The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Glass-Box Program Synthesis: A Machine Learning Approach

Konstantina Christakopoulou
University of Minnesota, Twin Cities
christa@cs.umn.edu

Adam Tauman Kalai
Microsoft Research, New England
Adam.Kalai@microsoft.com

Abstract

Recently proposed models which learn to write computer programs from data use either input/output examples or rich execution traces. Instead, we argue that a novel alternative is to use a *glass-box* scoring function, given as a program itself that can be directly inspected. Glass-box optimization covers a wide range of problems, from computing the greatest common divisor of two integers, to learning-to-learn problems.

In this paper, we present an intelligent search system which learns, given the partial program and the glass-box problem, the probabilities over the space of programs. We empirically demonstrate that our informed search procedure leads to significant improvements compared to brute-force program search, both in terms of accuracy and time. For our experiments we use rich context free grammars inspired by number theory, text processing, and algebra. Our results show that (i) running our framework iteratively can considerably increase the number of problems solved, (ii) our framework can improve itself even in domain agnostic scenarios, and (iii) it can solve problems that would be otherwise too slow to solve with brute-force search.

Introduction

For computers to program computers, we must first address how programming problems will be represented and how performance will be evaluated. In the field of program synthesis, the two main approaches for specifying problems are: (a) by examples (Gulwani, Harris, and Singh 2012), in which a number of example input-output pairs (x_i, y_i) are provided as input and the goal is to output a function f satisfying $f(x_i) = y_i \ \forall i$ while possibly minimizing other criteria (e.g., being short); and (b) by specification (Manna and Waldinger 1980), in which a formal specification in some particular language is given. More generally, there is a utility (usefulness score) for any synthesized program. Assuming this utility function can be written as a program-scoring program, we propose the approach of giving the synthesis direct access to the scoring-program's source code: (c) program synthesis as optimizing the *glass-box*[1] program scoring objective that will be used to evaluate it. In this paper, we illustrate the potential of the glass-box program synthesis approach by designing a system that learns to synthesize programs that maximize the corresponding utilities of various glass-box objectives.

[1]Ironically, the term *white box* is commonly used to indicate transparency even though white boxes are not necessarily transparent. Hence, we use *glass box*.

Input x	**Output** y
BUBQJ	B
ZMFXI	Z
NEJOL	N
RVOII	I

```
def score(x: string, y: char):
    return x.count(y)
sum([score(x, f(x)) for x in tests()])
```

Figure 1: Two representations of the problem of finding the most frequent character in a string. **Top**: example input-output (x, y) pairs. **Bottom**: glass-box representation, summing scores on test strings. The score is the number of occurrences of y in x, and `tests()` randomly generates strings.

Glass-box program synthesis. To better understand the glass-box representation, consider a program synthesis contest. In programming contests among humans, problems are often described in English and scored by automated *scoring programs* based on their output on certain inputs (and other factors such as runtime and time of submission). Certainly, it is difficult for computers to understand English descriptions, so instead, we propose describing the problem to the synthesis system through the source code of the scoring program. In this "glass-box" model, there is no need for a separate problem description or examples – just the scoring program. As illustrated in Figure 1, if the problem was to find the most frequent character in a string, the score of a program outputting a character y on a string x would be the number of occurrences of y in x and the total score would be the sum of its scores over some randomly generated strings. Note that to specify the problem by input-output examples (Figure 1 top), one needs to solve the problem on several examples, and there may be ambiguities in that there may be multiple different functions f mapping x to y.

In casting the problem of programming as optimizing a

glass-box program-scoring program, one must write a program that precisely defines the utility of the synthesized program. However, this is arguably a necessary step to posing a problem in general, not only for programming contests. The term *glass-box* contrasts with black-box access. Black-box access would mean the ability to score arbitrary programs without any other access to the scoring program. Glass-box access is of course at least as powerful as black-box access because one can run the scoring program itself. A few cases where glass-box program synthesis can be applied are:

- Traditional optimization problems such as linear programming or the Traveling Salesman Problem (where the objective is the length of the tour).
- Number theory problems such as Greatest Common Divisor (GCD) or factoring. These problems can be efficiently scored because it is easy to verify factors and primality.
- Programming by example (PBE). In this case, a program P is scored by its accuracy mapping fixed inputs x_i to outputs $P(x_i) = y_i$, combined with a regularization term, e.g., $-\lambda(\text{program length})$, to prevent over-fitting.
- Optimizing an algorithm's performance in simulation. An example would be designing a network protocol to be evaluated in a network simulator. In this case, the scoring program could measure performance in a large network.
- Meta-optimization, learning to learn to learn. The problem of synthesizing program synthesis can itself be posed in the form of a meta-scorer that generates problems (i.e., scorers) from various domains, runs the candidate synthesizer on these problems, and averages the resulting program scores.

Learning to synthesize solutions by synthesizing problems. After defining the representation and evaluation of a programming problem, we demonstrate the feasibility of this approach through a system that learns to synthesize programs. Just as athletes do various exercises to improve performance at a sport, a program synthesis system may improve by practicing and learning from synthesizing solutions to various problems. Menon et al. (2013) introduced a Machine Learning (ML) approach to PBE synthesis that learns to synthesize across problems. Because their repository of real-world problems was relatively small, they selected a small number of features suited to text-processing PBE.

