HIERARCHICAL NEURAL PROGRAM SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Program synthesis aims to automatically construct human-readable programs that satisfy given task specifications such as input/output pairs or demonstrations. Recent works have demonstrated encouraging results in a variety of domains such as string transformation, tensor manipulation, and describing behaviors of embodied agents. Most existing program synthesis methods are designed to synthesize programs from scratch, generating a program token by token, line by line. This fundamentally prevents these methods from scaling up to synthesize programs that are longer or more complex. In this work, we present a scalable program synthesis framework that instead synthesizes a program by hierarchically composing programs. Specifically, we first learn a task embedding space and a program decoder that can decode a task embedding into a program. Then, we train a high-level module to comprehend the task specification (*e.g.* input/output pairs or demonstrations) from long programs and produce a sequence of task embeddings, which are then decoded by the program decoder and composed to yield the synthesized program. We extensively evaluate our proposed framework in a string transformation domain with input/output pairs. The experimental results demonstrate that the proposed framework can synthesize programs that are significantly longer and more complex than the programs considered in prior program synthesis works.

1 INTRODUCTION

Program synthesis aims to automatically synthesize a program structured in a domain-specific language (DSL) that satisfies given task specifications. Recently, encouraging results have been achieved across a variety of domains such as string and tensor manipulation, computer commands, graphics programs, and describing embodied agent behaviors (Devlin et al., 2017b; Balog et al., 2017; Bunel et al., 2018; Wu et al., 2017; Lin et al., 2018; Sun et al., 2018; Trivedi et al., 2021).

Most existing program synthesis methods are designed to sequentially synthesize programs—generating program tokens one by one based on previously produced tokens until the synthesis process has finished. However, without any inductive bias on the program space, synthesizing desired programs token by token can become difficult when we scale from short programs with simple behaviors to longer programs with more complex behaviors. Furthermore, these methods often learn to maximize the likelihood of each token with losses that weigh their importance in the program equally regardless of how they can affect the program's behavior. Such a training scheme suffers from the program aliasing problem and having weak supervision for critical tokens. For example, while a program output may not change significantly when a non-critical token is replaced with a different one, such as "nullptr" and "NULL" which are mostly interchangeable in C++, the output is completely changed when replacing input values of key functions (*i.e.* the position index token of an array is changed from a[0] to a[1], resulting in a different value). Thus, we argue that this token-by-token paradigm for generation and training fundamentally prevents current program synthesis methods from scaling up to synthesizing more complex programs.

In this work, we aim to develop a program synthesis framework that can scale up to longer programs which can induce more complex behaviors. Our key idea is to compose shorter programs with simple behaviors to form longer programs with more complex behaviors. However, the space of meaningful short programs is intractable in most cases – searching for correct sequences of short programs to compose into long programs can still be challenging. Therefore, we not only propose an architecture that can learn to hierarchically compose programs, denoted the Hierarchical Neural

Program Synthesizer (HNPS), but also devise a training schema that allows for efficiently learning this hierarchical synthesizer.

To this end, we first construct a *short program dataset* that contains short programs with their corresponding input/output pairs. Then, we learn a task embedding space that represents the space of short program behaviors, along with a program decoder that can decode a task embedding into a short program. Next, we create a *composed program dataset* in which the composed programs are generated by sequentially composing short programs. We then train a "program composer" on this dataset to produce a sequence of task embeddings, each representing a latent short program, which are decoded by the program decoder into short programs and sequentially composed to yield the synthesized program. This training schema allows for generating programs via program-by-program composition rather than token-by-token composition, thus enabling additional supervision through the task embeddings instead of only maximizing the likelihood based on produced program tokens.

To evaluate the proposed framework, we consider program synthesis in a string transformation domain, where a task specification consists of input/output strings (*e.g.* [`John-01/John`,`Mary-02/Mary`]) and a program consists of a sequence of string manipulation operations (*e.g.* `SubStr(0, Regex("-", 0, Start))`), similar to Devlin et al. (2017b); Hong et al. (2021). Experiments show that our framework is superior at synthesizing longer programs which induce more complex string manipulations. Furthermore, we perform extensive ablation studies which justify the hierarchical architecture and the latent supervision made possible by the proposed training schema and composed program dataset.

2 Related Work

Neural program induction. Program induction methods (Neelakantan et al., 2015; Graves et al., 2014; Kaiser & Sutskever, 2016; Gaunt et al., 2017; Reed & De Freitas, 2016; Xiao et al., 2018; Burke et al., 2019; Lázaro-Gredilla et al., 2019; Pierrot et al., 2019) are designed to implicitly induce underlying programs to mimic the behaviors demonstrated in given task specifications. To this end, most methods employ external memory (Graves et al., 2014; Zaremba et al., 2016), modular frameworks with modularized supervision (Reed & De Freitas, 2016; Cai et al., 2017; Xu et al., 2018), or sophisticated attention mechanisms (Devlin et al., 2017a) to acquire programmatic behaviors. In contrast, we are interested in explicitly synthesizing human-readable programs.

Neural program synthesis. Program synthesis methods (Bošnjak et al., 2017; Parisotto et al., 2017; Devlin et al., 2017b; Bunel et al., 2018; Shin et al., 2018; Chen et al., 2019; Liu et al., 2019; Sun et al., 2018; Lin et al., 2018; Liao et al., 2019; Ellis et al., 2019; 2020; Balog et al., 2017; Liskowski et al., 2020; Abolafia et al., 2018; Hong et al., 2021; Silver et al., 2020; Trivedi et al., 2021; Verma et al., 2018) explicitly generate programs that can be executed to perform the tasks from given specifications. These task specifications can range from strings of program input/output pairs to demonstrations of agent behaviors. Compared to these works, we are particularly interested in developing a framework that can synthesize longer programs with complex behaviors.

Recently, pre-trained large language models (LLMs) have been employed for program synthesis from task specifications such as descriptions of desired program behavior or direct input/output examples (Nijkamp et al., 2022; Li et al., 2022; Chen et al., 2021). These large models are trained on massive corpora of code which allow them to demonstrate impressive performance in synthesizing programs. However, they still generate tokens sequentially and can encounter the same issues as domain-specific token-by-token synthesis methods. In addition, transformer-based models (Vaswani et al., 2017) have been adapted with novel program synthesis-tailored attention mechanisms (Shi et al., 2022). Our method is distinct in that it incorporates supervision against the latent embedding vectors in addition to the individual program tokens. The training framework we propose is complementary to the advances made by LLMs and can be combined with such architectures and massive datasets in the future.

3 Problem Formulation

Our goal is to synthesize programs from task specifications. In this section, we formally describe our definition of programs and task specifications as well as the specific problem formulation.

Program and Domain Specific Language. Programs considered in this work are constructed based on a Domain Specific Language (DSL), which establishes the possible space of valid programs as well as enables sampling programs. A DSL example is shown in Figure 1. The DSL defines a set of operations (*e.g.* `SubStr(`k_1`, `k_2`)`, `Concat(`e_1`, `e_2`, ...)`, `ConstStr(`s`)`) and their parameters, such as numbers (*e.g.* k_1, k_2) and strings (*e.g.* s). A program consists of a sequence of such operations and determines how input strings are transformed into their corresponding output strings. For example, a program `SubStr(0, Regex("-", 0, Start))` takes an input string `John-01` and produces an output string `John`.

Task Specification. A task specification describes a desired user intent consisting of inputs and desired program execution results. For example, [Input : `October`, Output : `oct`] can serve as an input/output pair task specification for a string transformation program. Note that several different programs could satisfy this example specification, so in practice, multiple input/output pairs are usually specified to the program synthesis network. Importantly, the mappings between task specifications and programs are many-to-many—multiple task specifications can be used to describe the same desired program and many programs can satisfy the same task specification, signaling the difficulty of program synthesis. In general, providing a task specification should be easier than asking non-expert users to write the program. Commonly used representations for task specifications include input/output (I/O) pairs, demonstrations, or natural language instructions.

