Towards Specification-Directed Program Repair

Richard Shin∗ & Dawn Song
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720, USA
{ricshin,dawnsong}@cs.berkeley.edu

Illia Polosukhin
NEAR
illia@near.ai

Abstract

Several recent papers have developed neural network program synthesizers by using supervised learning over large sets of randomly generated programs and specifications. In this paper, we investigate the feasibility of this approach for program repair: given a specification and a candidate program assumed similar to a correct program for the specification, synthesize a program which meets the specification. Working in the Karel domain with a dataset of synthetically generated candidates, we develop models that can make effective use of the extra information in candidate programs, achieving 40% error reduction compared to a baseline program synthesis model that only receives the specification and not a candidate program.

1 Introduction

Several recent papers have proposed neural network-based approaches to program synthesis from input/output examples (Parisotto et al., 2017; Devlin et al., 2017b; Bunel et al., 2018). In these works, the goal is to generate a program in a domain-specific language from a small number of input/output examples, that can also generalize to unseen examples. The models are trained from a large corpus of programs and a set of input/output examples for each program.

In our work, we consider a variant of the above setting which we dub specification-directed program repair, where, in addition to a set of input/output examples that specify the desired semantics, we also get a candidate program which we assume is textually similar to a correct program. In real-world applications of program synthesis, we may have access to a slightly erroneous program to use as a candidate, and making effective use of this information would allow us to achieve greater performance than possible with only a specification. For example, many programming tasks involve solving problems similar to existing ones for which we already have solutions.

We demonstrate our methods in the Karel domain, also considered by past work in program synthesis (Devlin et al., 2017a; Bunel et al., 2018). We show that candidate programs indeed contain useful information for synthesizing programs that satisfy I/O examples; our methods achieve 40% error reduction compared to a baseline program synthesis model that does not have access to a candidate program.

Papers such as DeepFix (Gupta et al., 2017), skp (Pu et al., 2016), SynFix (Bhatia & Singh, 2016; Wang et al., 2018) and Devlin et al. (2017c) also consider the task of program repair. Unlike these prior works, we focus entirely on semantic errors and do not consider syntactic ones, and consider a very large number of possible task specifications and errors rather than a handful.

2 Task and Model Description

2.1 Problem definition

In the program synthesis setting, we are given a set of input/output examples \((I_1, O_1), \ldots, (I_k, O_k)\) and our goal is to find a program \(\pi\) such that \(\pi(I_1) = O_1, \ldots, \pi(I_k) = O_k\). \(k\) is typically small;

∗Work partially performed at NEAR.
for our experiments, we use $k = 5$. In a neural program synthesis approach, we use a dataset containing $N$ programs each with $k$ input/output examples, for training a neural network to represent $p_0(\pi | (I_1, O_1), \ldots, (I_k, O_k))$ such that it will put high probability on programs which not only satisfy all $(I_i, O_i)$ but will generalize to other I/O examples.

In this work, we consider the setting of program repair. In addition to a set of input/output examples $(I_1, O_1), \ldots, (I_k, O_k)$, we are also given $\pi_{\text{cand}}$, and our goal is to find a $\pi$ such that $\pi(I_1) = O_1, \ldots, \pi(I_k) = O_k$ and also generalizes to other unseen I/O examples. We assume that there exists such a satisfactory $\pi$ where $\text{dist}(\pi, \pi_{\text{cand}}) \leq d$ for some distance measure between programs.

2.2 KAREL DOMAIN

Karel is an educational programming language [Pattis 1981] that has been used in introductory Stanford classes and also in past program synthesis and induction work (Bunel et al. 2018; Devlin et al. 2017a). In Karel, a program consists of instructions for an agent inside a grid. At each cell, the grid (which is between $2 \times 2$ and $16 \times 16$) can contain either an obstacle, or between 0 and 10 markers. The agent starts at some cell in the grid (which may contain markers but no obstacle), and has move, turnLeft, turnRight as actions to move, and pickMarker, putMarker to manipulate markers. The language contains if, ifElse, while constructs with conditionals {front, left, right}isClear, markersPresent, their negations. repeat allows for a fixed number of repetitions. There are no functions or variables.

2.3 DATA GENERATION

We used a Karel dataset made publicly available by [Devlin et al. 2017a] containing 1,116,854 training examples and 2,500 test examples. Each example is $\{\pi^{(i)}, ((I_1, O_1^{(i)}), \ldots, (I_k, O_k^{(i)}))\}$; 5 of the I/O pairs are given to the model and the remaining 1 is held out for testing.