To get around this shortage of data, we generate our own problems which we then use to practice synthesizing solutions. In particular, once we have a set of practice problems, we iteratively find improved solutions to these problems by interleaving search and training a logistic regression-based model which guides the search intelligently, following Menon et al. (2013). Recall that in glass-box synthesis, a problem is a scoring function, so we immediately know how to score any synthesized program. A similar approach of creating artificial problems was introduced independently by Balog et al. (2016).

Experiments. We show that in practice, it is possible to synthesize solutions for a number of problems of interest, such as the GCD, that would be prohibitively slow with a naive approach. Also, we show that our Glass-Box Program Synthesis system (GlassPS), can improve itself even in a domain agnostic framework, where a union of grammars from various domains is considered; although better results can be achieved with domain-specific grammars.

Contributions. In this paper, we formalize learning to synthesize successful solutions to programming problems as a machine learning problem, using glass-box optimization. Our main contributions are three-fold:

1. We introduce glass-box program-scoring programs as a novel alternative for specifying problems.
2. We formalize a machine learning framework for glass-box optimization by synthesizing practice problems and learning patterns among the problems and the as of now discovered solutions.
3. We present experiments that demonstrate the ability of our framework to learn to generate well-formed python programs across domains.

The rest of this paper is organized as follows. After discussing related work, we introduce key concepts needed to understand our approach, and then present the details of our proposed learning to write programs framework: GlassPS. Then, we present experimental results that evaluate the performance of GlassPS in a range of problem domains.

Related Work

Learning to write computer programs has recently received a lot of attention from many viewpoints; however, it is a long-studied problem in Artificial Intelligence (Manna and Waldinger 1980; Lavrac and Dzeroski 1994). Due to space limitations, we mention a few notable related works.

In *Programming by Example* (PBE), a program synthesis system attempts to infer a program from input/output (I/O) examples, searching for a composition of some base functions. PBE has had success in various domains (Gulwani 2012), one notable example being "Flash Fill" for string manipulation in Microsoft Excel (Gulwani, Harris, and Singh 2012). In (Raza et al. 2015) the end-user can give both I/O pairs and a natural language description of the task.

Recent advances in deep learning and the augmentation of deep networks with end-to-end trainable abstractions (Graves et al. 2016), such as Neural Turing Machine (Graves et al. 2014), Hierarchical Attentive Memory (Andrychowicz and Kurach 2016) and Neural Stack (Joulin and Mikolov 2015), have given rise to the *neural programming* paradigm (Zaremba and Sutskever 2014; Neelakantan et al. 2015). Most of these works are trained using I/O examples, except for (Reed and de Freitas 2015; Cai et al. 2017) where the model (neural programmer interpreter) is trained with rich supervision execution traces, i.e., sequences of calls to the immediate subroutines conditioned on the input.

The works of (Dechter et al. 2013; Menon et al. 2013; Yessenov et al. 2013; Balog et al. 2016; Parisotto et al. 2016; Devlin et al. 2017) combine the learning-to-program approaches of machine learning and program synthesis to perform *guided search* over the space of synthesized programs.

Other successful views have been the *probabilistic programming* perspective, i.e., representing a program as a gen-

erative probabilistic model (Lake at al. 2015), and the *programming via specification* approach (Solar-Lezama 2008; Gaunt et al. 2016b), i.e., specifying a partial program capturing the high-level structure of the implementation and letting the computer synthesize the low-level details.

Relationship to Other Works. Our work, using the perspective of machine learned program synthesis, introduces the novel glass-box introspection along with contextual features to inform the search. The two closest works to ours are (Parisotto et al. 2016) and (Balog et al. 2016). The key differences to our work are: (i) they use I/O examples to condition the search, while we propose and use the glass-box problem representation, (ii) they use deep networks, while we use logistic regression. Also, while in our work and (Parisotto et al. 2016), problem-specific learned weights and a partial program representation guide the search, in (Balog et al. 2016) a separate model has to be learned per task.

Another factor differentiating the various works is the expressiveness of the Domain Specific Language used. While many learning-to-program works demonstrate program-writing for single domains like string processing, our approach can generate code in a general-purpose programming language covering various domains such as number theory, strings, root finding; which can open up interesting possibilities for general problem solving (Mikolov et al. 2015).

Similar to (Balog et al. 2016), in our approach we utilize the thus far found problem-solution pairs to inform the continuous learning of our system; hence, our work can be put in the context of lifelong learning (Gaunt et al. 2016a).

Proposed Framework

Key Concepts

To formalize the problem of learning to synthesize programs as solutions to glass-box optimization problems, we start with a discussion of high-level concepts:

Program. A program P computes a function $p : \mathcal{X} \to \mathcal{Y}$ where $\mathcal{X}$ is a set of inputs and $\mathcal{Y}$ is a set of outputs. We distinguish the program P from the function it computes p because two different programs may compute the same function. In GlassPS, program input $\mathcal{X} = \mathcal{O}$ is the set of python objects (including numbers, strings, arrays, and functions) and program output $\mathcal{Y} = \mathcal{O} \cup \{\perp\}$ is the set of objects plus the special symbol $\perp$ that indicates that the program crashed or did not produce an output in the allotted time.

