Neural Generation of Source Code for Program Synthesis

Kensen Shi
Department of Computer Science
Stanford University
kensens@cs.stanford.edu

Abstract

Existing program synthesizers sometimes consider many unnatural candidate programs that a human programmer would immediately identify to be undesireable in some way. To improve the quality of these candidate programs, we propose a neural model that generates a sketch of a method body given the desired method signature (where the sketch would later be concretized and evaluated by a program synthesizer). Specifically, we use transfer learning: we first train the model on a large dataset of code scraped from GitHub, and afterward we retrain the model to obtain a different distribution of code phenomena that will be useful to a specific program synthesizer. Our model is able to produce reasonable-looking method sketches even for signatures unseen during training.

1 Introduction

Program synthesis is the task of automatically generating a program (source code) that meets a given specification. For example, a user might provide input-output pairs or a logical formula that describes the desired program's behavior, and a program synthesizer must find a program that adheres to the specification or otherwise satisfies the user. Many program synthesizers involve a search over programs, but the distribution over searched programs is often very different from the distribution over source code that experienced human programmers might write. For example, brute-force approaches that enumerate all programs in a grammar, and random searches that generate programs in a straightforward way, will typically consider programs that contain redundant or useless computation.

FrAngel[1] is a Java program synthesizer that accepts a desired method signature along with input-output examples and outputs a Java method implementation consistent with those examples. At a high level, FrAngel naïvely generates abstract syntax trees (ASTs) representing the source code in a top-down fashion, choosing nodes randomly among options that typecheck (given any type constraints implied by the parent nodes). Each generated method implementation is executed to check for a match on the given input-output pairs. Unfortunately, FrAngel suffers from the previously-described problem—it often generates code that a human programmer can immediately identify as excessively verbose, redundant, or trivial, such as the following:

```
String foo(String str) {
    int var1 = 0;
    String var2 = "";
    str = var2;
    return "";
}
```

In the above example, `var1` is declared but never used, the argument `str` is "thrown away" by being assigned to `var2` before `str` is used at all, and the return expression is a constant (with no dependence on any variables). These symptoms, among others, are quite common in FrAngel's randomly-generated code.

This project aims to use a neural model to generate *natural* source code, approximating the distribution of reasonable code that a human programmer might write for a given method signature. This can aid existing program synthesizers such as FrAngel by reducing the number of "obviously-bad" programs that must be executed or evaluated.

Our model takes as input the types in the desired method signature and outputs a sequence of tokens for the method body. We focus on code *sketches*, where variable names, method names, string literals, and other categories of tokens are abstracted, since such tokens do not generalize across different pro-

[1] A research project I worked on with Prof. Percy Liang, independent of CS 230.

grams. Hence, `return <Class>.<Method>(var1, <StringLiteral>);` might be a part of a code sketch produced by our generative model. Concretizing these code sketches while satisfying type constraints is a task left to the program synthesizer.

2 Data Collection

2.1 Data from GitHub

To create the *GitHub* dataset, we downloaded the 20 most-starred Java repositories on GitHub and used Eclipse's Java parser to obtain ASTs for all Java files. Because the AST does not directly provide a tokenization of the source code, we obtained a sequence of tokens for each method in the following manner. Each node of the AST corresponds to a substring of the source code. For instance, a node for array access might correspond to the code `arr[i + 1]`, and its children nodes correspond to the expressions `arr` and `i + 1`. The tokens contributed by the array access node are the consecutive characters that are not covered by its children, i.e., `[` and `]`. Hence, to obtain tokens, a node collects its children's tokens and inserts the remaining blocks of consecutive characters (with whitespace stripped) as tokens in the proper places, producing the sequence of tokens `arr`, `[`, `i`, `+`, `1`, and `]` in the previous example.

We standarized variable names (`arg1`, `arg2`, and so on for arguments; `var1` and so on for local variables, etc.) and grouped certain types of tokens (e.g., `12.3` and `230` are both replaced with `<NumberLiteral>`, and other groups include `<Class>`, `<Method>`, etc.). In this way, we refrain from learning the implementation-specific details of the dataset and focus instead on the general structure of the code.

We limit the *GitHub* dataset to include only static methods with 10 to 120 tokens in the method body, and we also exclude methods using certain Java constructions that FrAngel cannot handle (e.g., exception handling, lambda expressions, switch statements, etc.[2]). This results in over 10,000 examples. 10% of the examples were set aside for validation, and the remaining 90% were used for training.