Program ρ := Concat c(e_1 e_2 e_3 ... c)

Expression e n := ConstStr k(s k) | SubStr s(q_1 q_2 s)

Position q := ConstPos p(k p) | Regex r(x n b r)

Boundary := Start | End

Regex x := s | Word | Num | Alphanum | ... | Char

Position index k := -20 | -19 | ... | 19 | 20

Regex index n := -3 | -2 | ... | 2 | 3

Character s := "SPACE" | "." | "," | ... | "?"

Figure 1: The domain-specific language (DSL) for constructing string manipulation programs.

Problem Formulation. We develop a framework that can synthesize a program from a task specification as described above. In particular, we focus on long programs that can induce more complex execution results. Specifically, we consider a string transformation domain, where a task specification consists of a set of input/output string pairs (*e.g.* [`John-01/John`, `Mary-02/Mary`, `Bob-03/Bob`]) and a program comprises a sequence of string manipulation operations (*e.g.* `SubStr(0, Regex("-", 0, Start))`).

4 APPROACH

Our goal is to design a framework that can synthesize a long program given a task specification. To this end, we propose the Hierarchical Neural Program Synthesizer (HNPS), a framework that learns to compose programs to form longer programs, as illustrated in Figure 2. In the following, we describe how we learn a program embedding space in Section 4.1 from short programs and simpler task specifications. Then, we describe how we produce a dataset for learning to compose programs in Section 4.2. Finally, we present how HNPS learns to synthesize a program given a task specification by composing programs, described in Section 4.3.

4.1 LEARNING A TASK EMBEDDING SPACE

To learn a task embedding space, we build a short program dataset D_{short} by randomly sampling programs from the DSL, and we generate task specifications (*e.g.* input/output pairs) by running sampled programs with randomly generated inputs. We then train, on programs ρ and corresponding task specifications σ sampled from D_{short}, a neural program synthesis model that consists of a task encoder e_β which encodes a task specification σ to a task embedding, and a program decoder p_θ which synthesizes the program from the embedding. Both the task encoder e_β and the program decoder p_θ are recurrent networks trained to optimize the token-by-token cross-entropy loss to maximize the likelihood of the ground truth program tokens. We denote this the task embedding loss, $\mathcal{L}^{\text{TE}}_{\theta,\beta}$:

$$\mathcal{L}^{\text{TE}}_{\theta,\beta}(\boldsymbol{\rho}) = -\mathbb{E}_{(\boldsymbol{\sigma},\boldsymbol{\rho})\sim D_{\text{short}}} \left[\log p_\theta \left(\boldsymbol{\rho} | e_\beta \left(\boldsymbol{\sigma} \right) \right) \right]. \tag{1}$$

Ideally, this learned embedding space can serve as a good representation space for desired program behavior, grouping similar programs and their task specifications closer together, and keeping dissimilar programs and their task specifications farther apart.

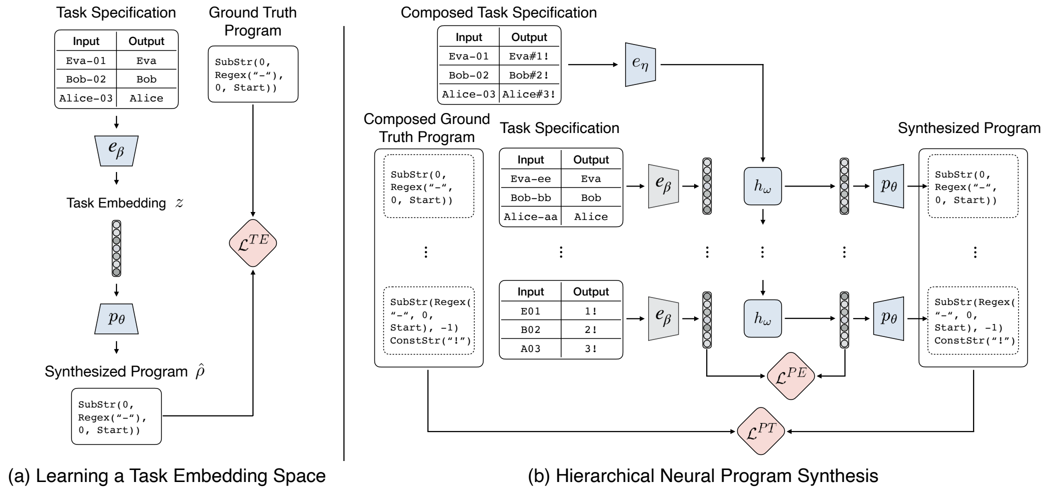

Figure 2: **Framework Overview.** **(a)** Learning a Task Embedding Space: We first learn a task embedding space by training a task encoder, e_β, to encode a task specification (e.g. I/O pairs) σ of a short program to a task embedding which is decoded by p_θ to a program $\hat{\rho}$. e_β and p_θ are trained with task embedding loss $\mathcal{L}^{\text{TE}}_{\theta,\beta}$. **(b)** Hierarchical Neural Program Synthesis: We then train a task interpreter e_η initialized with the weights from task encoder e_β to interpret the given I/O pairs. The program composer h_ω uses this encoded input to continually predict task embeddings to be decoded by the pretrained p_θ decoder. These decoded programs are finally composed to form the complete synthesized program. The model is trained to synthesize long programs with a program token loss $\mathcal{L}^{\text{PT}}$ and match the ground truth program embeddings with the program embedding loss $\mathcal{L}^{\text{PE}}$.

4.2 Creating The Composed Long Program Dataset

Our ultimate goal is to learn to compose short programs to form a longer program. Therefore, we create a dataset D_{composed} which consists of long programs obtained by composing shorter programs from the program dataset D_{short}. We denote these composed long programs $\bar{\rho} = [\rho_1, ..., \rho_N]$, where each ρ_i is a short program. Such a composed program dataset will give us access to ground truth information of how each long program is composed and allow the model to learn program composition. For all long programs in D_{composed}, we also include the task embeddings of their sub-programs, calculated with the task encoder e_β using the task specifications of short programs from D_{short}. The information from task embeddings will enable us to train our model to fully utilize our learned task embedding space.

4.3 Hierarchical Neural Program Synthesis

To synthesize a long program, program-by-program instead of token-by-token, we propose to leverage the program decoder p_θ learned in Section 4.1 as a low-level module to produce short programs. Then, we employ a *program composer* h_ω that produces a sequence of task embeddings which can be decoded into short programs and sequentially composed to form a long program.

Specifically, our Hierarchical Neural Program Synthesizer consists of the following three components. A task interpreter e_η learns to interpret given task specifications (*i.e.* string I/O pairs). We task specifications from D_{short} and D_{composed} are similar, parameters from task encoder e_β can be used as initialization for the task interpreter e_η. Then, the program composer h_ω takes the output of the task interpreter as input and sequentially produces task embeddings until a long program that satisfies the task specification is generated, or until the maximum program length is reached. Finally, the program decoder p_θ trained during the construction of the task embedding space will generate the corresponding short program. Since predicting the exact task embedding can be difficult, the program decoder p_θ is fine-tuned along with other components and the task embedding space is allowed to drift. To train the Hierarchical Neural Program Synthesizer, we propose to optimize the following objectives.

Program Token Loss. The program token loss $\mathcal{L}^{\text{PT}}$ simply aims to maximize the log likelihood of the ground truth program tokens in the synthesized program via the cross-entropy loss. After sampling a composed program $\bar{\rho}$ and its task specifications σ, we apply the cross-entropy loss to maximize the log likelihood of each individual short program $\rho_1 ... \rho_N$ that makes up $\bar{\rho}$. This loss is propagated throughout all three networks. Note that h_ω is a recurrent network so its predictions depend on earlier short programs; therefore, losses from later shorter programs will be backpropagated through time to update both e_η and h_ω to improve overall reconstruction of the entire long program:

$$\mathcal{L}^{\text{PT}}_{\theta,\omega,\eta} = -\mathbb{E}_{(\boldsymbol{\sigma},\bar{\boldsymbol{\rho}})\sim D_{\text{composed}}} \left[\frac{1}{N} \sum_{i=1}^{N} \log p_\theta \left(\boldsymbol{\rho_i} | h_\omega \left(e_\eta \left(\boldsymbol{\sigma} \right) \right) \right) \right]. \tag{2}$$

Program Embedding Loss. Even though the task embedding space may change as p_θ is fine-tuned, embeddings from the initial task embedding space still provide useful grounding for the program composer to output sensible task embeddings. To provide regularization on the output space of the program composer, we embed the task specifications ρ_i of each short program that composes longer programs in D_{composed} with the original fixed task encoder e_β. Then, we apply a program embedding loss $\mathcal{L}^{\text{PE}}$ that aims to minimize the Euclidean distance between this ground truth (*i.e.* encoded) program embedding and the program embedding produced by the program composer, for all N subprograms in the composed program:

$$\mathcal{L}^{\text{PE}}_{\eta,\omega} = \mathbb{E}_{(\boldsymbol{\sigma},\boldsymbol{\sigma_1},\dots,\boldsymbol{\sigma_N})\sim D_{\text{composed}}} \left[\frac{1}{N} \sum_{i=1}^{N} || h_\omega \left(e_\eta \left(\boldsymbol{\sigma} \right) \right) - e_\beta \left(\boldsymbol{\sigma_i} \right) ||_2 \right]. \tag{3}$$

In our experiments in Section 5.3, we find that this loss is especially useful in scenarios with less training data to ground the program composer to the space of sensible program embeddings.

