Program Synthesis Using Deduction-Guided Reinforcement Learning

Yanju Chen[1(✉)], Chenglong Wang[2], Osbert Bastani[3], Isil Dillig[4], and Yu Feng[1(✉)]

[1] University of California, Santa Barbara, Santa Barbara, CA 93106, USA
{yanju,yufeng}@cs.ucsb.edu
[2] University of Washington, Seattle, WA 98115, USA
clwang@cs.washington.edu
[3] University of Pennsylvania, Philadelphia, PA 19104, USA
obastani@seas.upenn.edu
[4] The University of Texas at Austin, Austin, TX 78712, USA
isil@cs.utexas.edu

Abstract. In this paper, we present a new program synthesis algorithm based on reinforcement learning. Given an initial policy (i.e. statistical model) trained off-line, our method uses this policy to guide its search and gradually improves it by leveraging feedback obtained from a deductive reasoning engine. Specifically, we formulate program synthesis as a reinforcement learning problem and propose a new variant of the *policy gradient* algorithm that can incorporate feedback from a deduction engine into the underlying statistical model. The benefit of this approach is two-fold: First, it combines the power of deductive and statistical reasoning in a unified framework. Second, it leverages deduction not only to *prune* the search space but also to *guide* search. We have implemented the proposed approach in a tool called Concord and experimentally evaluate it on synthesis tasks studied in prior work. Our comparison against several baselines and two existing synthesis tools shows the advantages of our proposed approach. In particular, Concord solves 15% more benchmarks compared to Neo, a state-of-the-art synthesis tool, while improving synthesis time by 8.71× on benchmarks that can be solved by both tools.

1 Introduction

Due to its potential to significantly improve both programmer productivity and software correctness, *automated program synthesis* has gained enormous popularity over the last decade. Given a high-level specification of user intent, most modern synthesizers perform some form of backtracking search in order to find a

This work was sponsored by the National Science Foundation under agreement number of 1908494, 1811865 and 1910769.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 587–610, 2020.
https://doi.org/10.1007/978-3-030-53291-8_30

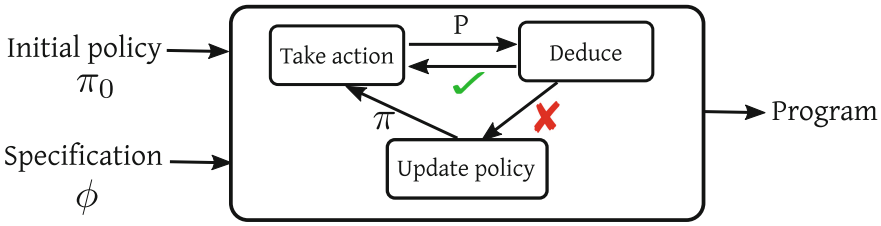

Fig. 1. Overview of our synthesis algorithm

program that satisfies the specification. However, due to the enormous size of the search space, synthesizers additionally use at least one of two other techniques, namely deduction and statistical reasoning, to make this approach practical. For example, many recent synthesis techniques use lightweight program analysis or logical reasoning to significantly prune the search space [18,19,39,53]. On the other hand, several recent approaches utilize a statistical model (trained off-line) to bias the search towards programs that are more likely to satisfy the specification [2,4,7,19]. While both deductive and statistical reasoning have been shown to dramatically improve synthesis efficiency, a key limitation of existing approaches is that they do not tightly combine these two modes of reasoning. In particular, although logical reasoning often provides very useful feedback at synthesis time, existing synthesis algorithms do not leverage such feedback to improve their statistical model.

In this paper, we propose a new synthesis algorithm that meaningfully combines deductive and statistical reasoning. Similar to prior techniques, our approach starts with a statistical model (henceforth called a *policy*) that is trained off-line on a representative set of training problems and uses this policy to guide search. However, unlike prior techniques, our method *updates* this policy on-line at synthesis time and gradually improves the policy by incorporating feedback from a deduction engine.

To achieve this tight coupling between deductive and statistical reasoning, we formulate syntax-guided synthesis as a reinforcement learning (RL) problem. Specifically, given a context-free grammar for the underlying DSL, we think of partial (i.e., incomplete) programs in this DSL as states in a Markov Decision Process (MDP) and actions as grammar productions. Thus, a *policy* of this MDP specifies how a partial program should be extended to obtain a more specific program. Then, the goal of our reinforcement learning problem is to improve this policy over time as some partial programs are proven infeasible by an underlying deduction engine.

While the framework of reinforcement learning is a good fit for our problem, standard RL algorithms (e.g., policy gradient) typically update the policy based on feedback received from states that have *already* been explored. However, in the context of program synthesis, deductive reasoning can also provide feedback about states that have *not* been explored. For example, given a partial program that is infeasible, one can analyze the root cause of failure to infer *other* infeasible

programs [18,54]. To deal with this difficulty, we propose an *off-policy* reinforcement learning algorithm that can improve the policy based on such additional feedback from the deduction engine.

As shown schematically in Fig. 1, our synthesis algorithm consists of three conceptual elements, indicated as "Take action", "Deduce", and "Update policy". Given the current policy π and partial program P, "Take action" uses π to expand P into a more complete program P'. Then, "Deduce" employs existing deductive reasoning techniques (e.g., [18,32]) to check whether P' is feasible with respect to the specification. If this is not the case, "Update policy" uses the feedback provided by the deduction engine to improve π. Specifically, the policy is updated using an off-policy variant of the *policy gradient* algorithm, where the gradient computation is adapted to our unique setting.

We have implemented the proposed method in a new synthesis tool called Concord and empirically evaluate it on synthesis tasks used in prior work [2,18]. We also compare our method with several relevant baselines as well as two existing synthesis tools. Notably, our evaluation shows that Concord can solve 15% more benchmarks compared to Neo (a state-of-the-art synthesis tool), while being 8.71× faster on benchmarks that can be solved by both tools. Furthermore, our ablation study demonstrates the empirical benefits of our proposed reinforcement learning algorithm.

To summarize, this paper makes the following key contributions:

- We propose a new synthesis algorithm based on reinforcement learning that tightly couples statistical and deductive reasoning.
- We describe an off-policy reinforcement learning technique that uses the output of the deduction engine to gradually improve its policy.
- We implement our approach in a tool called Concord and empirically demonstrate its benefits compared to other state-of-the-art tools as well as ablations of our own system.

The rest of this paper is structured as follows. First, we provide some background on reinforcement learning and MDPs (Sect. 2) and introduce our problem formulation in Sect. 3. After formulating the synthesis problem as an MDP in Sect. 4, we then present our synthesis algorithm in Sect. 5. Sections 6 and 7 describe our implementation and evaluation respectively. Finally, we discuss related work and future research directions in Sect. 8 and 9.

2 Background on Reinforcement Learning

At a high level, the goal of reinforcement learning (RL) is to train an agent, such as a robot, to make a sequence of decisions (e.g., move up/down/left/right) in order to accomplish a task. All relevant information about the environment and the task is specified as a *Markov decision process (MDP)*. Given an MDP, the goal is to compute a policy that specifies how the agent should act in each state to maximize their chances of accomplishing the task.

In the remainder of this section, we provide background on MDPs and describe the policy gradient algorithm that our method will build upon.