The following are two examples from the *GitHub* dataset. Tokens are space-separated, and newlines and indentation were added for clarity. The first token in the signature is always the method's return type, and remaining tokens are the argument types.

```
Signature: int int[]
Method:
int var1 = <NumberLiteral> ;
for ( int elem1 : arg1 ) {
    var1 *= elem1 ;
}
return var1 ;
```

```
Signature: void <Class> <Class>
Method:
String var1 = <Method> ( arg1 ) ;
<Class> . <Method> ( arg2 , var1 ) ;
```

2.2 Data from FrAngel

We noticed that the *GitHub* dataset does not include a high proportion of looping control structures, which are frequently necessary to solve tasks in FrAngel's benchmarks. Hence, to obtain training data that more closely matches the desired distribution of code phenomena, we also collected method implementations by running FrAngel 20 times on our existing benchmark suite of 90 tasks.

Because we ran FrAngel multiple times, and FrAngel uses a random search, many tasks have multiple solutions. Hence, we weight the examples such that each task has equal weight, and the task's total weight is divided equally among its solutions. Thus, the dataset is not biased toward tasks with many different solutions. We refer to this dataset of FrAngel-generated solutions as the *Solutions* dataset, containing about 500 examples.

In addition to solutions to tasks, FrAngel also outputs some "partial successes" (method implementations that pass some but not all of the test cases). These partial successes are often similar in structure to the solutions, so we include them (along with the solutions) in a *Synthesizer* dataset, containing about 3000 partial successes in addition to the 500 solutions. Examples are weighted where half of a task's weight is distributed among the task's partial successes, and the other half among the task's solutions.

The *Solutions* and *Synthesizer* datasets were split into train/val/test sets by task. Specifically, 9 tasks were randomly chosen to form the validation set, 9 remaining tasks formed the test set, and the remaining 72 tasks formed the training set. (For consistency, the same tasks were selected for both datasets.) In

[2]This was not done exhaustively—simpler constructions such as the ternary operator were left in the dataset, even though FrAngel does not produce ternary operators.

this way, the validation and test sets cover tasks and signatures that are not present in the training set.

We use a vocabulary including all 63 distinct tokens in the *Synthesizer* dataset, plus the most common tokens in the *GitHub* dataset to reach a vocab size of 100. We discarded the 1.7% of examples in the *GitHub* dataset with tokens not in the vocabulary.

3 Approach

3.1 Model Architecture

Our recurrent model takes as input (at each time step) the previous token and the method signature, and outputs a probability distribution for the text token. (The model only produces the method body, not including the method signature.) The method signature is described by the return type, followed by argument types, padded if necessary to become a fixed length.[3] Tokens (including types in the signature) are mapped to a 64-dimensional embedding, which is learned during training. The embeddings are passed through a 2-layer LSTM with 512 hidden dimensions per layer, and finally a fully-connected layer with softmax activation transforms the LSTM output into a probability distribution over tokens. We implemented this model using PyTorch [5].

We trained the model using a modified cross entropy loss that takes into account the example weight and the output sequence length. Specifically, the loss function minimized is

$$\left(-\sum_{i=1}^{m} \frac{w^{(i)}}{|y^{(i)}|} \sum_{j=1}^{|y^{(i)}|} \log \hat{p}\left(y^{(i)\langle j\rangle}\right) \right) \Bigg/ \sum_{i=1}^{m} w^{(i)}$$

where m is the number of examples, $w^{(i)}$ is the weight of the i-th example, $y^{(i)}$ is the output sequence of the i-th example, $y^{(i)\langle j\rangle}$ is the j-th token of $y^{(i)}$, and $\hat{p}(t)$ is the model's predicted probability of token t (note that $\hat{p}$ is computed by a recurrent model and changes at each time step). We normalize by the length to avoid a bias toward longer sequences, which contribute more terms to the inner sum.

Sampling from the model (given a method signature) is done in the standard way—sampling one token at a time from the model's output probability distribution, and feeding each token back into the model to produce the next token, until an end-of-sequence token is sampled.

[3] We handled methods with up to 4 arguments, even though FrAngel's benchmarks only involve up to 3 arguments.

