Grammar Filtering for Syntax-Guided Synthesis

Kairo Morton,[1] **William Hallahan,**[2] **Elven Shum,**[3] **Ruzica Piskac,**[2] **Mark Santolucito**[2]
[1]George School, [2]Yale University, [3]Deerfield Academy
mortonk@georgeschool.org, {william.hallahan, ruzica.piskac, mark.santolucito}@yale.edu,
eshum20@deerfield.edu

Abstract

Programming-by-example (PBE) is a synthesis paradigm that allows users to generate functions by simply providing input-output examples. While a promising interaction paradigm, synthesis is still too slow for realtime interaction and more widespread adoption. Existing approaches to PBE synthesis have used automated reasoning tools, such as SMT solvers, as well as works applying machine learning techniques. At its core, the automated reasoning approach relies on highly domain specific knowledge of programming languages. On the other hand, the machine learning approaches utilize the fact that when working with program code, it is possible to generate arbitrarily large training datasets. In this work, we propose a system for using machine learning in tandem with automated reasoning techniques to solve Syntax Guided Synthesis (SyGuS) style PBE problems. By preprocessing SyGuS PBE problems with a neural network, we can use a data driven approach to reduce the size of the search space, then allow automated reasoning-based solvers to more quickly find a solution analytically. Our system is able to run atop existing SyGuS PBE synthesis tools, decreasing the runtime of the winner of the 2019 SyGuS Competition for the PBE Strings track by 47.65% to outperform all of the competing tools.

Introduction

The term "program synthesis" refers to automatically generating code to satisfy some specification. That specification describes *what* the code should do, without going into details about *how* it should be done. The specification could be given as a set of constraints (Manna and Waldinger 1979; Kuncak et al. 2010), it can be deduced from the program and its environment (Gvero et al. 2013; Feng et al. 2017), or it can be inferred from a large corpus (Balog et al. 2017; Santolucito et al. 2017).

One paradigm of program synthesis is called *programming by example* (Cypher et al. 1993) (PBE). In the PBE approach, a user only provides a set of pairs of input-output examples that illustrate the desired behavior of the code. From these examples, the PBE engine should then generate code that generalizes from the examples to create a program which covers the unspecified examples as well.

The idea of automated code synthesis is an area of research with a long history (cf. the Church synthesis problem (Church 1963)). However, due to the problem's undecidability and high computational complexity for decidable fragments, for almost 50 years the research in program synthesis was mainly focused on addressing theoretical questions and the size of synthesized programs was relatively small. However, the state of affairs has drastically changed in the last decade. By leveraging advances in automated reasoning and formal methods, there has been a renewed interest in software synthesis. The research in program synthesis has recently focused on developing efficient algorithms and tools, and synthesis has even been used in industrial software (Gulwani 2011). Today, machine learning plays a vital role in modern software synthesis and there are numerous tools and startups that rely on machine learning and big data to automatically generate code (cod 2019; Balog et al. 2017).

With numerous synthesis tools and formats being developed, it was difficult to empirically evaluate and compare existing synthesis tools. The Syntax Guided Synthesis (SyGuS) format language (Alur et al. 2013; Raghothaman and Udupa 2019) was introduced in an effort to standardize the specification format of program synthesis, including PBE synthesis problems. The SyGuS language specifies synthesis problems through two components - a set of constraints (eg input-output examples), and a grammar (a set of functions). The goal of a SyGuS synthesis problem is to construct a program from functions within the given grammar that satisfies the given constraints. With this standardized synthesis format and an ever expanding set of benchmarks, there is now a yearly competition of synthesis tools (Alur et al. 2019), which pushes the frontier of scalable synthesis further.

The SyGuS Competition splits synthesis problems into tracks, for example PBE Strings or PBE BitVectors, assigning a different grammar for each track - and sometimes even varying the grammar within a single track. As the grammar defines the search space in SyGuS, this allows benchmark designers to ensure problems are relatively in-scope of current tools. However, when synthesis is deployed in real-world applications, we must allow for larger grammars that account for the wide range of use-cases users require (San-

tolucito, Hallahan, and Piskac 2019). While larger grammars allow for more expressive power in the synthesis engine, it also slows down the whole synthesis process.

In our own experimentation, we found that by manually removing some parts of the grammar from the SyGuS Competition benchmarks, we can significantly improve synthesis times. Accordingly, we sought to automate this process. Removing parts of a grammar is potentially dangerous though, as we may remove the possibility of finding a solution altogether. In fact, understanding the grammar's impact on synthesis algorithms is a complex problem, connected to the concept of overfitting (Padhi et al. 2019).