In summary, we propose the following objective for learning to synthesize a program by processing given task specifications and composing programs: $\min_{\eta,\omega,\theta} \lambda_1 \mathcal{L}^{\text{PT}} + \lambda_2 \mathcal{L}^{\text{PE}}$ where λ_1 and λ_2 are hyperparameters controlling the importance of each loss. Note that the program embedding loss comes from our contribution of composing a dataset with shorter programs to construct a dataset with decomposable longer programs.

5 Experiments

We evaluate our framework on datasets generated on the aforementioned string transformation task, detailed in Section 5.1. We then justify the design decisions of HNPS by comparing against baselines and ablations detailed in Section 5.2. Finally, we present the results of our experiments in Section 5.3.

5.1 Datasets

We create two datasets, D_{short} and D_{composed}, where D_{composed} consists of long programs which are obtained by composing shorter programs from D_{short}. Once programs are generated, string I/O pairs that correspond to these programs are also created. D_{short} has 100,000 programs, which are created by randomly sampling one expression-long programs from our DSL. For each sampled program, we randomly generate 1,000 strings as potential input strings, and the first 20 input strings that can be executed without exceptions are stored in our dataset along with their corresponding output strings. If less than 20 input strings in the 1,000 generated are executed without exceptions for a program, then that program is discarded. For D_{composed}, we also generate 200,000 programs by composing 2-4 randomly sampled programs from D_{short}. For each composed program in D_{composed}, the same method is used to generate I/O pairs. We split each dataset into training and validation sets to be able to evaluate model generalization performance during training.

Finally, we generate a long program dataset, D_{long}, which consists of 30,000 programs, each containing 2-4 expressions by directly sampling from our DSL. Notably, D_{long} is not constructed by composing short programs. We evaluate the performance of HNPS on this testing dataset to evaluate its ability to synthesize unseen, long programs.

5.2 Baseline and Ablations

We evaluate HNPS against baselines and its variants. The following baselines represent the family of neural synthesis methods that are designed to synthesize a program in a token-by-token manner.

- **Naïve**: A naïve synthesis baseline trained on D_{composed} which encodes the I/O pair tokens to produce an embedding which is used to synthesize the full-length program. This baseline is expected to struggle at learning to synthesize long programs with complex behaviors from scratch.
- **Naïve-short**: Naïve but the model is only trained on D_{short}. This baseline is expected to learn well from D_{short} but should zero-shot generalize poorly to D_{long}.
- **Naïve-short-finetune**: Naïve-short but the model is finetuned on D_{composed}. This baseline utilizes both D_{short} and D_{composed} in the same order as our proposed method. The difference is that it has a flat architecture without hierarchy like ours. The performance gap between this baseline and ours should justify the hierarchical design of our proposed framework.

To justify our design choices, the following variants of HNPS are considered for ablation studies.

- **H-Naïve-PT**: A naïve hierarchical synthesis baseline in which the decoder is trained from scratch (*i.e.* we skip the learning embedding space stage), and only the program token loss $\mathcal{L}^{\text{PT}}$ is applied. This ablation learns to compose short programs to create longer ones, except that it does not learn in a two-stage fashion. The performance gap between this ablation and our proposed framework should justify the importance of learning a task embedding space.
- **HNPS-PT**: An ablation in which only the program token loss $\mathcal{L}^{\text{PT}}$, which still provides strong supervision for relatively shorter programs, is applied. This ablation first learns a task embedding space from D_{short} and then learns from D_{composed} while optimizing $\mathcal{L}^{\text{PT}}$. Here, we seek to analyze in which contexts the absence of the program embedding loss may be more or less detrimental to execution accuracy performance, given that ground truth task embeddings can be noisy.
- **HNPS (HNPS-PT+PE)**: The full HNPS method. It first learns a task embedding space from D_{short}. Then, it optimizes both losses (the program token loss $\mathcal{L}^{\text{PT}}$ and the program embedding loss $\mathcal{L}^{\text{PE}}$) while learning from D_{composed}.

5.3 Results

We present results of HNPS and comparison methods on synthesizing programs in both an unseen, non-composed, long program dataset and in unseen programs in test set of D_{composed} in Section 5.3.1. Then, we analyze the task embedding space learned by HNPS and how it aids long program synthesis in Section 5.3.2. Finally, we carefully examine the effect of the program embedding loss on synthesizing programs of different lengths and with different dataset sizes in Section 5.3.3.

5.3.1 Execution Accuracy on Long Programs

Table 1: **Performance on Unseen Test Programs.** We evaluate execution accuracy over *unseen* test programs in our dataset D_{long}, where execution accuracy refers to the percent of outputs of synthesized programs that matches the expected outputs on given I/O pairs. The x-axis represents the length of the ground truth program (number of program tokens).

	Number of program tokens			
Method	**10-25**	**25-40**	**40-55**	**55-70**
Naïve	52.19%	18.39%	1.32%	0.02%
Naïve-short	0.01%	0.01%	0.00%	0.00%
Naïve-short-finetune	51.43%	25.40%	3.82%	0.09%
H-Naïve-PT	72.92%	40.17%	10.71%	0.91%
HNPS-PT	**77.65%**	**46.88%**	16.61%	1.73%
HNPS (ours-full)	76.46%	46.73%	**16.68%**	**2.11%**

Evaluation on Non-Composed Long Programs. We evaluate execution accuracy over *unseen* test programs in our dataset D_{long}, where execution accuracy refers to the percent of outputs of synthesized programs that matches the expected outputs on given input-output pairs.

Results from our experiments are shown in Table 1. We additionally provide a visualization of relative performance on different program lengths in Figure 3, where we normalize the execution accuracy of each method by scaling the accuracy of HNPS to 1.0. For programs longer than 40 tokens, our method is able to achieve better performance than the baselines and ablations with 16.68% execution accuracy on programs 40-55 tokens long and 2.11% on programs 55-70 tokens long. For programs shorter than 40 tokens, HNPS-PT, where program embedding loss is not applied, yields the best performance with 77.65% execution accuracy on programs with 10-25 tokens and 46.88% on program with 25-40 tokens. We further analyze performance differences between HNPS-PT and HNPS in shorter programs in later sections.

With only the hierarchical architecture and without the pretrained task embedding space, H-Naïve-PT is still able to outperform other non-hierarchical methods. However, its relative performance compared to HNPS drops significantly as program length increases, reaching less than 50% of HNPS's execution accuracy on 55-70 token programs. This demonstrates that the task embedding space is crucial in training hierarchical models.

The non-hierarchical baselines Naïve, Naïve-short, and Naïve-short-finetune perform significantly worse than our method. All three baselines have close to zero execution accuracy on 55-70 token programs, even for Naïve-short-finetune which finetunes the same task encoder and program decoder as HNPS on the same datasets. This lends credence to our claim that the token-by-token generation scheme is fundamentally limiting to a program synthesis model's ability to scale to longer programs.