3.2 Transfer Learning

Note that the *GitHub* dataset is relatively large but does not contain the ideal distribution of code phenomena actually required by solutions to FrAngel's benchmarks. On the other hand, the *Solutions* and *Synthesizer* datasets are generated by FrAngel and are much closer to this desired distribution, but these datasets are much smaller. We therefore use transfer learning—first training on the *GitHub* dataset, and then training on the *Solutions* or *Synthesizer* datasets—to let the model learn general properties of code (i.e., where to place semicolons) from the larger *GitHub* dataset while refining the output distribution to mimic the smaller synthesizer-generated datasets. This application of transfer learning to obtain a more desirable output distribution is a primary contribution of this project.

3.3 Hyperparameters and Training

The number of LSTM layers and hidden dimensions, learning rate (0.002), and weight decay (ℓ_2 regularization parameter of 0.0002) were all chosen with a hyperparameter search using the *Synthesizer* validation loss of the best model (among all models, trained with and without transfer learning). We used a minibatch size of 128 and trained the models for 50 epochs with the Adam optimizer. However, after every epoch we computed the validation loss, and we selected the model that minimized that loss, which typically occured before epoch 30.

4 Results

Table 1 summarizes our results. The first three rows correspond to models trained only on a single dataset (without transfer learning), and the last two rows correspond to transfer learning models trained first on the *GitHub* dataset and then on either the *Solutions* or *Synthesizer* dataset. The columns correspond to the train and test sets of the *Solutions* and *Synthesizer* datasets. The loss and token-level accuracy are listed in the cells. Accuracy is computed with the following formula (weighted and normalized by sequence length, similar to the loss):

$$\frac{-\sum_{i=1}^{m} \frac{w^{(i)}}{|y^{(i)}|} \sum_{j=1}^{|y^{(i)}|} 1\left[y^{(i)\langle j\rangle} = \arg\max_{t \in V} \hat{p}(v)\right]}{\sum_{i=1}^{m} w^{(i)}}$$

Training method	Sol-Train	Sol-Test	Syn-Train	Syn-Test
GitHub	1.23 (70.6%)	1.27 (72.4%)	1.37 (68.5%)	1.33 (70.5%)
Solutions	0.72 (78.6%)	1.26 (62.6%)	0.93 (73.0%)	1.33 (60.7%)
Synthesizer	0.44 (87.0%)	1.09 (69.4%)	0.52 (84.0%)	1.13 (68.4%)
Transfer-Sol	0.26 (92.7%)	1.00 (74.6%)	0.54 (85.7%)	1.06 (73.0%)
Transfer-Syn	0.22 (93.7%)	**0.95 (75.1%)**	0.32 (90.4%)	**0.99 (73.8%)**

Table 1: Loss (and token accuracy) for models trained on the 3 datasets and two transfer learning models. The Transfer-Syn model, trained on the *GitHub* and *Synthesizer* datasets, performs the best overall.

Figure 1 shows some example programs generated by Transfer-Syn for signatures not present during training. We note that the implementations generally look reasonable for the given signatures, although not all of them will compile (after appropriately concretizing the method sketches).

4.1 Discussion

Table 1 shows that the transfer learning approach is successful. That is, Transfer-Sol and Transfer-Syn consistently achieve lower loss and higher accuracy than the Solutions and Synthesizer models respectively, on both test sets. Furthermore, while the Solutions and Synthesizer models obtain lower accuracy than the GitHub model (which is not unreasonable since the *Solutions* and *Synthesizer* datasets are very small), both transfer learning models outperform the GitHub model on both metrics. Thus, we conclude that the transfer learning approach is an effective way to learn from scraped and synthesizer-produced code.

The results also show that models trained on the full *Synthesizer* dataset (with and without transfer learning) outperform models trained only on the *Solutions* subset, even when tested on the *Solutions* subset. From this observation, we conclude that expanding a synthesizer-produced dataset with partial successes in addition to final solutions can significantly boost overall performance.

The Transfer-Syn model, trained first on the *GitHub* dataset and then on the *Synthesizer* dataset, is consistently the best model in all metrics. By using transfer learning with the full *Synthesizer* dataset, it is positively affected by both of the above observations. The model exceeds 75% token-level accuracy on the *Solutions* test set, which is quite impressive considering that there are many reasonable method implementations for a given input signature, so it is impossible to achieve 100% accuracy.

Our models do not always produce method sketches that will compile (after concretization). The most common errors we observed were mismatched types, incorrect variable naming, and missing or extraneous parentheses or braces. For instance, the second method in Figure 1 attempts multiply by `arg2`, but `arg2` is a `String` according to the signature. The first method in Figure 1 declares the variable `elem1` twice, when it should have used `elem2` for the inner loop.

