# Reward-Guided Synthesis of Intelligent Agents with Control Structures

GUOFENG CUI, Rutgers University, USA
YUNING WANG, Rutgers University, USA
WENJIE QIU, Rutgers University, USA
HE ZHU, Rutgers University, USA

Deep reinforcement learning (RL) has led to encouraging successes in numerous challenging robotics applications. However, the lack of inductive biases to support logic deduction and generalization in the representation of a deep RL model causes it less effective in exploring complex long-horizon robot-control tasks with sparse reward signals. Existing program synthesis algorithms for RL problems inherit the same limitation, as they either adapt conventional RL algorithms to guide program search or synthesize robot-control programs to imitate an RL model. We propose ReGuS, a reward-guided synthesis paradigm, to unlock the potential of program synthesis to overcome the exploration challenges. We develop a novel hierarchical synthesis algorithm with decomposed search space for loops, on-demand synthesis of conditional statements, and curriculum synthesis for procedure calls, to effectively compress the exploration space for long-horizon, multi-stage, and procedural robot-control tasks that are difficult to address by conventional RL techniques. Experiment results demonstrate that ReGuS significantly outperforms state-of-the-art RL algorithms and standard program synthesis baselines on challenging robot tasks including autonomous driving, locomotion control, and object manipulation.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Program Synthesis, Sequential Decision Making

**ACM Reference Format:**
Guofeng Cui, Yuning Wang, Wenjie Qiu, and He Zhu. 2024. Reward-Guided Synthesis of Intelligent Agents with Control Structures. *Proc. ACM Program. Lang.* 8, PLDI, Article 217 (June 2024), 25 pages. https://doi.org/10.1145/3656447

## 1 INTRODUCTION

Inspired by the success of program synthesis in various applications, a growing body of research has explored *programmatic reinforcement learning* [3, 25, 41, 44, 48–50, 54] that harnesses *domain-specific programs* as the representation of reinforcement learning (RL) models for robot learning. These methods either adapt existing deep RL algorithms to guide the search of robot-control programs or synthesize a program to mimic a trained RL agent in a robot environment. Unfortunately, while these methods have made significant strides in improving the interpretability of learned robot controllers, they are limited by the capabilities of existing RL algorithms. *Consequently, they are unable to effectively handle tasks that prove challenging for conventional RL approaches.*

Authors' addresses: Guofeng Cui, gc669@cs.rutgers.edu, Rutgers University, USA; Yuning Wang, yw895@rutgers.edu, Rutgers University, USA; Wenjie Qiu, wenjie.qiu@rutgers.edu, Rutgers University, USA; He Zhu, hz375@cs.rutgers.edu, Rutgers University, USA.


ACM 2475-1421/2024/6-ART217
https://doi.org/10.1145/3656447

***In what scenarios is deep reinforcement learning ineffective for robot learning?*** Robotics tasks typically involve a *long-horizon*, extending over thousands of control steps, and are characterized by *sparse rewards*, where a reward indicating success or failure is given only at the end of task execution. RL methods exhibit limited effectiveness when dealing with long-horizon and sparse-reward tasks due to the exponential increase in robot-environment interactions necessary for exploration, which grows with the number of steps required to discover rewarding signals [29]. As a result, they struggle with long-horizon multi-stage tasks such as putting an object into a drawer through the *sequence* of opening the drawer, placing the object, and subsequently closing the drawer, where intrinsic logic deduction are essential. They also face challenges when tackling long-horizon tasks that feature repetitive patterns, such as navigating through *multiple* rooms with complex floor plans, as illustrated in Fig. 1 where an ant robot [47] is tasked with locating a target object. In our experience, controllers learned using state-of-the-art RL techniques tend to get trapped in the second room [22, 35, 36] or fail to generalize to similar floor plans with more rooms [6, 32, 38, 40].

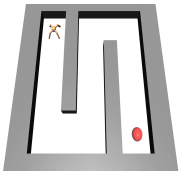


Fig. 1. A long-horizon, sparse-reward robot task.

**This paper.** We present ReGuS - *reward-guided synthesis* - to unlock the potential of program synthesis to overcome the inherent exploration challenges encountered by conventional RL techniques (which also perplex existing programmatic RL methods) in handling long-horizon, multi-stage, and procedural robot-control tasks with sparse reward signals. ReGuS aims to generate robot-control programs that guide agents within a robotic environment to sequentially take control actions, maximizing the expected reward received from the environment across all potential initial states.

**The main idea** behind ReGuS to address sparse-reward, long-horizon, multi-stage, and procedural robot-control tasks is to compress the agent exploration space by leveraging standard language constructs such as state-conditioned loops, conditional statements, and procedure calls. For example, in Fig. 1, after learning a procedure capable of traversing the first room, ReGuS can encapsulate this procedure within a loop program to iteratively explore all the remaining rooms based on this learned knowledge, leading to a systematic approach to environment exploration.

**Key Assumption.** To make program synthesis practical for high-dimensional continuous robotics environments, ReGuS fires on a domain-specific language (DSL) incorporated with *user-defined state abstraction predicates* [1, 14]. Such predicates construct a higher-level representation of the robot's environment based on observed sensor data. This higher-level representation can then be reasoned about using standard language constructs, such as loops and conditionals, to trigger suitable actions from a current state. For example, consider a simplified navigation environment `doorkey` in Fig. 2. The robot has to pick up the key (marker) in the left room to unlock the door, and then get into the right room to place the key on top of another key (marker). The DSL for a navigation program can be designed to include perceptual functions such as `leftIsClear()`, `rightIsClear()`, `frontIsClear()`, and `present(marker)` to enable the robot to detect obstacles and locate desired objects. These perceptual functions form an effective state abstraction for decision making e.g. turning left or right when the front is not clear.

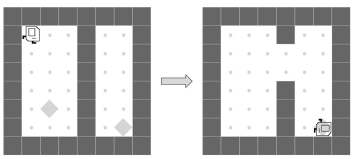


Fig. 2. The DoorKey Environment.

ReGuS does not impose restrictions on the types of state abstractions. It operates with arbitrary user-defined state abstraction, even in cases where it does not adhere to the Markov property. As an example, our state abstraction for the `doorkey` environment is not Markov. An RL controller depending solely on the abstracted states cannot accurately determine whether it should pick up a marker or place a marker when a maker is present. This is because abstraction throws away

information - the state abstraction should have included information about the room in which the robot is located. However, defining such information in the abstraction is environment-specific and not generalizable (e.g. to multiple rooms). Instead, ReGuS synthesizes a program that explicitly divides the task into two sequential loops, with the first loop specifically for handling key pick-up and the other for dropping the key at the right position, resulting in a program generalizable to multiple rooms.

**The ReGuS Framework.** While programs with rich control-flow structures compress the exploration space for long-horizon tasks in robot learning, the highly combinatorial program search space renders a pure enumeration-based synthesis process intractable. For example, there are an infinite number of syntactic elaborations of the loop body for each loop construct, which crucially depends on the loop condition and could be executed in an arbitrary number of iterations. Fig. 3 visualizes ReGuS's solution to this challenge. It tackles the exponential growth of program search space using a combination of three key ideas:

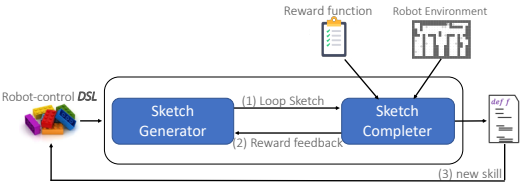



Fig. 3. Overview of the reward-guided synthesis (ReGuS) framework for robotics tasks.

***(1) Hierarchical Synthesis of Loop Sketches:*** ReGuS is a two-stage synthesis technique. At the high level, a sketch generator synthesizes a loop sketch that only captures the "skeletal" loop structures (sequential or nested) while leaving the loop bodies as "holes", *yet to be determined*. Given a loop sketch, the low-level loop sketch completer synthesizes a complete program and provides feedback to guide the sketch generator. The feedback is the highest reward among all the programs searched in the sketch completion phase. The high-level sketch generator, implemented as a variant of Monte Carlo Tree Search, uses the feedback to prioritize a search path toward the best quality sketch for low-level sketch completion. While leveraging reward feedback for program search tree expansion has already been explored in related works such as [2, 4], the novelty of ReGuS is to direct high-level sketch generation by backpropagating rewards obtained from low-level sketch completion. The prior works [2, 4] assume access to input-output examples and thus can evaluate candidate programs based on the inputs and compare the output similarity to generate rich reward information. However, in sparse-reward settings without input-output examples (aka user demonstrations), if we were to apply the Monte Carlo reward estimation to synthesize an entire program (including loops, conditionals, and agent actions) rather than just a loop sketch, the likelihood of reaching a complete program with a nonzero averaged reward is low. This reward feedback mechanism also differs ReGuS from existing two-stage program synthesis techniques [19, 20, 51, 53] that only enumerate (example-consistent) sketches to be filled in without learning from the sketch completion process.

***(2) On-demand Synthesis of Conditional Statements:*** In the loop sketch completion phase, our method guides top-down enumerative search by executing partially synthesized programs to prioritize exploring partial programs with the highest reward performance (regularized by their structure cost). During this evaluation, ReGuS lifts concrete states before and after an action $c$ into presumed precondition $\varphi$ and postcondition $\psi$ by state abstraction. ReGuS uses inferred preconditions and postconditions to synthesize conditional statements on an as-needed basis to vastly reduce the program search space. If there is a disagreement between the inferred precondition $\varphi$ and the current state $\sigma$ before executing $c$ (i.e., $\sigma$ does not satisfy $\varphi$), ReGuS explores the option of appending a conditional statement before $c$ with the branch condition evaluating to true for $\sigma$ to handle the newly encountered state $\sigma$. ReGuS merges the two branches of this conditional statement at a common point with consistent postconditions (Sec. 4.3).

***(3) Curriculum Synthesis of Procedural Calls:*** Synthesizing programs for robot control has the additional benefit of enabling knowledge transfer across related tasks via reusable procedures. For various tasks defined in similar environments, ReGuS ranks the tasks according to their complexity e.g. the number of objects to manipulate, and adds programs synthesized to solve simpler tasks as new skills via callable procedures so that they can be building blocks to constitute sophisticated programs for more complex tasks. While curriculum synthesis can principally be incorporated into existing programmatic RL approaches, these methods involve either adapting deep RL algorithms to steer program search or synthesizing a program to imitate a trained RL agent. Consequently, the integration must also extend to the deep RL algorithms, introducing additional complexity.

**Evaluation.** We evaluated ReGuS on several application domains including classic discrete environments, highway autonomous driving, locomotion control of a quadruped ant robot to explore goals in mazes of complex shapes, and object manipulation through a 7-DoF Fetch Mobile Manipulator. All of our benchmarks are equipped with sparse rewards and involve inducing programs with complex control structures to solve long-horizon tasks. The results demonstrate that ReGuS significantly outperforms standard program synthesis baselines and state-of-the-art reinforcement learning techniques.

**Contributions.** To summarize, this paper makes the following key contributions:

- We propose a reward-guided synthesis (ReGuS) technique for robot learning. ReGuS overcomes the exploration challenges in long-horizon sparse-reward robot tasks by synthesizing programs that incorporate state-conditioned loops, conditionals, and procedural calls. These programming constructs act as strong inductive biases, enabling exploration space compression and the generation of generalizable robot controllers for multi-stage and procedural tasks.
- We evaluate ReGuS on challenging robot tasks needing logical deduction and generalization and show that it outperforms program synthesis and deep RL baselines by large margins.

## 2 OVERVIEW

**Why Reward-guided Synthesis?** ReGuS uses reward functions over program traces to define the intended program behavior. The synthesis problem is framed as generating a program that maximizes the expected reward over all potential initial states. Reward functions are black boxes to the synthesis algorithm. ReGuS does not rely on examples or counterexamples from specifications. The motivation behind this choice stems from the significant cost required to obtain example robot behavior (demonstrations) from human experts in robotic tasks [15, 26]. More importantly, synthesizing programs from demonstrated robot behaviors can be infeasible when robots have only partial visibility into the world. Consider a maze environment in Fig. 4 where the agent begins at the bottom facing east. There are two target positions (markers). In each episode, one of the targets is randomly selected as the agent's goal to reach. Although it is feasible to collect demonstrations by determining the shortest paths to the targets using an A* planner, programming by example would fail to identify a valid program based on these demonstrations. This is because, at the starting position, due to the agent's limited field of view, a synthesized program cannot condition the invisible goal position to instruct the agent whether it should go forward or turn right.