Glass-box problems. Glass-box synthesis is defined over a set of *problems*. Each problem is represented by a glass-box scoring program $P \in \mathcal{P}$. A problem P computes a function $p : \mathcal{S} \to \mathbb{R}$, which measures the score of the solution S to the problem P, i.e., $p(S)$.

Synthesizer. A *synthesizer* $Z : \mathcal{P} \to \mathcal{S}$ generates a solution program based on the program with which it will be scored. Hence, the goal of a synthesizer is to attempt to find the optimal solution that maximizes the score, i.e., $Z(P) \approx \arg\max_{S \in \mathcal{S}} p(S)$.

Importantly, Z takes the scoring-program P's source code as glass-box input. Though this can be used to simulate black-box access to p by generating I/O examples, the synthesizer Z can potentially achieve higher scores than with black-box access alone.

Solutions. We use $\mathcal{S}$ to denote the set of programs output by the synthesizer, which we refer to as *solutions*. To highlight the distinction between problems and solutions, consider the "find a sorting algorithm" problem: While the solution program we are after is a fast sorting algorithm, the glass-box scoring function is a function checking for many random arrays, whether the output of the candidate solution program applied on the input array is a sorted version of that array; the scoring program does not need to be fast.

Grammars. Notice that both problems and solutions are programs. A program can be expressed as a composition of building blocks; these blocks constitute the rules of a context-free *grammar* (CFG) that allows for recursive and compositional structure. We denote the set of grammar rules for solution programs by $\mathbb{S}$ and for problem programs by $\mathbb{P}$.

The solution CFG $\mathbb{S}$ GlassPS uses, generates program trees that are converted to strings and are then evaluated by the python interpreter. It includes rules such as,

$$\text{R1: } E \to (E + E), \quad \text{R2: } E \to (\text{lambda } x\text{: } E)$$
$$\text{R3: } E \to (E).\text{lower}(), \quad \text{R4: } E \to 1, \quad \text{R5: } E \to x$$

For instance, (R1) generates code to add numbers, concatenate strings, or combine any two objects supported by the python $+$ operator. Rule (R2) creates a function of one variable, x. Rule (R3) converts a string to lower-case. The grammar supports iteration through recursion. We avoid halting issues (Skiena 1998) by bounding the total number of routine calls allowed. For simplicity, $\mathbb{S}$ does not have types and uses only one non-terminal; leaving it to the learning system to *learn* to generate programs that do not raise exceptions. Our CFG does not support directly "reaching in" to the scoring program, though such functionality can be added.

The problem CFG $\mathbb{P}$ GlassPS uses, also generates code to be evaluated by the python interpreter. $\mathbb{P}$ contains multiple non-terminals, roughly grouped by python type, so as to facilitate generating well-formed scoring programs.

Program Tree. Glass-box/Solution programs are derived from the corresponding CFG ($\mathbb{P}/\mathbb{S}$) and are represented as rooted trees in which each node is associated with a rule from the CFG. Problem/Solution trees are constructed top-down probabilistically, sampling from the rule probabilities of $\mathbb{P}$ or $\mathbb{S}$ respectively. The choice of trees for representing programs is convenient, as it is easy to extract features from trees for machine learning purposes.

Learning from (practice problems, solutions)

It is natural to try to *learn* to synthesize programs based on a collection of problem-solution pairs. To do so, one would ideally have access to a large repository of samples of problem and solution programs. Menon et al. (2013) provide a small set of problem/solution pairs for text processing PBE. Since the set is relatively small, they use domain knowledge to hand-code a small number of features for learning.

Instead, similarly to Balog et al. (2016), we synthesize practice problems of our own, and synthesize solutions to these problems. In our glass-box synthesis approach, this amounts to synthesizing scorers, i.e., the glass-box prob-

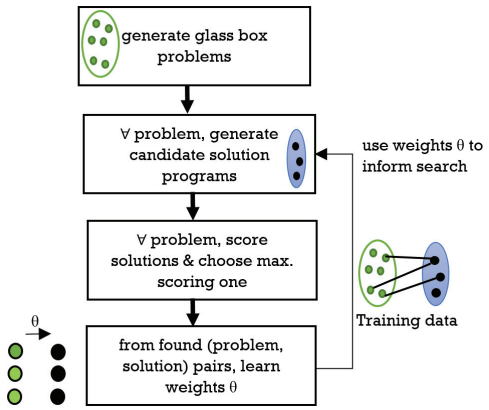

Figure 2: Framework Overview.

lems. For each such scorer, we synthesize a number of solution programs specific to that scorer and choose the highest scoring program. We then learn from this collection of problem-solution pairs to improve the model used in synthesis. We iteratively find improved solutions to these problems by interleaving search and training a model that helps guide the search intelligently. An overview of our framework is shown in Figure 2 and the specifics are described next.

Synthesizing practice problems. Since glass-box problems are programs themselves, they can be represented as program trees and they are synthesized by randomly expanding nodes from the CFG $\mathbb{P}$ (uniform probabilities), on which the set of problems can be expressed. Duplicate problems are removed, and the synthesized problems are divided into training and test sets, with a 90-10 random train/test split. The training (practice) problems are denoted $P_1, \ldots, P_m \in \mathcal{P}$ and the test problems $T_1, \ldots, T_n$.