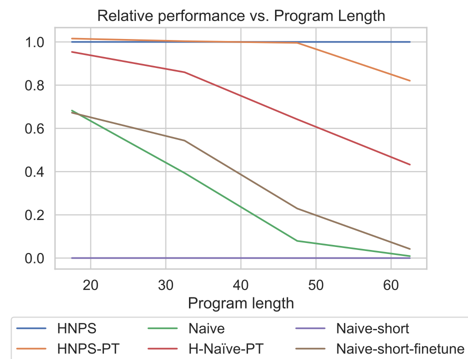

Figure 3: Relative performance of all baselines and HNPS evaluated on D_{long}. We normalize the execution accuracy of each method by scaling the accuracy of HNPS to 1.0.

Evaluation on Composed Long Programs. We also evaluate execution accuracy of all methods over training programs and *unseen* test programs in our D_{composed} dataset. Exact results are detailed in appendix Section D.2. Similar to the results from evaluation on D_{long}, HNPS & HNPS-PT are able to outperform H-Naïve-PT, which further highlights the importance of constructing a task embedding space. We also observe that non-hierarchical baselines (Naïve-*) have significantly lower performance than hierarchical methods on both training and test programs. This supports our assertion that without the hierarchical architecture, the model's ability to fit to a distribution of long programs is fundamentally limited.

We also analyze the "generalization gap," *i.e.* the difference between training and test set performance, on D_{composed}. HNPS has a smaller generalization gap than both HNPS-PT and H-Naïve-PT. This implies that the program embedding loss can help prevent overfitting in hierarchical models. This may be due to the fact that the program decoder learns to synthesize a diverse set of short programs during D_{short} training. When training on D_{composed}, the embedding loss grounds the decoder to the original task embedding space, encouraging it to not overfit to only the short programs needed for the D_{composed} training set. Therefore, the program embedding loss not only helps the composer to create longer programs (which we will explore further in Section 5.3.2) but also helps ensuring the model does not overfit to the subset of short programs that are used to compose D_{composed}.

Qualitative Examples. In Figure 4, we show an example of programs generated by HNPS, HNPS-PT, and Naïve-short-finetune compared with the ground truth program. To assess our method's ability to create a meaningful embedding space for program composition, we first introduce two concepts: critical/non-critical tokens and critical/non-critical errors. We define critical tokens as program tokens that if changed, will alter the behavior of the program and thus produce a different program output for a given string input. Following this, critical errors are characterized by errors of the program synthesis module on critical tokens. Conversely, non-critical tokens are those that can be replaced without affecting program behavior, and non-critical errors are mistakes on non-critical tokens. In the figure, we mark all non-critical errors orange and all critical errors red.

In the program generated by HNPS, there are 3 tokens that are different from the ground truth program, all of them confusing the "End" token with the "Start" token. However, even with these incorrect tokens, the HNPS-generated program still achieves correct behavior. This is because the DSL syntax dictates that "Regex" expressions that only capture one string token will refer to the same index position regardless of whether the "Start" or "End" token is specified. So, even with three token errors, the output of HNPS is able to satisfy the I/O specification.

Figure 4: **Qualitative Examples.** We show an example of a program generated by HNPS, HNPS-PT, and Naïve-short-finetune compared with the ground truth program. Non-critical errors, where the changes in tokens do not effect program behavior, are marked orange. Critical errors, where the changes in tokens result in different program behavior, are marked red.

Ground Truth

```
Concat c( SubStr s( Regex r( "@" 1 End r) Regex r( Allcaps -2 End r) s)
SubStr s( Regex r( "%" 1 Start r) Regex r( Digit -2 End r) s)
SubStr s( Regex r( "#" 2 End r) Regex r( Word -3 End r) s)
SubStr s( ConstPos p( 4 p) Regex r( "," -3 End r) s) c)
```

HNPS-PT

```
Concat c( SubStr s( Regex r( "@" 1 End r) Regex r( Allcaps -2 End r) s)
SubStr s( Regex r( "%" 1 Start r) Regex r( Digit -2 Start r) s)
SubStr s( Regex r( "#" -3 End r) Regex r( Word -3 End r) s)
SubStr s( ConstPos p( 4 p) Regex r( "," 2 Start r) s) c)
```

Naive-short-finetune

```
Concat c( SubStr s( Regex r( "@" 1 Start r) Regex r( Word -3 End r) s)
SubStr s( Regex r( "%" 1 Start r) Regex r( Allcaps -2 End r) s)
SubStr s( Regex r( "#" 1 Start r) Regex r( Digit -2 Start r) s)
SubStr s( Regex r( Char 0 End r) Regex r( "," -3 Start r) s) c)
```

HNPS

```
Concat c( SubStr s( Regex r( "@" 1 End r) Regex r( Allcaps -2 End r) s)
SubStr s( Regex r( "%" 1 Start r) Regex r( Digit -2 Start r) s)
SubStr s( Regex r( "#" 2 Start r) Regex r( Word -3 End r) s)
SubStr s( ConstPos p( 4 p) Regex r( "," -3 Start r) s) c)
```

HNPS-PT's generated program has 4 token errors, which is just one more error than HNPS. Yet, 2 of the 4 errors made by HNPS-PT are in critical parameters of the "Regex" function that completely change the function output. This example correspond to a trend we observed in generated programs where HNPS tends to make same amount or more token errors compared to HNPS-PT, but HNPS makes less error in critical tokens, resulting in better execution accuracy. We suspect that this is because supervision for HNPS-PT on critical tokens is weak without the program embedding loss, resulting in more critical errors.

For Naïve-short-finetune, the generated program has several errors in regex pattern parameters and even a wrong function. This demonstrates that even though Naïve-short-finetune is trained on the same datasets, its performance is still limited by the token-by-token prediction scheme, making it unable to recognize correct program behavior and synthesize accurate programs.

5.3.2 Analyzing the Task Embedding Space

We now analyze how learning the task embedding space aids with hierarchical program synthesis. In Figure 5, we provide a visualization of the tasks embeddings of 5 programs randomly drawn from the test set of D_{short}. Dimensionality reduction is performed with principal component analysis (PCA) in order to project the task embeddings to a 2D space. We represent one embedding of an I/O pair with a dot in the figure and represent different programs using various colors. As shown in the figure, I/O pairs from the same program are clustered together and I/O pairs from different programs are separable, demonstrating that a meaningful embedding space is able to be learned and leveraged in our training procedure.

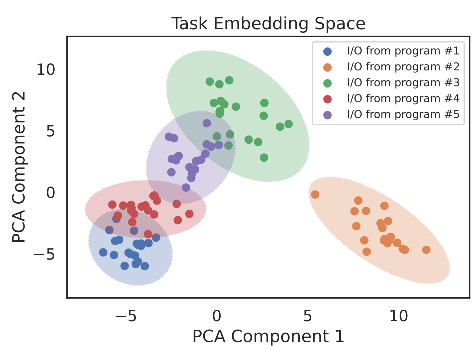

Figure 5: We perform dimensionality reduction with PCA on the task embeddings of 5 programs randomly drawn from the test set of D_{short}. Each dot represents the embedding of an I/O pair and each color represent a different program.

We are also interested in whether our embedding space can identify "critical tokens," tokens that would greatly change program behavior if swapped out for a different one. If nearly-identical programs with minor critical changes are embedded farther apart, then our program composer is more resilient to the critical token problem that can affect standard neural program synthesis methods. To do so, we sampled 100 programs from D_{short}. For each sampled program ρ, we either make a change to one non-critical token to create $\rho_{\text{non-critical}}$, or change one critical token to create ρ_{critical}. To get the task embedding of $\rho_{\text{non-critical}}$ and ρ_{critical}, we generate 20 I/O pairs for each program and pass them through our task encoder e_β. In Table 2, we show that the average euclidean distance between embeddings of programs with a difference in one critical token is significantly smaller than distance between programs with difference in one non-critical token. We thus confirm that our embedding space captures some degree of program semantics, as changes in critical tokens alter the behavior of the program more than changes in non-critical tokens, and our embedding space accordingly places the embedding of ρ_{critical} further from ρ than it does for the embedding of $\rho_{\text{non-critical}}$. We hypothesize that this smooth embedding space enables the program composer to more easily synthesize programs that match given task specifications.