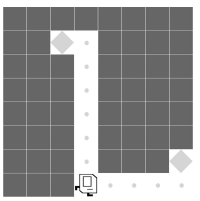
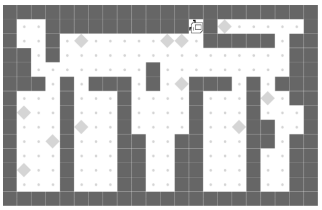


Fig. 4. Partial Observability.

**Sparse Reward Functions.** Synthesizing programs by rewards presents a significant challenge in constructing informative reward functions. One may be tempted to use dense reward functions as they offer frequent task-related information. For example, Fig. 5a presents a simplified household environment `CleanHouse` for home-assisted robots where the goal is to navigate the house to pick up trash cans (markers) placed adjacent to walls. The agent starts at the entry and must return



(a) CleanHouse environment.

```
while (not present(marker)) {
    while (not present(marker)) {
        if (leftIsClear()) turnLeft();
        if (not frontIsClear()) turnRight();
        move();
    } pickUp(marker);
}; pickUp(marker);
```

(b) CleanHouse controller.

Fig. 5. Fig. 5a depicts the CleanHouse environment where the agent starts at the entrance of the house. Fig. 5b is a program that can pick up all trash cans and return them to the dumpster from all possible initial states.

to one side of the entry where two dumpsters (green and blue) are located, with two markers in this area. A dense reward function may be inversely proportional to the robot's distance from the dumpster location. However, using dense rewards can lead to a learning agent becoming trapped in local minima [23]. For example, rewarding the robot to be close to the dumpster location may inadvertently hinder its ability to collect trash cans at the far end of the house. ReGuS targets sparse rewards that are given either after the robot completes the entire task or sporadically when it achieves critical steps. In `CleanHouse`, by the end of a control episode, the agent receives a reward based on the percentage of markers picked up and whether the dumpster location is reached.

### 2.1 Program and Domain-Specific Language

In this work, we focus on application domains for robot navigation and manipulation. We assume that a robot takes an object-centric view as input (formally defined in Sec. 3) obtained by e.g. using object detection [33, 34, 42] or discovery [13, 31] methods. These models return the detected objects in the view and their associated attributes e.g. object class, positions, and velocities.

ReGuS synthesizes robot-control programs based on a generic DSL depicted in Fig. 6, powered by user-defined state abstraction predicates. Based on the observation that a robot typically needs to engage with various environmental objects in a navigation or manipulation task, such as grasping objects and detecting nearby obstacles, our design principle is to specify state abstraction predicates as perception functions aimed at describing spatial relationships and physical orientations among environmental objects (including the robot itself). For example, in a manipulation task of grasping a block on a table, a predicate for ascertaining whether the robot's gripper is positioned above the block is important. For the `CleanHouse` task, we define state abstraction predicates as:

$$\texttt{frontIsClear}() : \neg\exists o.\, \texttt{front}(self,\, o) \qquad \texttt{leftIsClear}() : \neg\exists o.\, \texttt{left}(self,\, o)$$
$$\texttt{rightIsClear}() : \neg\exists o.\, \texttt{right}(self,\, o) \qquad \texttt{present(o)} : \texttt{front}(self,\, o)$$

where $self$ represents the robot itself and $o$ is an arbitrary object in the scene. State abstraction predicates as such enable the robot to locate desired objects or obstacles to construct a higher-level representation of its environment. This, in turn, facilitates the search for controllers capable of updating the environment representations to achieve a desired goal state.

$$\begin{aligned}
\text{State Abstraction } h &::= \text{Domain-dependent predicates}\\
\text{Control Action } c &::= \text{Domain-dependent skills}\\
\text{Condition } b &::= h \mid \text{not } h\\
\text{Statement } S &::= \textbf{while } (b)\ \{S\} \mid \textbf{if } (b)\ S_1\ \textbf{else}\ S_2 \mid S_1; S_2 \mid c
\end{aligned}$$

Fig. 6. Our DSL for robot-control program synthesis.

In the DSL, a control action $c$ is an application of a low-level *skill* that refers to *task-agnostic* capabilities of a robot. Skills add a layer of speed and steering control on top of the robot's low-level continuous state space, enabling robot navigation at a desired speed or object manipulation in a specific orientation. These skills can be derived from either robot APIs (primitive actions) or pre-trained RL controllers. Conceptually, skills are modular and reusable, and can be likened to building blocks or subroutines that contribute to the overall control policy of the robot in diverse contexts. For instance, in the context of the `CleanHouse` task, skills such as object manipulation `pickup(marker)`, navigation `move()`, `turnLeft()` and `turnRight()` facilitate picking up objects, forward movement, and changing the orientation.

Our DSL allows for standard language constructs, such as state-conditioned loops and conditional statements, which serve as powerful inductive biases to address the exploration challenges in handling long-horizon and multi-stage robot tasks. A program that achieves the full reward for `CleanHouse` is depicted in Fig. 5b. It uses the state abstraction predicates to extract state conditions (e.g. `present(marker)`) and based on these state conditions invokes a sequence of control actions for navigation (e.g. `turnRight`) and object manipulation (e.g. `pickUp(marker)`).

## 2.2 The Synthesis Procedure

ReGuS handles the exponentially expanding program search space for the DSL in Fig. 6 by leveraging the following novel strategies:

**Hierarchical Synthesis of Loop Sketches.** ReGuS is a *hierarchical* program synthesis procedure, as visualized in Fig. 3. The high-level sketch generator searches a "skeletal" loop sketch comprising solely of sequential or nested loop structures, where the loop bodies are represented as placeholders yet to be completed. For the `CleanHouse` example, a loop sketch could be

$$\textbf{while}\ (\texttt{not present(marker)})\ \{\ ??_{C1}\ \};??_{C2} \tag{1}$$

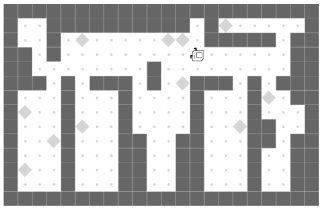
where $??_C$ represents a missing hole. The low-level sketch completer uses top-down enumeration to fill out a loop sketch. To best use the reward information, it executes partially synthesized programs, prioritizing exploration of those demonstrating high reward performance. The sketch completer feeds back the highest reward attained by completing the loop sketch to the sketch generator. The sketch generator, implemented as a variant of Monte Carlo Tree Search, uses the reward feedback to improve its future sketch selections. ReGuS demonstrates exceptional performance, particularly when task rewards are sporadically provided upon the robot's completion of critical steps. The directed high-level sketch generation procedure effectively utilizes the reward structures to identify subtasks to be completed within separate loops. For example, in `CleanHouse`, ReGuS discovers a program on top of the loop sketch in Equation 1 that can clean at least one marker (trash can) out of the total markers in a house, and the other single-loop sketches with a different loop condition hardly lead to a nonzero-reward program. ReGuS gives priority to loop sketches akin to Equation 1 and suggests the following sketch for generating a program capable of iteratively collecting all the markers:

$$\textbf{while}\ (\texttt{not present(marker)})\ \{\textbf{while}\ (\texttt{not present(marker)})\ \{\ ??_{C1}\ \};\ ??_{C2}\};\ ??_{C3} \tag{2}$$

**On-demand Synthesis of Conditional Statements.** ReGuS *lazily* synthesizes conditional statements on an as-needed basis to reduce the search space for sketch completion. It infers pre- and postconditions of each statement in a synthesized program by abstracting the states encountered before and after executing the statement using the state abstraction predicates. A conditional statement is inserted prior to an action statement when the state encountered before executing the action does not meet the inferred pre-condition of the action, indicating an unanticipated scenario.

```
while (not present(marker)) {
    while (not present(marker)) {
        { ¬ leftIsClear() ∧frontIsClear()
        ∧ ¬ present(marker) }
        move();
    }; ??C2
}; ??C3
```

(a) A partial program of the CleanHouse controller.



(b) Executing the program in Fig. 7a in CleanHouse.

Fig. 7. Fig. 7a is a partial program enumerated by ReGuS. The inferred precondition of the move action is depicted in blue. Fig. 7b is the execution result of the partial program in Fig. 7a from the state in Fig. 5a.

For example, Fig. 7a is a partial program enumerated by ReGuS to solve the CleanHouse task. The agent is initially at the entrance of the house (Fig. 5a), and the inferred precondition of move is drawn in blue. Particularly, the precondition asserts that in any state before executing move, the left to the controlled agent is not clear (i.e. there is a wall). The precondition is valid until the state in Fig. 7b is encountered in a subsequent inner loop iteration. The precondition is not satisfied as the left of the agent in Fig. 7b is clear. At this point, the axiomatic semantics of the partial program in Fig. 7a and the *actual* execution results in Fig. 7b are out-of-sync. ReGuS then explores the option of synthesizing a new conditional statement to synchronize the precondition and the actual state before executing move by constructing a new sketch depicted in Fig. 8. With on-demand conditional statement synthesis, ReGuS eventually finds the desired program in Fig. 5b.

```
while (not present(marker)) {
    while (not present(marker)) {
        if (leftIsClear()) { ??S }
        { ¬ leftIsClear() ∧frontIsClear()
        ∧ ¬ present(marker) }
        move();
    }; ??C2
}; ??C3
```

Fig. 8. ReGus adds a conditional statement to Fig. 7a on-demand.

**Curriculum Synthesis of Procedural Calls.** ReGuS uses procedures to enable knowledge transfer across related tasks in similar environments. ReGuS incorporates programs learned for simpler tasks into DSLs, treating them as procedures that can be utilized as skills. As the library of skills expands, the learned procedures serve to compress the program search space, expediting the synthesis of larger programs (Section 4.5). For example, the CleanHouse procedure (Fig. 5b) can serve as a skill that a program can iteratively invoke in an outer loop to clean each floor of a multi-floor house.

## 3 PROBLEM FORMULATION

We define a robot-control environment as a tuple $(\Lambda, d, O, C)$ where $\Lambda$ is a finite set of object types, the map $d : \Lambda \to N$ defines the dimensionality of the real-valued feature vector for each type, and $O$ is an object set where each object has a type drawn from $\Lambda$. As illustrated in Sec. 2.1, $C$ is a finite set of task-agnostic skills for low-level control in the continuous state space. A skill $f(\lambda_0, \ldots, \lambda_{M-1}) \in C$, e.g. pickup$(\cdot)$, have typed parameters $(\lambda_0, \ldots, \lambda_{M-1})$ where $M \geq 0$ and $\lambda_i \in \Lambda$. We frame reward-guided synthesis (ReGuS) as the problem of synthesizing domain-specific programs to solve robot-control tasks formalized as Markov decision processes (MDPs).