Challenge Problems. Apart from the practice problems that we synthesize, and the set of test problems held out from this pool of synthesized problems for validation, we also use "challenge" problems. Challenge problems are ten problems that we created manually, written in terms of $\mathbb{P}$, and intended to be "representative" of programming challenges that are naturally expressed as glass-box synthesis problems. These problems are given in the experiments section.

Synthesizing solutions. For each practice/test/challenge problem, a fixed number of candidate solution program trees are synthesized top-down probabilistically using the CFG $\mathbb{S}$, and the best one, according to the scoring function of the problem at hand, is chosen.

Every rule of the CFG $\mathbb{S}$ is associated with a probability. The probability that our process will synthesize a certain program is the product of the probabilities of the rules that comprise it. Before learning, the probabilities of the rules in $\mathbb{S}$ are equal (uniformly random expansion). The purpose of using machine learning is to learn the problem-specific rule probabilities of $\mathbb{S}$ successfully, so as to make the successful solution programs more probable, and thus easier to find. In what follows, we specify how to formalize the *learning to write programs that optimize glass-box functions* as a machine learning problem.

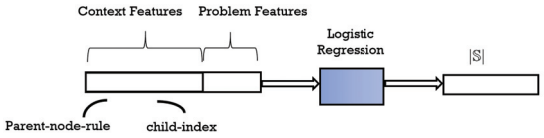

Figure 3: Mapping input features ϕ to class/rule $y \in \mathbf{R}^{|\mathbb{S}|}$.

Input Features. As input to the machine learning classifier, we use two types of features: *glass-box problem features* and *context features*. The *problem* features used for learning are a "bag-of-rules" representation of the problem. That is to say, for each problem rule in $\mathbb{P}$, there is a feature for its number of occurrences in the problem's program tree. By *context* we refer to the partial candidate program created so far. Specifically, since synthesis is top-down, we mean the path of rules from the root of the generated tree to the current node whose rule is to be assigned. For simplicity, to represent the context, we only use the one-hot-encoding of the parent node's rule (i.e., a sparse vector with 1 in the entry corresponding to the parent node's rule, and 0 elsewhere) and the one-hot-encoding of what the current node's child index is (i.e., is it the first/second/etc. child of the parent node). Thus, the one-hot-encoding vectors of the glass-box problem, the parent-node-rule and the child-index are concatenated, and comprise the input ϕ to the learner.[2]

Learning rule probabilities of $\mathbb{S}$. For every node of a candidate solution program tree, we want to predict using the derived features ϕ described above, which rule of $\mathbb{S}$ is the most probable. Thus, the problem of program inference, i.e., searching over the space of programs expressed by $\mathbb{S}$, can be reduced to a multi-label classification problem, as a solution program S is a collection of rules that are *simultaneously* present. This can be further reduced to multi-class classification, predicting for each class-rule separately whether it should be present or not. The number of classes equals $|\mathbb{S}|$.

We represent the target label of the correct rule in the successful solution program S_P^* for glass-box problem P as a one-hot-encoding vector $y \in \mathbf{R}^{|\mathbb{S}|}$ with 1 in the the next node rule present in S_P^*, and 0 in the other $|\mathbb{S}| - 1$ entries.

Given the general formulation, any multi-class classifier is applicable. In this work, for the purposes of a proof-of-concept illustration, we use a multi-class logistic regression model with parameters Θ; in particular, we learn $|\mathbb{S}|$ such parameter vectors, one for each class. An illustration of the mapping of features to solution rule is shown in Figure 3.

Importantly, the parameters Θ are learned based on a constantly updated dataset consisting of the thus far found successful (problem, solution) program pairs. Given the featurization process described, each found successful solution program with e.g. r nodes contributes r samples for the learning of the classifier; for each of these samples the input features will differ as the context will change.

Scoring Solution Programs. In order to include a suc-

[2]Although the "bag-of-words" representation may seem simplistic, preliminary experiments with richer representations did not show major improvements to outweigh the computational costs.

cessful problem-solution pair in the constantly updated training dataset, a notion of success needs to be specified. Recall that a solution program is successful if it maximizes the score of the corresponding problem. Although our framework is entirely capable of handling arbitrary continuous scoring functions, the nature of the problems induced by the grammars used in our experiments is such that all scoring programs return scores in $\{0, 1\}$ except for a solution that throws an exception, in which case the score is -1. This makes it easier to evaluate our system as the average score indicates the fraction of problems optimally solved. With iterations of learning, given that Θ are used to inform the problem specific search over $\mathbb{S}$, the poor-scoring programs become less likely.

Learning to write typed programs. While it would be nice if GlassPS generated a subset of python programs, in fact it initially generates many nonsensical (un-typable) programs because it uses only a single terminal E. Thus, unsurprisingly, at first the grammar mostly generates programs that raise exceptions by doing things like trying to lowercase the number 1. As a result, our learning system, utilizing the thus far found pairs of (glass-box problems, solutions), will progressively learn to compose well-formed programs, in addition to learning to solve problems.