Markov Decision Process. We formalize a *Markov decision process (MDP)* as a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{S}_I, \mathcal{S}_T, \mathcal{A}, \mathcal{F}, \mathcal{R})$, where:

- $\mathcal{S}$ is a set of *states* (e.g., the robot's current position),
- $\mathcal{S}_I$ is the initial state distribution,
- $\mathcal{S}_T$ is a set of the final states (e.g., a dead end),
- $\mathcal{A}$ is a set of actions (e.g., move up/down/left/right),
- $\mathcal{F} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a set of transitions,
- $\mathcal{R} : \mathcal{S} \to \mathbb{R}$ is a reward function that assigns a reward to each state (e.g., 1 for reaching the goal and 0 otherwise).

In general, transitions in an MDP can be stochastic; however, for our setting, we only consider deterministic transitions and rewards.

Policy. A policy for an MDP specifies how the agent should act in each state. Specifically, we consider a (stochastic) *policy* $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where $\pi(S, A)$ is the probability of taking action A in state S. Alternatively, we can also think of π as a mapping from states to distributions over actions. Thus, we write $A \sim \pi(S)$ to denote that action A is sampled from the distribution for state s.

Rollout. Given an MDP $\mathcal{M}$ and policy π, a *rollout* is a sequence of state-action-reward tuples obtained by sampling an initial state and then using π to make decisions until a final state is reached. More formally, for a rollout of the form:

$$\zeta = ((S_1, A_1, R_1), ..., (S_{m-1}, A_{m-1}, R_{m-1}), (S_m, \varnothing, R_m)),$$

we have $S_m \in \mathcal{S}_T$, $S_1 \sim \mathcal{S}_I$ (i.e., S_1 is sampled from an initial state), and, for each $i \in \{1, ..., m-1\}$, $A_i \sim \pi(S_i)$, $R_i = \mathcal{R}(S_i)$, and $S_{i+1} = \mathcal{F}(S_i, A_i)$.

In general, a policy π induces a distribution $\mathcal{D}_\pi$ over the rollouts of an MDP $\mathcal{M}$. Since we assume that MDP transitions are deterministic, we have:

$$\mathcal{D}_\pi(\zeta) = \prod_{i=1}^{m-1} \pi(S_i, A_i).$$

RL Problem. Given an MDP $\mathcal{M}$, the goal of reinforcement learning is to compute an *optimal* policy π^* for $\mathcal{M}$. More formally, π^* should maximize *cumulative expected reward*:

$$\pi^* = \arg\max_\pi J(\pi)$$

where the *cumulative expected reward* $J(\pi)$ is computed as follows:

$$J(\pi) = \mathbb{E}_{\zeta \sim \mathcal{D}_\pi} \left[\sum_{i=1}^{m} R_i \right]$$

Policy Gradient Algorithm. The *policy gradient algorithm* is a well-known RL algorithm for finding optimal policies. It assumes a parametric policy family π_θ with parameters $\theta \in \mathbb{R}^d$. For example, π_θ may be a deep neural network (DNN), where θ denotes the parameters of the DNN. At a high level, the policy gradient algorithm uses the following theorem to optimize $J(\pi_\theta)$ [48]:

Theorem 1. *We have*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\zeta \sim \mathcal{D}_{\pi_\theta}}[\ell(\zeta)] \qquad \textit{where} \qquad \ell(\zeta) = \sum_{i=1}^{m-1} \left(\sum_{j=i+1}^{m} R_j \right) \nabla_\theta \log \pi_\theta(S_i, A_i). \tag{1}$$

In this theorem, the term $\nabla_\theta \log \pi_\theta(S_i, A_i)$ intuitively gives a direction in the parameter space that, when moving the policy parameters towards it, increases the probability of taking action A_i at state S_i. Also, the sum $\sum_{j=i+1}^{m} R_j$ is the total future reward after taking action A_i. Thus, $\ell(\zeta)$ is just the sum of different directions in the parameter space weighted by their corresponding future reward. Thus, the gradient $\nabla_\theta J(\pi_\theta)$ moves policy parameters in a direction that increases the probability of taking actions that lead to higher rewards.

Based on this theorem, we can estimate the gradient $\nabla_\theta J(\pi_\theta)$ using rollouts sampled from $\mathcal{D}_{\pi_\theta}$:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{n} \sum_{k=1}^{n} \ell(\zeta^{(k)}), \tag{2}$$

where $\zeta^{(k)} \sim \mathcal{D}_{\pi_\theta}$ for each $k \in \{1, ..., n\}$. The policy gradient algorithm uses stochastic gradient ascent in conjunction with Eq. (2) to maximize $J(\pi_\theta)$ [48].

3 Problem Formulation

In this paper, we focus on the setting of syntax-guided synthesis [1]. Specifically, given a domain-specific language (DSL) L and a specification ϕ, our goal is to find a program in L that satisfies ϕ. In the remainder of this section, we formally define our synthesis problem and clarify our assumptions.

DSL. We assume a domain-specific language L specified as a context-free grammar $L = (V, \Sigma, R, S)$, where V, Σ denote non-terminals and terminals respectively, R is a set of productions, and S is the start symbol.

Definition 1 (Partial program). *A* partial program P *is a sequence* $P \in (\Sigma \cup V)^*$ *such that* $S \overset{*}{\Rightarrow} P$ *(i.e., P can be derived from S via a sequence of productions). We refer to any non-terminal in P as a hole* hole*, and we say that P is* complete *if it does not contain any holes.*

$$
\begin{aligned}
S &\rightarrow N \mid L \\
N &\rightarrow 0 \mid \ldots \mid 10 \mid x_i \\
L &\rightarrow x_i \mid \texttt{take}(L, N) \mid \texttt{drop}(L, N) \mid \texttt{sort}(L) \\
&\quad \mid \texttt{reverse}(L) \mid \texttt{add}(L, L) \mid \texttt{sub}(L, L) \mid \texttt{sumUpTo}(L)
\end{aligned}
$$

Fig. 2. A simple programming language used for illustration. Here, `take` (resp. `drop`) keeps (resp. removes) the first N elements in the input list. Also, `add` (resp. `sub`) compute a new list by adding (resp. subtracting) elements from the two lists pair-wise. Finally, `sumUpTo` generates a new list where the i'th element in the output list is the sum of all previous elements (including the i'th element) in the input list.

Given a partial program P containing a hole H, we can fill this hole by replacing H with the right-hand-side of any grammar production r of the form $H \rightarrow e$. We use the notation $P \overset{r}{\Rightarrow} P'$ to indicate that P' is the partial program obtained by replacing the first occurrence of H with the right-hand-side of r, and we write $\textsc{Fill}(P, r) = P'$ whenever $P \overset{r}{\Rightarrow} P'$.

Example 1. Consider the small programming language shown in Fig. 2 for manipulating lists of integers. The following partial program P over this DSL contains three holes, namely L_1, L_2, N_1:

$$\texttt{add}(L_1, \texttt{take}(L_2, N_1))$$

Now, consider the production $r \equiv L \rightarrow \texttt{reverse}(L)$. In this case, $\textsc{Fill}(P, r)$ yields the following partial program P':

$$\texttt{add}(\texttt{reverse}(L_1), \texttt{take}(L_2, N_1))$$

Program Synthesis Problem. Given a specification ϕ and language $L = (V, \Sigma, R, S)$, the goal of program synthesis is to find a *complete* program P such that $S \overset{*}{\Rightarrow} P$ and P satisfies ϕ. We use the notation $P \models \phi$ to indicate that P is a complete program that satisfies specification ϕ.