**Markov Decision Process (MDP).** Given an environment $(\Lambda, d, O, C)$, a robot task $e$ is modeled as an MDP in a structure $e = (\mathcal{S}, C, \mathcal{T} : \mathcal{S} \times C \times \mathcal{S} \to [0, 1], \mu(\mathcal{S}_0), \mathcal{R} : \{\mathcal{S} \times C \to \mathbb{R}\})$. $\mathcal{S}$ is an infinite set of continuous states. In each state $\sigma \in \mathcal{S}$, every object $o \in O$ is mapped to a feature vector in $\mathbb{R}^{d(\text{type}(o))}$. The action space $C$ is induced by the objects $O$ and the low-level skills $C$ where an action $c := f(o_0, \ldots, o_{M-1})$ applies a skill $f(\lambda_0, \ldots, \lambda_{M-1}) \in C$ to objects $o_0, \ldots, o_{M-1}$

$$\frac{\sigma' \sim \mathcal{T}(\cdot|\,\sigma, c) \quad r = \mathcal{R}(\sigma', c)}{\langle \sigma,\ c\rangle \Downarrow \sigma',\ r} \qquad \frac{\langle \sigma,\ S_1\rangle \Downarrow \sigma',\ r \quad \langle \sigma',\ S_2\rangle \Downarrow \sigma'',\ r'}{\langle \sigma,\ S_1; S_2\rangle \Downarrow \sigma'',\ r + r'} \qquad \frac{b(\sigma) \quad \langle \sigma,\ S_1\rangle \Downarrow \sigma',\ r}{\langle \sigma,\ \textbf{if}\ (b)\ S_1\ \textbf{else}\ S_2\rangle \Downarrow \sigma',\ r}$$

$$\frac{\neg b(\sigma) \quad \langle \sigma,\ S_2\rangle \Downarrow \sigma',\ r}{\langle \sigma,\ \textbf{if}\ (b)\ S_1\ \textbf{else}\ S_2\rangle \Downarrow \sigma',\ r} \qquad \frac{b(\sigma) \quad \langle \sigma,\ S\rangle \Downarrow \sigma',\ r \quad \langle \sigma',\ \textbf{while}\ (b)\ S\rangle \Downarrow \sigma'',\ r'}{\langle \sigma,\ \textbf{while}\ (b)\ S\rangle \Downarrow \sigma'',\ r + r'} \qquad \frac{\neg b(\sigma)}{\langle \sigma,\ \textbf{while}\ (b)\ S\rangle \Downarrow \sigma,\ \mathbf{0}}$$

Fig. 9. DSL operational semantics and reward evaluation.

and the objects must have types matching the typed parameters $\lambda_0, \ldots, \lambda_{M-1}$ of skill $f$, such as `pickup(marker)`. We use type information to filter out unreasonable control actions, such as `pickup(door)`. $\mathcal{T}$ captures the state transition probabilities. $\mathcal{R}$ denotes the reward function. We assume that any initial state $\sigma_0 \in \mathcal{S}_0$ of $e$ is sampled from a state distribution $\mu(\mathcal{S}_0)$. It is important to note that $\mathcal{T}$, $\mathcal{R}$, and $\mu(\mathcal{S}_0)$ are *unknown* to ReGuS.

**State Abstraction.** A predicate $h := g(o_0, \ldots, o_{p-1})$ is a binary state classifier $g(O^p) : S \longrightarrow$ {true, false} characterized by an ordered list of types $(\lambda_0, \ldots, \lambda_{p-1})$ where $g(O^p)$ is defined only when each object $o_i$ has type $\lambda_i$, e.g. `present(marker)`.

**Operational Semantics with Rewards.** ReGuS synthesizes programs based on the DSL introduced in Fig. 6. We outline the DSL operational semantics in Fig. 9. A distinct feature is that the DSL operational semantics tracks the cumulative reward achieved during program execution. The rules are in the shape of $\langle \sigma,\ S\rangle \Downarrow \sigma',\ r$. It specifies the semantics of executing a program $S$ in the DSL from a state $\sigma \in \mathcal{S}$. The resulting state of the execution is $\sigma'$. The execution receives a reward of $r$ from the environment. The Action rule is defined for a control action $c$. The action $c$ is executed upon a state $\sigma$ and the MDP transits to another state $\sigma'$ as the effect of the action according to the stochastic transition model $\mathcal{T}(\sigma'|\sigma, c)$. The immediate reward of taking action $c$ at $\sigma$ is given by the reward function $r = \mathcal{R}(\sigma', c)$.

**Program Synthesis Objective.** The objective of ReGuS is searching a robot-control program $\pi^*$ in the DSL in Fig. 6 for a task MDP $e$ such that

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\sigma_0 \sim \mu,\ \langle \sigma_0,\ \pi\rangle \Downarrow \sigma_f,\ r_f}\left[\ r_f\ \right]$$

where $\sigma_f$ and $r_f$ are the resulting state and final reward by executing $\pi$ in the task MDP from a state $\sigma_0$ sampled from the initial state distribution $\mu$ of $e$.

## 4 REWARD-GUIDED PROGRAM SYNTHESIS

We present the core algorithms for reward-guided synthesis. We first provide ReGuS's sketch language, then present the top-level synthesis algorithm, and then describe its key components.

### 4.1 Sketch Language

As outlined in the ReGuS framework in Fig. 3, ReGuS employs a hierarchical synthesis procedure. At the high level, ReGuS enumerates only "skeletal" programs. A skeletal program $??_S$ is a *loop sketch* that only contains sequential or nested loop structures with their loop bodies as "holes" *yet to be determined*. We rewrite the grammar in Fig. 6 to describe the syntax of loop sketches:

$$\text{Loop Sketch } ??_S \ ::= ??_C;\ \textbf{while}\ b\ \{??_S\};\ ??_S \mid ??_C \tag{3}$$

where $??_C$ is a code block sketch, $??_c$ is an unknown control action subject to enumerative search, and `skip` is a special command used by ReGuS to complete a block:

$$\text{Block } ??_C \ ::= ??_c;\ ??_C \mid \texttt{skip} \tag{4}$$

### 4.2 Top-level Algorithm

**Algorithm 1** Reward-guided Synthesis (ReGuS)

1: **procedure** ReGuS($e$: MDP, $\xi$: reward threshold, $N$: seeds)
2: $\varepsilon,\ r_\varepsilon,\ r_{best},\ \mathbb{Q} \leftarrow ??_{\mathbb{S}},\ -\infty,\ -\infty,\ \{\ \}$
3: **while** $r_{best} < \xi$ **do**
4: $\varepsilon^* \leftarrow \text{ReGuS}_{Sket}(\varepsilon,\ r_\varepsilon)$
5: **if** $\varepsilon^* \notin \mathbb{Q}$ **then** $\mathbb{Q}[\varepsilon^*] \leftarrow \emptyset$
6: $r_{\varepsilon^*} \leftarrow \text{ReGuS}_{Prog}(e,\ \varepsilon^*,\ \mathbb{Q}[\varepsilon^*],\ \xi,\ N)$
7: **if** $r_{\varepsilon^*} > r_{best}$ **then** $r_{best} \leftarrow r_{\varepsilon^*}$
8: $\varepsilon,\ r_\varepsilon \leftarrow \varepsilon^*,\ r_{\varepsilon^*}$

The top-level ReGuS algorithm is outlined in Algorithm 1. The algorithm takes as input a robot MDP $e$, a reward threshold $\xi$, and $N$ random seeds with each seed initializing a random initial MDP state (e.g. randomized object positions). The $N$ random seeds are used to create **diverse MDP instances** for robot learning. The goal is to synthesize a program whose averaged reward across all potential initial states of $e$ meets $\xi$. The reward threshold $\xi$ should be set as the maximum task reward. As the reward scale can vary across tasks, ReGuS allows the user to specify $\xi$. For example, for tasks with binary sparse rewards that only signal if a task succeeds (1.0) or fails (0.0), $\xi$ is set to 1.0.

The algorithm repeatedly finds a loop sketch $\varepsilon^*$ derived from the start symbol $??_{\mathcal{S}}$ via the sketch generator $\text{ReGuS}_{Sket}$ (Line 4), following the loop sketch grammar expressed in Equation 3. Each loop sketch $\varepsilon^*$ is equipped with a unique program search queue $\mathbb{Q}[\varepsilon^*]$. The sketch completer $\text{ReGuS}_{Prog}$ (Line 6) attempts to fill the missing code blocks $??_C$ in $\varepsilon^*$, following the code block grammar expressed in Equation 4, to search a program that has the best-averaged reward among all possible executions from the initial states of $e$. In Line 4, $\text{ReGuS}_{Sket}$ leverages the reward feedback from sketch completion to prioritize the search order of highly likely loop sketches. In the following discussion, we explain $\text{ReGuS}_{Sket}$ and $\text{ReGuS}_{Prog}$ in more detail.

### 4.3 Low-level Loop Sketch Completion

In this section, we illustrate the core ideas of sketch completion $\text{ReGuS}_{Prog}$ using the `FourCorners` task taken from [48]. The goal of the agent is to put a marker at each of the four corners of a room depicted in Fig. 11. The agent gets a 0.25 reward for each marker put in a corner. However, if a marker is placed in other positions, the final reward is 0. For simplicity, we assume the agent starts beside one of the walls. The state abstraction and actions for this example in the DSL (Fig. 6) are:

$$\begin{aligned} \text{Object } o &\in \{\ \texttt{marker}\ \} \\ \text{State Abstraction } h &::= \texttt{present}(o), \texttt{leftIsClear}(), \texttt{rightIsClear}(), \texttt{frontIsClear}() \\ \text{Control Action } c &::= \texttt{put}(o), \texttt{pickUp}(o), \texttt{turnLeft}(), \texttt{turnRight}(), \texttt{move}() \end{aligned} \tag{5}$$

We further assume that the loop sketch given by the high-level sketch generator is:

$$\textbf{while}\ (\texttt{not present(marker)})\ \{\ ??_{C1}\ \};\ ??_{C2} \tag{6}$$

where $??_{C1}$ and $??_{C2}$ are missing code blocks in the sketch.

The sketch completion algorithm $\text{ReGuS}_{Prog}$ is outlined in Algorithm. 2. It takes a task MDP $e$, a loop sketch $\varepsilon$, a program search queue $Q$ for $\varepsilon$, the reward threshold $\xi$ and $N$ random seeds for creating diverse MDP instances for synthesizing a generalizable program. It returns a program $\varepsilon_{best}$ empirically yielding the best reward $r_{best}$ averaged over all potential instances of $e$. $\text{ReGuS}_{Prog}$ performs an execution-guided top-down enumerative search and on-demand synthesis of conditional statements. As standard in top-down search, it utilizes the notion of partial programs [18, 19], which can be thought of as an abstract syntax tree where some of the nodes are labeled with non-terminals $??_C$ to be expanded later. By evaluating each partial program generated during synthesis in robot

**Algorithm 2** Reward-guided Loop Sketch Completion (ReGuS$_{Prog}$).

1: **procedure** ReGuS$_{PROG}$ ($e$: MDP, $\varepsilon$: sketch, $Q$: search queue, $\xi$: reward threshold, $N$: seeds)
2: $r_{best} \leftarrow -\infty$ ▷ $i$ holds the current random seed
3: **if** $Q = \emptyset$ **then** $Q$.ADD($\{\varepsilon,\ 0,\ \bot,\ -\infty,\ -\infty\}$) ▷ If $Q \neq \emptyset$, continue the search from $Q$
4: **while** $Q \neq \emptyset \wedge$ (*max search budget not reached*) **do**
5: $\varepsilon,\ i,\ \sigma_f,\ r_f,\ r_{avg} \leftarrow Q$.REMOVE$\big(\arg\max_{\{\varepsilon,\ i,\ \sigma_f,\ r_f,\ r_{avg}\} \in Q} r_{avg} - \lambda \cdot \text{COST}\ (\varepsilon)\big)$
6: **if** $r_{avg} \geq r_{best}$ **then** $r_{best} \leftarrow r_{avg}$
7: **if** $i \geq N$ **then** ***break*** ▷ $\epsilon$ has surpassed the reward threshold across all $N$ seeds
8: **if** $\sigma_f \neq (\bot, \_) \wedge r_f \geq \xi$ **then**
9: $\sigma \leftarrow$ RESET$(e, i+1)$ ▷ $\varepsilon'$ meets the reward threshold on seed $i$; search on seed $i+1$
10: $\varepsilon',\ \sigma_f,\ r_f \leftarrow$ EXEC$(\sigma,\ \varepsilon)$ ▷ Apply synthesis rules in Fig. 10
11: $Q$.ADD($\{\varepsilon',\ i+1,\ \sigma_f,\ r_f,\ \text{EVAL}(\varepsilon)\}$) ▷ Eval avg. reward of $\varepsilon'$ across all $N$ seeds
12: **else if** $\sigma_f = (\bot, ??_{\mathbb{A}})$ **then** ▷ $\varepsilon$ is a partial program with nonterminal $\mathbb{A}$
13: **for** *each produce rule* $l$ `for` $\mathbb{A}$ **do**
14: $\varepsilon' \leftarrow$ EXPAND$(\varepsilon, \mathbb{A}, l)$ ▷ Expand $\mathbb{A}$ using production rules
15: $\sigma \leftarrow$ RESET $(e, i)$ ▷ Reset $e$ with random seed $i$ to an initial state.
16: $\varepsilon'',\ \sigma_f,\ r_f \leftarrow$ EXEC$(\sigma,\ \varepsilon')$ ▷ Apply synthesis rules in Fig. 10
17: $Q$.ADD($\{\varepsilon'',\ i,\ \sigma_f,\ r_f,\ \text{EVAL}(\varepsilon'')\}$) ▷ Eval avg. reward of $\varepsilon''$ across all $N$ seeds
18: **return** $r_{best}$

environments, it uses reward information to prioritize exploring program search directions leading to the highest reward performance. We extend the DSL operational semantics in Fig. 9 to support evaluating partial programs:

$$\langle \sigma, ??_C \rangle \Downarrow (\bot, ??_C),\ 0$$

where $??_C$ is a nonterminal symbol in the partial program reached during execution that prevents it from further proceeding and $\bot$ is an indicator denoting that a program cannot be executed beyond this point. The reward is 0 since a nonterminal cannot be evaluated. For example, given a partial program (`move()`; $??_{C1}$), the resulting state is ($\bot$, $??_{C1}$) i.e. the execution cannot resume beyond $??_{C1}$. The reward of this partial program is the reward of taking `move()` in the task MDP.

