Program Synthesis of Intelligent Agents from Rewards

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. In this paper, we propose PBR, a programming-by-reward 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 search space for long-horizon, multi-stage, and procedural robot-control tasks that are difficult to explore using deep RL techniques. Experiment results demonstrate that PBR significantly outperforms state-of-the-art deep RL algorithms and standard program synthesis baselines on challenging RL tasks including video games, autonomous driving, locomotion control, object manipulation, and embodied AI - operating home-assisted robots in complex household environments.

1 INTRODUCTION

Deep reinforcement learning (RL) has emerged as a promising approach for developing intelligent agents for robot control. However, understanding, debugging, and maintaining a deep RL model is challenging due to the difficulty in interpreting the learned logic within the intricate deep network structure. Inspired by the success of program synthesis in various applications, a growing body of research on *programmatic reinforcement learning* has investigated *domain-specific programs* as an RL model representation, aiming to promote interpretability [Bastani et al. 2018; Inala et al. 2020a; Qiu and Zhu 2022; Silver et al. 2020; Trivedi et al. 2021; Verma et al. 2019, 2018; Yang et al. 2021]. These methods adapt existing RL algorithms to guide program search or synthesize a robot-control program to imitate an optimized RL agent. Unfortunately, while these methods have made significant strides in improving the transparency of learned robot-control policies, they are limited by the capabilities of existing RL algorithms. Consequently, they are unable to effectively handle tasks that prove challenging for conventional deep RL approaches.

We observe that RL methods exhibit limited effectiveness when confronted with long-horizon tasks characterized by sparse rewards, where logic deduction and generalization are important. For example, they face challenges in scenarios where complex intrinsic logic is essential, such as putting an object into a drawer through the *sequence* of opening the drawer, placing the object, and subsequently closing the drawer. As another example, these methods struggle with robot-control tasks that involve repetitive behaviors, such as traversing *multiple* rooms within household environments that feature arbitrarily complex floor plans to locate a target object. Fig. 1 visualizes such an environment where a home-assisted robot is instructed to navigate to a specific type of object (e.g. a laptop). The agent is only rewarded with a positive value when it successfully locates the target object. Otherwise, it receives a reward of 0. State-of-the-art RL algorithms (e.g. DD-PPO [Wijmans et al. 2020]) can only achieve 47% success rate [Khandelwal et al. 2022] after extensive training from 200 million environment interactions. The inherent difficulty of long-horizon tasks with sparse rewards is that the number of environment interactions needed to solve a task with exploration increases exponentially with the number of steps to find rewarding trajectories [Langford 2010].

Fig. 1. A virtually realistic 3D household environment.

This Paper. In this paper, we aim to unlock the potential of program synthesis to overcome the inherent challenges faced by conventional RL techniques (which also perplex existing programmatic RL methods) in exploring complex long-horizon robot-control tasks characterized by sparse reward signals. We develop a programming-by-reward paradigm called PBR to synthesize robot-control programs in a given domain-specific language (DSL). Such programs instruct intelligent agents to sequentially take control actions in a robotic environment to maximize the total reward they receive from the environment. PBR overcomes the exploration challenges by synthesizing programs with state-conditioned *loops*, conditionals, and procedural calls serving as strong inductive biases to effectively compress the search space. PBR composes these programming constructs to naturally represent generalizable solutions to repetitive, multi-stage, and procedural robot tasks that are difficult to explore by deep RL techniques. Specifically, PBR leverages four key insights:

(1) State Abstraction: To make program synthesis practical for robotics environments with high-dimensional continuous state spaces, PBR fires on DSLs incorporated with state abstraction to 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 `markerPresent()` to enable the robot to detect obstacles and locate desired objects. These perceptual functions define an effective state abstraction for decision making e.g. turning left or right when the front is not clear.

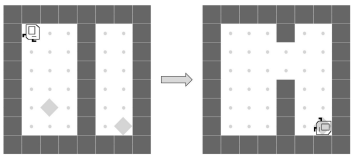


Fig. 2. The DoorKey Environment.