In this paper, we utilize machine learning to automate an analysis of a SyGuS grammar and a set of synthesis constraints. We generate a large number of SyGuS problems, and use this data to train a neural network. Given a new SyGuS problem, the neural network predicts how likely it is for a given grammar element to be critical to synthesizing a solution to that problem. Our key insight is that, in addition to criticality, we predict how much time we expect to save by removing this grammar element. We combine these predictions to efficiently filter grammars to fit a specific synthesis problem, in order to speed up synthesis times. Even with these reduced grammars, we are still able to find solutions to the problems.

We implemented our approach in a modular tool, GRT, that can be attached to any existing SyGuS synthesis engine as a blackbox. We evaluated GRT by running it on the SyGuS Competition Benchmarks from 2019 in the PBE Strings track. We found GRT outperformed CVC4, the winner of the SyGuS Competition from 2019, reducing the overall synthesis time by 47.65%. Additionally, GRT was able to solve a benchmark for which CVC4 timed out.

In summary, the core contributions of our work are as follows:

1. A methodology to generate models that can reduce time needed to synthesize PBE SyGuS problems. In particular, our technique reduced the grammar by identifying which functions to try to eliminate to increase the efficiency of a SyGuS solver. It also learns a model to predict which functions are critical for a particular PBE problem.
2. A demonstration of the effectiveness of our methodology. We show experiments on existing SyGuS PBE Strings track that demonstrates the speed up resulting from using our filtering as a preprocessor for an existing SyGuS solver. Over the set of benchmarks, our techniques decreases the total time taken by synthesis by 47.65%.

Related

One approach to SyGuS is to directly train a neural network to satisfy the input/output examples (Andrychowicz and Kurach 2016; Devlin et al. 2017b; Graves, Wayne, and Danihelka 2014; Joulin and Mikolov 2015; Kaiser and Sutskever 2015; Chen, Liu, and Song 2017). However, such approaches struggle to generalize, especially when the number of examples is small (Devlin et al. 2017a). Some existing work (Wang et al. 2018; Bunel et al. 2018) aims to represent aspects of the syntax and semantics of a language in a neural network. In contrast to these existing approaches, which aim to outright solve SyGuS problems, our work acts as a preprocessor for a separate SyGuS solver. However, one could also explore using our work as a preprocessor for one of these existing neural network directed synthesis approaches. Other works have explored combining logic-directed and machine learning guided synthesis approaches (Nye et al. 2019). This work sought to split synthesis tasks between generating high level sketches with neural networks, and fill in the holes of the sketch with an enumerative solver. Our work could be complementary to this, by assisting in pruning of the search space needed to fill in the holes.

Like our work, DeepCoder (Balog et al. 2017) and Neural-Guided Deductive Search (NGDS) (Kalyan et al. 2018) identify pieces of a grammar that should be removed from the grammar. However, in our parlance, these works only consider *criticality*, which measures how important a part of the grammar is to completing synthesis. Unlike our work, they do not consider the time savings from removing or keeping a part of the grammar. NGDS (Kalyan et al. 2018) does note that different models could be trained for different pieces of a grammar, however, it provides no means of automating this process. Rather, the user would have to manually elect to train individual neural networks for different grammatical elements. Work by Si et al (Si et al. 2018) aims to learn an efficient solver for a SyGuS from scratch, rather than, as in our work, acting as a preprocessor for a separate solver.

Background

A SyGuS synthesis problem is a tuple (C, G) of constraints, C, and a context-free grammar, G. In our case we restrict the set of constraints to the domain of PBE, so that all constraints are in the form of pairs (i, o) of input-output examples. We write $G \setminus g$ to denote the grammar G, but without the terminal symbol g. The set of terminal symbols are the component functions that can be used in constructing a program (e.g. +, -, str.length). We also use the notation, $\pi(G)$, to denote the projection of G into its set representation, which is the set of the terminal symbols in the grammar.

The problem statement of syntax-guided synthesis (SyGuS) is; given a grammar, G, and a set of constraints C, find a program, $P \in G$, such that the program satisfies all the constraints – $\forall c \in C . P \vdash c$. For brevity, we equivalently write $P \vdash C$. If our synthesis engine is able to find such a program in t seconds or less, we write that $(G, C) \rightsquigarrow_t P$. We use the notation T_G^C to indicate the time to run $(G, C) \rightsquigarrow_t P$. If the SyGuS solver is not able to find a solution within the timeout ($T_G^C > t$), we denote this as $(G, C) \not\rightsquigarrow_t P$. We typically set a timeout on all synthesis problems of 3600 seconds, the same value of the timeout used in the SyGuS competition. We write $(G, C) \rightsquigarrow P$ and $(G, C) \not\rightsquigarrow P$ as shorthand for $(G, C) \rightsquigarrow_{3600} P$ and $(G, C) \not\rightsquigarrow_{3600} P$, respectively.