ReGuS$_{Prog}$ uses the priority queue $Q$ to manage partially synthesized programs derived from the provided loop sketch $\varepsilon$. Each element in $Q$ is a tuple comprising a (partial) program $\varepsilon$, the current random seed $i$ used in the search for $\varepsilon$, final state $\sigma_f$ and final reward $r_f$ obtained by executing $\varepsilon$ in the MDP instance initialized by random seed $i$, averaged reward $r_{avg}$ acquired by evaluating $\varepsilon$ *across all N MDP instances induced by the N random seeds.* $Q$ ranks partial programs by their reward performance $r_{avg}$ regularized by the structural cost COST $(\varepsilon)$, as indicated in Line 5 of Algorithm 2. Structure costs bias ReGuS to generate simpler programs that are more likely to generalize. Let each production rule of the DSL have a non-negative cost defined as `cost`$(l)$. The structural cost of a program $\varepsilon$ is `cost`$(\varepsilon) = \sum_{l \in \mathcal{L}(\varepsilon)}$ `cost`$(l)$, where $\mathcal{L}(\varepsilon)$ is the multiset of production rules used to construct $\varepsilon$. The hyper-parameter $\lambda$ dictates the balance between structure cost and rewards.

In each iteration of the loop in Algorithm 2 lines 4–17, ReGuS$_{Prog}$ dequeues a partial program $\varepsilon$ in $Q$ (Line 5). If the execution of $\varepsilon$ in the MDP instance induced by random seed $i$ did not encounter any nonterminal symbol and the final reward obtained by the execution meets the reward threshold (Line 8), the algorithm goes on searching for $\varepsilon$ using a new initial MDP state $\sigma$ induced by the next random seed $i+1$ (Line 9). At line 10, the EXEC procedure executes $\varepsilon$ from $\sigma$ to obtain its final state $\sigma_f$ and reward $r_f$ in the new MDP instance. It may also return an updated version of $\varepsilon$ through on-demand conditional statement synthesis. The formalization of the EXEC procedure

$$\textsc{Action-Hoare}\ \frac{\varphi(\sigma)\ \text{or}\ \varphi \equiv \mathit{False} \qquad \langle \sigma, c\rangle \Downarrow \sigma', r}{\sigma, \{\varphi\}c\{\psi\} \triangleright \{\varphi \sqcup \alpha(\sigma)\}\ c\ \{\psi \sqcup \alpha(\sigma')\}, \sigma', r}$$

$$\textsc{Action-LazyBranch}\ \frac{\neg\varphi(\sigma) \qquad b \in \mathcal{H} \qquad b(\sigma)}{\sigma, \{\varphi\}c\{\psi\} \triangleright \textbf{if}(b)\ \{??_S\};\ \{\varphi\}c\{\psi\},\ (\bot, ??_S),\ \mathbf{0}}$$

$$\textsc{Compound-Hoare}\ \frac{\sigma,\ S;\triangleright S',\ \sigma',\ r \qquad \sigma' \neq (\bot, \_)}{\sigma,\ \{\varphi\}\ S\ \{\psi\} \triangleright \{\varphi \sqcup \alpha(\sigma)\}\ S\ \{\psi \sqcup \alpha(\sigma')\},\ \sigma',\ r}$$

$$\textsc{Comp1}\ \frac{\sigma,\ S_1;\triangleright S_1',\ \sigma',\ r \qquad \sigma' = (\bot, \_)}{\sigma,\ S_1;\ S_2 \triangleright S_1';\ S_2,\ \sigma',\ r}$$

$$\textsc{Comp2}\ \frac{\sigma,\ S_1;\triangleright S_1',\ \sigma',\ r_1 \qquad \sigma',\ S_2 \triangleright S_2',\ \sigma'',\ r_2}{\sigma,\ S_1;\ S_2 \triangleright S_1';\ S_2',\ \sigma'',\ r_1 + r_2}$$

$$\textsc{While-Skip}\ \frac{\neg b(\sigma)}{\sigma,\ \textbf{while}\ (b)\ \{S\} \triangleright \textbf{while}\ (b)\ \{S\},\ \sigma,\ \mathbf{0}}$$

$$\textsc{While-Sketch1}\ \frac{\sigma,\ S \triangleright S',\ \sigma',\ r \qquad b(\sigma) \qquad \sigma' = (\bot, \_)}{\sigma,\ \textbf{while}\ (b)\ \{S\} \triangleright \textbf{while}\ (b)\ \{S'\},\ \sigma',\ r}$$

$$\textsc{While-Sketch2}\ \frac{\sigma,\ S \triangleright S,\ \sigma',\ r \qquad \sigma' \neq (\bot, \_) \qquad b(\sigma) \qquad \sigma',\ \textbf{while}\ (b)\ \{S\} \triangleright \textbf{while}\ (b)\ \{S'\},\ \sigma'', r'}{\sigma;\ \textbf{while}\ (b)\ \{S\} \triangleright \textbf{while}\ (b)\ \{S'\},\ \sigma'',\ r + r'}$$

$$\textsc{If-Erase}\ \frac{S_1 \equiv c_1;\ ??_S \qquad c_1 = c_2 \qquad \langle \sigma, c_2\rangle \Downarrow \sigma', r}{\sigma,\ \textbf{if}\ (b)\ \{S_1\};\ \{\varphi_2\}\ c_2\ \{\psi_2\} \triangleright \{\alpha(\sigma) \sqcup \varphi_2\}\ c_2\ \{\alpha(\sigma') \sqcup \psi_2\},\ \sigma',\ r}$$

$$\textsc{If-Merge}\ \frac{S_1 \equiv s_1^1;\ s_2^1;\ \cdots;\ \{\varphi_k^1\}\ s_k^1\ \{\psi_k^1\};\ \texttt{skip} \qquad (S = \texttt{join}(S_1, S_2)) \neq \bot \qquad \sigma,\ S \triangleright S',\ \sigma',\ r}{\sigma,\ \textbf{if}\ (b)\ \{S_1\};\ S_2 \triangleright S',\ \sigma',\ r}$$

$$\textsc{If-Merge-Fail}\ \frac{S_1 \equiv s_1^1;\ s_2^1;\ \cdots;\ \{\varphi_k^1\}\ s_k^1\ \{\psi_k^1\};\ \texttt{skip} \qquad \texttt{join}(S_1, S_2) = \bot}{\sigma,\ \textbf{if}\ (b)\ \{S_1\};\ S_2 \triangleright \bot,\ \bot,\ 0}$$

$$\textsc{If-T}\ \frac{b(\sigma) \qquad \sigma,\ S_1 \triangleright S_1',\ \sigma',\ r}{\sigma,\ \textbf{if}\ (b)\ S_1\ \textbf{else}\ S_2 \triangleright \textbf{if}\ (b)\ S_1'\ \textbf{else}\ S_2,\ \sigma', r}$$

$$\textsc{If-F}\ \frac{\neg b(\sigma) \qquad \sigma,\ S_2 \triangleright S_2',\ \sigma',\ r}{\sigma,\ \textbf{if}\ (b)\ S_1\ \textbf{else}\ S_2 \triangleright \textbf{if}\ (b)\ S_1\ \textbf{else}\ S_2',\ \sigma',\ r}$$

Fig. 10. The Exec Procedure: $\sigma,\ \varepsilon \triangleright \varepsilon',\ \sigma',\ r$. The synthesis rules are based on the syntax of $\varepsilon$. $\Downarrow$ defines the semantics (Fig. 9). $\alpha$ abstracts a concrete state using state abstraction predicates associated with the DSL.

will be detailed shortly. The program and its execution results are enqueued for further processing at Line 11. The search process in ReGuS$_{Prog}$ continues until a synthesized program achieves the maximum reward on all $N$ random seeds, terminating the search (Line 7).

*Example 1.* In the maze environment depicted in Fig. 4, consider an instance where the goal position is near the bottom. ReGuS$_{Prog}$ could generate a program that guides the agent to move forward and turn left whenever obstructed by a wall to reach the goal. However, this program does not generalize to the other instances with the goal position at the top. For such cases, Exec inserts an if-statement into the program by need, enabling ReGuS$_{Prog}$ to eventually find a program that instructs the agent to turn right when the path to the right is clear for reaching the top goal.

If the execution of a partial program $\varepsilon$ on the MDP instance induced by seed $i$ does encounter a nonterminal symbol (Line 12), ReGuS$_{Prog}$ expands the nonterminal by enumerating all suitable production rules for the missing hole (Line 14). For each expanded program $\varepsilon'$, ReGuS$_{Prog}$ uses the Exec procedure to execute $\varepsilon'$ in the MDP instance induced by the current seed $i$ (Line 16), then adds $\varepsilon'$ and its final state and reward back to $Q$ for further processing (Line 17).

**The Exec Procedure.** We describe Exec using the synthesis rules of the following shape:

$$\sigma,\ \varepsilon \triangleright \varepsilon',\ \sigma_f,\ r_f$$

```
while (not present(marker)) {
 {leftIsClear() ∧ frontIsClear()∧¬rightIsClear() ∧ ¬present(marker)}
 move();
 {leftIsClear() ∧ frontIsClear()∧¬rightIsClear ∧ ¬present(marker)}
}; ??C2
```

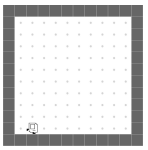


Fig. 11. Pre and postcondition inference of `while (not present(marker)) {move; }; ??`$_{C_2}$ in FourCorners.

```
while (not present(marker)) {
 {leftIsClear() ∧ frontIsClear()∧¬rightIsClear() ∧ ¬present(marker)}
 move();
 {leftIsClear()∧¬rightIsClear() ∧ ¬present(marker)}
}; ??C2
```

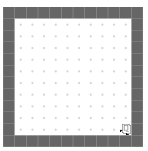


Fig. 12. Widen the postconditon of move in Fig. 11 to include an unseen state resulting from executing move.

where $\varepsilon$ represents a (partial) program, and $\sigma$ is the state from which $\varepsilon$ is to be executed. The Exec procedure accomplishes three tasks: (1) it evaluates $\varepsilon$ using the DSL semantics $\langle \sigma, S \rangle \Downarrow \sigma_f, r_f$ (as depicted in Fig. 9), where $\sigma_f$ represents the final state, and $r_f$ is the reward obtained from the execution. (2) It deduces (underapproximated) pre- and post-conditions for each program statement. (3) It may modify $\varepsilon$ into a new program $\varepsilon'$ by leveraging inferred preconditions and postconditions to synthesize conditional statements on an as-needed basis. Fig. 10 depicts the synthesis rules.