State abstraction techniques have been widely incorporated into modern reinforcement learning [Andre and Russell 2002; Dietterich 1999]. This enables the agent to generalize its learned policies across similar states, leading to improved sample efficiency. However, RL algorithms typically require a Markov state representation which holds if and only if the representation contains enough information to accurately characterize environment transition dynamics, whereas abstract state representations are not necessarily Markov. 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. While one can apply more advanced deep RL algorithms to learn to augment non-Markov states with a learnable external or internal "memory" unit, it further complicates the exploration challenges (Sec. 5). In contrast, PBR synthesizes a program that explicitly divides the task into two sequential loops, with each loop specifically handling the key in one room.

(2) Loop Sketches: Although synthesizing programs with loops compresses the search space for long-horizon tasks, there are an infinite number of syntactic elaborations of the loop body (possibly with nested loops) for each loop construct, which crucially depends on the loop condition and could be executed an arbitrary number of iterations. Fig. 3 visualizes PBR's solution to this challenge. At the high level, PBR synthesizes a loop sketch that only captures the "skeletal" sequential or nested while-loop structures of the program with the loop bodies as "holes" *yet to be determined*. We highlight that a loop sketch contains no conditionals. The low-level loop sketch completion

phase provides feedback to guide the high-level synthesizer. In PBR, the feedback is provided as the highest reward achieved by the low-level synthesizer on a loop sketch. The high-level synthesizer, 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. This feedback mechanism differs PBR from existing two-stage program synthesis techniques [Feng et al. 2017a,b; Wang et al. 2017, 2020] that only enumerate (example-consistent) sketches to be filled in without learning from the sketch completion process.

(3) On-demand Synthesis of Conditionals: 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, PBR lifts concrete states before and after an action c into presumed precondition φ and postcondition ψ by state abstraction. PBR uses inferred preconditions and postconditions to synthesize conditional statements on an as-needed basis to vastly reduce the program search space. If c is evaluated in a subsequent loop iteration or from a different initial state, and there is a disagreement between the inferred precondition φ and the current state σ before executing c (i.e., σ does not satisfy φ), PBR explores the option of appending a conditional statement before c with the branch condition evaluating to True for σ to handle the newly encountered state σ. PBR merges the two branches of this conditional statement at a common point with consistent postconditions (Sec. 4.1).

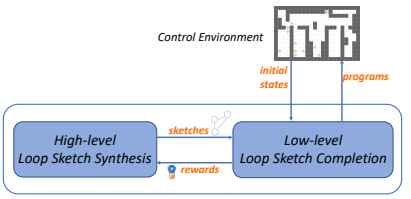


Fig. 3. Overview of the programming by reward framework for discrete environments.

(4) Curriculum Synthesis: Synthesizing programs for robot control has the additional benefit to enable knowledge transfer across related tasks via reusable procedures. For various tasks defined in similar environments, PBR ranks the tasks according to their complexity e.g. the number of objects to manipulate, and adds programs synthesized to solve simpler tasks as callable procedures so that they can be building blocks to constitute sophisticated programs for more complex tasks.

From State-based Programs to Vision-based Neurosymbolic Programs. Defining state abstraction for robot-control program synthesis becomes challenging in vision-based robot tasks that rely on RGB images for decision-making. To address this, we make the assumption of having access to a simulator modeling the environment of the real robot, a common practice in the RL community. Leveraging the simulator, PBR synthesizes state-based robot-control programs using available state information. Subsequently, we employ imitation in simulation to distill these state-based programs into *neurosymbolic* vision-based programs, where state abstraction predicates such as `present(key)` are learned as neural net primitives (e.g., for segmentation and classification). By constructing a higher-level representation of the scene through abstraction, our neurosymbolic programs can be executed in the real world, where accurate object states are unavailable.

Evaluation. We have evaluated PBR on several application domains that require different abilities including classic discrete environments, navigation in a 3D video game Minecraft, highway autonomous driving, locomotion control of a quadruped ant robot to explore goals in mazes of complex shapes, object manipulation through a 7-DoF Fetch Mobile Manipulator, and operating an intelligent home-assistant robot in complex household environments. All of our benchmarks involve inducing programs with (multiple) loops and (nested) conditionals to solve long-horizon tasks. Many environments are only equipped with sparse rewards that only signal if a task succeeds or fails. The results demonstrate that PBR significantly outperforms standard program synthesis baselines and state-of-the-art reinforcement learning techniques.