Deduction Engine. In the remainder of this paper, we assume access to a *deduction engine* that can determine whether a partial program P is *feasible* with respect to specification ϕ. To make this more precise, we introduce the following notion of feasibility.

Definition 2 (Feasible partial program). *Given a specification ϕ and language $L = (V, \Sigma, R, S)$, a partial program P is said to be* feasible *with respect to ϕ if there exists any complete program P' such that $P \overset{*}{\Rightarrow} P'$ and $P' \models \phi$.*

In other words, a feasible partial program can be refined into a complete program that satisfies the specification. We assume that our deduction oracle over-approximates feasibility. That is, if P is feasible with respect to specification ϕ, then $\textsc{Deduce}(P, \phi)$ should report that P is feasible but not necessarily vice versa. Note that almost all deduction techniques used in the program synthesis literature satisfy this assumption [18,19,21,27,53].

Example 2. Consider again the DSL from Fig. 2 and the specification ϕ defined by the following input-output example:

$$[65, 2, 73, 62, 78] \mapsto [143, 129, 213, 204, 345]$$

The partial program $\texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N))$ is infeasible because, no matter what production we use to fill non-terminal N, the resulting program cannot satisfy the provided specification for the following reason:

- Given a list l and integer n where $n < \mathit{length}(l)$, $\texttt{take}(l, n)$ returns the first n elements in l. Thus, the length of $\texttt{take}(l, n)$ is smaller than that of l.
- The construct $\texttt{reverse}(l)$ reverses its input; thus, the size of the output list is the same as its input.
- Finally, $\texttt{add}(l_1, l_2)$ constructs a new list by adding the elements of its input lists pair-wise. Thus, `add` expects the two input lists to be the same size.
- Since the outputs of `reverse` and `take` do not have the same size, we cannot combine them using `add`.

Several techniques from prior work (e.g., [18,19,39,53]) can prove the infeasibility of such partial programs by using an SMT solver (provided specifications are given for the DSL constructs).

Beyond checking feasibility, some deduction techniques used for synthesis can also provide additional information [18,32,54]. In particular, given a partial program P that is infeasible with respect to specification ϕ, several deduction engines can generate a set of other infeasible partial programs $P_1, \ldots, P_n$ that are infeasible for the same reason as P. To unify both types of feedback, we assume that the output of the deduction oracle $\mathcal{O}$ is a set S of partial programs such that S is empty if and only if $\mathcal{O}$ decides that the partial program is feasible.

This discussion is summarized by the following definition:

Definition 3 (Deduction engine). *Given a partial program P and specification ϕ,* Deduce*(P, ϕ) yields a set of partial programs S such that (1) if $S \neq \varnothing$, then P is infeasible, and (2) for every $P' \in S$, it must be the case that P' is infeasible with respect to ϕ.*

Example 3. Consider again the same infeasible partial program P given in Example 2. Since $\texttt{drop}(l, n)$ drops the first n elements from list l (where $n < \mathit{length}(l)$), it also produces a list whose length is smaller than that of the input. Thus, the following partial program P' is also infeasible for the same reason as P:

$$P' \equiv \texttt{add}(\texttt{reverse}(x), \texttt{drop}(x, N))$$

Thus, Deduce(P, ϕ) may return the set $\{P, P'\}$.

4 MDP Formulation of Deduction-Guided Synthesis

Given a specification ϕ and language $L = (V, \Sigma, R, S)$, we can formulate the program synthesis problem as an MDP $\mathcal{M}_\phi = (\mathcal{S}, \mathcal{S}_I, \mathcal{S}_T, \mathcal{A}, \mathcal{F}, \mathcal{R})$, where:

- States $\mathcal{S}$ include all partial programs P such that $S \overset{*}{\Rightarrow} P$ as well as a special label $\perp$ indicating a syntactically ill-formed partial program
- $\mathcal{S}_I$ places all probability mass on the empty program S, i.e.,

$$\mathcal{S}_I(P) = \begin{cases} 1 \text{ if } P = S \\ 0 \text{ if } P \neq S \end{cases}$$

- $\mathcal{S}_T$ includes complete programs as well as infeasible partial programs, i.e.,

$$P \in \mathcal{S}_T \iff \textsc{IsComplete}(P) \ \vee \ \textsc{Deduce}(P, \phi) \neq \varnothing \vee P = \perp$$

- Actions $\mathcal{A}$ are exactly the productions R for the DSL
- Transitions $\mathcal{F}$ correspond to filling a hole using some production i.e.,

$$\mathcal{F}(P,\ r = (H \to e)) = \begin{cases} \perp & \text{if } H \text{ is not a hole in } P \\ \textsc{Fill}(P, r) & \text{otherwise} \end{cases}$$

- The reward function penalizes infeasible programs and rewards correct solutions, i.e.,

$$\mathcal{R}(P) = \begin{cases} 1 & \text{if } P \models \phi \\ -1 & \text{if } P = \perp \vee \textsc{Deduce}(P, \phi) \neq \varnothing \vee (\textsc{IsComplete}(P) \wedge P \not\models \phi) \\ 0 & \text{otherwise.} \end{cases}$$

Observe that our reward function encodes the goal of synthesizing a complete program P that satisfies ϕ, while avoiding the exploration of as many infeasible programs as possible. Thus, if we have a good policy π for this MDP, then a rollout of π is likely to correspond to a solution of the given synthesis problem.

Example 4. Consider the same specification (i.e., input-output example) ϕ from Example 2 and the DSL from Example 1. The partial program

$$P \equiv \texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N))$$

is a terminal state of $\mathcal{M}_\phi$ since $\textsc{Deduce}(P, \phi)$ yields a non-empty set, and we have $\mathcal{R}(P) = -1$. Thus, the following sequence corresponds to a rollout of $\mathcal{M}_\phi$:

$$(S, S \to L, 0),\ (L, L \to \texttt{add}(L, L), 0),\ (\texttt{add}(L_1, L_2), L \to \texttt{reverse}(L), 0)$$
$$(\texttt{add}(\texttt{reverse}(L_1), L_2), L \to x, 0),\ (\texttt{add}(\texttt{reverse}(x), L), L \to \texttt{take}(L, N), 0)$$
$$(\texttt{add}(\texttt{reverse}(x), \texttt{take}(L, N)), L \to x, 0), (\texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N)), \varnothing, -1).$$

Simplified Policy Gradient Estimate for $\mathcal{M}_\phi$. Since our synthesis algorithm will be based on policy gradient, we will now derive a simplified policy gradient for our MDP $\mathcal{M}_\phi$. First, by construction of $\mathcal{M}_\phi$, a rollout ζ has the form

$$(P_1, r_1, 0), ..., (P_m, \varnothing, q)$$

where $q = 1$ if $P_m \models \phi$ and $q = -1$ otherwise. Thus, the term $\ell(P)$ from Eq. 1 can be simplified as follows:

$$\ell(P_m) = \sum_{i=1}^{m-1} q \cdot \nabla_\theta \log \pi_\theta(P_i, r_i), \tag{3}$$

where $P_m \sim \mathcal{D}_{\pi_\theta}$ is a final state (i.e., complete program or infeasible partial program) sampled using π_θ. Then, Eq. 1 is equivalently

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{n} \sum_{k=1}^{n} \ell(P^{(k)}), \tag{4}$$

where $P^{(k)} \sim \mathcal{D}_{\pi_\theta}$ for each $k \in \{1, ..., n\}$.