We define G as the grammar constructed from the maximal set of terminal symbols we consider for synthesis. We call a terminal, g, within a grammar, *critical* for a set of constraints, C, if $(G \setminus g, C) \not\rightsquigarrow P$. For any given set of constraints, if a solution exists with G, there is also a grammar,

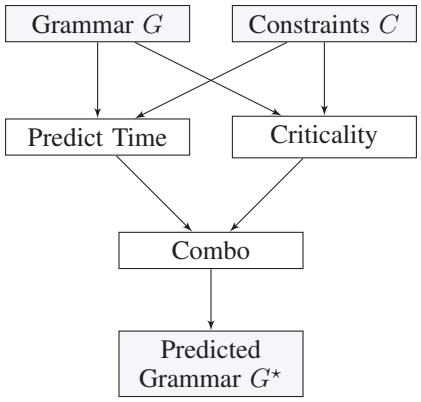

Figure 1: GRT uses the grammar G and constraints C to predict how critical each function is, and the amount of time that would be saved by eliminating it from the grammar. Then, it outputs a new grammar $G^\star$, which it expects will speed up synthesis over the original grammar (that is, it expects that $T^C_{G^\star} < T^C_G$).

G_{crit}, that contains exactly the critical terminal symbols required to find a solution. More formally, G_{crit} is constructed such that

$$(G_{crit}, C) \rightsquigarrow P \wedge \forall g \in G_{crit}.\ (G \setminus g, C) \not\rightsquigarrow P$$

Note that G_{crit} is not unique.

The goal of our work is to find a grammar, $G^\star$, where $\pi(G_{crit}) \subseteq \pi(G^\star) \subseteq \pi(G)$. This will yield a grammar that removes some noncritical terminal symbols so that the search space is smaller, but still sufficient to construct a correct program.

Overview

Our system, GRT, works as a preprocessing step for a SyGuS solver. The goal of GRT is to remove elements from the grammar and thus, by having a smaller search space, save time during synthesis. To do this we combine two metrics, as shown in Figure 1: our predicted confidence that a grammar element is not needed, and our prediction of how much time will be saved by removing that element. We focus on removing only elements where we are both confident that the grammar element is noncritical, and that removing the grammar element significantly impacts synthesis times. By giving the constraints and the grammar definition to GRT, we predict which elements of the grammar can be safely removed. By analyzing running times we predict which of these elements are benefical to remove. We describe GRT in three sections, addressing dataset generation, the training stage, and our evaluation.

Data Generation

In order to learn a model for GRT, we need to generate a labelled dataset that maps constraints to grammar components in G_{crit}. This will allow us to predict, given a new set of constraints C', which grammar elements are noncritical for synthesis, and accordingly prune our grammar. The generation of data for application to machine learning for program synthesis is a nontrivial problem, requiring careful construction of the dataset (Shin et al. 2019). We break the generation of this dataset into two stages: first, we generate a set of programs, $\mathcal{P}$ from G. Then, for each program in $\mathcal{P}$, we generate constraints for that program. We additionally need a dataset of synthesis times, in order to predict how long synthesis takes for a given set of constraints.

Criticality Data

To generate a set of programs $\mathcal{P}$, that can be generated from a grammar G, we construct a synthesis query with no constraints. We then run CVC4 with the command `--sygus-stream`, which instructs CVC4 to output as many solutions as it can find. With no constraints, all functions satisfy the specification, and CVC4 will generate all permutations of (well-formed and well-typed) functions in the grammar, until the process is terminated (we terminate after generating n programs). Because CVC4 generates solutions of increasing size, we collect all generated programs, then shuffle the order to prevent data bias with respect to the order (size) in which CVC4 generated programs.

After generating programs, we generate corresponding constraints (in the form of input-output examples for PBE) for these functions. To do this, for each program, P, we randomly generate a set of inputs I, and compute the input-output pairs $C = \{(i, P(i)) \mid i \in I\}$. We then form a SyGuS problem (G, C), where we know that the program P satisfies the constraints, and is part of the grammar: $P \vdash C$ and $P \in G$. This amounts to programs that *could* be synthesized from the constraints (i.e. $(G, C) \rightsquigarrow_\infty P$). It is important that our dataset represent programs that *could* be synthesized, as opposed to what *can* be synthesized (i.e. $(G, C) \rightsquigarrow_{3600} P$). This is important because we will use this data set to try to learn the "semantics" of constraints, and we do not want to use this data set to additionally, inadvertently learn the limitations of the synthesis engine.

At this point, we have now constructed a dataset of triples of grammars (fixed for all benchmarks), constraints, and programs, $D = \{(G, C_1, P_1) \ldots (G, C_n, P_n)\}$. In order to use D to helps us predict G_{crit}, we break up each triple by splitting each constraint set C into its individual constraints. For a triple (G, C, P), where $C = \{c_1 \ldots c_m\}$, we generate a new set of triples $\{(G, c_1, P) \ldots (G, c_m, P)\}$. The union of all these triples of individual constraints form our training set, $\mathcal{TR}_{crit}$, that will be used to predict critical functions in the grammar for a given set of constraints.

