358



Beyond applying the same processes used in Natural Language to Programming Languages, Yue Wang [29] proposed an encoder-decoder transformer model named CodeT5, pre-trained on several tasks that utilize the characteristics of program tokens, such as span prediction and identifier (token type) prediction. CodeT5 unifies the language model framework to solve various code-related tasks, including code generation, code translation, code summarization, and code refinement.

Following this, Le et al. [12] proposed CodeRL, which uses unit tests as rewards, similar to the approach of Bunel et al. [3]. By combining the important signals from specifications with the advantages of pre-trained transformer models, they achieved excellent results. The current state-of-the-art code generation models are predominantly trained using similar concepts and extensive amounts of high-quality data.

2.2 Demo2Program

"Demo2program," proposed by Sun et al. [22], stands for "*learning to synthesize program from diverse demonstration videos*". In this line of work, the model is trained to understand the decision-making process of the machine and interpret it as a program. They use entire intermediate execution states as examples in the Programming By Example (PBE) paradigm. Long Short-Term Memory (LSTM) networks are employed to encode each execution sequence (or demonstration video) into a vector. These vectors are then aggregated to summarize all videos into a compact vector, which is subsequently used to decode program tokens.

Additionally, they use a multi-task objective to decode all action and perception sequences from the encoded demonstration vector. This ensures that the model captures

doi:10.6342/NTU202401358

every event occurring in each step of the encoding process while performing supervised learning on the program tokens.

This multi-task objective method leverages more information from the execution process, emphasizing that actions and perceptions are important signals. Actions inherently correspond to functions used during program execution, while perceptions represent conditions within the program. These elements form the basis for decision-making during program execution. Building on this concept, Dang-Nhu [6] directly predicted the perceptions occurring in each frame of the state and the actions occurring between two adjacent states using a Convolutional Neural Network (CNN). These actions and perceptions were then treated as specifications, and a rule-based solver was used to search for a program that satisfies these specifications.

However, Dang-Nhu's [6] method caused the model to lose its ability to tolerate noise, similar to methods used before the advent of neural network approaches. Additionally, exhaustively searching for programs using a rule-based solver is time-consuming, especially when the program structure is complex or the length of the programs and videos is long. Therefore, in our project (detailed in Chapter 4), we use a Large Language Model (LLM) instead of a rule-based solver, combining the advantages of neural networks with improved noise tolerance and efficiency.

Furthermore, previous works have evaluated their methods on synthetic data generated using the same heuristics as the training data. This artificial data is created by random sampling, making it task-agnostic and containing many meaningless behaviors that do not appear in actual decision-making situations. Therefore, we evaluate our methods on videos from Programmatic Reinforcement Learning (PRL) tasks, where the machine's

doi:10.6342/NTU202401358

behavior is more strategic compared to human-created data.

2.3 Programmatic Reinforcement Learning

In contrast to policies in Deep Reinforcement Learning (DRL) that directly generate actions given a state, Programmatic Reinforcement Learning (PRL) synthesizes programs structured in a Domain-Specific Language (DSL) to guide agent behavior. Program policies can be parsed and executed based on an initial state when applied to the environment. Before application, users can interpret the policy and filter out any risky programs, allowing for human inspection as an additional safety measure. Furthermore, the structured nature of the program enhances the policy's generalizability to unseen scenarios.

Unlike the Programming By Example (PBE) paradigm, where the model is trained to fulfill a specification, the model in PRL is trained to optimize a reward given a task with a clear goal. This makes the policy a strategy with a specific purpose. Executing this policy on several initial states generates demonstrations that serve as an ideal target for evaluating our method. If we can synthesize programs based on demonstrations from PRL, it would demonstrate our method's superiority in learning a decision-making process.

doi:10.6342/NTU202401358

Chapter 3 Bridging Neural Program Synthesis and Execution

As discussed in Section 1.2.1, learning neural program synthesis and learning neural program execution both require an understanding of the structure and semantics of programs, as well as the ability to interpret dynamic information from the environment in which the instructions or programs are applied. Our research aims to explore the similarities between these two approaches and investigate whether training in neural program synthesis can enhance execution.

In the context of Programming By Example (PBE), the data formats differ between the two approaches. For Neural Program Synthesis, the model is provided with example input-output (I/O) pairs and is tasked with predicting the corresponding programs. Conversely, for Neural Program Execution, the model is given the programs and example inputs, and it aims to predict the desired outputs. Hence, the structure design of the encoder and decoder should be both different between NPS and NPE.

In this chapter, I will discuss the problem formulation, model design, and the result discussion.

doi:10.6342/NTU202401358

3.1 Problem Overview

3.1.1 Problem Formulation

The goal of neural program synthesis is to token-by-token generate a program prediction P' that can correctly produce the desired outputs given a set of k input-output (I/O) string examples $\{(I_k, O_k) \mid k = 1, 2, \ldots, K\}$. Here O_k represents the output obtained by executing the ground truth program P on the k-th input I_k. The synthesized program P' should be able to produce the same output O_k when applied to the corresponding input I_k.

The goal of neural program execution is to predict the output O_k for a given set of k input string examples I_k and a program P. The model should accurately replicate the output O_k as if the ground truth program P were executed on the k-th input I_k.

Both paradigms use Cross-Entropy Loss to calculate the difference between the ground truth and the prediction at a token wise level. We minimize Cross-Entropy Loss as our objective function.

3.1.2 Domain Specific Language

$I_1 =$ `January`	$O_1 =$ `jan`
$I_2 =$ `February`	$O_2 =$ `feb`
$I_3 =$ `March`	$O_3 =$ `mar`
$P =$ `ToCase(Lower, SubStr(1,3))`	

Table 3.1: FlashFill Example

Following the approach of Robustfill [7] we conducted our experiments in a string transformation domain, structuring our program P in a Domain Specific Language (DSL) as defined in Figure A.1. The inputs I_k and the outputs O_k are strings composed of multiple

doi:10.6342/NTU202401358

characters, and the program P functions as a transformation mechanism, converting one string into another.