5 RL-Based Synthesis Algorithm

In this section, we describe our synthesis algorithm based on reinforcement learning. Our method is an *off-policy* variant of the standard (on-policy) policy gradient algorithm and incorporates additional feedback – in the form of other infeasible programs – provided by the deduction engine when improving its policy parameters. We first give a high-level overview of the synthesis algorithm and then explain how to update the policy.

5.1 Overview of Synthesis Algorithm

Our RL-based synthesis algorithm is presented in Fig. 3. In addition to specification ϕ and domain-specific language L, this algorithm also takes as input an initial policy π_0 that has been trained off-line on a representative set of training problems.[1] In each iteration of the main synthesis loop, we first obtain a rollout of the current policy by calling the GetRollout procedure at line 7. Here, each rollout either corresponds to a complete program P or an infeasible partial program. If P is complete *and* satisfies the specification, we return it as a solution in line 8. Otherwise, we use feedback $\mathcal{C}$ provided by the deduction engine to improve the current policy (line 9). In the following subsections, we explain the GetRollout and UpdatePolicy procedures in more detail.

5.2 Sampling Rollouts

The GetRollout procedure iteratively expands a partial program, starting from the start symbol S of the grammar (line 11). In each iteration (lines 12–19), we first check whether the current partial program P is feasible by calling Deduce. If P is infeasible (i.e., $\mathcal{C}$ is non-empty), then we have reached a terminal

[1] We explain how to train this initial policy in Sect. 6.

```
1: procedure SYNTHESIZE(L, φ, π_0)
2:     input: Domain-specific language L = (V, Σ, R, S)
3:     input: Specification φ; initial policy π_0
4:     output: Complete program P such that P ⊨ φ
5:     π_θ ← π_0
6:     while true do
7:         (P, C) ← GETROLLOUT(L, φ, π_θ)
8:         if C = ∅ then return P
9:         else π_θ ← UPDATEPOLICY(π_θ, C)
10: procedure GETROLLOUT(L, φ, π_θ)
11:     P ← S
12:     while true do
13:         C ← DEDUCE(P, φ)
14:         if C ≠ ∅ then return (P, C)
15:         choose r ∼ π_θ(P) ∧ LHS(r) ∈ HOLES(P)
16:         P ← FILL(P, r)
17:         if ISCOMPLETE(P) then
18:             if P ⊨ φ then return (P, ∅)
19:             else return (P, {P})
20: procedure UPDATEPOLICY(π_θ, C)
21:     for k ∈ {1, ..., n'} do
22:         P^(k) ∼ Uniform(C)
23:     θ' ← θ + η Σ_{k=1}^{n'} ℓ(P^(k)) · D_{π_θ}(P^(k)) / (1/|C|)
24:     return π_θ'
```

Fig. 3. Deduction-guided synthesis algorithm based on reinforcement learning

state of the MDP; thus, we return P as the final state of the rollout. Otherwise, we continue expanding P according to the current policy π_θ. Specifically, we first sample an action (i.e., grammar production) r that is applicable to the current state (i.e., the left-hand-side of r is a hole in P), and, then, we expand P by calling the FILL procedure (defined in Sect. 3) at line 16. If the resulting program is complete, we have reached a terminal state and return P; otherwise, we continue expanding P according to the current policy.

5.3 Improving the Policy

As mentioned earlier, our algorithm improves the policy by using the feedback $\mathcal{C}$ provided by the deduction engine. Specifically, consider an infeasible program P explored by the synthesis algorithm at line 7. Since DEDUCE(P, ϕ) yields a set of infeasible programs, for every program $P' \in \mathcal{C}$, we know that the reward should be -1. As a consequence, we should be able to incorporate the rollout used to construct P into the policy gradient estimate based on Eq. (3). However, the challenge to doing so is that Eq. (4) relies on *on-policy* samples – i.e., the

programs $P^{(k)}$ in Eq. (4) must be sampled using the current policy π_θ. Since $P' \in \mathcal{C}$ is not sampled using π_θ, we cannot directly use it in Eq. (4).

Instead, we use *off-policy* RL to incorporate P' into the estimate of $\nabla_\theta J(\pi_\theta)$ [28]. Essentially, the idea is to use *importance weighting* to incorporate data sampled from a different distribution than $\mathcal{D}_{\pi_\theta}$. In particular, suppose we are given a distribution $\tilde{\mathcal{D}}$ over final states. Then, we can derive the following gradient:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{P\sim\mathcal{D}_{\pi_\theta}}[\ell(P)] \\ &= \mathbb{E}_{P\sim\tilde{\mathcal{D}}}\left[\ell(P)\cdot\frac{\mathcal{D}_{\pi_\theta}(P)}{\tilde{\mathcal{D}}(P)}\right]\end{aligned} \tag{5}$$

Intuitively, the *importance weight* $\frac{\mathcal{D}_{\pi_\theta}(P)}{\tilde{\mathcal{D}}(P)}$ accounts for the fact that P is sampled from the "wrong" distribution.

Now, we can use the distribution $\tilde{\mathcal{D}} = \text{Uniform}(\textsc{Deduce}(P', \phi))$ for a randomly sampled final state $P' \sim \mathcal{D}_{\pi_\theta}$. Thus, we have[2]:

Theorem 2. *The policy gradient is*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{P'\sim\mathcal{D}_{\pi_\theta}, P\sim \mathit{Uniform}(\textsc{Deduce}(P',\phi))}\left[\ell(P)\cdot\frac{\mathcal{D}_{\pi_\theta}(P)}{1/|\textsc{Deduce}(P',\phi)|}\right]. \tag{6}$$

Proof. Note that

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{P'\sim\mathcal{D}_{\pi_\theta}}[\nabla_\theta J(\pi_\theta)] \\ &= \mathbb{E}_{P'\sim\mathcal{D}_{\pi_\theta}, P\sim\text{Uniform}(\textsc{Deduce}(P',\phi))}\left[\ell(P)\cdot\frac{\mathcal{D}_{\pi_\theta}(P)}{1/|\textsc{Deduce}(P',\phi)|}\right],\end{aligned}$$

as claimed. □

The corresponding estimate of $\nabla_\theta J(\pi_\theta)$ is given by the following equation:

$$\nabla_\theta J(\theta) \approx \frac{1}{n}\sum_{k=1}^{n}\frac{1}{n'}\sum_{k'=1}^{n'}\ell(P^{(k,k')})\cdot\frac{\mathcal{D}_{\pi_\theta}(P^{(k,k')})}{1/|\textsc{Deduce}(P^{(k)},\phi)|},$$

where $P^{(k)} \sim \tilde{\mathcal{D}}$ and $P^{(k,k')} \sim \text{Uniform}(\textsc{Deduce}(P^{(k)}, \phi))$ for each $k \in \{1, ..., n\}$ and $k' \in \{1, ..., n'\}$. Our actual implementation uses $n = 1$, in which case this equation can be simplified to the following:

$$\nabla_\theta J(\theta) \approx \frac{1}{n'}\sum_{k'=1}^{n'}\ell(P)\cdot\frac{\mathcal{D}_{\pi_\theta}(P^{(k')})}{1/|\textsc{Deduce}(P,\phi)|}, \tag{7}$$

[2] Technically, importance weighting requires that the support of $\tilde{\mathcal{D}}$ contains the support of $\mathcal{D}_{\pi_\theta}$. We can address this issue by combining $\tilde{\mathcal{D}}$ and $\mathcal{D}_{\pi_\theta}$—in particular, take $\tilde{\mathcal{D}}(P) = (1-\epsilon)\cdot\text{Uniform}(\textsc{Deduce}(P',\phi))(P) + \epsilon\cdot\mathcal{D}_{\pi_\theta}(P)$, for any $\epsilon > 0$.

where $P \sim \tilde{\mathcal{D}}$ and $P^{(k')} \sim \text{Uniform}(\textsc{Deduce}(P, \phi))$ for each $k' \in \{1, ..., n'\}$.