5.3.3 Ablating the Program Embedding Loss

Now we analyze the effect of the program embedding loss specifically. In Table 3 we compare HNPS, our full method, and HNPS-PT, our method without the program embedding loss $\mathcal{L}^{\text{PE}}$, on D_{long} as a function of the size of the composed program dataset. We examine these two methods on: (1) our full D_{composed} containing 200,000 programs, (2) 50% of D_{composed}, and (3) 25% of D_{composed}. When trained with the full D_{composed}, HNPS only performs better than HNPS-PT on programs longer than 40 tokens. However, when using 50% or 25% of the data, HNPS's performance become significantly better than HNPS-PT across all program lengths. Especially for for programs longer than 55 tokens on 100k dataset, HNPS's execution accuracy is 0.16% which is 8 times the performance of HNPS-PT. This suggests that the supervision from program embedding loss is particularly effective when data is scarce. We hypothesize that this is because in practice, ground truth task embeddings tend to be noisy due to the imperfect task encoder. When a good support for the program space is provided by training data, the model can learn to compose short programs from the program token loss alone, making a noisy program embedding loss less useful as supervision. However, when training data is scarce, learning from the token loss alone is difficult and it will need the information provided by program embedding loss to output sensible task embeddings despite the noisiness of the embedding loss.

Table 2: **Embedding Distance with Changes in Tokens.** We sampled 100 programs each from the training set and test set of D_{short} to compare the average euclidean distance between task embeddings of programs with a difference in one critical token and programs with a difference in one non-critical token.

difference type	train	test
critical token	2.92 ± 0.58	2.99 ± 0.63
non-critical token	6.71 ± 1.98	10.50 ± 1.79

Table 3: **Effect of Dataset Size on Program Embedding Loss.** We examine HNPS and HNPS-PT on: (1) our full D_{composed} containing 200,000 programs, (2) 50% of D_{composed} (100k), and (3) 25% of D_{composed} (50k). We report their execution accuracy on D_{long}.

		Number of program tokens			
Dataset Size	**Method**	**10-25**	**25-40**	**40-55**	**55-70**
200k	HNPS	76.46%	46.72%	**16.68%**	**2.11%**
	HNPS-PT	**77.65%**	**46.88%**	16.61%	1.73%
100k	HNPS	**68.03%**	**30.84%**	**4.44%**	**0.16%**
	HNPS-PT	63.47%	23.12%	1.54%	0.02%
50k	HNPS	**55.94%**	**12.83%**	**0.30%**	0.00%
	HNPS-PT	50.76%	8.19%	0.09%	0.00%

6 Conclusion

In this work, we study learning to synthesize programs from task specifications such as input/output pairs (I/O pairs). In particular, we are interested in scaling up current neural program synthesis methods to synthesize long programs with more complex behaviors. To this end, we propose the Hierarchical Neural Program Synthesis (HNPS) framework; instead of producing a program token-by-token like most existing methods, we propose to synthesize a program subprogram-by-subprogram. Specifically, we first learn a task embedding space from short programs and their I/O pairs that continuously parameterizes diverse program behaviors. Then, we create a composed program dataset that provides intermediate supervision for learning a program composer, which efficiently learns to hierarchically compose short programs to form long and complex task-solving programs. Experimental results on a string transformation domain demonstrate the effectiveness of our proposed framework. Ablation studies provide detailed analysis on learned task embedding spaces as well as justify the proposed training schema that leverages a composed program dataset.

REFERENCES

Daniel A Abolafia, Mohammad Norouzi, Jonathan Shen, Rui Zhao, and Quoc V Le. Neural program synthesis with priority queue training. *arXiv preprint arXiv:1801.03526*, 2018.

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

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter. In *International Conference on Machine Learning*, 2017.

Rudy R 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.

Michael Burke, Svetlin Penkov, and Subramanian Ramamoorthy. From explanation to synthesis: Compositional program induction for learning from demonstration. *arXiv preprint arXiv:1902.10657*, 2019.

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In *International Conference on Learning Representations*, 2017.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *Advances in Neural Information Processing Systems*, 2017a.

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*, 2017b.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *Neural Information Processing Systems*, 2019.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *Proceedings of International Conference on Machine Learning (ICML)*, 2017.

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

Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. Latent programmer: Discrete latent codes for program synthesis. In *International Conference on Machine Learning*, 2021.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. In *International Conference on Learning Representations*, 2016.

Miguel Lázaro-Gredilla, Dianhuan Lin, J Swaroop Guntupalli, and Dileep George. Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *Science Robotics*, 2019.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

Yuan-Hong Liao, Xavier Puig, Marko Boben, Antonio Torralba, and Sanja Fidler. Synthesizing environment-aware activities via activity sketches. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *International Conference on Language Resources and Evaluation*, 2018.

Paweł Liskowski, Krzysztof Krawiec, Nihat Engin Toklu, and Jerry Swan. Program synthesis as latent continuous optimization: Evolutionary search in neural embeddings. In *Genetic and Evolutionary Computation Conference*, 2020.

Yunchao Liu, Jiajun Wu, Zheng Wu, Daniel Ritchie, William T. Freeman, and Joshua B. Tenenbaum. Learning to describe scenes with programs. In *International Conference on Learning Representations*, 2019.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. In *International Conference on Learning Representations*, 2015.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis. *arXiv preprint*, 2022.

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

Thomas Pierrot, Guillaume Ligner, Scott E Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. Learning compositional neural programs with recursive tree search and planning. In *Neural Information Processing Systems*, 2019.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations*, 2016.

Kensen Shi, Joey Hong, Manzil Zaheer, Pengcheng Yin, and Charles Sutton. Compositional generalization and decomposition in neural program synthesis. In *Deep Learning for Code (DL4C) Workshop at ICLR*, 2022.

Eui Chul Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. In *Neural Information Processing Systems*, 2018.

Tom Silver, Kelsey R Allen, Alex K Lew, Leslie Pack Kaelbling, and Josh Tenenbaum. Few-shot bayesian imitation learning with logical program policies. In *Association for the Advancement of Artificial Intelligence*, 2020.

Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, 2018.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. Learning to synthesize programs as interpretable and generalizable policies. In *Advances in Neural Information Processing Systems*, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of Neural Information Processing Systems (NeurIPS)*, 2017.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, 2018.

Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural scene de-rendering. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Da Xiao, Jo-Yu Liao, and Xingyuan Yuan. Improving the universality and learnability of neural programmer-interpreters with combinator abstraction. In *International Conference on Learning Representations*, 2018.

Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. Neural task programming: Learning to generalize across hierarchical tasks. In *International Conference on Robotics and Automation*, 2018.

Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples. In *International Conference on Machine Learning*, 2016.

APPENDIX

Table of Contents

A ARCHITECTURE

A.1 TASK ENCODER

The task encoder e_β is a recurrent neural network which encodes an I/O task specification σ to a task embedding. The encoded representation is then used by the program composer h_ω to synthesize programs. Specifically, the encoder is a gated recurrent unit (GRU) module with an input feature size of 256 and a hidden state size of 512. An embedding vector is calculated for the input string and separately for the output string. These embeddings are passed through a joint encoding module, which consists of a block of linear layers, dropout of 0.2, and rectified linear unit (ReLU) activation, followed by batch normalization.

A.2 PROGRAM DECODER

First trained during the embedding space learning stage, the program decoder p_θ is a recurrent neural network which decodes a task embedding produced by the task encoder e_β into a program $\hat{\rho}$. The model is trained such that the reconstructed program is as close as possible to the ground truth program whose task specification was the input to the task encoder. In HNPS, the program decoder takes as input the program embeddings from the program composer module and outputs program tokens. Under the hood, the decoder is a GRU module with an input feature size of 343 and a hidden state size of 256. The previously produced program tokens are encoded and concatenated with the program composer module output, passed through the GRU, and output logits are generated via two linear layers and the Tanh activation function. Finally, the program token is generated via a softmax over these logits.

A.3 PROGRAM COMPOSER

The program composer h_ω is a recurrent neural network which takes the output of the task interpreter as input and sequentially produce tasks embeddings. Specifically, at each step, h_ω takes in previously produced task embeddings, and last hidden state program decoder as input to predict next task embedding. It consists of a GRU module with an input feature size of 768 and a hidden state size of 256. This is followed by a single block of a linear layer and leaky ReLU activation.