An example of this process is illustrated in Table 3.1. The program `ToCase(Lower, SubStr(1,3))` truncates the input strings to obtain substrings from positions 1 to 3 and then converts these substrings to lowercase.

3.1.3 Training Data

However, the *FlashFillTest* dataset used by Devlin et al. [7] in Robustfill is not publicly available. Therefore, we generated our own dataset based on the DSL of *FlashFillTes*. Our data generation method follows the approach of Shin et al. [20]:

1. We first sample program tokens token-by-token to form a valid program P that can be correctly parsed and executed.

2. Then, we sample string characters token-by-token to form a valid input string that is syntactically correct for the program to process and generate the corresponding output string.

Step 2 is repeated until we have 16 pairs of input-output (I/O) strings and a corresponding program that satisfies the requirements, which are then added as one data point to the dataset.

We created a dataset with 100,000 training examples, 5,000 validation examples, and 5,000 testing examples.

Table 3.2 presents an example of our generated data. Let's take I_1 as an example: The program P is a nested function where the innermost components are "`Regex r(`

doi:10.6342/NTU202401358

inputs	outputs
I_1 = Ne{Tetvr}Agu,m[Ngz.Hctbg]Dd?Op I_2 = Dk@q,Aqh(Xao}Cxqp:Ov@Qey,Czc# I_3 = rD)Nun)It,Sdb$Zkmz:Hv#3Vr!Sy)	O_1 = Tetvr O_2 = Aqh O_3 = It
P = Concat c(GetToken n(Propcase -3 n) v(SubStr s(Regex r(Alphanum 0 End r) Regex r(Propcase -4 End r) s) v) c)	

Table 3.2: Our String Transformation Dataset Example

Alphanum 0 End r)" and "Regex r(Propcase -4 End r)". These expressions represent "*find the ending position of the 0th string in alphanumeric character*" and "*find the ending position of the -4th string in properly cased character*," respectively.

- "{" is identified as the 0th string in alphanumeric character.
- "Ngz" is identified as the -4th string in properly cased character.

Next, the two positions found by the above functions are used as indices for the SubStr s(index1 index2 s) function, which truncates characters before index1 and after index2.

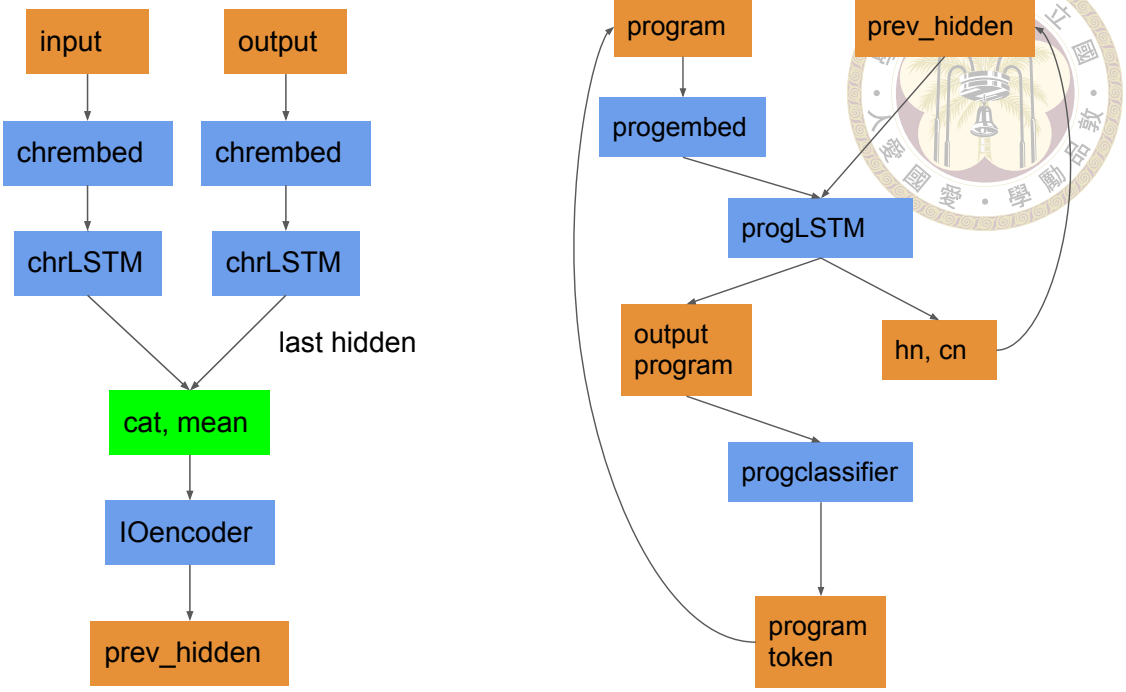
The resulting substring "{Tetvr}Agu,m[Ngz" is then passed as an argument to GetToken n(Propcase -3 n) v(arguement v) to obtain the -3th string in properly cased character Tetvr.

After forming an expression of length 1, we use Concat c(c) to concatenate more expressions if the program's expression length is greater than 1.

3.2 Methods and Preliminary Results

In this section, we discuss the encoder-decoder architecture in the program synthesizer for Neural Program Synthesis and in the program executor for Neural Program Ex-

doi:10.6342/NTU202401358



(a) **The encoder of program synthesizer** (b) **The decoder of program synthesizer**

Figure 3.1: **The structure of the a simple version of synthesizer.** Brown squares are representing vectors, blue squares are for trainable module, green squares are for functions of arithmetic computation.

ecution. Also, we discuss how the setting of teacher forcing and the scheduled-sampling affect the result.