Algorithmic Procedure

To summarize our framework, a formal description of the procedure followed is given in Algorithm 1. Our system operates on iterations $j = 1, \ldots, T$. At j round, GlassPS calls the SOLVE module, in order to attempt to solve the $T_1, \ldots, T_n$ test problems, which are synthesized based on $\mathbb{P}$. In SOLVE, the system generates its own practice problems $P_1, \ldots, P_m$ from the CFG $\mathbb{P}$, using uniform probabilities. To solve these generated practice problems, the system calls the TRAIN module. The goal of TRAIN is to a) construct a training dataset of problems - solutions, so that b) the parameters Θ of the model are learned. To achieve a), the system uses the parameters of the previous round's logistic regression model, to guide how the program trees of the solution programs will be built (SEARCH). Specifically, for every node of the under construction solution program tree, features are extracted via the module FEATURIZE (discussed in the previous section), which creates the bag-of-words features ϕ for the problem and context. Given these features, the current learner's model is used to predict, what the probability of every rule of the solution CFG $\mathbb{S}$ is (inside the module LR-predict). These inferred probabilities are used as per rule weights to perform weighted sampling for which rule should be next in the candidate program tree. This is how the learning guides the search over programs.

Via this procedure, for every problem, a set of candidate solution programs is constructed. Each candidate program S_i is scored by the corresponding scoring program $P(S_i)$. The programs which successfully solve the respective glass-box problems, i.e., with the maximum score using shortest length to break the ties, are used to construct a (problem, context) $\rightarrow$ solution rule training data set, calling again the FEATURIZE routine for the ϕ representation of (problem, context). This constructed training dataset is then used to learn a new logistic regression model, which will be used to find solutions for the test and challenge problems, and which will be subsequently used in the next $j+1$ learning round to guide the search over the candidate solution programs.

Algorithm 1 Learning algorithm for glass-box synthesis

procedure SOLVE($m, T_1, \ldots, T_n$) ▷ learn & solve
  for $i = 1$ to m **do**:
    $P_i \leftarrow$ GENRANDOMPRACTICE
  $\theta \leftarrow$ TRAIN($P_1, P_2, \ldots, P_m$)
  for $i = 1$ to n **do**:
    $S_i \leftarrow$ SEARCH(T_i, θ)
  return $S_1, S_2, \ldots, S_N$

procedure TRAIN($P_1, \ldots, P_m$) ▷ fit θ
  $\theta \leftarrow \theta_0$
  for $j = 1$ to num-epochs **do**
    for $i = 1$ to m **do**
      $S_i \leftarrow$ SEARCH(P_i, θ)
    Featurize and train LR on all nodes in $\langle (P_i, S_i) \rangle_{i=1}^{m}$
    Update θ
  return θ

procedure SEARCH(P, θ) ▷ solve a problem
  for $i = 1$ to num-candidates **do**
    $S_i \leftarrow$ NODE(P, "root", θ)
  return S_i with greatest score $P(S_i)$

procedure NODE(P, c, θ) ▷ node from prblm, context
  $\phi \leftarrow$ FEATURIZE(P, c)
  $i \leftarrow$ weighted-sample(LR-predict$_\theta(\phi)$) ▷ rule r_i
  children = []
  for $j = 1$ to number-of-rule-children(r_i) **do**
    children.append (NODE($P, (r_i, j), \theta$))
  return new node with rule r_i and children

Illustrative Example: how guided search works

Before we move to our experimental results, to better understand how our learning framework works, let us demonstrate how we could successfully synthesize the solution program of computing the greatest common divisor (GCD) of two positive integers m, n using our framework GlassPS.

The particular glass-box scoring function for GCD is $\sigma(m, n, y) = \begin{cases} 0 & \text{if } m \bmod y \neq 0 \\ 0 & n \bmod y \neq 0 \\ y & \text{otherwise} \end{cases}$. We can see that this function achieves maximum score y, which happens when y is a factor of both m and n. In python, this program can be written succinctly as `lambda m, n, y: ntprog(-mod(m, y) or (-mod(n, y) or y))`. The function `ntprog` simply evaluates the synthesized program over a domain of m, n pairs. It also implements early stopping optimality criteria, which improves efficiency without changing the behavior of the algorithm.

The successful solution program that we are after S^* is `def f(m, n): f(mod(n, m), m) if n else`

Toy Glass-box Problem CFG $\mathbb{P}$	
W1: Loss $\rightarrow$ ntprog(Obj)	W6: $NT \rightarrow y$
W2: $Obj \rightarrow$ lambda $m, n, y : NT$	W7: $NT \rightarrow n$
W3: $NT \rightarrow (NT$ or $NT)$	W8: $NT \rightarrow 0$
W4: $NT \rightarrow$ mod(NT, NT)	W9: $NT \rightarrow -NT$
W5: $NT \rightarrow m$	W10: $NT \rightarrow NT < 0$
Toy Solution Program CFG $\mathbb{S}$	
R1: $E \rightarrow$rec(lambda $m, n : E$)	R9: $E \rightarrow 0$
R2: $E \rightarrow$callrec(E, E)	R10: $E \rightarrow 1$
R3: $E \rightarrow E$ if E else E	R11: $E \rightarrow \log(E)$
R4: $E \rightarrow$mod(E, E)	R12: $E \rightarrow$ arctan(E)
R5: $E \rightarrow m$	R13: $E \rightarrow E.upper()$
R6: $E \rightarrow n$	R14: $E \rightarrow$[E for i in E]
R7: $E \rightarrow E + E$	R15: $E \rightarrow set(E)$
R8: $E \rightarrow abs(E)$	R16: $E \rightarrow (E, E)$

Table 1: Toy grammars for illustration purposes. `callrec` refers to the call of a function recursively, `rec` is the outer function to be called recursively, `ntprog` is a function for early-stopping based on evaluations of the synthesized program in various argument values. $\mathbb{P}$ contains many terminals ($Loss, Obj, NT$), while $\mathbb{S}$ contains a single terminal E.