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

- Our main contribution is a novel Programming by Reward (PBR) technique for robot-control program synthesis. PBR overcomes the exploration challenges in long-horizon robot tasks with sparse rewards by synthesizing programs that incorporate state abstractions, loops, conditionals, and procedural calls. These programming constructs act as strong inductive biases, enabling search space compression and the generation of generalizable robot controllers for multi-stage and procedural tasks with repetitive patterns.
- We evaluate PBR on challenging robot tasks needing logical deduction and generalization and show that it outperforms program synthesis and deep RL baselines by large margins.

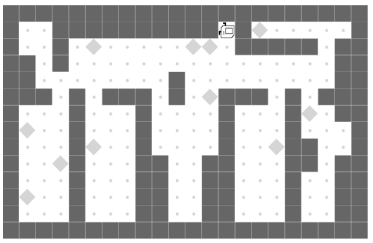
2 OVERVIEW

Motivating Example. Consider the task of `CleanHouse` depicted in Fig. 4a, set in a simplified household environment within a grid-world representation (in Sec. 5 we assess the performance of our synthesis algorithm in virtually realistic household environments as illustrated in Figure 1). The goal here is to synthesize a robot controller to pick up trash cans (markers) positioned near walls within a house. The agent starts at the entry point and must return to one side of the entry where two dumpsters (green and blue) are located, with two markers in this area.

Programming by Reward. Our program synthesis technique PBR (Program by Reward) diverges from mainstream synthesis algorithms that rely on example-based demonstrations, aiming instead to use task rewards as specifications for robot-control programs. The motivation behind this choice stems from the significant cost and effort required to obtain demonstrations from human experts in robot tasks. Prior research in robot manipulation tasks often depended on large numbers of expert demonstrations, ranging from tens to hundreds of thousands. In PBR, we assume the existence of a reward function that assigns a real-valued reward signal to each program execution run in a robotics environment. The synthesis problem is framed as the generation of a program that maximizes the expected reward when executed on all potential initial states of the environment.

Sparse Reward Functions. Synthesizing robot control programs using 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 instance, in `CleanHouse`, 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 [Hadfield-Menell et al. 2017]. For example, rewarding the robot to be close to the dumpster location may inadvertently hinder the agent's ability to effectively search the far end of the house and collect trash cans located there. Our approach targets sparse reward functions. These rewards can be given either after the robot completes the entire task (terminal reward) or sporadically when the robot achieves critical steps (step-wise rewards). Sparse rewards are easier to define compared to dense rewards. In the `CleanHouse` task, upon program termination or at the end of a control episode, the agent receives a reward based on the percentage of locations cleaned (markers picked up) and whether the dumpster location is reached.

Main Challenge. However, due to the lengthy action sequences required for many tasks, using sparse rewards introduces challenges in exploration and credit assignment. In particular, long-horizon tasks with sparse reward signals are challenging because the number of environment interactions needed to solve a task with exploration increases exponentially with the number of steps to get a reward [Langford 2010], a key reason why state-of-the-art reinforcement learning algorithms with sparse rewards often result in failure or require extensive amounts of data. In PBR, we aim to develop *data-efficient* program synthesis algorithms to effectively learn from sparse rewards.



(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. 4. Fig. 4a depicts the Cleanhouse environment where the agent is at the entrance of the house. Fig. 4b is a successful program that achieves the full reward on all possible initial states.

2.1 Program and Domain-Specific Language.

PBR synthesizes robot-control programs based on a generic DSL depicted in Fig. 5. As illustrated in Sec. 1, to make program synthesis practical for high-dimensional continuous robotics environments, we design the DSL based on state abstraction, which is powered by domain-dependent perception predicates, to construct a higher-level representation of the robot's environment. We assume the robot receives an object-centric view from its environment (formally defined in Sec. 3) that segments the world into objects and classifies them into different categories from the scene. In practice, one can employ perception algorithms such as object identification to extract relevant information and create a representation focused on individual objects from the robot's sensor data. Each object in the view has an associated continuous feature vector including its positions and velocities. Perception predicates h encode relations between environment objects and indicate circumstances in the current environment state that can be perceived by a robot. For example, state abstraction predicates for `cleanhouse` include

$$\texttt{frontIsClear()} : \neg\exists o.\, \texttt{front}(self,\, o) \qquad \texttt{leftIsClear()} : \neg\exists o.\, \texttt{left}(self,\, o) \tag{1}$$

$$\texttt{rightIsClear()} : \neg\exists o.\, \texttt{right}(self,\, o) \qquad \texttt{present(o)} : \texttt{front}(self,\, o) \tag{2}$$

where $self$ represents the robot itself and o is an arbitrary object. The atomic predicates `front`, `left`, and `right` describe spatial relationships that capture the physical orientations between objects. End users have the freedom to define arbitrary domain-dependent perception predicates. As discussed in Sec. 1, the PBR synthesis algorithm does not assume that abstract state representations for a Markov Decision Process (MDP) preserve the Markov property. Synthesized programs can rely on control flow information to deal with different environment states mapped to the same abstract state (e.g. two loops for Doorkey in Fig. 2).