B Hyperparameters and Training Details

In the datasets (D_{short}, D_{composed}) on which we train our model, we use a training/validation/test split of 70% / 15% / 15%. Our datasets contain 20 I/O string pairs for each program. During training, we randomly sample 10 of those 20 I/O pairs to use as the task specification.

We use the PyTorch framework for implementing HNPS.

B.1 Learning a Program Embedding Space

In this stage, our model aims to generate a program that is as close as possible to the program whose behavior is described by the input task specification. To do this, we employ cross-entropy loss between the generated program tokens and the ground truth program tokens. We employ a training regime with teacher forcing behavior, where the previously predicted program tokens are replaced with the ground truth program tokens.

B.2 Hierarchical Neural Program Synthesis

Here, in addition to the cross-entropy loss for individual program tokens, we employ a mean squared error loss for program embeddings. We use ground truth program embeddings for the program embedding loss during finetuning, but we let the embedding space drift. Even though ground truth program embeddings can be noisy, they still offer a valuable source of supervision for composing programs.

To determine when our synthesis module should stop outputting more program tokens, we initially tried to learn the prediction of a stop token. However, we observed that this was not robust. Instead, we take advantage of the fact that in the program synthesis domain, we can verify the execution of a program at any point. So, as we are composing programs together, if the addition of another subprogram causes a reduction in execution accuracy, then we terminate the program at this point - we found this heuristic performed well.

B.3 Baseline Implementation Details

Naïve, Naïve-short, and Naïve-short-finetune are implemented with a GRU encoder-decoder model. The encoder is implemented with the same architecture and hyperparameters as the HNPS task encoder, and the decoder uses the same architecture and hyperparameters as the HNPS program decoder.

B.4 Hyperparameters

Training runs use the Adam optimizer with an initial learning rate of 0.0001, and a learning rate scheduler decays the rate with a gamma of 0.95. A weight decay of 0.00001 is used. The model is optimized for 150 epochs of the dataset.

Hyperparameters used in our experiments are below:

- Number of I/O String Pairs in Task Specification: 10
- Optimizer: Adam
- Learning Rate: 0.0001
- Learning Rate Scheduler Gamma: 0.95
- Weight Decay: 0.00001
- Training Epochs: 150

Task Encoder:

- Model Type: GRU
- Model Type: GRU
- Input Size: 256
- Hidden State Size: 512
- Dropout: 0.2
- Activation Function: ReLU

- Number of Linear Layer Blocks: 1
- I/O Embedding Pooling Function: Mean

Program Decoder:

- Model Type: GRU
- Input Size: 256
- Hidden State Size: 256
- Activation Function: Tanh
- Number of Linear Layers: 2

Program Composer:

- Model Type: GRU
- Input Size: 768
- Hidden State Size: 256
- Activation Function: Leaky ReLU
- Number of Linear Layer Blocks: 1

HNPS Loss Coefficients:

- λ_1: 1.0
- λ_2: 1.0

B.5 SYNTAX CHECKER

Across all methods in our experiments, we utilize a syntax checker to make sure our synthesis module is outputting valid program tokens per the DSL. This aids in producing programs that can actually be compiled and executed. For example, if an open parentheses token is generated, the syntax checker will ensure that the expression in the parentheses is eventually closed with a close parentheses token. Another use case for the syntax checker is making sure regular expressions are formatted correctly during synthesis (*i.e.* the regular expression function type must be one of the following: `Word`, `Num`, `Alphanum`, `Allcaps`, `Propcase`, `Lower`, `Digit`, `Char`.

C PROGRAM DATASET

Once we have trained the embedding space, we use the trained task encoder to fetch the embeddings of each of the short programs. These short programs are composed together for the dataset D_{composed}. We store the short program embeddings along with the programs that those short programs combine together to form, so that we can calculate our program embedding loss $\mathcal{L}^{\text{PE}}$.

C.1 DATASET STATISTICS

D_{short} has 100,000 programs, which are created by randomly sampling one expression-long programs from our DSL. For each sampled program, we randomly generate 1,000 strings with 50-70 characters as potential input strings, and the first 20 input strings that can be executed without exceptions are stored in our dataset along with their corresponding output strings. If less than 20 input strings in the 1,000 generated are executed without exceptions for a program, then that program is discarded. For D_{composed}, we also generate 200,000 programs by composing sampled programs from D_{short}. In D_{composed}, 60,000 programs are composed with 2 short programs, 70,000 programs are composed with 3 short programs, and 70,000 programs are composed with 4 short programs. For each composed program in D_{composed}, the same method is used to generate I/O pairs.

We generate a long program dataset, D_{long}, which consists of 30,000 programs, each containing 2-4 expressions by directly sampling from our DSL. D_{long} contains 10,000 2 expression-long programs, 10,000 3 expression-long programs, and 10,000 4 expression-long programs.

To demonstrate the distribution gap between the 3 datasets, in Table 4, we show the frequency of important DSL tokens in each of our dataset. We calculate token frequency by counting total number of occurrences of each token divided by the amount of programs in the dataset.

Table 4: Frequency of important DSL tokens in each of our datasets. We calculate token frequency by counting total number of occurrences of each token divided by the amount of programs in the dataset.

token	D_{short}	D_{composed}	D_{long}
Concat	1.00	1.00	1.00
ConstStr	0.06	0.23	0.11
SubStr	0.94	2.82	2.89
Regex	1.64	4.92	5.21
ConstPos	0.23	0.72	0.57
Word	0.09	0.25	0.32
Num	0.04	0.11	0.34
Alphanum	0.09	0.25	0.32
Allcaps	0.08	0.25	0.33
Propcase	0.09	0.26	0.33
Lower	0.09	0.25	0.34
Digit	0.09	0.26	0.32
Char	0.06	0.21	0.31
All number tokens	1.87	5.66	5.78
All char tokens	1.08	3.31	2.71
Start	0.82	2.46	2.60
End	0.82	2.46	2.61