`m) )` , i.e., a recursive form of Euclid's GCD algorithm which computes recursively the GCD of the remainder and m if $n \neq 0$, else it returns m.

For illustration purposes, let us consider the toy CFGs shown in Table 1, which are actually subsets of the grammars used in our experiments. They contain rules typically useful for number theory problems.

Calling the FEATURIZE module of Algorithm 1, the glass-box problem of GCD is written as a one-hot-encoding vector $\mathbf{x}_{\text{prob}}^{\text{GCD}} \in \mathbf{R}^{|\mathbb{P}|}$, with non-zero values $W1 : 1, W2 : 1, W3 : 2, W4 : 2, W5 : 1, W6 : 3, W7 : 1, W9 : 2$, where the values are the occurrences of the respective rule.

Recall that to decide which rule will become the next program tree node, a learned Logistic Regression (LR) model based on the so far found pairs of problems-solutions will be used in *inference* mode, to predict given the problem & context features ϕ, which target class y (i.e., rule of the solution CFG) is the most likely. Thus, to construct the successful finding-GCD program tree S^* top-down, guided search will proceed by *sampling rules* from $\mathbb{S}$ for the next node using the predicted LR probabilities:

1. For the root node: the features ϕ will be $\mathbf{x}_{\text{prob}}^{\text{GCD}}$ concatenated with an all-zero vector, as the root does not have a parent. The target *sampled* class y, i.e., the rule with the maximum sampled predicted probability from LR, should be $R1$ (one-hot-encoding with 1 in the entry of $R1$).
2. For the second node, ϕ will be $[\mathbf{x}_{\text{prob}}^{\text{GCD}}, \mathbf{x}_{\text{parent}}, \mathbf{x}_{\text{child}}]$, where $\mathbf{x}_{\text{prob}}^{\text{GCD}}$ is the same as for the previous node, $\mathbf{x}_{\text{parent}} \in \mathbf{R}^{|\mathbb{S}|}$ is the one-hot-encoding vector with 1 in $R1$ (as $R1$ is the rule of the parent node), and $\mathbf{x}_{\text{child}}$ is the one-hot vector with 1 in the first entry, as this is the first child. The sampled target class from the predicted probability vector should be $R3$ (the if rule).
3. Continue for each program tree node, until no more nodes are to be expanded, i.e., they are leaf nodes, and the entire S^* program tree is constructed.

In short, the LR model should learn that the rules $R1, R2, R3, R4, R5, R6$ should have high probability under the $\mathbf{x}_{\text{prob}}^{\text{GCD}}$ and the certain parent and child index contexts. When finding S^* corresponding to the glass-box GCD P_{GCD}, the (P_{GCD}, S^*) pair is added to the training dataset used to learn next round's LR.

Experiments

Setup. We use as an evaluation metric the fraction of test problems successfully solved. The sets of practice/train and test problems do not change throughout the experiment. Programs were limited to be at most size 20 nodes. For generating solutions, instead of generating programs independently at random (which results in a vast majority of duplicates), we generate the programs that are the most probable according to the learned parameters, using the search algorithm of Menon et al. 2013, which does not produce duplicates. Since problem generation is not a bottleneck, problems are generated more simply by sampling uniformly from the grammar $\mathbb{P}$ and then removing the duplicates.

Domains. We consider CFGs from the following domains:

- *Number Theory.* Example target problems include GCD or finding the largest non-trivial factor of a (small) number, using a brute force approach. Numerous trivial functions also arise in the practice set, such as given two numbers, output the smaller of the two.
- *Finding Roots.* Example target problems are finding the root of algebra expressions such as $\log(y/2) - x^2 = 0$, i.e., solve for y as a function of x.
- *Summation Formulas.* Example target problems are finding the closed-form expression of computing the sum of a function of the first n numbers $f(n) = \sum_{i=1}^{n} g(i)$. For example $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$.
- *Strings*. For simplicity, we considered problems where the desired output is a single character from the input meeting a certain objective. An example problem is finding the most frequent character in a string or finding the alphabetically first character in a string.

Verification. For each of these problems, the performance is evaluated on a bounded domain X of possible inputs and range Y of possible outputs, which makes it easy to check for optimality, and enables early stopping for efficiency. Although we report results for binary program-scoring functions for efficiency and ease of understanding, we observed qualitatively similar results when using real-valued scores.