In the DSL, control actions denoted as c represent domain-specific low-level *skills* that refer to *task-agnostic* capabilities of a robot within the continuous state space. They add a layer of speed and steering control on top of the continuous state space, enabling the robot to automatically navigate in a desired direction at a specified velocity. These skills can be derived from robot APIs (primitive actions) or pre-trained neural networks. Conceptually, a skill c can be likened to building blocks or subroutines that contribute to the overall control policy of the robot. Skills are modular and reusable, allowing agents to apply them in diverse contexts or combinations to solve complex problems. For instance, in the context of the `cleanhouse` task, skills such as object manipulation `pickup(marker)`, naviagation `move()`, `turnLeft()` and `turnRight()` facilitate picking up objects, forward movement at a desired speed (fixed for this environment), and altering the target direction that serves as setpoints for the low-level controllers.

The key idea behind the DSL is to use state abstractions to construct a higher-level representation of the robot's environment, which can then be reasoned about using standard language constructs to trigger the suitable skill from an environment state. Our DSL supports programs with state-conditioned loops and conditional statements. A program that achieves the full reward for `CleanHouse` is depicted in Fig. 4b. It invokes the abstraction predicates to extract state conditions (e.g. `present(marker)`) and invokes a sequence of control actions for navigation (e.g. `turnRight`) and object manipulation (e.g. `pickUp(marker)`) based on the state conditions.

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

Fig. 5. Our DSL for Neurosymbolic Programming for RL.

2.2 The Synthesis Procedure

The simplest strategy for program search is to enumerate programs in order of increasing complexity measured by structural cost. However, the exponential growth of the search space with longer programs renders the synthesis process intractable. For `CleanHouse`, enumerative program search has to explore an extremely large number of programs before reaching the desired program in Fig. 4b as the abstract syntax tree depth of this program is 9 and the size of its abstract syntax tree is 15. Presumably one could use more advanced synthesis algorithms developed over the years for programming by example to address programming by reward. However, since input-out examples are not readily available or informative in RL contexts, reusing search bias or pruning heuristics in programming by example for programming by reward is not always possible.

The main contribution of this paper is a new synthesis algorithm called PBR for programming by reward. PBR uses the following novel strategies to address the aforementioned challenges.

Hierarchical Program Synthesis over Loop Sketches. PBR is a *hierarchical* program synthesis procedure. We visualize it in Fig. 3. At the high level, PBR synthesizes a "skeletal" programs which represents a *loop sketch* that only contains sequential or nested while loop structures with the loop bodies as "holes" *yet to be determined*. For the `Cleanhouse` example, a loop sketch could be

$$\textbf{while } (\texttt{not present(marker)})\ \{\ ??_{C_1}\ \};??_{C_2} \tag{3}$$

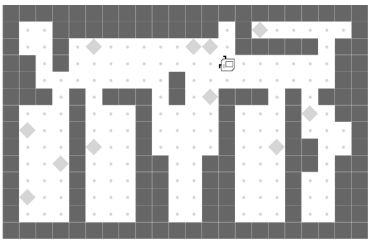
where $??_C$ represents a missing hole. At the low level, we synthesize the missing pieces in a loop sketch. Notably, as visualized in Fig. 3, the low-level synthesizer provides feedback to guide the high-level synthesizer. In our method, the feedback is provided as the highest reward achieved by the low-level synthesizer on a high-level loop sketch. The high-level synthesizer, 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. The feedback mechanism differs PBR from existing two-stage program synthesis techniques [Feng et al. 2017a,b; Wang et al. 2017, 2020] that only enumerate (example-consistent) sketches to be filled in without learning from the sketch completion process.