**Action Synthesis Rules.** The Action-Hoare rule is applied to an action $c$. For any new program statement that is just enumerated to fill a missing hole in a sketch, we assume its initial pre- and post-conditions are *False*. ReGuS executes an action $c$ only if the current program state $\sigma$ satisfies the precondition $\varphi$ of $c$ or $\varphi$ is the initial *False*. Following the DSL operational semantics, this execution produces the resulting state $\sigma'$ and receives a reward $r$ from the environment. We define $H$ as the set of state abstraction predicates equipped within the DSL (Equation 5) and $\mathcal{H}$ as the lattice space of the (conjunctive) predicate abstraction domain formed by these state abstraction predicates. We lift concrete states $\sigma$ and $\sigma'$ before and after executing $c$ into its precondition $\varphi$ and postcondition $\psi$ by leveraging the state abstraction predicates $H$ :

$$\alpha(\sigma) \equiv \{h \text{ if } h(\sigma) \text{ otherwise } \neg h \mid \forall h \in H\}$$

The state abstraction function $\alpha(\sigma)$ constructs a Boolean conjunction of literals where each literal is a predicate $h(\sigma)$ or its negation $\neg h(\sigma)$ evaluated on $\sigma$. The rule widens the current precondition $\varphi$ (*resp.* postcondition $\psi$) by joining it with $\alpha(\sigma)$ (*resp.* $\alpha(\sigma')$) in the abstract domain $\mathcal{H}$.

*Example 2.* Assume that the sketch in Equation. 6 is expanded to the partial program in Fig. 11. ReGuS abstracts the states before and after executing move (assuming that the agent is initially at the left bottom corner facing east). When ReGuS continues to execute the partial program in Fig. 12, it discovers that the inferred postcondition of move is no longer valid when the agent touches the right wall in the state depicted in Fig. 12. The postcondition is widened to include this new state.

The Action-LazyBranch rule allows ReGuS to synthesize conditional statements on demand. It applies when the state $\sigma$ encountered before executing an action $c$ does not satisfy the previously inferred precondition $\varphi$ of $c$, which indicates that $c$ is about to be executed in an unknown state. This situation commonly occurs when $c$ is evaluated in a subsequent loop iteration or from a different initial state. The rule appends an **if** statement with a sketch $??_S$ before $c$, creating a branch point. The intuition is that since $\sigma$ is new to the program, a new piece of statements might be needed to handle it. The **if** condition $b \in \mathcal{H}$ evaluates to true on the unseen state.

*Example 3.* When we continue to execute move from the state in Fig. 12, the precondition is no longer valid (as the front of the agent is not clear before executing move). The inferred axiomatic

semantics of the partial program in Fig. 12 (left) and the *actual* execution state of the program in Fig. 12 (right) are out-of-sync. ReGuS resolves the disagreement by exploring the option to synthesize a new conditional statement to synchronize the precondition and the actual state before executing `move`. The new sketch is depicted in Fig. 13.

The Compound-Hoare rule applies to all compound statements (e.g. loops and conditional statements). It infers the precondition and postcondition of a compound statement $S$ by abstracting the states encountered before and after executing $S$ via the state abstraction function $\alpha$.

```
while (not present(marker)) {
    if (not frontIsClear()) { ??_S }
    { leftIsClear() ∧ frontIsClear() ∧
    ¬ rightIsClear() ∧ ¬ present(marker) }
    move();
    { leftIsClear() ∧
    ¬ rightIsClear() ∧ ¬ present(marker) }
}; ??_C2
```

Fig. 13. ReGuS synthesizes a new conditional as the current running state in Fig. 12 before executing `move` does not meet the inferred precondition of `move`.

**Sequential Statement Synthesis Rules.** The Comp1 rule applies to a sequential statement $S_1; S_2$ when the execution of $S_1$ results in a sketch $S_1'$ due to on-demand conditional statement synthesis. The resulting state is $\bot$ as the program is incomplete. The Comp2 rule is straightforward.

**Loop Statement Synthesis Rules.** The While-Sketch1 rule applies when the execution of the first iteration of a loop encounters a nonterminal e.g. the loop body $S$ is incomplete. The execution cannot proceed beyond it. The While-Sketch2 rule is standard. It allows on-demand conditional statement synthesis at any subsequent loop iteration beyond the first one, updating the loop body $S$ to $S'$ with an appended conditional statement, as exemplified in Fig. 13.

**Conditional Statement Synthesis Rules.** The If-Erase rule relates to an **if** $(b)$ $\{??_{S_1}\}$ statement appended before an action statement $c_2$ by the Action-LazyBranch rule. If the first statement to fill $??_{S_1}$ is an action $c_1$ that is equivalent to $c_2$, the conditional statement is unnecessary as $c_1$ is executed on both branches regardless of the Boolean condition $b$. The rule joins the preconditions and postconditions of the shared action $c_2$ in both branches while eliminating the conditional statement. The If-Merge rule evaluates an **if** $(b)$ $\{S_1\}$ statement appended by Action-LazyBranch at a branch point before a compound statements $S_2$. The rule applies when the expansion of $S_1$ has been completed with a `skip` command, per the sketch grammar in Equation 3 and 4. Intuitively, $S_1$ and $S_2$ perform different actions depending on the Boolean condition $b$. This rule expects the control of either computation to return to a merge point with consistent postconditions. The `join` function below searches a merge point in $S_2$ by comparing the last statement $s_1^k$ of $S_1$ with each statement in $S_2$, where we use $S_{[j:k]}$ to denote the subsequence of statements from the $j^{\text{th}}$ to the $k^{\text{th}}$ statement in $S$:

$$
\begin{aligned}
&\texttt{join}(S_1, S_2) = \\
&\quad \texttt{assume}\ S_1 \ \equiv\ S_{1\,[1:k]};\ \{\varphi_1^k\}\ s_1^k\ \{\psi_1^k\};\ \texttt{skip} \\
&\quad \texttt{if}\ \exists\ j.\ S_2 \ \equiv\ S_{2\,[1:j]};\ \{\varphi_2^j\}\ s_2^j\ \{\psi_2^j\};\ S_{2\,[j+1:]} \bigwedge s_1^k = s_2^j \bigwedge \psi_1^k \Rightarrow \psi_2^j \\
&\quad \texttt{then return}\ \{\ \textbf{if}\ (b)\ S_{1\,[1:k]}\ \textbf{else}\ S_{2\,[1:j]};\ \{\varphi_1^k \sqcup \varphi_2^j\}\ s_2^j\ \{\psi_2^j\};\ S_2\ \}\ \texttt{else return}\ \bot
\end{aligned}
$$

If $s_1^k$ is syntactically equivalent to the $j$-th statement $s_2^j$ in $S_2$ and the postcondition of $s_1^k$ in $S_1$ implies that of $s_2^j$ meaning that the resulting state of taking $s_1^k$ has previously been seen, a merge point is found at $j$. At the merge point, we join the postconditions of $s_1^k$ and $s_2^j$.

*Example 4.* Assume that ReGuS has expanded the body $??_S$ of the appended **if** statements in Fig. 13 to a sequence of actions in Fig. 14. ReGuS favors the action `put(marker)` as performing this action can receive a reward of 0.25 from the environment for placing a marker in the bottom right

```
while (not present(marker)) {
 if (not frontIsClear()) {
  put(marker);
  turnLeft();
  move();
  {leftIsClear() ∧ frontIsClear()∧¬rightIsClear() ∧ ¬present(marker)}
  skip
 }
 move();
 {leftIsClear()∧¬rightIsClear() ∧ ¬present(marker)}
}; ??C2
```

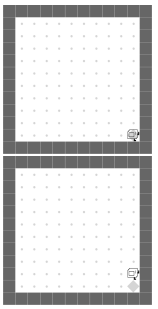


Fig. 14. ReGuS fills out the **if** statement in Fig. 13. The postconditions of the two move statements are shown.

corner. In Fig. 14, move is a merge point - the postcondition of the first move within the **if** body implies the postcondition of the same move action outside the **if** scope. Fig. 15 shows a program derived through the If-Merge rule. This program can successfully place a marker on each of the four corners.

The If-Merge-Fail rule applies when a merge point cannot be found by the join function. In this case, the program is reduced to $\perp$ to notify the sketch completer to prune away this search direction. This rule does not affect the completeness of ReGuS. Interested readers can find more details in the extended version [12]. This design choice is particularly beneficial for robot learning where diverse agent behaviors often need to converge at key sub-goals. For example, in a task for household cleaning, when a home-assisted robot encounters a locked door, its desired behavior converges at the subgoal of passing through the door, regardless of whether it currently has the key. The If-Merge-Fail rule prunes away partial programs that cannot open the door when the robot does not initially have the key.

```
while (not present(marker)) {
    if (not frontIsClear()) {
        put(marker);
        turnLeft();
    } else { }
    move();
}; ??C2
```

Fig. 15. ReGuS applies the If-Merge synthesis rule (Fig. 10) to join the branches of the conditional statement in Fig. 14.

### 4.4 High-Level Loop Sketch Generation

ReGuS generates high-level loop sketches containing loop bodies as "holes" to be completed by the sketch completer (Sec. 4.3). We approach loop sketch generation as a tree search. Each node in the tree represents a sketch allowed by the sketch grammar. (same as Equation 3):

$$\text{Loop Sketch } ??_S \ ::= ??_C;\ \textbf{while}\ b\ \{??_S\};\ ??_S \mid ??_C$$

In the following, we use the terms "tree node" and "sketch" interchangeably. The root node corresponds to the empty sketch $??_S$. A tree edge $(u, u')$ encodes a single-step application of a production rule in the sketch grammar to obtain the sketch $u'$ by replacing a nonterminal $??_S$ in the sketch $u$. A *terminal loop sketch* only contains nonterminals $??_C$ (code block) and can be filled out by the sketch completer $\text{ReGuS}_{Prog}$ (Algorithm 2). Terminal loop sketches are found on the leaf nodes of a search tree. Given a robot task MDP $e$, we measure the reward $r$ of a terminal loop sketch $\varepsilon^*$ as:

$$r(\varepsilon^*) = \text{ReGuS}_{Prog}(e, \varepsilon^*, \mathbb{Q}[\varepsilon^*], \xi, N)$$

where $\mathbb{Q}[\varepsilon^*]$ is the program search queue for $\varepsilon^*$, $\xi$ is the task reward threshold, and $N$ is the number of random seeds to initialize MDP instances in Algorithm 1 and 2.

The sketch generator $\text{ReGuS}_{Sket}$ is a variant of Monte Carlo Tree Search (MCTS). In each iteration, it samples one terminal loop sketch from the search tree. The best reward achieved by the sketch

completer on this sketch is used to weigh the nodes in the search tree so that better sketches are more likely to be chosen in the future. ReGuS$_{Sket}$ repeats four steps at each search iteration:

- Selection: The tree is traversed from the root guided by the score of each node (defined below) until reaching either a leaf or a node $u$ that still has one or more unexplored $??_S$ nonterminals.
- Expansion: If sketch $u$ has unexplored $??_S$ nonterminals, we create a child node $u'$ which is obtained by expanding the leftmost nonterminal $??_S$ of $u$ using either the sketch production rule $??_S ::= ??_C$ or $??_S ::= ??_C$; **while** $b$ $\{??_S\}$; $??_S$, as presented in Equation 3.
- Synthesis: This step assesses the reward of the new expansion. If sketch $u$ is already a leaf node, we set $\varepsilon^* = u$. Otherwise, we iteratively expand any remaining nonterminal $??_S$ on the newly created node $u'$ until a terminal loop sketch $\varepsilon^*$ is derived. ReGuS then evaluates the reward of $\varepsilon^*$ using the function $r(\varepsilon^*)$.
- Backpropagation: The reward feedback $r(\varepsilon^*)$ from the sketch completer is propagated recursively to all the traversed tree nodes to update the averaged reward of branching from these nodes.