3.2.1 Examine the Effects of the Attention Layer in Program Synthesizer and the Effects of the Scheduled Sampling

Simple Version of Program Synthesizer We first try a simple version of LSTM-based program synthesizer as in **??**:

Encoder (Simple Version) Each character in the input and output strings is embedded into a vector with a hidden size of 100. These embedded character vectors, along with an initial zero hidden vector and cell vector, are then fed sample-wise into a long short-term memory (LSTM) network.

doi:10.6342/NTU202401358

We concatenate the final hidden vector of the input string with the final hidden vector of the output string, and similarly concatenate the final cell vector of the input string with the final cell vector of the output string within the LSTM. We then compute the mean value across all 16 examples.

This concatenated input/output vector is fed into an I/O encoder, which consists of multiple linear layers with a hyperbolic tangent activation function, completing the encoding process for the input/output strings. The resulting encoded I/O vector has a size of 300, aligning with the size of the program embedding.

Decoder (Simple Version) Each program token in the program is embedded into a vector with a hidden size of 300. These embedded program vectors, along with an initial hidden and cell vector from the encoded I/O vector, are then fed into a long short-term memory (LSTM) network. The output program tokens are fed into a token classifier, which consists of multiple linear layers with a Leaky Rectified Linear Unit (LeakyReLU) activation function. The token with the highest probability is then sampled as the program token.

Decoder Training Policies: Teacher Forcing and Scheduled Sampling When training sequence-to-sequence models, such as those used in neural program synthesis, the choice of decoder training policy can significantly impact the model's performance. One common approach is Teacher Forcing, where the model is trained by providing the ground truth token as the next input token at each step, rather than using the model's own predictions. This method helps the model learn the correct sequence of tokens more quickly and ensures stable training by always feeding the correct context. However, a major drawback of

doi:10.6342/NTU202401358

Teacher Forcing is that it can lead to a discrepancy between training and inference, as the model is never exposed to its own errors during training.

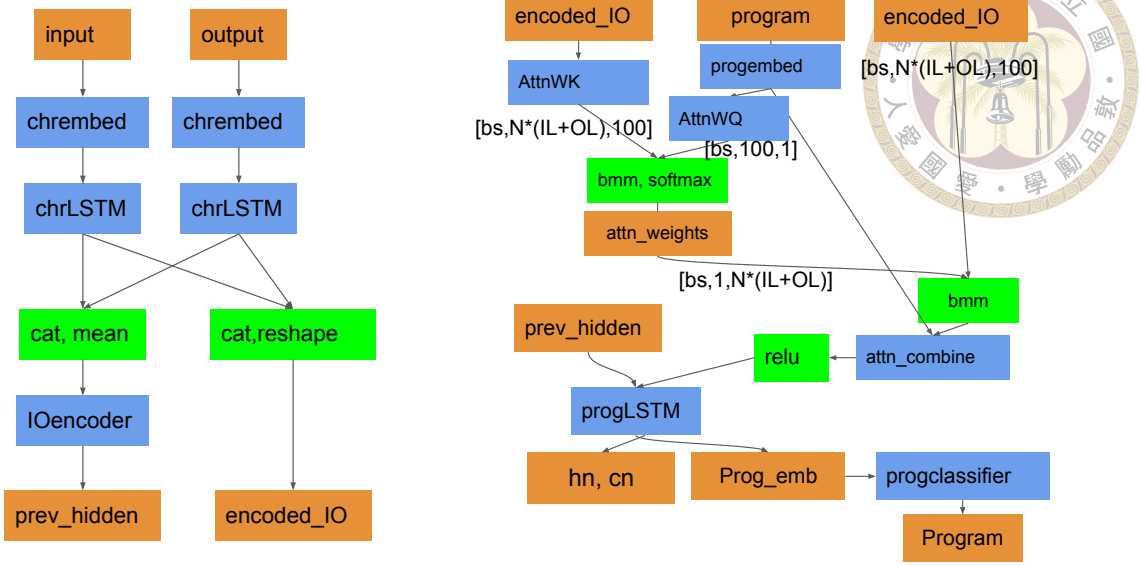
To address this issue, Scheduled Sampling is introduced as an alternative training policy. Scheduled Sampling gradually transitions from Teacher Forcing to using the model's own predictions during training. Initially, the ground truth tokens are used as inputs, but over time, the model increasingly relies on its own predicted tokens. This approach helps mitigate the exposure bias introduced by Teacher Forcing, as the model learns to handle its own mistakes and better adapt to the conditions it will face during inference. Scheduled Sampling strikes a balance between stability and robustness, aiming to improve the model's generalization and performance on unseen data.

Here, we implement scheduled sampling by linearly decreasing the ratio of using ground truth tokens from 1 to 0 during the training phases, spanning from the 50th epoch to the 100th epoch. Both scheduled sampling and teacher forcing are trained for 150 epochs to facilitate a thorough comparison of our results.

Adding attention layer of program synthesizer To evaluate the effectiveness of incorporating attention mechanisms in neural program synthesis, we aim to compare its performance against the simpler structure described earlier. The motivation behind introducing attention is to enable the model to focus on relevant parts of the input/output examples, potentially enhancing its ability to capture complex relationships within the data. We hypothesize that the attention mechanism will allow the model to dynamically weigh the importance of different I/O pair tokens when generating the output program, thus improving its performance in synthesizing programs from input/output examples.

Table 3.3 presents the preliminary training status during the selection of the neural

doi:10.6342/NTU202401358



(a) **The encoder with attention** (b) **The decoder with attention**

Figure 3.2: **The structure of the synthesizer with attention layer.** Brown squares are representing vectors, blue squares are for trainable module, green squares are for functions of arithmetic computation.