Timing Data

In addition to a training set for predicting G_{crit}, we also need a separate training set for predicting the time that can be saved by removing a terminal from the grammar. This dataset maps grammar elements $g \in G$ to the effect on synthesis times, $\mathbb{R}$, when g is dropped from the grammar. To do this we require synthesis problems that more closely model the types of constraints that humans typically write. We collect these set of benchmarks from users of the live coding interface for SyGuS (Santolucito, Hallahan, and Piskac 2019).

Because we had limited number of human-generated constraint examples, we augmented this with constraints generated from $\mathcal{TR}_{crit}$.

We run synthesis for each problem with the full grammar, as well as with all grammars constructed by removing one element, g. For every synthesis problem benchmark, $1 \leq i \leq m$, we record the difference in synthesis times between running with the full grammar, and removing g:

$$T_G^{C_i} - T_{G\setminus g}^{C_i} \tag{1}$$

Thus, we create a training set, $\mathcal{TR}_{time}$, relating each terminal $g \in \pi(G)$ and a set of constraints, to the time it takes to synthesize a solution without that terminal.

Training

Predicting criticality

Our goal is to predict, given a set of constraints C, if a terminal g belongs to the set of terminals $\pi(G_{crit})$ for C. To do this, we use a Feedforward Neural Network (Multi-Layer Perceptron), with an extra embedding layer to encode the string valued input-output examples into feature vectors. We train the neural network to predict the membership of each terminal $g \in \pi(G)$ to the critical set $\pi(G_{crit})$, based on a single constraint $c \in C$. This prediction produces a 1D binary vector of length $|\pi(G)|$, where 1 at position i in the binary vector indicates the terminal in position i is predicted to belong to the critical set.

When a SyGuS problem has multiple ($|C| \geq 2$) constraints, we run our prediction on each constraint individually. We then use a voting mechanism to come to consensus on the construction of $G^\star$. After computing $|C|$ binary vectors across all constraints, the vectors are summed to produce a final voting vector. The magnitude of each element in this final voting vector represents the number of votes "from each constraint" that the terminal represented by that element is in the critical set. We then use this final voting vector in combination with our time predictions.

Predicting time savings

It is only worthwhile to remove a terminal symbol g from a grammar G if $T_{G\setminus g}^C$ is less than T_G^C. If a g stands to only give us a small gain in synthesis times, it may not be worth the risk that we incorrectly predicted its criticality.

To predict the amount of time saved by removing a terminal g we examine the distribution of times in our training set $\mathcal{TR}_{time}$. For each terminal g, we calculate A_g, the average time increase that results from removing g from the grammar. Denoting the time to run $(G, C) \rightsquigarrow P$ as T_G^C, we can write A_g as:

$$A_g = \frac{\sum_{i=1}^{n} T_G^{C_i} - T_{G\setminus g}^{C_i}}{n}$$

If a terminal g has a negative A_g, then removing it from the grammar actually slows down synthesis, on average. As such, dropping the terminal from the grammar is not generally helpful. Thus, we only consider those terminals with a positive A_g in our second step.

Combining predictions

With our predictions of the criticality a terminal g and of time saved by removing g, we must make a final decision on whether or not we should remove g. To do this, we take the top three terminals with the greatest average positive impact on synthesis time over the training set, as computed with A_g. These tended to be terminals that mapped between types which saved more time due to the internal mechanisms and heuristics of the CVC4 solver. We then use the final voting vector from our criticality prediction to choose only two out of the three to remove from G to form $G^\star$. We chose to remove only two terminals from G in order to minimize the likelihood of generating a $G^\star$, such that $\pi(G^\star) \subseteq \pi(G_{crit})$. We conjecture that the number of terminals removed is a grammar-dependent parameter that must be selected on a per grammar basis, just as the number of terminals with $A_g > 0$ is grammar specific.

Falling back to the full grammar

There is some danger that $G^\star$ will, in fact, not be sufficient to synthesize a program. Thus, we propose a strategy that

- first, tries to synthesize a program with the grammar $G^\star$
- second, if synthesis with $G^\star$ is unsuccessful, falls back to attempting synthesis with the full grammar G.

We determine how long to wait before switching from $G^\star$ to G by finding an x that minimizes:

$$\sum_{i=1}^{n} \begin{Bmatrix} T_{G^\star}^{C_i} & T_{G^\star}^{C_i} < x \\ \min(x + T_G^{C_i}, t) & T_{G^\star}^{C_i} > x \end{Bmatrix} \tag{2}$$

where $C_1 \ldots C_n$ are the constraints from the training set, and t is the timeout for synthesis.

