Latent Execution for Neural Program Synthesis

Xinyun Chen, Dawn Song, Yuandong Tian
UC Berkeley, Facebook AI Research
{xinyun.chen, dawnsong}@berkeley.edu, yuandong@fb.com

1 Overview

Program synthesis from input-output (IO) examples has been a long-standing challenge. While recent works demonstrated limited success on domain-specific languages (DSL), it remains highly challenging to apply them to real-world programming languages, such as C. Due to complicated syntax and token variation, there are three major challenges: **(1)** unlike many DSLs, programs in languages like C need to compile first and are not executed via interpreters; **(2)** the program search space grows exponentially when the syntax and semantics of the programming language become more complex; and **(3)** collecting a large-scale dataset of real-world programs is non-trivial.

To address these challenges, we propose `LaSynth` that learns the latent representation to approximate the execution of partially generated programs, even if they are incomplete in syntax (addressing **(1)**). The learned execution significantly improves the performance of next token prediction over existing approaches, facilitating search (addressing **(2)**). Finally, once trained with randomly generated ground-truth programs and their IO pairs, `LaSynth` can synthesize more concise programs that resemble human-written code. Furthermore, retraining our model with these synthesized programs yields better performance with fewer samples for both Karel and C program synthesis, indicating the promise of leveraging the learned program synthesizer to improve the dataset quality for input-output program synthesis (addressing **(3)**). In the evaluation, `LaSynth` achieves 55.2% accuracy on generating simple C code with tens of tokens including loops and branches, outperforming existing approaches without executors by around 20%.

This work is under review for NeurIPS 2021, and the preprint is available here: `https://arxiv.org/abs/2107.00101`. We present the key results below, and more details can be found in the preprint.

2 Approach

2.1 Latent Execution Framework

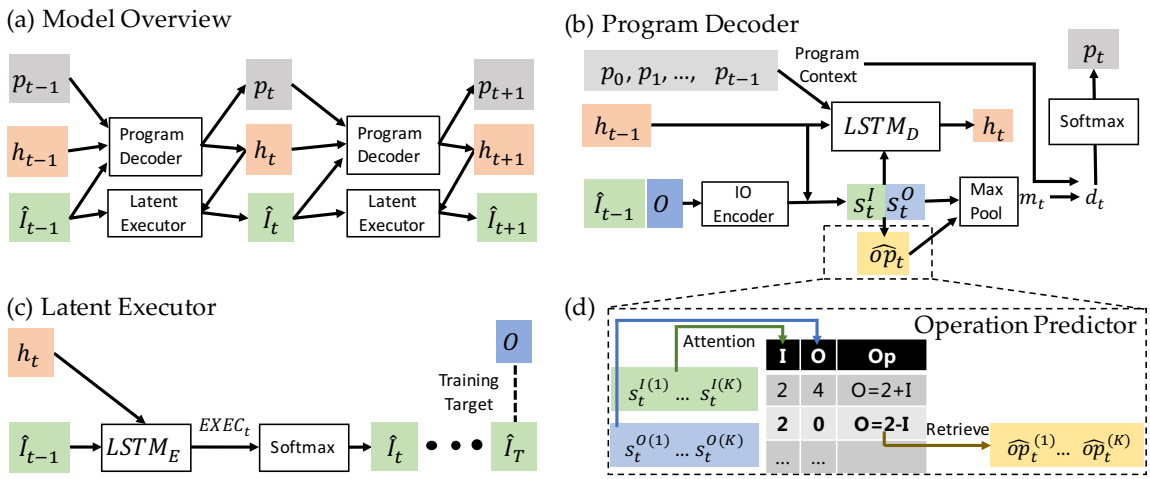

Figure 1: *(a) An overview of* `LaSynth` *model architecture. (b), (c), and (d) present the details of the program decoder, latent executor, and the operation predictor.*

As shown in Figure 1, we propose the latent execution framework that improves over standard program synthesis architectures. The standard program decoder architecture typically does not achieve strong performance when the program complexity increases. One main reason is that the standard program decoder only takes the initial IO pairs as the input without considering the program execution, thus the learned representation for the partial program does not effectively guide

Table 1: Results on Karel (left) and C (right) datasets.

Approach	Gen	Exact
`LaSynth`	83.68%	41.12%
Exec	**86.04%**	39.40%
Bunel et al.	77.12%	32.17%
Shin et al.	81.30%	**42.80%**

Approach	Accuracy
`LaSynth`	**55.2%**
`NoAttentionInDecoding`	53.5%
`NoOpPredictor`	53.7%
`NoPartialExecutor`	42.9%
`NoExecutor`	38.6%
RobustFill	37.6%
Property Signatures	34.5%

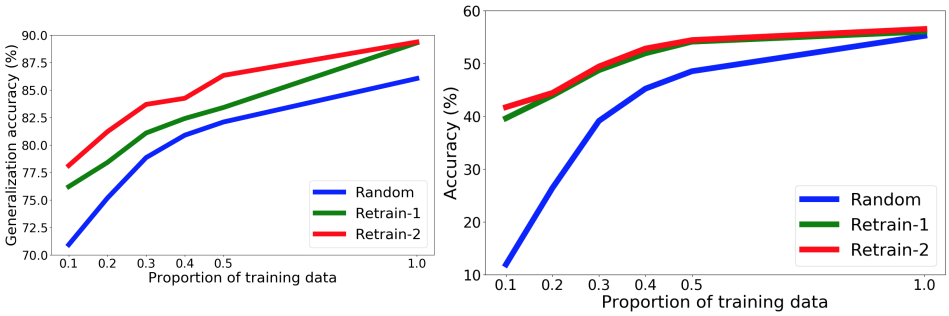

Figure 2: *Results of iterative retraining on Karel (left) and C (right) datasets.*

the synthesis process. To address this challenge, our *latent executor* (Figure 1(c)) maintains a second representation $\hat{I}_t$ during program decoding. Intuitively, $\hat{I}_{t-1}$ models the *hypothetical input* of the partial program $p_{t...T}$ so that its output becomes O.

2.2 Data Regeneration and Iterative Retraining

Interestingly, once our model is trained on the initial random generated programs, the predicted program becomes more concise and resembles human-written code. While the exact token match accuracy is low even on the training set, the model still satisfies the IO pairs for many problems. We leverage such a phenomenon to construct a new dataset with higher-quality programs. Specifically, we run beam search on the trained model to predict program given input-output pairs in the training set. If model prediction satisfies all the input-output examples and held-out cases, we replace the original program with the prediction. Afterward, we re-train the model on the new dataset.

3 Results

We present the overall results of `LaSynth` and baseline model architectures in Table 1. On both Karel and C benchmarks, `LaSynth` outperforms all baselines that do not incorporate the partial program execution information. In Figure 2, we further show the effectiveness of iterative retraining. We observe that retraining for one iteration is sufficient, and it significantly improves the generalization accuracy especially when the training sample size is small.

4 Future Work

We plan to extend our `LaSynth` framework to synthesize more complicated real-world code, and we list some promising directions as follows.

- Train the latent execution framework on real-world C code. Given that the C programs in our benchmark are randomly generated, their complexity and diversity still do not match the human-written code. We plan to train our model on a large-scale collection of human-written programs, and evaluate the synthesis performance on real-world code.
- Design a better model architecture for the latent executor, and build upon the recent advancements in large-scale pre-trained language models.
- Extend our iterative retraining scheme to improve the quality of benchmarks including real-world code. We can also combine our technique with other approaches for reducing the bias of data distributions, so that the trained models could better generalize to new coding tasks.