Method	accuracy/train	accuracy/eval	loss/train	loss/eval
teacher forcing	0.9789	0.7234	0.0581	6.956
scheduled sampling	0.9087	0.7708	0.2563	1.069
attention layer + teacher forcing	0.9705	0.1641	0.08274	10.741
attention layer + scheduled sampling	0.9342	**0.8101**	0.1869	0.7713

Table 3.3: The preliminary training status during architecture selection

program synthesis architecture. The table includes four different training configurations: (1) Teacher Forcing, (2) Scheduled Sampling, (3) Attention Layer with Teacher Forcing, and (4) Attention Layer with Scheduled Sampling. For each configuration, the table shows the accuracy [1] and loss metrics both during training (accuracy/train, loss/train) and evaluation (accuracy/eval, loss/eval).

In the results, we observe that when utilizing Teacher Forcing alone, the model achieves high training accuracy of 97.89%. However, the evaluation accuracy drops notably to 72.34%, indicating potential overfitting. On the other hand, Scheduled Sampling shows

[1] Accuracy here refers to the ratio of predicted program tokens that match the ground truth.

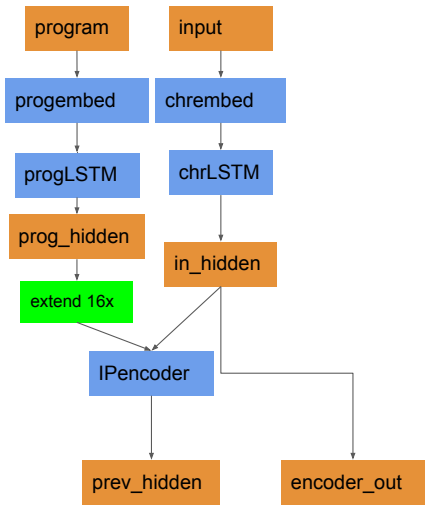
doi:10.6342/NTU202401358

a significant improvement in evaluation accuracy (77.08%), suggesting that it effectively mitigates overfitting by introducing a mechanism that forces the model to rely less on the ground truth during training. This only comes with a slight decrease in training accuracy (90.87%) despite the model has already fully relied on its own prediction here.

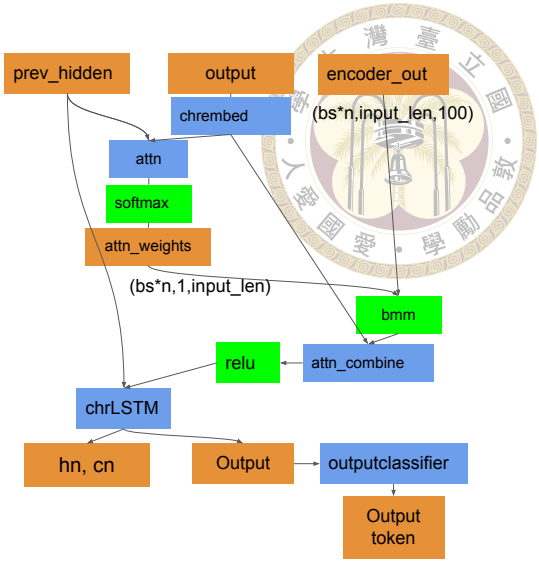
Introducing an attention layer along with Teacher Forcing dramatically decreases the evaluation accuracy to 16.41%, indicating a substantial drop in generalization performance. This observation aligns with the notion that the attention mechanism, while beneficial for focusing on relevant parts of the input, may lead to overfitting by allowing the model to memorize the training data more effectively. However, when the attention layer is combined with Scheduled Sampling, the evaluation accuracy improves significantly to 81.01%, surpassing all other configurations. This suggests that the attention mechanism, when used in conjunction with Scheduled Sampling, enhances the model's ability to synthesize programs from input/output examples by enabling it to dynamically weigh the importance of different input tokens, thereby achieving better generalization.

In conclusion, the results demonstrate that employing an attention layer alongside Scheduled Sampling yields the best performance in terms of evaluation accuracy. This combination effectively balances the benefits of attention in capturing complex relationships within the data while mitigating overfitting through Scheduled Sampling, making it the optimal choice for neural program synthesis tasks.

doi:10.6342/NTU202401358



(a) **The encoder of program executor**



(b) **The decoder of program executor**

Figure 3.3: **The structure of the program executor.** Brown squares are representing vectors, blue squares are for trainable module, green squares are for functions of arithmetic computation.

	Train	Valid	Test
Exact match of synthesis	73.65	29.92	29.58
All-sample-right of synthesis	75.75	37.92	35.43
Exact match of execution	0.66	0.34	0.37
Partial-sample-right of execution	27.05	17.47	16.53

Table 3.4: Result of neural program synthesis and neural program execution

3.2.2 Experiment results on Neural Program Synthesis and Unsuccessful results on Neural Program Execution

After decide the architecture of program synthesizer, we also design our execution predictor with similar concept (illustrated in Figure 3.3). However, in the context of neural program execution, the performance of the executor fell short of expectations. (Result is presented in Table 3.4.) [2] This was compounded by the complexity of both the training settings and the evaluation metrics. Consequently, the original intention of utilizing neural program execution to aid in neural program synthesis became less feasible.

[2] *Exact match*: The ratio of predicted program is same as the ground truth. *All-sample-right*: The ratio of predicted program meets all I/O pairs specification, it consider aliasing program as correct. *Partial-sample-right of execution*: The ratio that predicted output string is correct samplewisely.

doi:10.6342/NTU202401358

3.3 Discussion with Contributions from Team Member

Furthermore, our team member, Wei-Hsu Lee, delved into another facet of the research as documented in his master thesis [13]. Instead of focusing on investigating the structure of LSTM, he conducted experiments with CodeT5 [29]. Lee directly fed input data into the code-generating language model, CodeT5, and fine-tuned it. His experimental setup yielded the following observations:

1. Training with neural program execution did not significantly aid neural program synthesis.
2. The effectiveness varied when training with different I/O and program settings: single program with multiple I/O, multiple programs with single I/O, and multiple inputs with multiple outputs.
3. The model was confused by different programs with partially identical input-output pairs.
4. Utilizing natural language data from Wikipedia as evaluation data performed poorly compared to the original testing set.