In the selection step, the score of each tree node $u$ is computed by UCT (Upper Confidence Bound 1 applied to trees) [28] as follows: $UCT_u = (\overline{R_u} + 2C_p\sqrt{\frac{2\ln T}{T_u}})$ where $\overline{R_u}$ is the averaged reward feedback received on terminal loop sketches branching from node $u$, $T$ is the number of visits of the parent of $u$ during tree search, $T_u$ is the number of visits of $u$ and $C_p$ is the exploration constant. The child node with the greatest $UCT$ is selected during tree exploration. UCT is a standard mechanism to balance between exploitation (the first component of the formula) and exploration (the second component of the formula) for tree search. An example is given in the extended version [12].

### 4.5 Curriculum Synthesis

Robots are often assigned a variety of related tasks rather than being tasked with a single specific problem. These tasks differ in their complexity, involving various objects to manipulate and different goals to achieve. Synthesizing programs for robot control has the additional benefit of enabling knowledge transfer across tasks via reusable procedures. ReGuS ranks a sequence of related tasks according to their complexity e.g. the number of objects to manipulate to form a curriculum for the synthesizer. ReGuS adds programs synthesized to solve simpler tasks as new skills in the form of callable procedures to the DSL in Fig. 6. These new procedures serve as building blocks to constitute sophisticated programs for more complex tasks.

```
def skill_j(obj):
  while (not present(obj)) {
    while (not present(obj)) {
      ...
    }; pickUp(obj);
  }; pikcUp(obj);
```

Fig. 16. Skill Lifting.

To lift a program to a procedure, we abstract the various environment objects $O$ manipulated by the program to the parameters of the procedure. For instance, the `CleanHouse` program depicted in Figure 5b can be abstracted to a procedure skill$_j$ (where the specific name is not of significance), shown in Fig. 16. Adding this skill of cleaning one floor to the DSL, ReGuS can efficiently synthesize a robot-control program that iteratively invokes this procedure in a loop to clean a multi-floor house. As the set of robot skills grows in the DSL, the program search space becomes more efficiently compressed by these new skills.

## 5 EXPERIMENTS

ReGuS[1] is written in Python. In our implementation, to ensure program termination, we add "return" as a special action and add a penalty reward to programs that do not terminate within the task horizon for the agent to learn to terminate its execution when goal conditions are satisfied.

[1]The source code of ReGuS is available at https://github.com/RU-Automated-Reasoning-Group/ReGuS

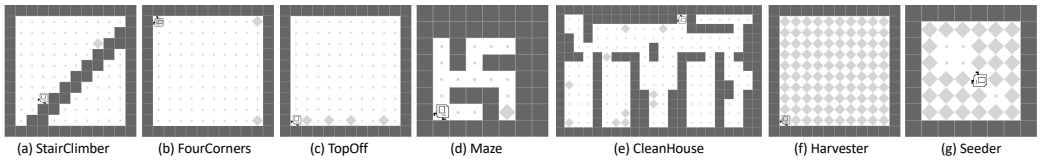



Fig. 17. Tasks of the "karel The Robot"

Our experiments are designed to answer the following research questions:

- **(RQ1)** How significant is the use of *loop sketch synthesis* and *on-demand conditional statement synthesis* in the ReGuS algorithm?
- **(RQ2)** Does *curriculum synthesis* enable ReGuS to expedite the generation of large programs needed for complex tasks, which would be challenging to synthesize?
- **(RQ3)** How effective is ReGuS at learning robot-control programs for tasks with *stochastic and continuous state spaces*?

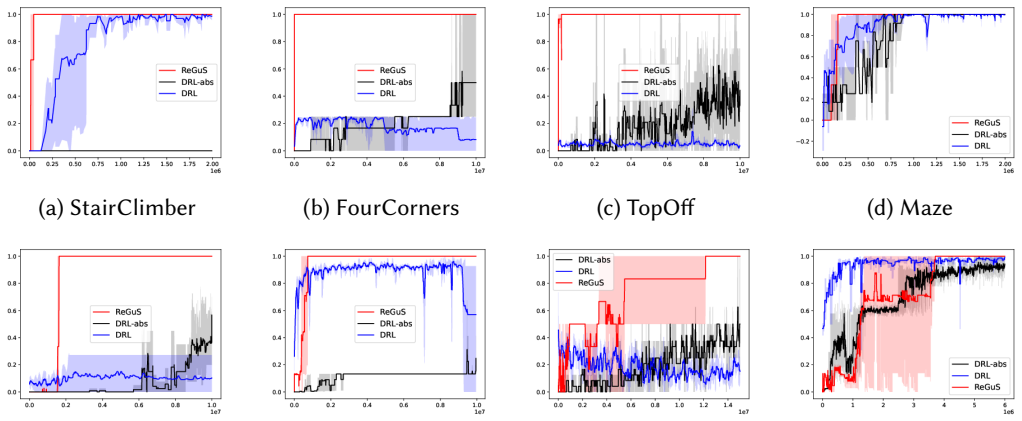
**Main Baselines.** Throughout the evaluation, we consider two important deep reinforcement learning (DRL) baselines.

- **DRL:** A deep reinforcement learning agent that takes raw environment states as input and produces a control action. The agent is implemented as a deep feedforward neural network trained by proximal policy optimization (PPO) [43], a state-of-the-art DRL algorithm.
- **DRL-abs:** A DRL agent that takes abstract states as input. This baseline allows for a fair comparison to ReGuS as both approaches utilize the same state abstraction. All returned values of abstraction predicates e.g. { `frontIsClear() == true, present(marker) == false, ...`} are concatenated as a binary vector, which is fed to a recurrent deep neural network as its input.

ReGuS and the baselines share the *same* control action space parameterized by the finite set of task-agnostic robot skills (as defined in Sec. 3).

### 5.1 RQ1: Loop Sketch Synthesis and On-demand Conditional Statement Synthesis

We use a suite of discrete state and action environments with the "Karel The Robot" simulator [37], taken from [48], to evaluate the capability of hierarchical loop sketch synthesis and on-demand conditional statement synthesis. In these environments visualized in Figure 17, an agent navigates inside a 2D grid world with walls and modifies the world state by interaction with markers. These tasks feature randomly sampled agent positions, walls, markers, and goal configurations. For these environments, we equip our DSL (Fig. 6) with the state abstraction predicates and actions defined in Equation 5. `DoorKey`, `CleanHouse` and `FourCorners` are running examples in the paper. In `DoorKey`, the agent gets a sporadic reward of 0.5 for picking up the first key and 1.0 for dropping it onto the second key. On `StairClimber`, the goal is to climb the stairs to reach the marker to get a reward of 1. The reward is $-1$ if the agent moves into an invalid position off the stairs and 0 otherwise. On `TopOff`, the goal is to put a marker in any position already with a marker and reach the rightmost square on the bottom row. The reward is defined as the number of markers correctly placed (the reward is normalized to a value between 0 and 1). On `Maze`, the goal is to find a marker inside a random maze to get a reward of 1. On `Harvester`, the goal is to pick up all the markers. Similarly, `Seeder` requires the agent to place exactly one marker on all grids. `Onestroke` requires traversing the environment without revisiting the same position as the visited grids become walls and the episode terminates if the agent hits a wall. The reward for these environments is determined by the ratio of successfully placed markers or visited grids.




(a) StairClimber (b) FourCorners (c) TopOff (d) Maze

(e) CleanHouse (f) Harvester (g) Doorkey (h) Seeder

Fig. 18. Comparing ReGuS against the baselines on the Karel domain. The x-axis denotes the number of robot-environment interactions (simulation steps). The y-axis records the rewards of learned controllers.

Table 1. Comparing the mean time in seconds (with standard deviation) used by ReGuS against baselines for synthesizing a program achieving the maximal reward 1.0 in the Karel domain.

| | StairClimber | FourCorners | TopOff | Maze | CleanHouse | Harvester | DoorKey | Seeder |
|---|---|---|---|---|---|---|---|---|
| EnumS | >2hrs | 1170.82 | 1609.06 | 3611.09 | >2hrs | >2hrs | >2hrs | >2hrs |
| EnumR | >2hrs | 1187.31 | 184.21 | 2980.31 | >2hrs | >2hrs | > 2hrs | >2hrs |
| ReGuS$_{Sket}$+EnumR | >2hrs | 221.79 | >2hrs | >2hrs | >2hrs | >2hrs | >2hrs | >2hrs |
| EnumS+ReGuS$_{Prog}$ | **20.13** | 3.76 | 3.10 | 135.04 | 890.12 | 1262.79 | >2hrs | 1599.97 |
| **ReGuS** | 22.02 | **3.52** | **2.85** | **124.83** | **537.67** | **171.38** | **2623.51** | **259.54** |

Fig. 18 depicts the learning performance in terms of the agent's rewards over 5 runs of each tool. We show the average (solid line) and std dev (shaded) rewards over the course of training. The rewards are collected by evaluating each program or neural policy for 1000 random initial states. ReGuS successfully synthesized programs getting 1.0 reward on all the tasks and exhibited the best data efficiency over the baselines.

**Ablation Study.** We studied the importance of hierarchical loop sketch synthesis and on-demand conditional statement synthesis through an ablation study.:

- EnumS: top-down enumeration of programs in order of increasing complexity measured by program structural cost (defined in Sec. 4.3).
- EnumR: top-down enumeration of (partial) programs ranked by both rewards and structural costs as in ReGuS but without loop sketch synthesis and on-demand conditional statement synthesis.
- ReGuS$_{Sket}$+EnumR: This ablation replaces the sketch completer ReGuS$_{Prog}$ in ReGuS with EnumR (sketch completion does not use on-demand conditional statement synthesis).
- EnumS+ReGuS$_{Prog}$: This ablation replaces the sketch generator ReGuS$_{Sket}$ in ReGuS with EnumS (sketch generation does not use reward feedback from sketch completion)[2].

Table. 1 presents the time taken by all the tools to find a program that achieves the maximal 1.0 reward across 1000 random initial states. The results are averaged over five trials. The enumeration baselines EnumS and EnumR perform significantly worse than ReGuS on all tasks, demonstrating the

[2]EnumR cannot be used because rewards cannot be assigned to "intermediate" sketches that still have ??$_S$ symbols.

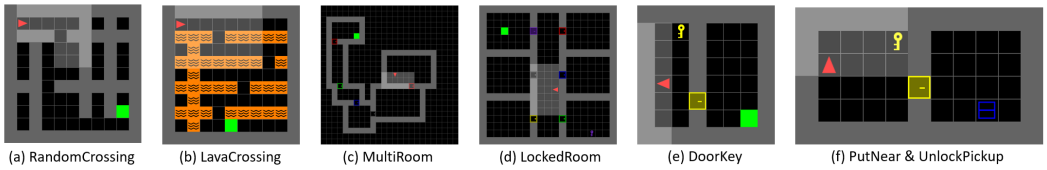



Fig. 19. Tasks of the MiniGrid Environments.

effectiveness of ReGuS's strategy for decomposed search space for loops. ReGuS$_{Sket}$+EnumR times out on most of the benchmarks whereas ReGuS solves all the tasks in a few minutes, highlighting the necessity of on-demand conditional statement synthesis to reduce the large program search space. ReGuS outperforms EnumS+ReGuS$_{Prog}$ on tasks requiring complex loop structures such as `CleanHouse`, `Harvester`, `seeder`, and `DoorKey`, showing the importance of hierarchical synthesis of loop sketches. The reward feedback from sketch completion acts as an effective heuristic to guide the generation of high-quality loop sketches.

### 5.2 RQ2: Curriculum Synthesis

We evaluate how ReGuS can expedite program synthesis for a stream of tasks with various complexity using MiniGrid [11], a collection of gridworld environments with goal-oriented tasks, widely used to evaluate state-of-the-art reinforcement learning algorithms. The agent can only detect if an object presents within its visible area (gray region). We focus on 7 tasks depicted in Fig. 19. **Crossing**: the agent has to reach a goal randomly placed in a room with intersecting walls. **LavaCrossing**: Similar to Crossing, but with lava streams that terminate the episode with zero rewards upon contact by the agent. **MultiRoom**: the agent has to navigate randomly connected rooms with closed doors to reach a goal. **LockedRoom**: the agent has to find a key in a room to unlock another room that houses the goal. **DoorKey**: similar to LockedRoom, but the key is initially in the same room as the agent (this is more challenging than Karel-DoorKey in Fig. 2 because this environment lacks the 0.5 reward for key pickup). **PutNear**: the agent needs to pick up a key in one room and place it next to another box hidden behind a locked door. **UnlockPickup**: the agent has to use a key to enter a locked room, drop the key, and then pick up a box. These environments feature random initial agent and object positions. As the tasks involve interacting with various objects, the state abstraction and control action space is significantly more complicated than that of Karel (Equation 5). In each environment, a positive reward of 1.0 is given only when the goal condition is satisfied. These tasks are known to be challenging to state-of-the-art RL algorithms due to their sparse reward setting, long horizon, and partial observability [11].