We also observed the models generalize beyond the signatures seen during training by learning good token encodings. In particular, the first method in Figure 1 has a signature similar to one in the training set, except with a `List` instead of a `Set`. As a result, the model is able to generate a method that is similar to the training example's solution, except using an `ArrayList` instead of a `HashSet` (and erroneously using `elem1` in place of `elem2` as discussed earlier).

5 Related Work

Many works in program synthesis use various deep learning approaches (sometimes combined with program synthesis techniques) to produce programs from natural language descriptions [1, 3, 6, 7] and/or input-output examples [2, 6]. However, these approaches primarily aim to find a correct solution to the task described by the input. Our setting is quite different, since the input is a method signature that corresponds to many possible method bodies. Furthermore, we do not want to produce a single "best" output—we instead seek a generative model that can produce a variety of method bodies for a single signature, to be used as a first step in a larger program synthesizer.

Maddison and Tarlow [4] propose generative models of *natural* source code. While their goal of naturalness is similar to ours, their model is not directly applicable to program synthesis because it is unconditional, whereas our setting requires generating source code conditioned on properties of the synthesis task (specifically, the desired method signature).

```
Signature: List String[] String[]
Method:
ArrayList var1 = new ArrayList();
for (String elem1 : arg1) {
    for (String elem1 : arg2) {
        if (elem1.<Method>(elem1)) {
            var1.<Method>(elem1);
        }
    }
}
return var1;

Signature: double <Class> String
Method:
return <Class>.<Field> * arg2 + arg1.<Method>();

Signature: <Class>[] <Class>[] String
Method:
for (int i1 = <NumberLiteral>;
        i1 <IneqOp> arg1.<Field>; i1++) {
    <Class>.<Method>(arg1, arg2);
}
return arg1;

Signature: void <Set> <Class> int
Method:
for (int i1 = <NumberLiteral>;
        i1 <IneqOp> arg3; i1++) {
    <Class>.<Method>(arg1, arg2.<Method>(arg3));
}

Signature: <Class> short
Method:
<Class> var1 = new <Class>();
var1.<Method>(<NumberLiteral>, arg1);
return var1;

Signature: Object boolean Object
Method:
for (<Class> elem1 :  arg1) {
    if (arg2.<Method>(elem1)) {
        return <Class>.<Method>();
    }
} }
return arg2;
```

Figure 1: Example programs generated by the best model, Transfer-Syn. None of these signatures appeared in any of the training sets. Whitespace is edited for clarity. Compilation errors are underlined in red.

The author is not aware of other work that specifically generates method bodies given the desired method signature.

6 Conclusion

We presented a model that generates source code sketches of a method body given the desired method signature. We found that transfer learning is an effective way to learn from code scraped indiscriminately at a large scale while approximating a different distribution of desired code phenomena. Our model produces natural-looking code even for signatures unseen during training. Overall, this approach seems to be a promising way to improve the candidate programs considered by search-based program synthesizers.

As future work, one can modify the model to output a tree instead of a sequence, since code inherently has a tree structure. This could be done using a tree-based LSTM structure [8, 9]. Furthermore, one can force the model to follow the language rules by remembering certain information (e.g., types of variables) during generation and sampling only from allowable options at each time step. Beam search can assist with this, allowing for "backtracking" in case the allowable options have small probability, or if none of the options are allowed.

Acknowledgements

The author would like to thank Abhijeet Shenoi, Prof. Percy Liang, and Jacob Steinhardt for insightful discussions and helpful advice.

References

[1] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[2] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy I/O. *CoRR*, abs/1703.07469, 2017.

[3] X. V. Lin, C. Wang, D. Pang, K. Vu, and M. D. Ernst. Program synthesis from natural language using recurrent neural networks. Technical report, Technical Report UW-CSE-17-03-01, University

of Washington Department of Computer Science and Engineering, Seattle, WA, USA, 2017.

[4] C. Maddison and D. Tarlow. Structured generative models of natural source code. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 649–657, 2014.

[5] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[6] I. Polosukhin and A. Skidanov. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.

[7] M. Rabinovich, M. Stern, and D. Klein. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017.

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

[9] X. Zhu, P. Sobihani, and H. Guo. Long short-term memory over recursive structures. In *International Conference on Machine Learning*, pages 1604–1612, 2015.