Ideally, as captured in the first line of the sum, $(C_i, G^\star) \rightsquigarrow_x P$ will finish before $T_{G^\star}^{C_i}$ = x. However, if a benchmark does not finish in that time, it will fall back on the full grammar. Then, either $(C_i, G^\star) \rightsquigarrow_{t-x} P$ will succeed, and synthesize the expression in total time $x + T_G^{C_i}$, or synthesis will timeout, in total time $(t - x) + x = t$.

Experiments

The SyGuS competition (Alur et al. 2017) provides public competition benchmarks and results from previous years. In particular, the PBE Strings dataset provides a collection of PBE problems over a grammar that includes string, integer, and Boolean manipulating functions. First, we describe our approach to generating a training set of PBE problems over strings. Then, we present our results running GRT against the 2019 competition's winner in the PBE Strings track, CVC4 (Nötzli et al. 2019; Barrett et al. 2011; Alur et al. 2019). We are able to reduce synthesis time by 47.65% and synthesize a new solution to a benchmark that was left unsolved by CVC4.

Technical details

The data triples generated during our initial data generation process of $\mathcal{TR}_{crit}$ are triples of strings. However, the neural network cannot process input-output pairs of type string as

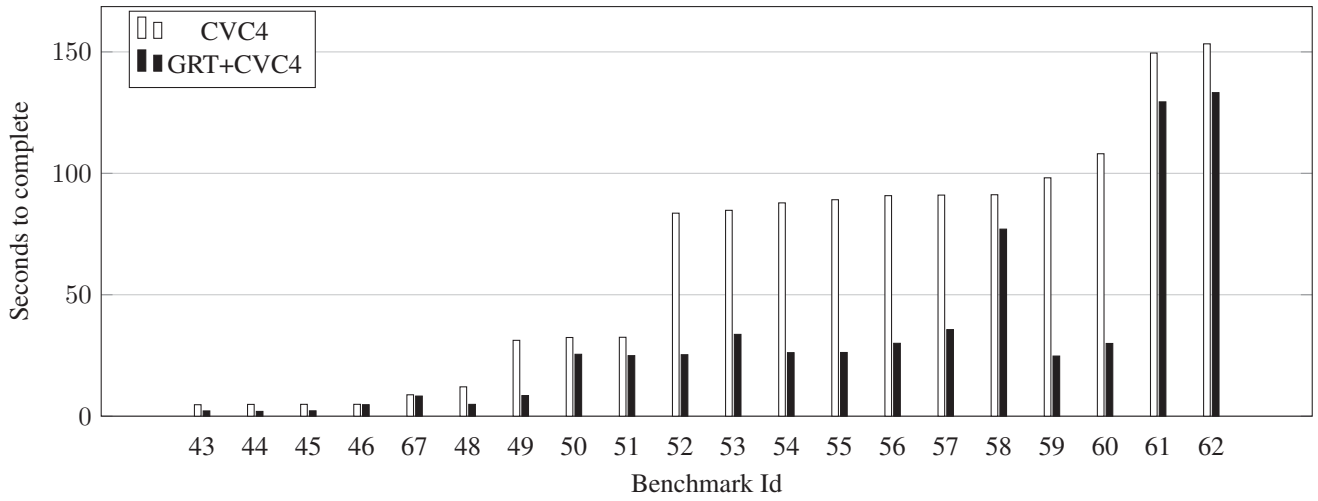

Figure 2: The top 20 problems with longest synthesis time for CVC4 (excepting timeouts), and the corresponding synthesis times for GRT+CVC4.

input. Thus, this data must be encoded numerically before it can be utilized to train the neural network. Each character in the input-output pairs is converted to its ASCII equivalent integer value. The size of each pair is then standardized by adding a padding of zeros to the end of each newly encoded input and output vector respectively. This creates two vectors: the encoded input and the encoded output, both of which have a length of 20. These two vectors are then concatenated to give us a single vector for training. By the end of this process the triples created in our first data generation step are now one vector of type $\mathbb{N}^{40}$ representing the input-output pair and a correct label P that will be predicted.

To generate the training set for predicting synthesis times, $\mathcal{TR}_{time}$, we combine human generated and automatically generated SyGuS problems. Specifically, we use 10 human generated SyGuS problems, and 20 randomly selected problems from $\mathcal{TR}_{crit}$.