Now, going back to our synthesis algorithm from Fig. 3, the UpdatePolicy procedure uses Eq. 7 to update the policy parameters θ. Specifically, given a set $\mathcal{C}$ of infeasible partial programs, we first sample n' programs $P^{(1)}, \ldots, P^{(n')}$ from $\mathcal{C}$ uniformly at random (line 22). Then, we use the probability of each $P^{(k)}$ being sampled from the current distribution $\mathcal{D}_{\pi_\theta}$ to update the policy parameters to a new value θ' according to Eq. 7.

Example 5. Suppose that the current policy assigns the following probabilities to these state, action pairs:

$$\pi_\theta((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{take}(L, N)) = 0.3$$
$$\pi_\theta((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{drop}(L, N)) = 0.3$$
$$\pi_\theta((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{sumUpTo}(L)) = 0.1$$

Furthermore, suppose that we sample the following rollout using this policy:

$$P \equiv \texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N)),$$

This corresponds to an infeasible partial program, and, as in Example 3, $\textsc{Deduce}(P, \phi)$ yields $\{P, P'\}$ where $P' \equiv \texttt{add}(\texttt{reverse}(x), \texttt{drop}(x, N))$. Using the gradients derived by Eq. 7, we update the policy parameters θ to θ'. The updated policy now assigns the following probabilities to the same state, action pairs:

$$\pi_{\theta'}((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{take}(L, N)) = 0.15$$
$$\pi_{\theta'}((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{drop}(L, N)) = 0.15$$
$$\pi_{\theta'}((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{sumUpTo}(L)) = 0.2$$

Observe that the updated policy makes it less likely that we will expand the partial program $\texttt{add}(\texttt{reverse}(x), L))$ using the `drop` production in addition to the `take` production. Thus, if we reach the same state $\texttt{add}(\texttt{reverse}(x), L)$ during rollout sampling in the next iteration, the policy will make it more likely to explore the `sumUpTo` production, which does occur in the desired program

$$\texttt{add}(\texttt{reverse}(x), \texttt{sumUpTo}(x))$$

that meets the specification from Example 2.

6 Implementation

We have implemented the proposed algorithm in a new tool called Concord written in Python. In what follows, we elaborate on various aspects of our implementation.

6.1 Deduction Engine

CONCORD uses the same deduction engine described by Feng et al. [18]. Specifically, given a partial program P, CONCORD first generates a specification φ of P by leveraging the abstract semantics of each DSL construct. Then, CONCORD issues a satisfiability query to the Z3 SMT solver [15] to check whether φ is consistent with the provided specification. If it is not, this means that P is infeasible, and CONCORD proceeds to infer other partial programs that are also infeasible for the same reason as P. To do so, CONCORD first obtains an unsatisfiable core ψ for the queried formula, and, for each clause c_i of ψ originating from DSL construct f_i, it identifies a set S_i of other DSL constructs whose semantics imply c_i. Finally, it generates a set of other infeasible programs by replacing all f_i's in the current program with another construct drawn from its corresponding set S_i.

6.2 Policy Network

Architecture. As shown by Fig. 4, CONCORD represents its underlying policy using a deep neural network (DNN) $\pi_\theta(r \mid P)$, which takes as input the current state (i.e., a partial program P) and outputs a probability distribution over actions (i.e., productions r in the DSL). We represent each program P as a flat sequence of statements and use a recurrent neural network (RNN) architecture, as this is a natural choice for sequence inputs. In particular, our policy network is a gated recurrent unit (GRU) network [13], which is a state-of-the-art RNN architecture. Our policy network has one hidden layer with 256 neurons; this layer is sequentially applied to each statement in the partial program together with the latent vector from processing the previous statement. Once the entire partial program P has been encoded into a vector, π_θ has a final layer that outputs a distribution over DSL productions r based on this vector.

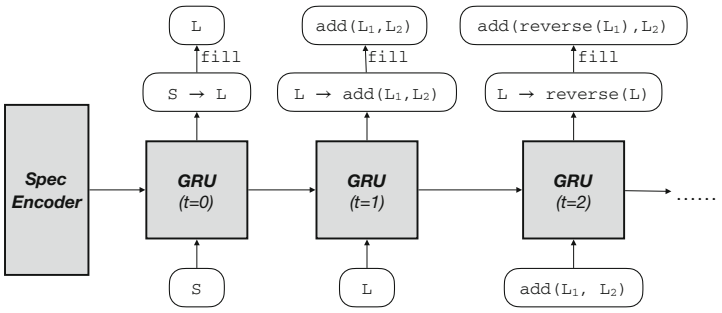

Fig. 4. The architecture of the policy network showing how to roll out the partial program in Example 4.

Pretraining the Initial Policy. Recall from Sect. 5 that our synthesis algorithm takes a input an *initial policy network* that is updated during the synthesis

process. One way to initialize the the policy network would be to use a standard random initialization of the network weights. However, a more effective alternative is to *pretrain* the policy on a benchmark suite of program synthesis problems [44]. Specifically, consider a representative training set X_{train} of synthesis problems of the form (ϕ, P), where ϕ is the specification and P is the desired program. To obtain an initial policy, we augment our policy network to take as input an encoding of the specification ϕ for the current synthesis problem – i.e., it has the form $\pi_\theta(r \mid P, \phi)$.[3] Then, we use supervised learning to train π_θ to predict P given ϕ—i.e.,

$$\theta^0 = \arg\max_\theta \sum_{(\phi,P)\in X_{\text{train}}} \sum_{i=1}^{|P|-1} \pi_\theta(r_i \mid P_i, \phi).$$

We optimize θ using stochastic-gradient descent (SGD) on this objective.

Given a new synthesis problem ϕ, we use π_{θ^0} as the initial policy. Our RL algorithm then continues to update the parameters starting from θ^0.

6.3 Input Featurization

As standard, we need a way to featurize the inputs to our policy network – i.e., the statements in each partial program P, and the specification ϕ. Our current implementation assumes that statements are drawn from a finite set and featurizes them by training a different embedding vector for each kind of statement. While our general methodology can be applied to different types specifications, our implementation featurizes the specification under the assumption that it consists of input-output examples and uses the same methodology described by Balog et al. [2].

6.4 Optimizations

Our implementation performs a few optimization over the algorithm presented in Sect. 5. First, since it is possible to sample the same rollout multiple times, our implementation uses a hash map to check whether a rollout has already been explored. Second, in different invocations of the GetRollout procedure from Fig. 3, we may end up querying the feasibility of the same state (i.e., partial program) *many* times. Since checking feasibility requires a potentially-expensive call to the SMT solver, our implementation also memoizes the results of feasibility checks for each state. Finally, similar to Chen et al. [11], we use a 3-model ensemble to alleviate some of the randomness in the synthesis process and return a solution as soon as one of the models in the ensemble finds a correct solution.