In order to create a candidate program $\pi^{(i)}_{\text{cand}}$ for training our model, we apply random mutations to $\pi^{(i)}$, where each mutation preserves the syntactic validity of the program. Specifically, we consider the following mutations: insert/delete/replace an action; wrap or unwrap with if/elseif/while/repeat, sampling a random condition if necessary; replace the conditional/number of repetitions in if/elseif/while/repeat. We consider applying 1, 2, or 3 mutations in sequence to $\pi^{(i)}$, to create $\pi^{(i)}_{\text{cand}}$ that differ from $\pi^{(i)}$ by varying amounts.

2.4 MODELS

Given a candidate program $\pi_{\text{cand}}$ consisting of tokens $t_1, \ldots, t_n$, we first embed each token and then apply a 2-layer bidirectional LSTM to obtain an encoding $e_1, \ldots, e_n$. Following [Bunel et al. 2018],

https://msr-redmond.github.io/karel-dataset/
Table 1: Accuracies of our models. $m$ indicates the number of sequential mutations applied to generate candidate $\pi_{\text{cand}}$; columns indicate the value of $m$ at test time. “Gen.” means the output program passed all tests and “Exact” means the output program textually matched the $\pi$ in the data. The bottom row shows how often we can recover a correct program when we apply $m$ random mutations to candidate programs; for “$m = 3$, top 64”, we attempt $m = 3$ mutations 64 times and report when any of the 64 attempts succeed.

<table>
<thead>
<tr>
<th>Model type</th>
<th>Train dist.</th>
<th>$m = 1$, top 1</th>
<th>$m = 2$, top 1</th>
<th>$m = 3$, top 1</th>
<th>$m = 3$, top 64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gen.</td>
<td>Exact</td>
<td>Gen.</td>
<td>Exact</td>
<td>Gen.</td>
</tr>
<tr>
<td>Tokens</td>
<td>$m = 1$</td>
<td>79.28%</td>
<td>75.88%</td>
<td>23.92%</td>
<td>16.56%</td>
</tr>
<tr>
<td>Tokens</td>
<td>$m = 1, 2$</td>
<td>83.24%</td>
<td><strong>76.16%</strong></td>
<td>72.68%</td>
<td>63.76%</td>
</tr>
<tr>
<td>Tokens</td>
<td>$m = 1, 2, 3$</td>
<td><strong>83.32%</strong></td>
<td>73.72%</td>
<td><strong>77.28%</strong></td>
<td><strong>65.92%</strong></td>
</tr>
<tr>
<td>Edits</td>
<td>$m = 1$</td>
<td>73.92%</td>
<td>71.56%</td>
<td>15.00%</td>
<td>7.80%</td>
</tr>
<tr>
<td>Edits</td>
<td>$m = 1, 2$</td>
<td>79.40%</td>
<td>73.36%</td>
<td>68.24%</td>
<td>61.00%</td>
</tr>
<tr>
<td>Edits</td>
<td>$m = 1, 2, 3$</td>
<td>79.32%</td>
<td>71.28%</td>
<td>70.88%</td>
<td>61.40%</td>
</tr>
<tr>
<td></td>
<td>Prog. synthesis baseline</td>
<td>70.36%</td>
<td>39.04%</td>
<td>70.36%</td>
<td>39.04%</td>
</tr>
<tr>
<td>Random mutations</td>
<td>5.08%</td>
<td>2.56%</td>
<td>0.72%</td>
<td>0.24%</td>
<td>0.4%</td>
</tr>
</tbody>
</table>

Table 1 summarizes our main experimental results. To measure the value of using $\pi_{\text{cand}}$, we compare against our baseline program synthesis model (which does not have access to $\pi_{\text{cand}}$). When using $\pi_{\text{cand}}$ with $m = 1$, our best program repair model achieves 83.32% top-1 generalization accuracy compared to 70.36% of the program synthesis model; a 40% reduction in error rate. When $m = 3$ (and therefore $\pi_{\text{cand}}$ is less useful), the top-1 generalization accuracy of 69.12% is lower than the baseline’s 70.36%, but the exact match rate of 57.04% is higher than the baseline’s 39.04%; top-64 generalization accuracy of 93.96% also beats the baseline’s 85.80%.

The edit-output model performs similar to but consistently worse than the token-output model, although it has the advantage of lower computational cost. Interestingly, when testing with $m = 1$ mutations to create $\pi_{\text{cand}}$, the models trained with the exact same data distribution ($m = 1$) perform worse than those trained with a wider variety of mutations ($m = 1, 2, 3$). We also found that using the $e_i$ (embeddings of the candidate program) at each step of decoding was highly critical for the model to work; variants without (not in table) either failed to learn or achieved very similar results to the baseline program synthesis model, indicating that it is not using information from the candidate program.

For all methods, we used a beam search with beam size 64. For top 1 accuracy, we only consider the most likely output of the beam search; for top 64, we check whether any of the 64 outputs are correct.
REFERENCES