For example, on `Cleanhouse`, PBR discovers a program on top of the loop sketch in Equation. 3 that can clean at least one marker (trash can) out of the total markers in a house. The other single-loop sketches with a different loop condition cannot lead to a nonzero-reward program. The high-level synthesizer prioritizes loop sketches based on this condition and may expand the loop sketch in Equation. 3 below for deriving a program that iteratively picks up all the markers:

$$\textbf{while } (\texttt{not present(marker)})\ \{\textbf{while } (\texttt{not present(marker)})\ \{\ ??_{C_1}\ \};\ ??_{C_2}\};\ ??_{C_3} \tag{4}$$

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

(a) A Sketch of the Cleanhouse controller.



(b) Executing the sketch in Fig. 6a in Cleanhouse.

Fig. 6. Fig. 6a is a partial program enumerated by PBR. The inferred precondition of the move action is depicted in blue. Fig. 6b is the execution results of the partial program in Fig. 6a from the state in Fig. 4a.

On-demand Synthesis of Conditionals. PBR guides top-down enumerative search to fill out a loop sketch. To best use the reward information, it executes partially synthesized programs to prioritize exploring partial programs with the highest reward performance. To effectively reduce the search space for low-level sketch completion, PBR infers presumed pre- and post-conditions of each statement in a synthesized program by abstracting the states encountered before and after executing the statement with predicate abstraction. PBR *lazily* synthesizes conditional statements on an as-needed basis only when the precondition of a statement *inferred* from past executions does not hold on an unseen state before executing the same statement from a new initial state or a new loop iteration.

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

Fig. 7. Lazily added conditional to Fig. 6a.

For example, Fig. 6a is a partial program enumerated by PBR to solve the `Cleanhouse` task. The agent is initially at the entrance of the house (Fig. 4a), and the inferred precondition of `move` is drew 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. 6b is encountered in a subsequent inner loop iteration. The precondition is not satisfied as the left of the agent in Fig. 6b is clear. At this point, the axiomatic semantics of the partial program in Fig. 6a and the *actual* execution results in Fig. 6b are out-of-sync. PBR then explores the option of synthesizing a new control flow structure to synchronize the precondition and the actual state before executing `move` by constructing a new sketch depicted in Fig. 7. With lazy control flow structure synthesis, PBR eventually finds the desired program in Fig. 4b.

Curriculum Synthesis. PBR incorporates programs learned for simpler tasks into DSLs, treating them as procedures that can be utilized as skills. As the library of robot skills expands, the learned procedures serve to compress the search space, expediting the synthesis of larger programs (Section 4.3). For example, the `cleanhouse` procedure can serve as a "macro" skill that a robot program can iteratively invoke in an outer loop to clean each floor of a multi-floor house.

2.3 Limitations

PBR requires state abstraction predicates in DSLs, which imposes an overhead. However, PBR aims to address the exploration challenges posed by long task horizons and sparse rewards, which are known to be difficult for traditional RL algorithms. State abstraction plays a pivotal role to enable PBR to surpass RL techniques even when they are equipped with the same abstraction.

$$\textsc{Action}\quad \frac{\sigma' \sim \mathcal{T}(\cdot \mid \sigma, c) \qquad r = \mathcal{R}(\sigma, c)}{\langle \sigma,\ c\rangle \Downarrow \sigma',\ r}$$

$$\textsc{Sequence}\quad \frac{\langle \sigma,\ S_1\rangle \Downarrow \sigma',\ r \qquad \langle \sigma',\ S_2\rangle \Downarrow \sigma'',\ r'}{\langle \sigma,\ S_1; S_2\rangle \Downarrow \sigma'',\ r + r'}$$

$$\textsc{Conditional-T}\quad \frac{b(\sigma) \qquad \langle \sigma,\ S_1\rangle \Downarrow \sigma',\ r}{\langle \sigma,\ \textbf{if}\ (b)\ S_1\ \textbf{else}\ S_2\rangle \Downarrow \sigma',\ r}$$

$$\textsc{Conditional-F}\quad \frac{\neg b(\sigma) \qquad \langle \sigma,\ S_2\rangle \Downarrow \sigma',\ r}{\langle \sigma,\ \textbf{if}\ (b)\ S_1\ \textbf{else}\ S_2\rangle \Downarrow \sigma',\ r}$$

$$\textsc{While-T}\quad \frac{b(\sigma) \qquad \langle \sigma,\ S\rangle \Downarrow \sigma',\ r \qquad \langle \sigma',\ \textbf{while}\ (b)\ S\rangle \Downarrow \sigma'',\ r'}{\langle \sigma,\ \textbf{while}\ (b)\ S\rangle \Downarrow \sigma'',\ r + r'}$$

$$\textsc{While-F}\quad \frac{\neg b(\sigma)}{\langle \sigma,\ \textbf{while}\ (b)\ S\rangle \Downarrow \sigma,\ \mathbf{0}}$$

Fig. 8. DSL operational semantics and reward evaluation.