Grammars. Briefly, our grammars, both for our typed program-scoring grammar $\mathbb{P}$ and of our solution grammar $\mathbb{S}$, include operators such as $+, -, *, /$, `mod`, math functions such as `pow`, `abs`, `tan`, `tanh`, `arctanh`, `log`, `exp`, constants such as 0, 1, 2, or the passed arguments, operators such as $<$, `max`, boolean operators such as `or`, `if` expressions, string functions such as `startswith`, `endwith`, `count`, `upper`, `lower`, `ord`. Additionally, the grammar $\mathbb{S}$ contains recursive functions (with a single, or two arguments passed), operators on lists, sets and tuples.

Each generated program was converted to a string and

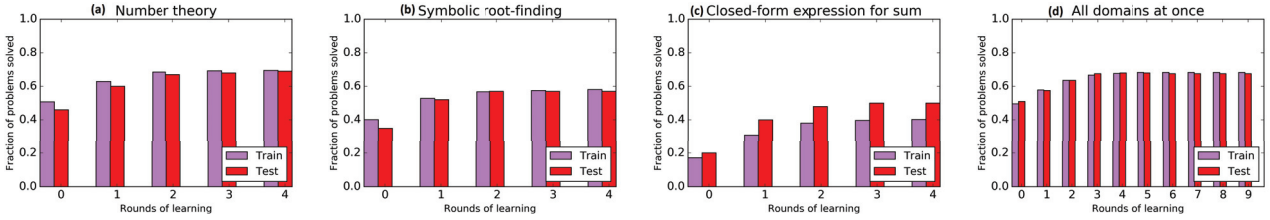

Figure 4: (a-c) Domain-specific experiment: Fraction of problems solved with multiple rounds of learning in each domain (every training round was run for 45 minutes). (d) All-domain experiment: Similar results are achieved with 90 minutes (instead of 45) for each learning round, and additional rounds beyond 5 do not seem to improve test accuracy.

Challenge problem number	1	2	3	4	5
Solution time without learning	>30:00	>2:15	>2:15	0:05	2:15
Solution time with all-domain learning	>2:15	>2:15	0:01	0:01	0:01
Solution time with domain-specific learning	0:10	>2:15	0:01	0:01	0:01
Challenge problem number	6	7	8	9	10
Solution time without learning	>2:15	>2:15	>2:15	>2:15	>2:15
Solution time with all-domain learning	0:03	>2:15	0:01	>2:15	0:01
Solution time with domain-specific learning	0:01	0:05	0:01	>2:15	0:01

Table 2: Time in hours and minutes for solving each of the 10 challenge problems with and without learning. Most problems were run for a max of 2:15, but GCD was run for 30 hours without learning.

passed to the native python eval. A timeout of 1 second was used per evaluation.

Results on Practice/ Test Problems

To empirically evaluate our proposed framework, we conducted experiments under two experimental conditions. In the first condition, we considered for the context-free grammars, only the one relevant to the problems we were trying to solve. For example, for solving string problems, we considered the string CFG, and so on. In the second condition we took the union of all grammar rules from our four different domains, and considered 250 problems from each of the four domains. We refer to this as the *all-domain experiment*.

For the first condition, we considered 1,000 practice train problems and 100 test problems. We progressively performed four training rounds, as more rounds did not seem to improve learning. Each round of training was run for 45 minutes. We show in Figure 4(a-c) the fraction of practice train/test problems solved by GlassPS as rounds of learning progress from 0 to 4. We can see that performing 4 rounds of our framework solves about (a) 70% of the target problems for the number theory domain, (b) $\sim$ 60% for the root-finding domain, and (c) 50% for the summation formula domain. The string domain is not shown because 100% of the problems were solved in each round, even prior to learning.

For the second setup of the all-domain experiment, we show the results in Figure 4(d), which includes the string domain making its results higher than Figure 4(a-c). In order to achieve similar fraction of train/test problems solved as in the first setup, every training round was run for double time, i.e., 90 mins instead of 45. We conclude that although domain-specific learning is better (as it needs half the time for the same results), the all-domain experiment is a proof-of-concept experiment that even without knowing the domain, GlassPS can improve itself over time.

Results on Challenge problems

Finally, we consider the performance of our framework in our following ten challenge problems.

1. GCD (number theory),
2. Greatest non-trivial factor of a number n.[3] (number theory),
3. Most frequent character in a string (strings),
4. Alphabetically first character in a string (strings),
5. Solve for y: $\log(y) - (x * x)/2 = 0$ (roots),
6. Solve for y: $y + (\text{pow}(x, (2 * 2))/2) = 0$ (roots),
7. Find a closed-form expression for: sum(1, n, lambda i: (i * i)) (sums),
8. Find a closed-form expression for: sum(1, n, lambda i: (i *(i * i))) (sums),
9. Find a closed-form expression for: sum(1, n, lambda i: pow(2, (-i))) (sums),
10. Find a closed-form expression for: sum(1, n, lambda i: 1 / (1, (i * (1 + i)))) (sums).

In Table 2, we report the time in hours and minutes needed to solve each of the 10 challenge problems with our system. We can compare the times needed for (i) when the search is not guided by learning, i.e., brute-force (2nd row), (ii) the all-domain setup (3rd row), and (iii) domain-specific learning (4th row), which is the best performing across problems.

[3]The time limits were such that a brute-force loop could find the largest factor in the alloted time, so no advanced factoring algorithms were necessary.