From these experiments, three key insights emerged:

1. Large Language Models (LLMs) demonstrate significantly greater power than handcrafted LSTM models.
2. Randomly generated datasets fail to generalize well to other data distributions.
3. Trivial settings have a substantial impact.

doi:10.6342/NTU202401358

Considering these findings, we have opted to conclude this project at its current stage. Our subsequent project Demo2Program+: Improve Neural Program Synthesis from Diverse Demonstration Videos and Evaluate on Demonstrations of Goal-Oriented Taskalso benefits from these insights.

doi:10.6342/NTU202401358

Chapter 4 Demo2Program+

Demo2Program+: Improve Neural Program Synthesis from Diverse Demonstration Videos and Evaluate on Demonstrations of Goal-Oriented Task.

As discussed in the introduction section (Section 1.2.2) and the literature review section (Section 2.2), we have two primary directions:

1. Enhancing the performance of Demo2Program based on insights gleaned from previous works [6, 8, 22].
2. Employing demonstrations from Programmatic Reinforcement Learning (PRL) tasks to assess our method, aiming for higher rewards.

In this chapter, I will discuss the problem overview, method design, and the result.

4.1 Problem Overview

4.1.1 Karel Environment and Domain Specific Language

We conduct our experiment on Karel domain. Karel is a programming language used in educational contexts, particularly in the field of computer science and programming education. In the Karel programming language, the language interacts with a simulated

doi:10.6342/NTU202401358

machine or environment where the robot (Karel) operates. This simulated environment typically consists of a grid world where Karel can move around, pick up, and put down markers. The Domain Specific Language (DSL) defined in Figure A.2 provides commands that control Karel's actions within this environment.

Programmatic Reinforcement Learning Tasks for evaluating includes 8 tasks: , TopOff, DoorKey, FourCorners, Harvester, Maze, Seeder, StairClimber, and OneStroke [14, 23]. Figure A.3 presents an example of the task DoorKey. We use demonstrations from an sample well-trained program policy from Liu et al. [14] as our specification.

4.1.2 Problem Formulation

Generally, the problem formulation remains the same as in Section 3.1.1, except that in Chapter 4, we use execution traces D_k instead of I/O pairs I_k, O_k. D_k is actually composed of the states S_k^t from several frames. Here, S_k^t represents the state at the t-th timestep of the k-th example.

We use original dataset generated by demo2program [22] with 25,000 training examples, 5,000 validation examples, and 5,000 testing examples. Also, we discuss the heuristic of generation in Section 4.2.3.

4.2 Methods

We utilize 5 technique to improve the performance of demo2program:

1. Utilizing Large Language Model (LLM) as our encoder-decoder.
2. Using Visual Words of Action/Perception as Model' s Input.

doi:10.6342/NTU202401358

3. Training on Data with Better Generating Heuristic.

4. Employing Action/Perception Prediction and Function/Branch Prediction as Auxiliary Objective Loss Functions.

5. Utilizing Exedec [19] to Break Down Long Horizon Execution Traces.

4.2.1 Utilizing Large Language Model as Our Encoder-Decoder

As previously concluded, transformer-based models are more powerful than LSTM-based models. Therefore, we utilize CodeT5 [29] as our model. We fine-tune CodeT5 on the demo2program dataset to enable it to learn the state and mechanisms of the Karel world. Leveraging its pretrained knowledge and the fine-tuning process, the model gains an understanding of the semantic and syntactic structures of the Karel Domain Specific Language.

4.2.2 Using Visual Words of Action/Perception as Model's Input

As Sun et al. [22] revealed, action and perception play important roles in synthesizing programs. Actions inherently correspond to functions used during program execution, while perceptions represent conditions within the program. These elements form the basis for decision-making during program execution. In fact, we don't even need the raw input state; the actions and perceptions alone determine the decision-making process of the program.

Hence, following the approach of Manchin et al. [15], we use actions and perceptions as visual words instead of the raw state of Karel to feed into the model. We first train a Convolutional Neural Network (CNN) classifier to predict the perception in each frame of

doi:10.6342/NTU202401358

state, and then train another CNN classifier to predict the action occurring between every two adjacent frames. We then encode the action/perception predictions into visual words as the model's input. The model synthesizes the output program from these input action/perception visual words. In this way, we can filter out the raw input state with noise or unseen states in new scenarios, which might be challenging for synthesis, but can still predict their actions and perceptions.

4.2.3 Training on Data with Better Generating Heuristic

Liu et al. [14] improved the data generation method of the demo2program dataset [22] in their work HPRL. They filtered out programs with meaningless movements such as adjacent `putMarker, pickMarker, turnLeft, and turnRight`. Moreover, they generated valid execution traces that cover all branches of the program to address all possible situations the program may encounter.

For the sake of convenient comparison with previous baselines [6, 8, 22], we primarily utilize the demo2program dataset directly. However, we can also leverage the data generation heuristic proposed by Liu et al. [14] to enhance the performance of our model.

4.2.4 Employing Action/Perception Prediction and Function/Branch Prediction as Auxiliary Objective Loss Functions

Inspired by the method proposed by Sun et al. [22], which decodes actions and perceptions as auxiliary objective functions, we also aim to incorporate some essential signals of programs into our framework as auxiliary objective functions. However, since we have already used action/perception as our visual words, it would be redundant for a power-

doi:10.6342/NTU202401358

ful transformer model to decode them again as an objective. To address this, we propose "*Function*" and "*Branch*" as our new auxiliary objective loss functions.