7 Evaluation

In this section, we describe the results from our experimental evaluation, which is designed to answer the following key research questions:

[3] Including the specification as an input to π_θ is unnecessary if we do not use pretraining, since ϕ does not change for a single synthesis problem.

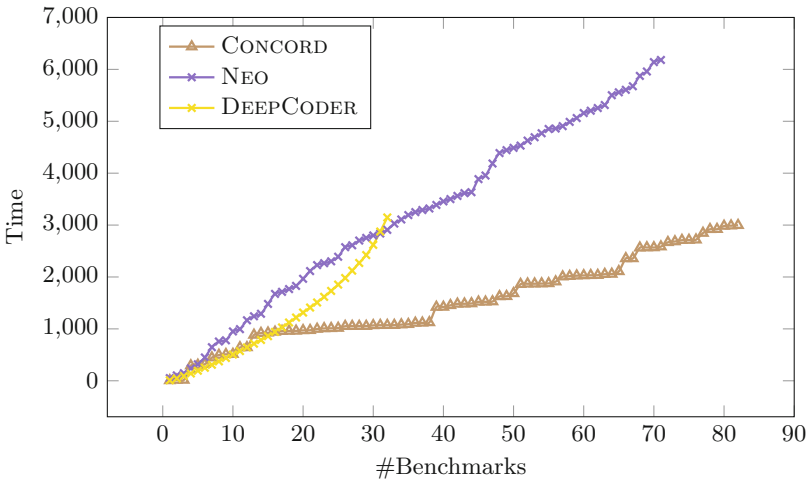

Fig. 5. Comparison between Concord, Neo, and DeepCoder

1. How does Concord compare against existing synthesis tools?
2. What is the impact of updating the statistical model during synthesis? (i.e., is reinforcement learning actually useful)?
3. How important is the proposed off-policy RL algorithm compared to standard policy gradient?
4. How important is it to get feedback from the deduction engine when updating the policy?

Benchmarks. We evaluate the proposed technique on a total of 100 synthesis tasks used in prior work [2,18]. Specifically, these synthesis tasks require performing non-trivial transformations and computations over lists using a functional programming language. Since these benchmarks have been used to evaluate both Neo [18] and DeepCoder [2], they provide a fair ground for comparing our approach against two of the most closely-related techniques. In particular, note that DeepCoder uses a pre-trained deep neural network to guide its search, whereas Neo uses both statistical and logical reasoning (i.e., statistical model to guide search and deduction to prune the search space). However, unlike our proposed approach, neither Neo nor DeepCoder update their statistical model during synthesis time.

Training. Recall that our algorithm utilizes a pre-trained initial policy. To generate the initial policy, we use the same methodology described in DeepCoder [2] and adopted in Neo [18]. Specifically, we randomly generate both programs and inputs, and we obtain the corresponding output by executing the program. Then, we train the DNN model discussed in Sect. 6 on the Google Cloud Platform with a 2.20 GHz Intel Xeon CPU and an NVIDIA Tesla K80 GPU using 16 GB of memory.

7.1 Comparison Against Existing Tools

To answer our first research question, we compare Concord against both Neo and DeepCoder on the 100 synthesis benchmarks discussed earlier. The result of this comparison is shown in Fig. 5, which plots the number of benchmarks solved within a given time limit for each of the three tools. As we can see from this figure, Concord outperforms DeepCoder and Neo both in terms of synthesis time as well as the number of benchmarks solved within the 5-min time limit. In particular, Concord can solve 82% of these benchmarks with an average running time of 36 s, whereas Neo (resp. DeepCoder) solves 71% (resp. 32%) with an average running time of 99 s (resp. 205 s). Thus, we believe these results answer our first research question in a positive way.

7.2 Ablation Study

To answer our remaining research questions, we perform an ablation study in which we compare Concord against three variants:

- **Concord-noRL:** This variant does not use reinforcement learning to update its policy during synthesis. However, it still uses the pre-trained policy to guide search, and it also uses deduction to prune infeasible partial programs. In other words, Concord-noRL is the same as the synthesis algorithm from Fig. 3 but it does not invoke the `UpdatePolicy` procedure to improve its policy during synthesis.
- **Concord-NoDeduce:** This variant uses reinforcement learning; however, it does not incorporate feedback from the deduction engine. That is, rather than checking feasibility of partial programs, it instead samples complete programs and uses the percentage of passing input-output examples as the reward signal. Note that this variant of Concord essentially corresponds to the technique proposed by Si et al. [44].[4]
- **Concord-StandardPG:** Recall that our algorithm uses an off-policy variant of the standard policy gradient algorithm to incorporate additional feedback from the deduction engine. To evaluate the benefit of our proposed approach, we created a variant called Concord-StandardPG that uses the standard (i.e., on-policy) policy gradient algorithm. In other words, ConcordStandardPG implements the same synthesis algorithm from Fig. 3 except that it uses Theorem 1 to update θ instead of Theorem 2.

The results from this evaluation are summarized in Table 1. Here, the first column labeled "# solved" shows the number of solved benchmarks, and the second column shows percentage improvement over Neo in terms of benchmarks solved. The third column shows average synthesis time for benchmarks that can

[4] We reimplement the RL algorithm proposed in [44] since we cannot directly compare against their tool. Specifically, the policy network in their implementation is tailored to their problem domain.

Table 1. Results of ablation study result comparing different variants.

	# solved	Delta to NEO	Avg. time (s)	Speedup over NEO
CONCORD-noRL	56	−21%	48	1.63×
CONCORD-NoDeduce	65	−8%	21	3.66×
CONCORD-StandardPG	65	−8%	27	2.88×
CONCORD	82	+15%	9	8.71×

be solved by *all* variants and NEO. Finally, the last column shows speed-up in terms of synthesis time compared to NEO.

As we can see from this table, all variants are significantly worse than CONCORD in terms of the number of benchmarks that can be solved within a 5-min time limit[5]. Furthermore, as we can see from the column labeled "Delta to NEO", all of our proposed ideas are important for improving over the state-of-the-art, as NEO outperforms all three variants but not the full CONCORD system, which solves 15% more benchmarks compared to NEO.

Next, looking at the third column of Table 1, we see that all three variants of CONCORD are significantly slower compared to CONCORD in terms of synthesis time. While both CONCORD and all of its variants outperform NEO in terms of synthesis time (for benchmarks solved by all tools), CONCORD by far achieves the greatest speed-up over NEO.

In summary, the results from Table 1 highlight that all of our proposed ideas (i.e., (1) improving policy at synthesis time; (2) using feedback from deduction; and (3) off-policy RL) make a significant difference in practice. Thus, we conclude that the ablation study positively answers our last three research questions.

8 Related Work

In this section, we survey prior work that is closely related to the techniques proposed in this paper.