Notably, our framework can solve problems that would be otherwise prohibitively slow without learning. For instance, GCD was found after 30 hours with no learning, while it took only 10 minutes using domain-specific learning, and 2 hours and 15 minutes for the all-domain experiment.

Discussion & Conclusions

Overview. We introduced a general framework for learning to write computer programs that maximize the score of glass-box objective functions. We have formulated this as a machine learning problem, showing as a proof-of-concept that a simple classifier such as logistic regression, can successfully learn patterns among the features of the scoring programs, and the features of the generated solution programs to be scored. We have shown experimentally that our framework learns over time to generate python typed code that solves problems across domains, even though we did not provide types and we did not give access to the domain type each problem comes from.

Conceptual Comparison with Existing Systems. This paper aims to provide glass-box representation as an alternative *tool* to guide the program generation, rather than competing with existing systems; as the various approaches best fit different types of problems and a different audience. Particularly, glass-box representations are better for capturing problems we normally think of as "algorithm design", e.g., as found in an algorithms book, while examples are better for representing tedious/repetitive tasks such as copying files or selecting last names from a list of names. The latter are often simpler and hence the example-based approaches are likely to have more immediate utility to lay-people, whereas the main utility of our approach would be to help design short clever programs for simple algorithms problems.

Future Directions. While the learning could certainly be improved (e.g. increasing the learning capacity, enriching the grammars, considering more and larger programs etc.), our approach is shown to be sound and to some extent flexible enough to combine multiple domains of synthesis in a single system. This opens up interesting directions for multi-domain learning systems that learn to solve problems. Exciting future research problems include *learning* good distributions over the glass-box scoring programs, and specifying meta-learning in our framework.

Acknowledgements

Most of this work was done during an internship at MSR. The authors thank Huseyn Melih Elibol for helpful discussions, and the anonymous reviewers for detailed feedback.

References

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. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

Cai, J.; Shin, R.; and Song, D. 2017. Making neural programming architectures generalize via recursion. *arXiv preprint arXiv:1704.06611*.

Dechter, E.; Malmaud, J.; Adams, R. P.; and Tenenbaum, J. B. 2013. Bootstrap learning via modular concept discovery. In *IJCAI*, 1302–1309.

Devlin, J.; Uesato, J.; Bhupatiraju, S.; Singh, R.; Mohamed, A.-r.; and Kohli, P. 2017. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*.

Gaunt, A. L.; Brockschmidt, M.; Kushman, N.; and Tarlow, D. 2016a. Lifelong perceptual programming by example. *arXiv preprint arXiv:1611.02109*.

Gaunt, A. L.; Brockschmidt, M.; Singh, R.; Kushman, N.; Kohli, P.; Taylor, J.; and Tarlow, D. 2016b. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*.

Graves, A.; Wayne, G.; Reynolds, M.; Harley, T.; Danihelka, I.; Grabska-Barwińska, A.; Colmenarejo, S. G.; Grefenstette, E.; Ramalho, T.; Agapiou, J.; et al. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538(7626):471–476.

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

Gulwani, S.; Harris, W. R.; and Singh, R. 2012. Spreadsheet data manipulation using examples. *Communications of the ACM* 55(8):97–105.

Gulwani, S. 2012. Synthesis from examples: Interaction models and algorithms. In *SYNASC*, 8–14.

Joulin, A., and Mikolov, T. 2015. Inferring algorithmic patterns with stack-augmented recurrent nets. In *NIPS*, 190–198.

Lake, B. M.; Salakhutdinov, R.; and Tenenbaum, J. B. 2015. Human-level concept learning through probabilistic program induction. *Science* 350(6266):1332–1338.

Lavrac, N., and Dzeroski, S. 1994. Inductive logic programming. In *WLP*, 146–160. Springer.

Manna, Z., and Waldinger, R. 1980. A deductive approach to program synthesis. *TOPLAS* 2(1):90–121.

Menon, A. K.; Tamuz, O.; Gulwani, S.; Lampson, B. W.; and Kalai, A. 2013. A machine learning framework for programming by example. In *ICML*, 187–195.

Mikolov, T.; Joulin, A.; and Baroni, M. 2015. A roadmap towards machine intelligence. *arXiv preprint arXiv:1511.08130*.

Neelakantan, A.; Le, Q. V.; and Sutskever, I. 2015. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*.

Parisotto, E.; Mohamed, A.-r.; Singh, R.; Li, L.; Zhou, D.; and Kohli, P. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*.

Raza, M.; Gulwani, S.; and Milic-Frayling, N. 2015. Compositional program synthesis from natural language and examples. In *IJCAI*, 792–800.

Reed, S., and de Freitas, N. 2015. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*.

Skiena, S. S. 1998. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media.

Solar-Lezama, A. 2008. *Program synthesis by sketching*. Ph.D. Dissertation, EECS Department, University of California, Berkeley.

Yessenov, K.; Tulsiani, S.; Menon, A.; Miller, R. C.; Gulwani, S.; Lampson, B.; and Kalai, A. 2013. A colorful approach to text processing by example. In *UIST*, 495–504.

Zaremba, W., and Sutskever, I. 2014. Learning to execute. *arXiv preprint arXiv:1410.4615*.