- *Function* is composed of 5 Boolean values representing whether specific perceptions are used as conditions for machine decision-making at specific timesteps during the execution of the program.
- *Branch* is composed of 5 Boolean values representing whether specific branching functions (for example: `while`, `if`, `else`, `ifelse`, `repeat`) are called at specific timesteps during the execution of the program.

We believe these two features, *Function* and *Branch*, are important aspects of a program. Therefore, we feed the output vector of the CodeT5 encoder into two decoders of multi-layer perceptron to predict *Function* and *Branch* respectively. We use the Cross-Entropy Loss between the *Function* and *Branch* prediction and the ground truth as our multi-task objective.

4.2.5 Utilizing ExeDec [19] to Break Down Long Horizon Execution Traces

Shi et al. [19] propose a decomposition-based program synthesis framework, ExeDec, along with a benchmark of 5 generalization tasks to evaluate the generalization ability of a program synthesizer. The ExeDec framework breaks down the synthesis process into steps of synthesizing smaller subtasks instead of directly tackling a complex task.

At each step of the process:

1. ExeDec uses a *subgoal model* to predict the intermediate subgoal states O_k^{t+1} that

doi:10.6342/NTU202401358

Model	Execution	Sequence
demo2program	72.1	41.0
watch-reason-code	74.7	43.3
PLANS	**91.6±1.3**	34.2±0.5
t5-small	86.64	50.9
codet5-small	90.96	**54.16**
codet5-base	90.7	54.1
codet5p-220M	87.26	52.06

Table 4.1: **Synthesis result (with method 1 and 2) on the dataset generated by demo2program.** The orange color means we reproduce other's work [15], red color is proposed by us.

will be achieved in the current step, instead of directly predicting the final desired output O_k.

2. The *synthesizer model* then predicts a subprogram P^t that can achieve the subgoal states O_k^{t+1} by executing the subprogram P^t on the current states O_k^t.

3. The prediction P^t of the *synthesizer model* is used to achieve the next current state E_k^t.

4. If the next current state E_k^t matches O_k for all k examples, we combine all the subprograms P^1, P^2, ... , P^t to form a final prediction P and terminate.

Shi et al. [19] tests ExeDec on their benchmark to show that this framework has a ability of generalization, which perfectly fits our needs. Therefore, we reproduce a ExeDec on Karel domain with a demonstration version instead of programming by I/O pairs. Both the *subgoal model* and the *synthesizer model* are CodeT5.

doi:10.6342/NTU202401358

4.3 Results and Discussion

4.3.1 Synthesis Results (with method 1 and 2

Table 4.1 show a result of applying visual word and fine-tuning CodeT5 (i.e. using method 1 and 2 mentioned in Section 4.2) on synthesizing program in demo2program dataset. *Execution* means the ratio of the program prediction that meet the specification on all the example, i.e., behave the same as the ground truth. *Sequence* means the ratio of the program prediction matches the ground truth exactly. We reproduce VT4 [15] on the Karel domain and try three different size of CodeT5 model. All the method perform good the human-generated dataset, we use CodeT5-small as our structure in later discussion.

Method	fourCorners	topOff	harvester	Maze	stairClimber	doorkey	oneStroke	seeder	average
ground truth	1	1	1	1	1	1	0.711	1	0.964
D2P_k=1	-0.001	-0.001	-0.001	-0.001	0	-0.001	-0.001	-0.001	-0.001
D2P_k=100	0	0	0.1944	-0.001	-0.001	-0.001	0.7111	0.0278	0.116
CodeT5_k=1	0	0	0	0.2	1	0	0.6222	0	0.228
CodeT5_k=100	0.250	0.9	0.1389	0.6	1	0	0.7889	0.1389	**0.477**

Table 4.2: Compare our model with D2P in PRL.

We then compared this setting with our reproduced demo2program (D2P) in PRL task. K means choose the best program that can get the highest reward among the k most probable candidates. Reward for each task is calculated from the mean of rewards yielded in all example. We can see that original structure of D2P can not generalize to a goal-oriented dataset other than their syntactic dataset, but our method can.

We want to further improved the performance of our method, so we tried the method 3, 4, 5 (mentioned in Section 4.2) to see whether there is an improved.

doi:10.6342/NTU202401358

Method	fourCorners	topOff	harvester	Maze	stairClimber	doorkey	oneStroke	seeder	average
ground truth	1	1	1	1	1	1	0.711	1	0.964
hprl 25k	1.000	1.000	0.139	1.000	1.000	0.000	0.119	0.139	**0.550**
demo2program	0.175	0.700	0.139	0.800	0.700	0.000	0.883	0.000	0.425
hprl 800k	0.250	1.000	0.000	1.000	1.000	0.200	0.575	0.028	0.507

Table 4.3: Testing the effect of fine-tuning on different dataset.

4.3.2 Comparing the Dataset Generation Method (with method 3)

We use the heuristic mentioned in Section 4.2.3 to generate datasets in different size: 25k and 800k. We fine-tuned our model in different dataset and evaluate it on PRL tasks. The results is presented in Table 4.3. We can see that the better heuristic help model learning, and the larger amount of data point may be redundant.