Program Synthesis. Over the past decade, there has been significant interest in automatically synthesizing programs from high-level expressions of user intent [2,6,21,23,25,39,40,46]. Some of these techniques are geared towards computer end-users and therefore utilize informal specifications such as input-output examples [23,40,50], natural language [24,42,55,56], or a combination of both [10,12]. On the other hand, program synthesis techniques geared towards programmers often utilize additional information, such as a program sketch [17,36,46,49] or types [33,39] in addition to test cases [20,30] or logical specifications [6,49]. While the synthesis methodology proposed in this paper

[5] To understand the improvement brought by the pre-trainedd policy, we also conduct a baseline experiment by using randomly initialized policy in CONCORD. Given the setting, CONCORD can solve as many as 27% of the benchmarks in the given 5-min time limit.

can, in principle, be applied to a broad set of specifications, the particular featurization strategy we use in our implementation is tailored towards input-output examples.

Deduction-Based Pruning. In this paper, we build on a line of prior work on using deduction to prune the search space of programs in a DSL [18,19,21,39,53]. Some of these techniques utilize type-information and type-directed reasoning to detect infeasible partial programs [20–22,37,39]. On the other hand, other approaches use some form of lightweight program analysis to prune the search space [18,19,53]. Concretely, Blaze uses abstract interpretation to build a compact version space representation capturing the space of all feasible programs [53]; Morpheus [19] and Neo [18] utilize logical specifications of DSL constructs to derive specifications of partial programs and query an SMT solver to check for feasibility; Scythe [50] and Viser [51] use deductive reasoning to compute approximate results of partial programs to check their feasibility. Our approach learns from deduction feedback to improve search efficiency. As mentioned in Sect. 6, the deductive reasoning engine used in our implementation is similar to the latter category; however, it can, in principle, be used in conjunction with other deductive reasoning techniques for pruning the search space.

Learning from Failed Synthesis Attempts. The technique proposed in this paper can utilize feedback from the deduction engine in the form of other infeasible partial programs. This idea is known as *conflict-driven learning* and has been recently adopted from the SAT solving literature [5,57] to program synthesis [18]. Specifically, Neo uses the unsat core of the program's specification to derive other infeasible partial programs that share the same root cause of failure, and, as described in Sect. 6, we use the same idea in our implementation of the deduction engine. While we use logical specifications to infer other infeasible programs, there also exist other techniques (e.g., based on testing [54]) to perform this kind of inference.

Machine Learning for Synthesis. This paper is related to a long line of work on using machine learning for program synthesis. Among these techniques, some of them train a machine learning model (typically a deep neural network) to directly predict a full program from the given specification [12,16,34,35]. Many of these approaches are based on sequence-to-sequence models [47], sequence to tree models [56], or graph neural networks [41] commonly used in machine translation.

A different approach, sometimes referred to as *learning to search*, is to train a statistical model that is used to *guide* the search rather than directly predict the target program. For example, DeepCoder [2] uses a deep neural network (DNN) to predict the most promising grammar productions to use for the given input-output examples. Similarly, R3NN [38] and NGDS [26] use DNNs to predict the most promising grammar productions conditioned on both the specification and the current partial program. In addition, there has been work on using concrete program executions on the given input-output examples to guide the DNN [11,52]. Our technique for pretraining the initial policy network is based

on the same ideas as these supervised learning approaches; however, their initial policies do not change during the synthesis algorithm, whereas we continue to update the policy using RL.

While most of the work at the intersection of synthesis and machine learning uses *supervised learning* techniques, recent work has also proposed using reinforcement learning to speed up syntax-guided synthesis [8,29,31,44] These approaches are all on-policy and do not incorporate feedback from a deduction engine. In contrast, in our problem domain, rewards are very sparse in the program space, which makes exploration highly challenging in a on-policy learning setting. Our approach addresses this problem using off-policy RL to incorporate feedback from the deduction engine. Our ablation study results demonstrate that our off-policy RL is able to scale to more complex benchmarks.

Reinforcement Learning for Formal Methods. There has been recent interest in applying reinforcement learning (RL) to solve challenging PL problems where large amounts of labeled training data are too expensive to obtain. For instance, Si et al. use graph-based RL to automatically infer loop invariants [43], Singh et al. use Q-learning (a different RL algorithm) to speed up program analysis based on abstract interpretation [45], Dai et al. [14] uses meta-reinforcement learning for test data generation, and Chen et al. [9] uses RL to speed up relational program verification. However, these approaches only use RL offline to pretrain a DNN policy used to guide search. In contrast, we perform reinforcement learning online during synthesis. Bastani et al. has used an RL algorithm called Monte-carlo tree search (MCTS) to guide a specification inference algorithm [3]; however, their setting does not involve any kind of deduction.

9 Conclusion and Future Work

We presented a new program synthesis algorithm based on reinforcement learning. Given an initial policy trained off-line, our method uses this policy to guide its search at synthesis time but also gradually improves this policy using feedback obtained from a deductive reasoning engine. Specifically, we formulated program synthesis as a reinforcement learning problem and proposed a new variant of the *policy gradient* algorithm that is better suited to solve this problem. In addition, we implemented the proposed approach in a new tool called Concord and evaluated it on 100 synthesis tasks taken from prior work. Our evaluation shows that Concord outperforms a state-of-the-art tool by solving 15% more benchmarks with an average speedup of 8.71×. In addition, our ablation study highlights the advantages of our proposed reinforcement learning algorithm.

There are several avenues for future work. First, while our approach is applicable to different DSLs and specifications, our current implementation focuses on input-output examples. Thus, we are interested in extending our implementation to richer types of specifications and evaluating our method in application domains that require such specifications. Another interesting avenue for future work is to integrate our method with other types of deductive reasoning engines.

In particular, while our deduction method is based on SMT, it would be interesting to try other methods (e.g., based on types or abstract interpretation) in conjunction with our proposed RL approach.

References

1. Alur, R., et al.: Syntax-guided synthesis. IEEE (2013)
2. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: learning to write programs. In: Proceedings of International Conference on Learning Representations. OpenReview (2017)
3. Bastani, O., Sharma, R., Aiken, A., Liang, P.: Active learning of points-to specifications. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 678–692 (2018)
4. Bavishi, R., Lemieux, C., Fox, R., Sen, K., Stoica, I.: AutoPandas: neural-backed generators for program synthesis. PACMPL **3**(OOPSLA), 168:1–168:27 (2019)
5. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, pp. 131–153 (2009)
6. Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and litmus tests. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 467–481 (2017)
7. Brockschmidt, M., Allamanis, M., Gaunt, A.L., Polozov, O.: Generative code modeling with graphs. In: ICLR (2019)
8. Bunel, R., Hausknecht, M., Devlin, J., Singh, R., Kohli, P.: Leveraging grammar and reinforcement learning for neural program synthesis. In: ICLR (2018)
9. Chen, J., Wei, J., Feng, Y., Bastani, O., Dillig, I.: Relational verification using reinforcement learning. PACMPL **3**(OOPSLA), 14:11–14:130 (2019)
10. Chen, Q., Wang, X., Ye, X., Durrett, G., Dillig, I.: Multi-modal synthesis of regular expressions (2019)
11. Chen, X., Liu, C., Song, D.: Execution-guided neural program synthesis. In: ICLR (2018)
12. Chen, Y., Martins, R., Feng, Y.: Maximal multi-layer specification synthesis. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, 26–30 August 2019, pp. 602–612 (2019)
13. Cho, K., et al.: Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078 (2014)
14. Dai, H., Li, Y., Wang, C., Singh, R., Huang, P., Kohli, P.: Learning transferable graph exploration. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, Vancouver, BC, Canada, 8–14 December 2019, pp. 2514–2525 (2019). http://papers.nips.cc/paper/8521-learning-transferable-graph-exploration
15. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
16. Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.R., Kohli, P.: RobustFill: neural program learning under noisy I/O. In: Proceedings of the 34th International Conference on Machine Learning, vol. 70, pp. 990–998. JMLR.org (2017)