C.2 Program and I/O Pair Samples

We include examples of programs from our D_{composed} dataset and show 5 out of 20 I/O pairs we sampled for each program.

```
Input:    ZL%@r#ENc#HMONNRZLEBHVZMYIDUYBZJWQPOQLVLXGLMREUJDXQI’aP@Q(dR(#/E@7#//
Output:   @7ZL%@r#ENc#HMONNRZLEBHVZMYIDUYBZJWQPOQLVLXGLMREUJDXQI’aP
          @Q(dR(#/E@7#/

Input:    ##/@@@KA?/WYYESBKMODJKGDBBYWPVXFQINPRGVGYOHRQWDEQZIOCYVGPJh/#
Output:   @KA?/WYYESBKMODJKGDBBYWPVXFQINPRGVGYOHRQWDEQZIOCYVGPJh/KA?/W
          YYESBKMODJKGDBBYWPVXFQINPRGVGYOHRQWDEQZIOCYVGPJh

Input:    ?!b:{#M@T@$BRKOCJZARAAHMHPZWAKAUy:YSLDHLLVOPHFMPYTZCFr#/X@///TLa#
Output:   @///TLaBRKOCJZARAAHMHPZWAKAUy:YSLDHLLVOPHFMPYTZCFr#/X@//

Input:    @/EOJHGAWTNYWAETCPPFNTTECQVPFYSb@#/#@/#EPGDIKWHJMVQNTVM3
Output:   @/EOJHGAWTNYWAETCPPFNTTECQVPFYSb@#/#@

Input:    MG#i/ZMTEMXUVXDKEMIDAFAUV8@QOOXGRGGFVQTSOZUAURSv@@7#%#kX?/L[W’@z{a#/(
Output:   @z{aMG#i/ZMTEMXUVXDKEMIDAFAUV8@QOOXGRGGFVQTSOZUAURSv@@7#
          %#kX?/L[W’@z{a#

Ground Truth Program:  Concat c( SubStr s( ConstPos p( 7 p) Regex r( "#" 1 End r) s)
                       SubStr s( Regex r( "(" -2 Start r) Regex r( "." -2 End r) s) c)
```

```
Input:    G#PZoez.Xo()$D(AJW3Av#q’@Xi?#P..TlY(O(N&XI#G: Q.hQT#dk)0Y(jSF
Output:   .Xo()$D(AJW3Av(N&XI#G: Q.hQT#dk)0Y(j

Input:    xN#iiu/HHfa4(!pEDz4{q(,(nlw.{JsGN#SK;h./CjDt}u.n0DgMuj#KBR,H#.(F;SR;R
Output:   HHfa4(!pEDz4{q(,(nlw.{JsGN(nlw.{JsGN#SK;h./CjDt}u.n0DgMuj#KBR,H#.(F;SR

Input:    ncL9.(Cbz3q[y9CXHkjjI84RA!sRX.T’;%G(kdk.@zb#j(D#X]#.&CHSwcdB([D}4 1
Output:   bz3q[y9CXHkjjI84RA!sRX.T’;%G(kdk.@zb#j(D(D#X]#.&CHSwcdB([D}4

Input:    78@aQ0(I(:#[h]#)8F3C.9(.5h9UXz#&L(2gZCR8d5iK33.Z.e[8
Output:   I(:#[h](.5h9UXz#&L(2gZCR8d5iK33.Z.e

Input:    :Jmq?T(R1hD.#;,.(Q9X?q7Dm}1Vq6e’0.J(##C8]s]5.X/a0?mC9(5k
Output:   R1hD.#;,.(Q9X?q7Dm}1Vq6e’0.J((##C8]s]5.X/a0?mC9(

Ground Truth Program:  Concat c( SubStr s( Regex r( "@" -1 End r) Regex r( "#" -1 End r) s)
                       SubStr s( Regex r( Allcaps 0 Start r) Regex r( "/" -1 End r) s) c)
```

```
Input:    !&]dr]1rqFpih3CdBEEICIvPaNkuHa4$0lX!{IA&eFkAUtbDBAs5GusY’&&
Output:   !&]dr]1rqFpih3CdBEEICIvPaNkuHa4$0lX!{IA&eFkAUtbDBAs5Gus
          $]dr]1rqFpih3CdBEEICIvPaNkuHa4$0lX!{IA&eFkAUtbDBAs5GusY’

Input:    a]&]&EqZWNGObvBcuKmtUvUzEtupBTYdHjfyMOj3R!f&wkhWgLeLk5!tShuzGfo4&
Output:   !f&wkhWgLeLk5!tShuzGf$]&]&EqZWNGObvBcuKmtUvUzEtupBTYdHjf
          yMOj3R!f

Input:    evFjtgoxhRAUsAoeAbNK4&$m]SfjBenvmUL9]!.MQBWzpFEbIcktLQSgasdlHc#&&Eg&!e
Output:   !.MQBWzpFEbIcktLQSgasdlHc#&&E$]SfjBenvmUL9]!.MQBWzpFEbIc
          ktLQSgasdlHc#&

Input:    f&]LP!rSAlV&INLbdV!&.soP%D@YDxCzkUaC63IgUV7sLsgxzcgnFsbWCz7!D2H DqA&]
Output:   !rSAlV&INLbdV!&.soP%D@YDxCzkUaC63IgUV7sLsgxzcgnFsbWCz7!D2
          H Dq$]LP!rSAlV&INLbdV!

Input:    Z](&SCh2!&(SWbZZjJOD1PC]FlgEAwCAXNZsLbdmJbzFLwd$V!&?4n&qVVQ
Output:   !&(SWbZZjJOD1PC]FlgEAwCAXNZsLbdmJbzFLwd$V!&?4n&qVV$](&
          SCh2!&(SWbZZjJOD1PC]FlgEAwCAXNZsLbdmJbzFLwd$V!

Ground Truth Program:   Concat c( SubStr s( Regex r( "!" 0 Start r) Regex r( Word -1 End r) s)
                        ConstStr k( "$" k)
                        SubStr s( Regex r( "]" 0 End r) Regex r( "&" 2 End r) s) c)
```

```
Input:    ;(qx{Y,,,FkAUtb}))D))DCAt1W,(2,S;BAspoCdBEEICIvPaNkuHajGdqFpi4;:F
Output:   (qx{Y,,))DCAFkAUtb}))D))DCAt1W,(2,S&

Input:    VI,r#;zGfo6,q);/X(3,,roqpvaDzj0,R;Z’)(&yK;ZDEq3)OSus$xGc)AqskKS7BX{lh
Output:   (3)OSus$xGc)AqskKS7BzGfo6,q)&

Input:    )S),);NpQFmcNGTMMQB4bbXkWitgy0,;(S,o;,)LmjUGnwzcm7,(IPpqrv10
Output:   (S);NpQFmcNGTMMQB4bbXkWitgy0,;(S,o;,)LmjUGnwzcQFmcNGTMMQB4bb
          XkWitgy0,&

Input:    HW(fMhJyyV0)T)ex,),iLiRno’E;;,),)yZXAIbscuV6(vUHgjyB7,;o
Output:   (fMhJyyV0)T)ex,),iLiRno’E;;),)yZXAIbscuMhJyyV0)T)ex,),iLiRno
          ’E;&

Input:    )zRxForqNOFp7FL,4))2;ksgGpWkZRRdGMvjxaT2((,,;1;4)jYeft9,FYrrzBY7,b
Output:   ((,)2;ksgGpWkZRRdGMvjxaT2((,,;1;4)jYefForqNOFp7FL,4))2;ksgGp
          WkZRRdGMvjxaT2((,,&

Ground Truth Program:   Concat c( SubStr s( Regex r( "(" 0 Start r) Regex r( "," -3 Start r) s)
                        SubStr s( Regex r( ")" -2 Start r) Regex r( Word -2 End r) s)
                        SubStr s( Regex r( Char 3 End r) Regex r( ";" 1 End r) s)
                        ConstStr k( "&" k) c)
```

```
Input:    2q@?QK6IAd}t2?))}!}!l?AV}UP,Hnriz1OoH?!CsyjyICX!!!Esi$80)DU5i}}
Output:   @?QK6IAd}t2?))}!}!l?AV}UP,Hnriz1OoH?!CsyjyICX))}!}!l?A
          V}Ut2?))}!}!l?AV?!

Input:    !WHj!?XUqQT }}yOI)x1[SWU’!KoVWd.RoutoK}!?}?Rocescz7!lWBxhk$OXu)}a))
Output:   Hj!?XUqQT }}yOI)x1[SWU’!KoVWd.RoutoK})x1[SWU’!KoVyOI)x1[S
          WU’!KoVWd.RoutoK?Rocescz7!lW

Input:    Xcxu6EQ1!Bx(}}1KMx?!}8U@}?!!?M)XpoeCWokeC!Lbiai.EQ7un}BH!})zy{QA(
Output:   xu6EQ1!Bx(}}1KMx?!}8U@}?!)XpoeCWokeC!Lbiai.E1KMx?!}8U@?
          M)XpoeC

Input:    EAb?u6!)DK]!Cp#?mFo$IR&cC/})Y!3Imegp3SW@CV(!}n}?MseO}Jbdt3W!Tr7}BKp
Output:   b?u6!)DK]!Cp#?mFo$IR&cC/})Y!3Imegp3SW@CV()DK]!Cp#?mFo$
          IR&cC/})Y!3Imegp3SDK]!Cp#?mFo$IR&cC/})Y!3Imegp3SW@CV(!}n?MseO}Jbdt3W!

Input:    CR@Vo,!?!))!NL’PBoSZ1}!ToKOOoja$TphjC!}BjhfwkJJO(?q}?}DrpRtNN2}
Output:   @Vo,!?!))!NL’PBoSZ1}))!NL’PBoSZ1}!ToKONL’PBoSZ1}!ToKOOoja
          $TphjC!}BjhfwkJJO(?q?}

Ground Truth Program:   Concat c( SubStr s( ConstPos p( 2 p) Regex r( "!" 3 Start r) s)
                        SubStr s( Regex r( ")" 0 Start r) Regex r( Allcaps -3 End r) s)
                        SubStr s( Regex r( Alphanum 2 Start r) Regex r( "}" -3 End r) s)
                        SubStr s( Regex r( "?" -1 Start r) Regex r( Propcase -2 Start r) s) c)
```

D Extended Results

D.1 Generated Programs

We show examples of programs generated by our HNPS model. Each program is generated given task specifications randomly picked from D_{composed}. We show a comparison between each generated program with the ground truth program given by the datsaset.