The overall architecture of our model can be categorized as a multi-layer perceptron (MLP) neural network. More specifically, our model is made up of five fully connected layers: the input layer, three hidden layers, and the output layer. By using the Keras Framework, we include an embedding layer along with our input layer which enables us to create unique vector embeddings of length 100 for any given input-output pair in the dataset. This embedding layer learns the optimal weights used to create these unique vectors through the training process. Thus, we create an encoding of the input-output pairs for training, while simultaneously standardizing the scale of the vector before it reaches the first hidden layer. The hidden layers of the model are all fully connected, and all use the sigmoid activation function. In addition, we implement dropout during training to ensure that overfitting does not occur. The size of the hidden layers was calculated using a geometric series to ensure that there was a consistent decrease in layer size as the layers get closer to the output layer. Specifically, the size of each hidden layer was calculated by:

$$HL_{size}(n) = input_{size}\left(\frac{output_{size}}{input_{size}}\right)^{\frac{n}{L_{num}+1}} \quad (3)$$

where L_{num} represents the total number of layers in the network. Our model used the Adam optimization method and the binary-cross entropy loss function as it is well suited for multi-label classification. Overall, our model was trained on 124928 data points for 15 epochs with a batch size of 200 producing a training time of 228 seconds.

Results

After generating our data sets and training our model, we wrote a wrapper script to run GRT as a preprocessor for CVC4's SyGuS engine. We compared the synthesis results of GRT+CVC4 with the synthesis results of running CVC4 alone. All experiments were run on MacBook Pro with a 2.9 GhZ Intel i5 processor with 8GB of RAM. CVC4 uses a default random seed, and is deterministic over the choice of that seed, so the results of synthesis from CVC4 on a given grammar and set of constraints are deterministic. We note that our training data in no way used any of the SyGuS benchmarks.

GRT+CVC4 outperformed directly calling CVC4 on 32 out of 64 benchmarks (50%), with a reduction in total synthesis time over all benchmarks from 1304.87 seconds with CVC4 to 683.09 seconds with GRT+CVC4. On one benchmark, CVC4 timed out and was not able to find a solution (even when the timeout was increased to 5000 seconds), while GRT+CVC4 found a solution within the timeout specified by the SyGuS Competition rules (3600 seconds). On one benchmark, both CVC4 and GRT+CVC4 timeout (TO) and are not able to find a solution. On the other 31 benchmarks, CVC4 performed the same (within ± 0.1s) with and without the preprocessor. All the benchmarks for which CVC4 performed the same as GRT+CVC4 finish in under 2 seconds, and 28 of the 31 finish in under a second. In

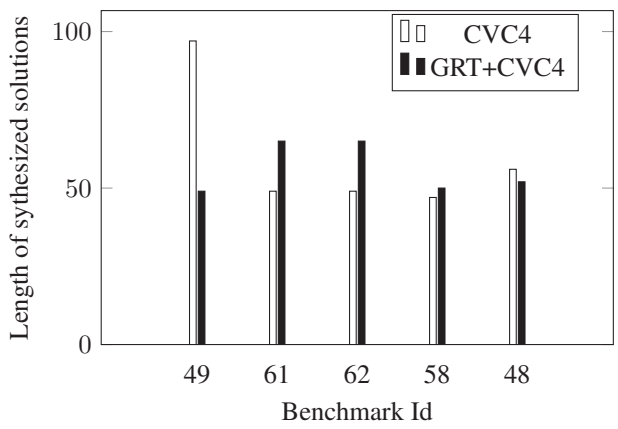

Figure 3: When the GRT+CVC4 found a different solution than CVC4, it was on average shorter than the solution found with the full grammar.

these cases there was little room for improvement even with GRT+CVC4.

Figure 4 shows the exact running times with both the full and reduced grammars from the benchmarks with the 30 largest running times with the full grammar. These are the benchmarks for which the synthesis times and size of the solution diverge most meaningfully, however all other data is available in the supplementary material for this paper. Figure 4 also shows $|P|$ and $|P^*|$, the sizes of the programs found by the CVC4 and GRT+CVC4, respectively. We define size of a program as the number of nodes in the abstract syntax tree of the program. In terms of the grammar G, this is the number of terminals (including duplicates) that were composed to create the program.

In Figure 2, we present a visual comparison of the results for the 20 functions that took CVC4 the longest, while still finishing in the 3,600 second time limit. We note that we have the largest gains on the problems for which CVC4 is the slowest. Problems that CVC4 already handles quickly stand to benefit less from our approach.

In order to get a better baseline to understand the impact of GRT on running times, we ran a version of GRT with only the criticality prediction, which we call GRTC. In this case, GRTC+CVC4 actually performed worse than CVC4 by itself, increasing the running time on 53 out of the 62 benchmarks that did not timeout on CVC4.

On all but 5 benchmarks, CVC4 synthesized the same program when running with G and $G^\star$. The sizes of the programs (in terms of the number of terminal symbols used) for the benchmarks on which CVC4 synthesized different programs are shown in Figure 3. While on some benchmarks GRT+CVC4 produced a larger solution than CVC4, as a whole the sum of the size of all solutions for CVC4 was 806, while for GRT+CVC4 it was 789. Thus, overall, we were able to outperform CVC4 on size of synthesis as well.