17. Ellis, K., Ritchie, D., Solar-Lezama, A., Tenenbaum, J.: Learning to infer graphics programs from hand-drawn images. In: Advances in Neural Information Processing Systems, pp. 6059–6068 (2018)
18. Feng, Y., Martins, R., Bastani, O., Dillig, I.: Program synthesis using conflict-driven learning. In: Proceedings of Conference on Programming Language Design and Implementation, pp. 420–435 (2018)
19. Feng, Y., Martins, R., Van Geffen, J., Dillig, I., Chaudhuri, S.: Component-based synthesis of table consolidation and transformation tasks from examples. In: Proceedings of Conference on Programming Language Design and Implementation, pp. 422–436. ACM (2017)
20. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.: Component-based synthesis for complex APIs. In: Proceedings of Symposium on Principles of Programming Languages, pp. 599–612. ACM (2017)
21. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 229–239 (2015)
22. Frankle, J., Osera, P., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 802–815 (2016)
23. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of Symposium on Principles of Programming Languages, pp. 317–330. ACM (2011)
24. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Mapping language to code in programmatic context. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, 31 October–4 November 2018, pp. 1643–1652 (2018). https://www.aclweb.org/anthology/D18-1192/
25. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of International Conference on Software Engineering, pp. 215–224. ACM/IEEE (2010)
26. Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., Gulwani, S.: Neural-guided deductive search for real-time program synthesis from examples. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, 30 April–3 May 2018, Conference Track Proceedings (2018). https://openreview.net/forum?id=rywDjg-RW
27. Lee, M., So, S., Oh, H.: Synthesizing regular expressions from examples for introductory automata assignments. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pp. 70–80 (2016)
28. Levine, S., Koltun, V.: Guided policy search. In: International Conference on Machine Learning, pp. 1–9 (2013)
29. Liang, C., Norouzi, M., Berant, J., Le, Q.V., Lao, N.: Memory augmented policy optimization for program synthesis and semantic parsing. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, Montréal, Canada, 3–8 December 2018, pp. 10015–10027 (2018). http://papers.nips.cc/paper/8204-memory-augmented-policy-optimization-for-program-synthesis-and-semantic-parsing
30. Long, F., Amidon, P., Rinard, M.: Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 727–739 (2017)

31. Mao, J., Gan, C., Kohli, P., Tenenbaum, J.B., Wu, J.: The neuro-symbolic concept learner: interpreting scenes, words, and sentences from natural supervision. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, 6–9 May 2019 (2019). https://openreview.net/forum?id=rJgMlhRctm
32. Martins, R., Chen, J., Chen, Y., Feng, Y., Dillig, I.: Trinity: an extensible synthesis framework for data science. Proc. VLDB Endow. **12**(12), 1914–1917 (2019)
33. Miltner, A., Maina, S., Fisher, K., Pierce, B.C., Walker, D., Zdancewic, S.: Synthesizing symmetric lenses. Proc. ACM Program. Lang. **3**(ICFP), 1–28 (2019)
34. Neelakantan, A., Le, Q.V., Abadi, M., McCallum, A., Amodei, D.: Learning a natural language interface with neural programmer. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, 24–26 April 2017, Conference Track Proceedings (2017). https://openreview.net/forum?id=ry2YOrcge
35. Neelakantan, A., Le, Q.V., Sutskever, I.: Neural programmer: inducing latent programs with gradient descent. In: 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, 2–4 May 2016, Conference Track Proceedings (2016). http://arxiv.org/abs/1511.04834
36. Nye, M.I., Hewitt, L.B., Tenenbaum, J.B., Solar-Lezama, A.: Learning to infer program sketches. In: Proceedings of the 36th International Conference on Machine Learning, ICML 2019, Long Beach, California, USA, 9–15 June 2019, pp. 4861–4870 (2019). http://proceedings.mlr.press/v97/nye19a.html
37. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 619–630 (2015)
38. Parisotto, E., Mohamed, A.R., Singh, R., Li, L., Zhou, D., Kohli, P.: Neuro-symbolic program synthesis. In: ICLR (2017)
39. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of Conference on Programming Language Design and Implementation, pp. 522–538 (2016)
40. Polozov, O., Gulwani, S.: FlashMeta: a framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, Part of SPLASH 2015, Pittsburgh, PA, USA, 25–30 October 2015, pp. 107–126 (2015)
41. Shin, E.C., Allamanis, M., Brockschmidt, M., Polozov, A.: Program synthesis and semantic parsing with learned code idioms. In: Advances in Neural Information Processing Systems, pp. 10824–10834 (2019)
42. Shin, R., Allamanis, M., Brockschmidt, M., Polozov, O.: Program synthesis and semantic parsing with learned code idioms. In: NeurIPS (2019)
43. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, Montréal, Canada, 3–8 December 2018, pp. 7762–7773 (2018)
44. Si, X., Yang, Y., Dai, H., Naik, M., Song, L.: Learning a meta-solver for syntax-guided program synthesis. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, 6–9 May 2019 (2019)
45. Singh, G., Püschel, M., Vechev, M.T.: Fast numerical program analysis with reinforcement learning. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, 14–17 July 2018, Proceedings, Part I, pp. 211–229 (2018)

46. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, 21–25 October 2006, pp. 404–415 (2006)
47. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, pp. 3104–3112 (2014)
48. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Advances in Neural Information Processing Systems, pp. 1057–1063 (2000)
49. Torlak, E., Bodík, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom, 09–11 June 2014, pp. 530–541 (2014)
50. Wang, C., Cheung, A., Bodik, R.: Synthesizing highly expressive SQL queries from input-output examples. In: Proceedings of Conference on Programming Language Design and Implementation, pp. 452–466. ACM (2017)
51. Wang, C., Feng, Y., Bodík, R., Cheung, A., Dillig, I.: Visualization by example. PACMPL **4**(POPL), 49:1–49:28 (2020). https://doi.org/10.1145/3371117
52. Wang, C., Huang, P., Polozov, A., Brockschmidt, M., Singh, R.: Execution-guided neural program decoding. CoRR abs/1807.03100 (2018). http://arxiv.org/abs/1807.03100
53. Wang, X., Dillig, I., Singh, R.: Program synthesis using abstraction refinement. In: Proceedings of Symposium on Principles of Programming Languages, pp. 63:1–63:30. ACM (2018)
54. Wang, Y., Dong, J., Shah, R., Dillig, I.: Synthesizing database programs for schema refactoring. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, 22–26 June 2019, pp. 286–300 (2019)
55. Yaghmazadeh, N., Wang, Y., Dillig, I., Dillig, T.: SQLizer: query synthesis from Natural Language. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 63:1–63:26. ACM (2017)
56. Yu, T., Yasunaga, M., Yang, K., Zhang, R., Wang, D., Li, Z., Radev, D.: SyntaxSQLNet: syntax tree networks for complex and cross-domain text-to-SQL task. In: Proceedings of EMNLP. Association for Computational Linguistics (2018)
57. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in Boolean satisfiability solver. In: Proceedings of International Conference on Computer-Aided Design, pp. 279–285. IEEE Computer Society (2001)