4.3.3 Multi Task Objective Loss - function&branch (with method 4)

dataset: demo2program	Exact			Execution		
	Train	Eval	Test	Train	Eval	Test
no loss	100	52.96	54.07	100	88.4	88.64
ap loss	100	53.08	54.68	100	89.68	89.92
all loss	100	53.24	**54.84**	100	89.78	**90.22**

Table 4.4: Performance metrics on synthetic dataset for using different auxiliary loss

Table 4.4 shows that using auxiliary loss of Action/Perception and Function/Branch slightly increase the performance in the synthetic dataset. But there is no obvious improvement when evaluating PRL tasks, as shown in Table 4.5.

	fourCorners	topOff	harvester	Maze	stairClimber	doorkey	oneStroke	seeder	average
ground truth	1.000	1.000	1.000	1.000	1.000	1.000	0.711	1.000	0.964
D2P, no loss	0.000	0.500	0.167	1.000	1.000	0.000	0.053	0.000	0.389
D2P, ap loss	0.250	0.500	0.167	0.800	0.900	0.000	0.028	0.000	0.342
D2P, all loss	0.075	0.900	0.167	1.000	0.700	0.000	-0.001	0.028	0.399
hprl, no loss	0.175	1.000	0.111	1.000	1.000	0.050	0.086	0.000	0.464
hprl, ap loss	0.150	1.000	0.139	1.000	1.000	0.000	0.111	0.028	0.468
hprl, all loss	0.175	0.300	0.028	1.000	1.000	0.000	0.108	0.167	0.372

Table 4.5: PRL Performance metrics for fine-tuning on different dataset and using different auxiliary loss

doi:10.6342/NTU202401358

4.3.4 ExeDec (with method 5)

ExeDec reproduce is implemented by my team member, Wei-Hsu Lee. The detail is elaborated in his thesis [13]. We can yield two observation: 1. *subgoal model* is really hard to train. 2. Even training *synthesizer model* with ground truth subgoal states, we still cannot obtain a correct program that terminate the process during evaluation.

We hypothesize that observation 1 is due to the fact that predicting a subgoal from an out of distribution input is no easier than predicting a program from an out of distribution input. We hypothesize that observation 2 is because there is still many prediction error from step by step, which may cause error propagation. As a result, this method can be beneficial if we carefully hand-craft the algorithm and use a good representation of subgoals.

4.4 Conclusion

Due to the difficulty in obtaining real-world labeled demonstration and program data, the demo2program approach typically relies on training with synthetic data. We use demonstrations of programmatic reinforcement learning tasks to mimic a execution of real world strategy making decision and propose five methods to generalize to a wider data distribution: 1. Utilizing Large Language Model as our encoder-decoder 2. Using visual words of action/perception as model’ s input 3. Training on data with better generating heuristic 4. Employing function/branch prediction as auxiliary objective loss functions 5. Utilizing ExeDec to break down long horizon execution traces

In our experiment, methods 1, 2, and 3 improved the performance of synthesizing

doi:10.6342/NTU202401358

program policies on PRL tasks. However, we did not observe significant improvements when directly using methods 4 and 5.

We analyzed that there are three potential cases of dataset distribution misalignment between the synthetic dataset and the real-world dataset: 1. Demonstrations generated by excessively long programs. 2. Demonstrations generated by overly deep programs (i.e., too many layers of branching/looping). 3. The sequence and combination of demonstration frames in the real-world dataset are not present in the synthetic dataset.

We believe that, if the method architecture is carefully designed, method 4 can slightly help with the second case, while method 5 can address the first and third cases.

doi:10.6342/NTU202401358

References

[1] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. arXiv preprint arXiv:2204.05862, 2022.

[2] Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In Proceedings of ICLR'17, 2017.

[3] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In International Conference on Learning Representations, 2018.

[4] Xiaocong Chen, Lina Yao, Julian McAuley, Guanglin Zhou, and Xianzhi Wang. Deep reinforcement learning in recommender systems: A survey and new perspectives. Knowledge-based systems, 2023.

[5] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In International Conference on Learning Representations, 2024.

[6] Raphaël Dang-Nhu. Plans: Neuro-symbolic program learning from videos. In Advances in Neural Information Processing Systems, 2020.

[7] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In International Conference on Machine Learning, 2017.

[8] Xuguang Duan, Qi Wu, Chuang Gan, Yiwei Zhang, Wenbing Huang, Anton van den Hengel,

doi:10.6342/NTU202401358

and Wenwu Zhu. Watch, reason and code: Learning to represent videos using program. In Proceedings of the 27th ACM International Conference on Multimedia, 2019.

[9] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In Proceedings of the 42nd acm sigplan international conference on programming language design and implementation, 2021.

[10] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. In Neural Information Processing Systems, 2020.

[11] Vidhi Jain, Maria Attarian, Nikhil J Joshi, Ayzaan Wahid, Danny Driess, Quan Vuong, Pannag R Sanketi, Pierre Sermanet, Stefan Welker, Christine Chan, et al. Vid2robot: End-to-end video-conditioned policy learning with cross-attention transformers. arXiv preprint arXiv:2403.12943, 2024.

[12] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In Neural Information Processing Systems, 2022.

[13] Wei-Hsu Lee. Enhancing program synthesis through fine-tuning techniques, contrastive learning, comprehensive training strategies, and real-world evaluation scenarios. Master's thesis, National Taiwan University, 2024.

[14] Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-Yi Lee, and Shao-Hua Sun. Hierarchical programmatic reinforcement learning via learning to compose programs. In International Conference on Machine Learning, 2023.

[15] Anthony Manchin, Jamie Sherrah, Qi Wu, and Anton van den Hengel. Program generation from diverse video demonstrations, 2023.

[16] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[17] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models

doi:10.6342/NTU202401358

to follow instructions with human feedback. In Neural Information Processing Systems, 2022.

[18] Richard E Pattis. Karel the robot: a gentle introduction to the art of programming. John Wiley & Sons, Inc., 1981.

[19] Kensen Shi, Joey Hong, Yinlin Deng, Pengcheng Yin, Manzil Zaheer, and Charles Sutton. Exedec: Execution decomposition for compositional generalization in neural program synthesis. In The Twelfth International Conference on Learning Representations, 2024.

[20] Richard Shin, Neel Kant, Kavi Gupta, Christopher Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic datasets for neural program synthesis. In International Conference on Learning Representations, 2019.