The SyGuS competition scores each tool using the formula: $5N + 3F + S$, where N is the number of benchmarks solved (non-timeouts), F is based on a "pseudo-logarithmic scale" (Alur et al. 2017) indicating speed of synthesis, and S is based on a "pseudo-logarithmic scale" indicating size of the synthesized solution. On all three of these measurements, GRT+CVC4 performed better than CVC4. There are number of other synthesis tracks available in the SyGuS competition, which do not involve PBE constraints. We note that our approach can selectively be applied as a preprocessing step for input in the PBE track without incurring an overhead on other synthesis tasks.

id	file	T_G^C	$T_{G^\star}^C$	$\|P\|$	$\|P^\star\|$
34	lastname-small.sl	1.80	1.84	4	4
35	bikes-long.sl	1.97	1.76	3	3
36	bikes-long-repeat.sl	2.08	1.71	3	3
37	lastname.sl	2.31	1.83	4	4
38	phone-6-short.sl	3.23	1.22	11	11
39	phone-7-short.sl	3.26	1.26	11	11
40	initials-long-repeat.sl	3.33	2.54	7	7
41	phone-5-short.sl	3.72	1.51	9	9
42	phone-7.sl	4.57	2.03	11	11
43	phone-8.sl	4.72	2.17	11	11
44	phone-6.sl	4.85	1.97	11	11
45	phone-5.sl	4.88	2.20	11	11
46	phone-9-short.sl	4.88	4.73	52	52
47	phone-10-short.sl	8.81	8.28	49	49
48	phone-9.sl	12.08	4.86	56	52
49	phone-10.sl	31.23	8.49	97	49
50	lastname-long.sl	32.40	25.49	4	4
51	lastname-long-repeat.sl	32.49	24.92	4	4
52	phone-6-long-repeat.sl	83.59	25.31	11	11
53	phone-5-long-repeat.sl	84.77	33.68	11	11
54	phone-7-long.sl	87.83	26.15	11	11
55	phone-7-long-repeat.sl	89.13	26.23	11	11
56	phone-5-long.sl	90.81	30.01	11	11
57	phone-8-long-repeat.sl	91.04	35.64	11	11
58	phone-9-long-repeat.sl	91.19	77.02	47	50
59	phone-6-long.sl	98.15	24.75	11	11
60	phone-8-long.sl	108.06	29.94	11	11
61	phone-10-long-repeat.sl	149.53	129.43	49	65
62	phone-10-long.sl	153.32	133.22	49	65
63	initials-long.sl	TO	TO	-	-
64	phone-9-long.sl	TO	3516.21	-	49

Figure 4: Synthesis results over the 30 longest running benchmarks from SyGuS Competition's PBE Strings track.

Although we implemented a strategy to manage a switch from the reduced grammar back to the full grammar, we found in practice that the optimal strategy for our system was to exclusively use the reduced grammar. Because we had conservatively pruned the grammar, we had no need to switch back to the full grammar.

Conclusions

In a way, by training on a dataset we generate from the output of the interpreter of the language, we are encoding an approximation of the semantics into our neural network. While the semantic approximation is too coarse to drive synthesis itself, we can use it to prune the search space of potential programs. By predicting terminals impact on synthesis time, we more conservatively remove only terminals likely to have

a positive impact. In conjunction with analytically driven tools, we can then significantly improve synthesis times with very little overhead.

While we have presented GRT, which demonstrates a significant gain in performance over all existing SyGuS solvers, we still have many opportunities for further improvement. In our prediction of the potential time saved by removing a terminal from the grammar, we have simply used the average expected value over all samples in the dataset. By using a neural network here, we may be able to leverage some property of the SyGuS problem constraints to have more accurate potential time savings predictions. This would allow us, possibly in combination with a more advance prediction combination strategy, to more aggressively prune the grammar. The drawback to this approach is that we may then potentially remove too much from the grammar. One of the key features of GRT is that it introduces no new timeouts, that is, it does not remove any critical parts of the grammar.

Additionally, our prediction of criticality of a terminal uses a voting mechanism to combine the prediction based on each constraint. While this worked well in practice, this strategy ignores the potential for interaction between constraints. In our preliminary exploration, we were not able to construct a model that captures this inter-constraint interaction in a useful way. This may be a path for future work. In a similar vein, there exist a number of other works that define a criticality measure for each terminal in the SyGuS grammar (Balog et al. 2017; Kalyan et al. 2018). It may be possible to leverage these in place of our criticality measure, and in combination with our time savings prediction, to achieve better results.

So far we have only explored the PBE Strings track of the SyGuS Competition. The competition also features a PBE BitVectors track where our technique may have significant gains as well. This would require a new encoding scheme, but the overall approach would remain similar. In general, extending this work to allow for other PBE types, as well as more general constraints, would broaden the potential real-world application of SyGuS.