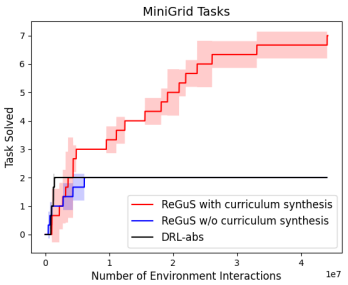



Fig. 20. ReGuS's performance on the MiniGrid tasks, evaluated over 5 trails. Each program is tested with 1000 random seeds.

We depict ReGuS's learning performance in Fig. 20. As these tasks increase in complexity as the number of objects to manipulate increases, ReGuS adds programs synthesized for a simpler task as a new skill that can be reused to constitute sophisticated programs for more complex tasks. For example, ReGuS adds the goal-reaching program synthesized for the third environment **Multiroom** as a new skill skill$_3(obj)$ to the DSL. This skill can then be reused as a new control action in the form of callable procedures such as skill$_3(key)$ and skill$_3(door)$. When synthesizing a program in **LockedRoom**, the agent can call skill$_3(key)$ to grab the key if it faces a locked door. It can then

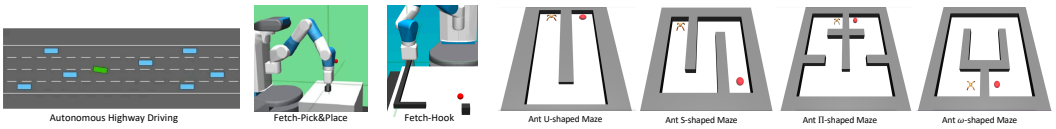



Fig. 21. Evaluating ReGuS in robot tasks with continuous state and action spaces.

return to the locked door via $\text{skill}_3(door)$ to open it. For a fair comparison, for DRL-abs, we also add learned neural policies for simpler tasks as a macro action that can be used by neural policies when learning for more complex tasks. DRL-abs consistently receives nearly zero rewards beyond the second environment. Using curriculum synthesis, **ReGuS achieves the maximum 1.0** reward on all the 7 environments evaluated over 1000 random seeds. Without curriculum synthesis, ReGuS failed to solve the more complex tasks as DRL-abs. The synthesized programs feature multiple loops with nested conditionals making them infeasible to be found by EnumS or EnumR.

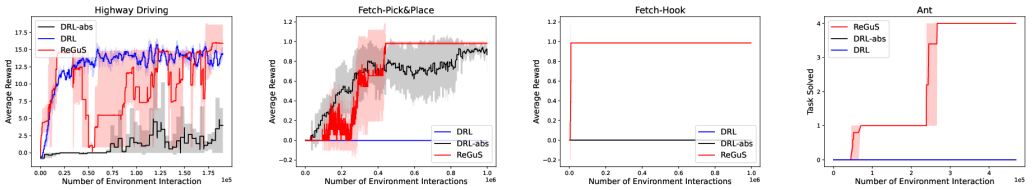
### 5.3 RQ3: Robotics Tasks in Stochastic Continuous State Spaces

We evaluate ReGuS using the following *continuous*, long horizon robotics tasks with sparse rewards.

**Highway Driving.** We consider the task of a self-driving car navigating a multilane highway depicted in Fig. 21 (left) adapted from [30]. In the DSL, we define a state abstraction predicate `frontIsClear()` to check if the predicted time-to-collision of observed vehicles on the same lane as the ego-vehicle is no less than a threshold based on their current speed and position. Additionally, we have `leftIsClear()` to check the clearance on the left lane assuming the ego-vehicle were to drive on the left lane. The `rightIsClear()` predicate is defined in the same way. The skills in our DSL `lane_left()`, `lane_right()`, `faster()` and `slower()` are borrowed from [30] which add a layer of speed and steering controllers on top of the continuous low-level control so that the ego-vehicle can follow the target lane at a desired speed. **The environment is stochastic** as the other vehicles' speeds and lane positions are uncertain e.g. at each timestep the front vehicle may maintain its speed or decelerate. The reward is 1.0 at each timestep only if the ego-car drives on the rightmost lane with the highest permissible speed while avoiding collisions. The horizon is 50.

**Fetch Environments.** The Fetch-Pick & Place task is for a manipulator to pick up a block and place it in a target position in mid-air as depicted in Fig. 21 (middle). The robot used is a 7-DoF Fetch Mobile Manipulator with a two-fingered parallel gripper. The environment is **stochastic** as the block is slippery and prone to falling from the gripper once grasped with 20% probability every step. We consider state abstraction predicates `Closing` (indicates if the gripper is closed), `NearObj` (indicates if the gripper is close to the block), `HoldingObj` (indicates if the gripper is holding the block), `AboveObj` (indicates if the gripper is above the block), and `ObjAt` (indicates if the block is in the goal region $g \in \mathbb{R}^3$). These predicates are borrowed from [27]. The continuous action space is $\mathbb{R}^4$, where the first 3 components encode the target gripper position and the last one is the target gripper width. The skills in our DSL are defined to include `openGripp()`, `closeGripp()`, `moveUp()`, `moveDown()`, `move(`$g$`)` that moves the gripper to a goal region $g$. The reward function is 1.0 if the block is moved to the goal region and 0 otherwise. The Fetch-Hook task is similar. However, the robot's gripper cannot directly reach the object and must utilize a hook to manipulate the object effectively. The table surface is scattered with random "bumps" that act as rigid obstacles, making the environment **stochastic**.

**Ant Navigation.** In the 4 maze environments depicted in Fig. 21 (right), a MuJoCo quadruped ant [47] is tasked to navigate to a goal position (red sphere). The objective is to synthesize a *single* program capable of reaching the goal regions in all 4 mazes. We adopt the same setting used in [16]: the ant receives range sensor readings about nearby obstacles as well as the goal.




(a) Highway Driving (b) Fetch-Pick&Place (c) Fetch-Hook (d) Ant Navigation

Fig. 22. Comparing the mean performance of ReGuS against the baselines over the robot learning tasks in Fig. 21 for 5 trails. The x-axis denotes the number of robot-environment interactions (simulation steps). The y-axis records the rewards of learned controllers (left and middle) or the numbers of solved tasks (right). Each program/neural policy is evaluated over 1000 trajectories.

The DSL is equipped with the state abstraction predicates and skills in Equation 5. The predicates `frontIsClear()`, `leftIsClear()`, `rightIsClear()`, and `present(goal)` use range sensory inputs for object detection. The skill module `move` can navigate the ant along the four cardinal directions by invoking four neural primitive skills `UP`, `DOWN`, `LEFT` and `RIGHT` pretrained by a deep RL algorithm SAC [22]. The other two skills `turnLeft` and `turnRight` update the cardinal direction of the ant based on which `move` invokes one of the four neural primitive skills correspondingly. Detailed descriptions of the predicates and skill modules are included in the extended version [12]. A positive reward is given only when the robot reaches the goal region within 800 timesteps.

Fig. 22 (left) compares ReGuS and the baselines on the Highway autonomous driving environment. As there is no goal in this environment, we show the rewards accumulated at each timestep. ReGuS learns an interpretable program that ensures safe avoidance, facilitates lane changes to the right, and maintains the highest permissible speed whenever possible. ReGuS ultimately surpasses the DRL baseline, which becomes stuck in a local minimum. DRL-abs tends to be overly cautious, always slowing down Fig. 22 (middle) shows that ReGuS generalizes well to the stochastic Pick&Place environments. Although the block is prone to falling off, the learned program iteratively picks it up in a loop to ensure eventual goal achievement. Curriculum synthesis further improves the data efficiency of ReGuS for the Hook environment. The synthesized program can utilize the Pick&Place skill to grasp the hook and align it with the block for movement toward the goal, enabling fast learning. In Fig. 22 (right), a task in Ant navigation is considered solved if the agent can achieve above 95% success rate for goal achievement over 1000 trajectories. The program synthesized for the Ant-U maze includes the behavior of turning left when facing a wall. When executed in the Ant-S environment, ReGuS adds to it a conditional statement, directing the ant to explore its right side when its right is clear. The program generalizes to the remaining two environments. The DRL and DRL-abs baselines struggle with the long-horizon, procedural Fetch and Ant Navigation tasks.

**Local Reward Optimums.** We further investigate whether ReGuS is subject to the risk of converging to suboptimal programs. Principally, ReGuS handles this issue by using UCT (Upper Confidence Bound 1 [28]) to balance sketch exploitation and sketch exploration in the high-level sketch generation phase (Sec. 4.4) and considering program structure cost in the low-level sketch completion phase (Sec. 4.3). For example, for the Fetch-Pick&Place task in Fig. 21 (middle), a synthesized program may attempt to move the gripper directly to the block to grab it. While this approach may occasionally succeed in getting a reward early, it is suboptimal because it could accidentally push the block off the table. Expanding partial programs excessively based on this behavior can result in increased structural complexity that outweighs the task reward. By leveraging structure costs, ReGuS navigates away from this local optima by synthesizing another program moving the gripper above the block before descending to grasp it. Although this behavior requires

more search steps to receive a positive reward, it can eventually lead to success. This phenomenon is evident in the fluctuations observed in the reward curve during learning for this task, as depicted in Fig. 22(b) (a similar trend can be observed for the other environments as well).

**Defining State-Abstraction Predicates.** To study the difficulty of defining predicates for continuous state environments, we firstly perform an ablation study to examine how the number of state abstraction predicates affects reward performance. In the Fetch-Pick&Place environment shown in Fig. 21 (middle), we iteratively eliminate predicates from the 5 perception-based predicates defined for this task and then rerun ReGuS. Our ablation study reveals that even when one of the three predicates—Closing, NearObj, or HoldingObj—is removed, ReGuS can still produce a correct program with full reward. However, ReGuS fails to solve this task when more than one predicate is removed, or when the other predicates are removed. We also investigate the influence of the quality of state abstraction predicates in the highway environment shown in Fig. 21 (left). The state abstraction predicates for this environment e.g. `frontIsClear()` assess if the predicted time-to-collision of observed vehicles is beyond a certain threshold. Our ablation study indicates that setting the threshold to 2 or 3 seconds yields programs with comparable rewards. However, setting the threshold to 4 leads to a program with a much lower reward, as the learned controller exhibited an overly cautious behavior, causing the ego vehicle to slow down prematurely.

We provided the synthesized programs for each robot task in our experiments in the extended version [12]. It also includes further comparisons between ReGuS and state-of-the-art deep RL algorithms and programmatic RL baselines. In the extended version, we demonstrated ReGuS's unique ability to generalize effectively to larger problem instances, and presented an ablation study on the hyperparameters of ReGuS, along with program execution trajectories on selected tasks.

## 6 LIMITATIONS

While ReGuS has demonstrated superior performance compared to state-of-the-art RL algorithms and standard program synthesis baselines in the robot-control tasks discussed in Sec. 5, our experiments have also revealed several limitations that require further improvement. The main limitation of ReGuS is its dependence on user-defined state abstraction predicates to construct the domain-specific language (DSL) used for synthesis. Our experiments demonstrate that specifying state abstraction predicates to describe spatial relationships among environmental objects is an effective strategy overall for robot-control tasks. The ablation study in Sec. 5.3 shows that determining suitable (constant) thresholds within these predicates for continuous environments may require domain expertise but can be adjusted as parameters by iterating with ReGuS.

ReGuS relies on curriculum synthesis to solve extremely sparse-reward tasks *characterized by multiple stages*, where a positive reward is given only upon the task's full completion. As shown in the experiment detailed in Sec. 5.2, this requires the presence of a series of related procedural tasks to form a reasonable curriculum. Exploring methods to automatically generate a curriculum for a *single* complex task is an interesting direction for future research.