Ground Truth Program:	Concat c(SubStr s(Regex r(Alphanum -3 Start r) Regex r("(" -2 Start r) s) SubStr s(Regex r(Alphanum -1 End r) Regex r("." -1 Start r) s) c)
HNPS Generated Program:	Concat c(SubStr s(Regex r(Alphanum -3 Start r) Regex r("(" -2 End r) s) SubStr s(Regex r(Alphanum -1 End r) Regex r("." -1 Start r) s) c)

Ground Truth Program:	Concat c(SubStr s(Regex r("!" 0 Start r) Regex r("$" -1 End r) s) SubStr s(ConstPos p(12 p) Regex r("." -1 End r) s) c)
HNPS Generated Program:	Concat c(SubStr s(Regex r("!" 0 End r) Regex r("." -1 End r) s) SubStr s(ConstPos p(12 p) Regex r("$" -1 End r) c)

Ground Truth Program:	Concat c(SubStr s(Regex r(Propcase -3 End r) Regex r("[" -2 Start r) s) SubStr s(Regex r("." 1 Start r) Regex r(Propcase 1 Start r) s) SubStr s(Regex r("@" -3 End r) ConstPos p(-8 p)) c)
HNPS Generated Program:	Concat c(SubStr s(Regex r(Propcase -3 End r) Regex r("[" -2 End r) s) SubStr s(Regex r("." 1 End r) Regex r(Propcase 1 Start r) s) SubStr s(Regex r("@" -3 Start r) ConstPos p(-8 p) c)

Ground Truth Program:	Concat c(SubStr s(Regex r("SPACE" -1 End r) Regex r("{" -2 End r) s) SubStr s(Regex r("?" 2 End r) Regex r(Lower 2 End r) s) SubStr s(Regex r(Allcaps 3 Start r) Regex r("," -1 Start r) s) c)
HNPS Generated Program:	Concat c(SubStr s(Regex r("SPACE" -1 Start r) Regex r("{" -2 End r) s) SubStr s(Regex r("?" -2 Start r) Regex r(Lower -1 End r) s) SubStr s(Regex r(Allcaps 3 Start r) Regex r("," -1 Start r) s) c)

Ground Truth Program:	Concat c(SubStr s(ConstPos p(14 p) Regex r("$" 0 End r) s) SubStr s(Regex r("'" -2 Start r) Regex r(Digit -2 Start r) s) SubStr s(Regex r(Digit 0 End r) Regex r(Word 3 End r) s) SubStr s(Regex r("!" 2 Start r) Regex r(Char -3 Start r) s c)
HNPS Generated Program:	Concat c(SubStr s(ConstPos p(14 p) Regex r(Char -1 End r) s) SubStr s(Regex r("'" 2 End r) Regex r(Digit -2 Start r) s) SubStr s(Regex r(Char 3 End r) Regex r(Word 3 End r) s) SubStr s(Regex r("!" 2 Start r) ConstPos p(-4 p) c)

Concat c(ConstStr k("SPACE" k)	SubStr s(Regex r("." 2 End r) Regex r("?" 2 End r) s) SubStr s(Regex r("@" -2 End r) Regex r(Alphanum 3 End r) s) SubStr s(Regex r(")" 1 Start r) Regex r(Num -1 Start r) s) s) c)
HNPS Generated Program:	Concat c(ConstStr k("SPACE" k) SubStr s(Regex r("." 2 End r) Regex r("?" 2 End r) s) SubStr s(Regex r("@" -2 End r) Regex r(Alphanum 3 End r) s) SubStr s(Regex r(")" 1 End r) Regex r(Num -1 Start r) s) c)

D.2 Evaluation on D_{composed}

We additionally evaluate execution accuracy of each method on our $D_{composed}$ dataset. In Table 5, we group programs by their length in terms of the number of tokens. In Table 6, we separate programs by the number of short programs used to composed them. From the results, we can see HNPS & HNPS-PT are able to outperform H-Naïve-PT. The gap in performance between these and H-Naïve-PT increases as programs get longer. For long programs composed of 2 short programs, H-Naïve-PT reaches 90% of HNPS's performance, but its execution accuracy is only 63% of that of HNPS. This highlights the importance of the task embedding space in program synthesis with hierarchical architectures. We also observe that non-hierarchical baselines (Naïve-*) have significantly lower performance than hierarchical methods on both training and test set programs. On programs

Table 5: The execution accuracy of each method on the training set and test set of D_{composed}. The programs are separated into groups by their length in terms of number of tokens.

Method	**Train**				**Test**			
	10-25	**25-40**	**40-55**	**55-70**	**10-25**	**25-40**	**40-55**	**55-70**
Naïve	59.6%	27.19%	3.22%	0.12%	55.24%	23.6%	2.41%	0.01%
Naïve-short	0.04%	0.01%	0.00%	0%	0.02%	0.01%	0.00%	0.00%
Naïve-short-finetune	60.64%	34.73%	8.52%	0.55%	58.78%	30.51%	5.79%	0.29%
H-Naïve-PT	81.95%	59.12%	32.93%	11.14%	75.17%	46.6%	15.89%	2.00%
HNPS-PT	**86.49%**	**67.57%**	**40.63%**	**15.88%**	**79.61%**	**54.7%**	21.42%	3.25%
HNPS (ours-full)	85.14%	65.19%	37.6%	13.29%	79.58%	53.64%	**21.88%**	**3.73%**

Table 6: The execution accuracy of each method on the training set and test set of D_{composed}. The programs are separated into groups by the amount of subprograms used to compose them.

Method	**Train**			**Test**		
	2	**3**	**4**	**2**	**3**	**4**
Naïve	33.01%	7.67%	0.87%	29.11%	5.84%	0.66%
Naïve-short	0.02%	0.00%	0.00%	0.01%	0.00%	0.00%
Naïve-short-finetune	41.32%	13.33%	1.53%	36.82%	9.79%	0.85%
H-Naïve-PT	63.27%	38.88%	16.69%	52.22%	22.43%	4.6%
HNPS-PT	**71.53%**	**47.89%**	**20.88%**	**60.42%**	28.82%	6.24%
HNPS (ours-full)	69.23%	44.23%	18.98%	58.67%	**29.23%**	**7.28%**

Table 7: We show the "generalization gap" of each method by calculating the the difference between training and test set performance.

Method	**By number of tokens**				**By number of short programs**		
	10-25	**25-40**	**40-55**	**55-70**	**2**	**3**	**4**
Naïve	-4.36%	-3.59%	-0.81%	-0.11%	-3.90%	-1.83%	-0.11%
Naïve-short	-0.02%	0.00%	0.00%	0.00%	-0.01%	0.00%	0.00%
Naïve-short-finetune	-1.86%	-4.22%	-2.73%	-0.26%	-4.50%	-3.54%	-0.68
H-Naïve-PT	-6.78%	-12.52%	-17.04%	**-9.14**%	-11.05%	-16.45	-12.09%
HNPS-PT	-6.88%	-12.87%	-19.21%	-12.63%	-11.11	-19.07%	-14.64
HNPS (ours-full)	**-5.56%**	**-11.55**	**-15.72**	-9.56%	**-10.56%**	**-15.00%**	**-11.7%**

composed of 4 short programs, none of the non-hierarchical baselines are able to achieve an execution accuracy higher than 2%, even on the training set, where all hierarchical variants reach more than 10% execution accuracy. This supports our assertion that without the hierarchical architecture, the model's ability to fit to a distribution of long programs is fundamentally limited.

In Table 7, we also present the difference between training and test set performance, or the "generalization gap," on D_{composed}. HNPS has a smaller generalization gap than both HNPS-PT and H-Naïve-PT. This difference is especially noticeable when comparing HNPS and HNPS-PT on programs longer than 40 tokens, where HNPS-PT performs worse on the test set despite obtaining a better execution accuracy on the training set. This implies that the program embedding loss helps prevent overfitting in hierarchical models.