References

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M. M.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, 1–8. IEEE.

Alur, R.; Fisman, D.; Singh, R.; and Solar-Lezama, A. 2017. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438*.

Alur, R.; Fisman, D.; Padhi, S.; Reynolds, A.; Singh, R.; and Udupa, A. 2019. The 6th competition on syntax-guided synthesis. https://sygus.org/comp/2019/results-slides.pdf. Accessed: 2019-11-20.

Andrychowicz, M., and Kurach, K. 2016. Learning efficient algorithms with hierarchical attentive memory. *arXiv preprint arXiv:1602.03218*.

Balog, M.; Gaunt, A. L.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2017. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Barrett, C.; Conway, C. L.; Deters, M.; Hadarean, L.; Jovanović, D.; King, T.; Reynolds, A.; and Tinelli, C. 2011. CVC4. In *International Conference on Computer Aided Verification*, 171–177. Springer.

Bunel, R.; Hausknecht, M.; Devlin, J.; Singh, R.; and Kohli, P. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*.

Chen, X.; Liu, C.; and Song, D. 2017. Learning neural programs to parse programs. *CoRR* abs/1706.01284.

Church, A. 1963. Application of recursive arithmetic to the problem of circuit synthesis. *Journal of Symbolic Logic* 28(4):289–290.

2019. codata - ai completions for your java ide. https://www.codota.com/. Accessed: 2019-09-03.

Cypher, A.; Halbert, D. C.; Kurlander, D.; Lieberman, H.; Maulsby, D.; Myers, B. A.; and Turransky, A., eds. 1993. *Watch What I Do: Programming by Demonstration*. Cambridge, MA, USA: MIT Press.

Devlin, J.; Bunel, R. R.; Singh, R.; Hausknecht, M.; and Kohli, P. 2017a. Neural program meta-induction. In *Advances in Neural Information Processing Systems*, 2080–2088.

Devlin, J.; Uesato, J.; Bhupatiraju, S.; Singh, R.; Mohamed, A.-r.; and Kohli, P. 2017b. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 990–998. JMLR. org.

Feng, Y.; Martins, R.; Wang, Y.; Dillig, I.; and Reps, T. W. 2017. Component-based synthesis for complex apis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, 599–612.

Graves, A.; Wayne, G.; and Danihelka, I. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*.

Gvero, T.; Kuncak, V.; Kuraj, I.; and Piskac, R. 2013. Complete completion using types and weights. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, 27–38.

Joulin, A., and Mikolov, T. 2015. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, 190–198.

Kaiser, Ł., and Sutskever, I. 2015. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*.

Kalyan, A.; Mohta, A.; Polozov, O.; Batra, D.; Jain, P.; and Gulwani, S. 2018. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*.

Kuncak, V.; Mayer, M.; Piskac, R.; and Suter, P. 2010. Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, 316–329.

Manna, Z., and Waldinger, R. 1979. A deductive approach to program synthesis. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence, IJCAI 79, Tokyo, Japan, August 20-23, 1979, 2 Volumes*, 542–551.

Nötzli, A.; Reynolds, A.; Barbosa, H.; Niemetz, A.; Preiner, M.; Barrett, C.; and Tinelli, C. 2019. Syntax-guided rewrite rule enumeration for smt solvers. *SAT*.

Nye, M.; Hewitt, L.; Tenenbaum, J.; and Solar-Lezama, A. 2019. Learning to infer program sketches. *arXiv preprint arXiv:1902.06349*.

Padhi, S.; Millstein, T. D.; Nori, A. V.; and Sharma, R. 2019. Overfitting in synthesis: Theory and practice. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, 315–334.

Raghothaman, M., and Udupa, A. 2019. Language to specify syntax-guided synthesis problems. https://sygus.org/assets/pdf/SyGuS-IF.pdf. Accessed: 2019-11-20.

Santolucito, M.; Zhai, E.; Dhodapkar, R.; Shim, A.; and Piskac, R. 2017. Synthesizing configuration file specifications with association rule learning. *PACMPL* 1(OOPSLA):64:1–64:20.

Santolucito, M.; Hallahan, W. T.; and Piskac, R. 2019. Live programming by example. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019.*

Shin, R.; Kant, N.; Gupta, K.; Bender, C.; Trabucco, B.; Singh, R.; and Song, D. 2019. Synthetic datasets for neural program synthesis. In *International Conference on Learning Representations*.

Si, X.; Yang, Y.; Dai, H.; Naik, M.; and Song, L. 2018. Learning a meta-solver for syntax-guided program synthesis.

Wang, C.; Huang, P.-S.; Polozov, A.; Brockschmidt, M.; and Singh, R. 2018. Execution-guided neural program decoding. *arXiv preprint arXiv:1807.03100*.