Another limitation is that ReGuS has only been evaluated in the domain of robot navigation and manipulation. We anticipate that ReGuS can be generalized to other domains, such as data-structure manipulations. For example, with a proper DSL for heap manipulation, a ReGuS agent could *move* along a linked list as long as there are elements *present* or *turn left* to traverse the left branch of a tree to *pick up* specific data structure elements (filtering). We leave this extension for future work.

## 7 RELATED WORK

**Reward-guided Program Synthesis.** There exist synthesis algorithms that design dense rewards over their own solutions to guide search directions. Probe [2] and Syntia [4] evaluate candidate programs based on the inputs from input-output examples and compare the output similarity to

generate rich reward information. Chen et al. [9] incorporate feedback from a deduction engine in its reward function, which is based on the feasibility of partial programs concerning a set of input-output examples. Faery [10] uses Monte Carlo estimation to sample user queries for obtaining additional input-output examples. In sparse-reward scenarios, attempting to synthesize complete programs (with complex control flow structures) using Monte Carlo reward estimation becomes challenging as the likelihood of finding a complete program with a nonzero averaged reward is low, making it difficult to guide the search algorithm effectively. ReGuS is designed to tackle sparse-reward tasks through its two-staged synthesis approach. The sketch generator leverages the reward feedback from the sketch completer to guide its sketch generation process.

**Quantitative Synthesis.** The goal of quantitative synthesis is to find a program that meets a given specification, like a set of examples or a logical formula, while also optimizing a quantitative cost function, such as minimizing the number of instructions in the program or its performance overhead. Cost functions require white-box access to whole programs [7] and must be in decidable theory [5]. However, agents for robot learning interact with complex dynamics environments, wherein the state-transition models may be black boxes. Moreover, quantitative synthesis techniques are rooted in programming by example and counterexample-guided inductive synthesis [21, 24]. ReGuS conducts synthesis without the dependency on examples.

**Execution-guided Program Synthesis.** There exists prior work in program synthesis that incorporates execution traces to guide program generation. Ellis et al. [17] expose program execution results encountered during a bottom-up search to a neural predictor. The predicted production rule probabilities are then used to prioritize more likely programs. Chen et al. [8] leverage intermediate values of partial programs that are consumed by a neural encoder-decoder model to guide top-down search. ReGuS differs in the sense that it generalizes states encountered in execution to pre- and postconditions to enable on-demand conditional statement synthesis.

**On-demand Synthesis of Conditional Statements.** ReGuS's conditional statement synthesis strategy is related to the lazy predicate synthesis algorithms described in [39, 52]. These algorithms learn a predicate $\psi^*$ to strengthen an existing abstraction $\phi$ such that the following properties hold $p^+ \Rightarrow (\phi \wedge \psi^*) \Rightarrow p^-$ where $p^+$ and $p^-$ are in general pre- and post-conditions or abstractions of input-output examples. However, ReGuS does not assume it knows the specification of a robot task nor does it assume access to a provided set of task execution examples. We justify this design choice in Sec. 2. Compared to [e,f], ReGuS is more suitable when only rewards are used to specify tasks and there is no example or formal specification.

**Generalized Planning.** ReGuS shares conceptual similarities with generalized planning techniques such as [45, 46], which aim to derive plans with loops to solve problem instances of unbounded sizes. However, planning techniques require the state transition model for each robot action within an abstracted state space to be provided. Similar requirements are imposed for component-based synthesis techniques with user-defined predicates, as seen in [9, 19]. In contrast, ReGuS automatically generates the pre- and postconditions of each task-agnostic robot skill within the abstracted state space of user-defined predicates. This model-free approach makes ReGuS better suited for application domains such as stochastic robot environments, where manually specifying each robot skill is challenging and requires extensive domain expertise.

## 8 CONCLUSION

We present ReGuS, reward-guided synthesis, to address the exploration challenges in deep reinforcement learning for robot environments with sparse rewards. Extensive experiment results demonstrate that by decomposed search space for loops, on-demand synthesis of conditional statements, and curriculum synthesis of robot skills as reusable procedures, ReGuS effectively compresses the exploration space for long-horizon, multi-stage, and procedural robot tasks.

## ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their help and feedback on this paper. This material is based upon work supported by the National Science Foundation under grant numbers CCF-2124155 and CCF-2007799.

## REFERENCES

[1] David Andre and Stuart J. Russell. 2002. State Abstraction for Programmable Reinforcement Learning Agents. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence*.

[2] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.* OOPSLA (2020).

[3] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Advances in Neural Information Processing Systems, NeurIPS 2018*.

[4] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium, USENIX Security 2017*.

[5] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*.

[6] Elliot Chane-Sane, Cordelia Schmid, and Ivan Laptev. 2021. Goal-Conditioned Reinforcement Learning with Imagined Subgoals. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021*.

[7] Swarat Chaudhuri, Martin Clochard, and Armando Solar-Lezama. 2014. Bridging boolean and quantitative synthesis using smoothed proof search. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014*.

[8] Xinyun Chen, Chang Liu, and Dawn Song. 2019. Execution-Guided Neural Program Synthesis. In *7th International Conference on Learning Representations, ICLR 2019*.

[9] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. Program Synthesis Using Deduction-Guided Reinforcement Learning. In *Computer Aided Verification - 32nd International Conference, CAV 2020*.

[10] Yanju Chen, Chenglong Wang, Xinyu Wang, Osbert Bastani, and Yu Feng. 2023. Fast and Reliable Program Synthesis via User Interaction. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023*.

[11] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. 2018. *Minimalistic Gridworld Environment for Gymnasium*. https://github.com/Farama-Foundation/Minigrid

[12] Guofeng Cui, Yuning Wang, Wenjie Qiu, and He Zhu. 2024. Reward-guided Synthesis of Intelligent Agents with Control Structures (Extended Version). https://github.com/RU-Automated-Reasoning-Group/ReGuS/regus.pdf

[13] Quentin Delfosse, Wolfgang Stammer, Thomas Rothenbacher, Dwarak Vittal, and Kristian Kersting. 2023. Boosting Object Representation Learning via Motion and Object Continuity. In *Machine Learning and Knowledge Discovery in Databases: Research Track - European Conference, ECML PKDD 2023*.

[14] Thomas G. Dietterich. 2000. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *J. Artif. Intell. Res.* (2000).

[15] Yan Duan, Marcin Andrychowicz, Bradly Stadie, OpenAI Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. 2017. One-Shot Imitation Learning. In *Advances in Neural Information Processing Systems, NeurIPS 2017*.

[16] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. 2016. Benchmarking Deep Reinforcement Learning for Continuous Control. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning, ICML 2016*.

[17] Kevin Ellis, Maxwell I. Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. 2019. Write, Execute, Assess: Program Synthesis with a REPL. In *Advances in Neural Information Processing Systems, NeurIPS 2019*.

[18] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*.

[19] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*.

[20] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*.

[21] Sumit Gulwani, Kunal Pathak, Arjun Radhakrishna, Ashish Tiwari, and Abhishek Udupa. 2019. Quantitative Programming by Examples. *CoRR* abs/1909.05964 (2019).

[22] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning, ICML 2018*.
[23] Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J. Russell, and Anca D. Dragan. 2017. Inverse Reward Design. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*.
[24] Qinheping Hu and Loris D'Antoni. 2018. Syntax-Guided Synthesis with Quantitative Syntactic Objectives. In *Computer Aided Verification - 30th International Conference, CAV 2018*.
[25] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. 2020. Synthesizing Programmatic Policies that Inductively Generalize. In *8th International Conference on Learning Representations, ICLR 2020*.
[26] Michael Janner, Sergey Levine, William T. Freeman, Joshua B. Tenenbaum, Chelsea Finn, and Jiajun Wu. 2019. Reasoning About Physical Interactions with Object-Oriented Prediction and Planning. In *7th International Conference on Learning Representations, ICLR 2019*.
[27] Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. 2021. Compositional Reinforcement Learning from Logical Specifications. In *Annual Conference on Neural Information Processing Systems, NeurIPS 2021*.
[28] Levente Kocsis and Csaba Szepesvári. 2006. Bandit Based Monte-Carlo Planning. In *17th European Conference on Machine Learning, ECML 2006*.
[29] John Langford. 2010. Efficient Exploration in Reinforcement Learning. In *Encyclopedia of Machine Learning*.
[30] Edouard Leurent. 2018. An Environment for Autonomous Driving Decision-Making. https://github.com/eleurent/highway-env.
[31] Zhixuan Lin, Yi-Fu Wu, Skand Vishwanath Peri, Weihao Sun, Gautam Singh, Fei Deng, Jindong Jiang, and Sungjin Ahn. 2020. SPACE: Unsupervised Object-Oriented Scene Representation via Spatial Attention and Decomposition. In *8th International Conference on Learning Representations, ICLR 2020*.
[32] Russell Mendonca, Oleh Rybkin, Kostas Daniilidis, Danijar Hafner, and Deepak Pathak. 2021. Discovering and Achieving Goals via World Models. In *Advances in Neural Information Processing Systems, NeurIPS 2021*.
[33] Chaitanya Mitash, Kostas E. Bekris, and Abdeslam Boularias. 2017. A self-supervised learning system for object detection using physics simulation and multi-view pose estimation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017*.
[34] Chaitanya Mitash, Abdeslam Boularias, and Kostas E. Bekris. 2018. Robust 6D Object Pose Estimation with Stochastic Congruent Sets. In *British Machine Vision Conference 2018, BMVC 2018*.
[35] Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. 2018. Data-Efficient Hierarchical Reinforcement Learning. In *Annual Conference on Neural Information Processing Systems, NeurIPS 2018*.
[36] Soroush Nasiriany, Vitchyr Pong, Steven Lin, and Sergey Levine. 2019. Planning with Goal-Conditioned Policies. In *Annual Conference on Neural Information Processing Systems, NeurIPS 2019*.
[37] Richard E. Pattis. 1981. *Karel the Robot: A Gentle Introduction to the Art of Programming* (1st ed.). John Wiley & Sons, Inc., USA.
[38] Silviu Pitis, Harris Chan, Stephen Zhao, Bradly C. Stadie, and Jimmy Ba. 2020. Maximum Entropy Gain Exploration for Long Horizon Multi-goal Reinforcement Learning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020*.
[39] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*.
[40] Vitchyr Pong, Murtaza Dalal, Steven Lin, Ashvin Nair, Shikhar Bahl, and Sergey Levine. 2020. Skew-Fit: State-Covering Self-Supervised Reinforcement Learning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020*.
[41] Wenjie Qiu and He Zhu. 2022. Programmatic Reinforcement Learning without Oracles. In *10th International Conference on Learning Representations, ICLR 2022*.
[42] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*. 779–788.
[43] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).
[44] Tom Silver, Kelsey R. Allen, Alex K. Lew, Leslie Pack Kaelbling, and Josh Tenenbaum. 2020. Few-Shot Bayesian Imitation Learning with Logical Program Policies. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*.
[45] Tom Silver, Rohan Chitnis, Joshua B. Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. 2021. Learning Symbolic Operators for Task and Motion Planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021*.

[46] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. 2011. A new representation and associated algorithms for generalized planning. *Artif. Intell.* (2011).
[47] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*.
[48] Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. 2021. Learning to Synthesize Programs as Interpretable and Generalizable Policies. In *Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021*.
[49] Abhinav Verma, Hoang Minh Le, Yisong Yue, and Swarat Chaudhuri. 2019. Imitation-Projected Programmatic Reinforcement Learning. In *Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*.
[50] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. 2018. Programmatically Interpretable Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*.
[51] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*.
[52] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.* POPL (2018).
[53] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration using Datalog Program Synthesis. *Proc. VLDB Endow.* (2020).
[54] Yichen Yang, Jeevana Priya Inala, Osbert Bastani, Yewen Pu, Armando Solar-Lezama, and Martin Rinard. 2021. Program Synthesis Guided Reinforcement Learning for Partially Observed Environments. In *Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021*.

Received 2023-11-16; accepted 2024-03-31