[21] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. Nature, 2016.

[22] Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In Proceedings of the 35th International Conference on Machine Learning, 2018.

[23] Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. Learning to synthesize programs as interpretable and generalizable policies. In Thirty-Fifth Conference on Neural Information Processing Systems, 2021.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in Neural Information Processing Systems, 2017.

[25] Letian Wang, Jie Liu, Hao Shao, Wenshuo Wang, Ruobing Chen, Yu Liu, and Steven L Waslander. Efficient reinforcement learning for autonomous driving with parameterized skills and priors. 2023.

[26] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. arXiv preprint, 2023.

doi:10.6342/NTU202401358

[27] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019.

[28] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In International Conference on Machine Learning, 2020.

[29] Shafiq Joty Steven C.H. Hoi Yue Wang, Weishi Wang. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In EMNLP, 2021.

[30] Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples, 2015.

doi:10.6342/NTU202401358

Appendix A — Domain Specific Language

A.1 String Transformation Domain

$$
\begin{array}{rcl}
\text{Program } p & := & \texttt{Concat}(\texttt{e}_1, \texttt{e}_2, \texttt{e}_3, ...) \\
\text{Expression } e & := & f \mid n \mid n_1(n_2) \mid n(f) \mid \texttt{ConstStr}(\texttt{c}) \\
\text{Substring } f & := & \texttt{SubStr}(\texttt{k}_1, \texttt{k}_2) \\
\text{Nesting } \texttt{n} & := & \texttt{GetToken}(\texttt{t}, \texttt{i}) \mid \texttt{ToCase}(\texttt{s}) \\
& \mid & \texttt{Replace}(\delta_1, \delta_2) \mid \texttt{Trim}() \\
& \mid & \texttt{GetUpto}(\texttt{r}) \mid \texttt{GetFrom}(\texttt{r}) \\
& \mid & \texttt{GetFirst}(\texttt{t}, \texttt{i}) \mid \texttt{GetAll}(\texttt{t}) \\
\text{Position } k & := & -30, -29, ..., 1, 2, ..., 30 \mid r \\
\text{Regex } r & := & t_1 \mid \cdots \mid t_n \mid \delta_1 \mid \cdots \mid \delta_m \\
\text{Type } t & := & \texttt{NUMBER} \mid \texttt{WORD} \mid \texttt{ALPHANUM} \\
& \mid & \texttt{ALLCAPS} \mid \texttt{PROPCASE} \mid \texttt{LOWER} \\
& \mid & \texttt{DIGIT} \mid \texttt{CHAR} \\
\text{Case } s & := & \texttt{PROPER} \mid \texttt{ALLCAPS} \mid \texttt{LOWER} \\
\text{Index } i & := & -5, -4, -3, -2, 1, 2, 3, 4, 5 \\
\text{Character } c & := & \texttt{A} - \texttt{Z}, \texttt{a} - \texttt{z}, \texttt{0} - \texttt{9} \mid \delta \\
\text{Delimiter } \delta & := & \&, .?!@()[]\%\{\}/ :; \$\#"' \\
\text{Boundary } y & := & \texttt{Start} \mid \texttt{End}
\end{array}
$$

Figure A.1: **Domain Specific Language (DSL) of string transformation domain:** Syntax of the string transformation DSL based on Robustfill [7].

doi:10.6342/NTU202401358

A.2 Karel Domain

Program ρ := DEF run m(s m)
Repetition n := 0..19
Perception h := frontIsClear | leftIsClear | rightIsClear |
markersPresent | noMarkersPresent
Condition b := perception h | not perception h
Action a := move | turnLeft | turnRight |
putMarker | pickMarker
Statement s := WHILE c(b c) w(s w) | $s_1 s_2$ | a |
REPEAT R=n r(s r) | IF c(b c) i(s i) |
IFELSE c(b c) i(s_1 i) ELSE e(s_2 e)

Figure A.2: **The Karel DSL grammar.** It describes the Karel domain-specific language's actions, perceptions, and control flows. The domain-specific language is obtained from Liu et al. [14].

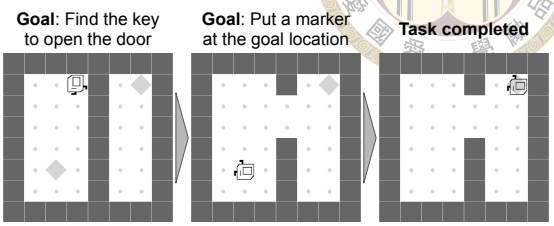
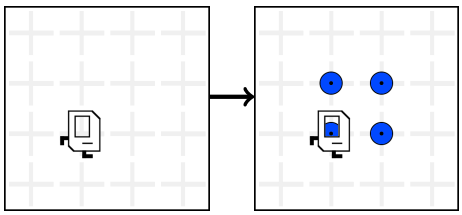


Figure A.3: **An example Karel task – DoorKey.** The agent first needs to find the key (marker) in the left room, which will open the door (wall) to the right room. Navigating to the goal marker in the right room and placing the picked marker on it will grant the full reward for the task. This sparse-reward task has been found to pose significant challenges to previous PRL methods, as it necessitates a greater capability in long-horizon strategy formulation.

doi:10.6342/NTU202401358

Appendix B — Program Aliasing



(a) Partial Specification as I/O Pair

Program A

```
def run():
  repeat(4):
    putMarker()
    move()
    turnLeft()
```

Program B

```
def run():
  while(noMarkersPresent):
    putMarker()
    move()
    turnLeft()
```

Figure B.4: **Program Aliasing** is one difficulty of program synthesis (pointed out by Bunel et al. [3]): For the input-output specification given in (B.4a), both programs are *semantically correct*. However, supervised training would penalize the prediction of Program B, if A is the ground truth.

doi:10.6342/NTU202401358