3 PROBLEM FORMULATION

We define a robot-control environment as a tuple (Λ, d, O, C) where Λ 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 Λ. 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 $f(\lambda_0, \ldots, \lambda_{N-1}) \in C$, e.g. `pickup(·)`, have typed parameters $(\lambda_0, \ldots, \lambda_{N-1})$ where $N \geq 0$ and $\lambda_i \in \Lambda$. We frame programming by reward (PBR) as the problem of synthesizing domain-specific programs to solve robot-control tasks formalized as Markov decision processes (MDPs).

Markov Decision Process (MDP). A robot-control 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 where a state $\sigma \in \mathcal{S}$ is a mapping from each object $o \in O$ 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 $c := f(o_0, \ldots, o_{N-1})$ applies a skill $f(\lambda_0, \ldots, \lambda_{N-1}) \in C$ to objects $o_1, \ldots, o_N$ and the objects must have types matching the typed parameters $\lambda_0, \ldots, \lambda_{N-1}$ of skill f, e.g. `pickup(marker)`. $\mathcal{T}$ 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 $\mu(\mathcal{S}_0)$. $\mathcal{T}$, $\mathcal{R}$, and $\mu(\mathcal{S}_0)$ are unknown to program synthesizers.

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

PBR synthesizes programs based on the DSL introduced in Fig. 5. A distinct feature is that the DSL operational semantics tracks the cumulative reward achieved during program execution.

Operational Semantics with Rewards. We outline the DSL operational semantics in Fig. 8. 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 task environment state $\sigma \in \mathcal{S}$. The resulting state of the execution is σ'. The execution receives a cumulative reward of r from the environment.

The Action rule is defined for a control action c. The action c is executed upon an MDP state σ and the MDP transits to another state σ' as the effect of the action according to the transition model $\mathcal{T}(\sigma' | \sigma, c)$. The immediate reward of taking action c at σ is given by the reward function $r = \mathcal{R}(\sigma, c)$ in the MDP after the transition. In the Conditional-T and Conditional-F rules, the control flow decision at a state σ is made on $b(\sigma)$ that queries the environment condition b which is a Boolean combination of state abstraction predicates. The rest of the rules are straightforward.

Stochasticity. In the main approach Sec.4, we simplify the analysis by assuming that the transition model $\mathcal{T}$ deterministically generates next environment states. However, in our experiments (Sec.5), $\mathcal{T}$ reflects the stochastic nature of robot-control environments - there is a great possibility that

a single execution of a skill c may not succeed. PBR readily generalizes to these scenarios by repeatedly calling the skill in a loop.

Program Synthesis Objective. The objective of PBR is searching a robot-control program π^* in the DSL in Fig. 5 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 σ_f and r_f are the resulting state and final reward by executing π in the task MDP from a state σ_0 sampled from the initial state distribution μ of e.

4 PROGRAMMING BY REWARDS

In this section, we present our synthesis technique for solving the programming by reward problem defined in the previous section. As outlined in the PBR framework in Fig. 3, PBR employs a hierarchical synthesis procedure. At the high level, PBR 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. 5 to describe the syntax of loop sketches:

$$\text{Loop Sketch } ??_S \ ::= ??_C;\ \textbf{while}\ b\ \{??_S\};\ ??_S \mid ??_C \tag{5}$$

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 PBR to complete a block:

$$\text{Block } ??_C \ ::= ??_c;\ ??_C \mid \texttt{skip} \tag{6}$$

By equation. 5, for tasks that do not need loops, a loop sketch can simply be reduced to a block.

The high-level synthesizer only has access to the loop sketch grammar in Equation. 5 and can generate arbitrarily loop structures. The low-level synthesizer fills the missing pieces $??_C$ in a sketch using Equation. 6. PBR leverages the rewards of programs from sketch completion as feedback to prioritize the search order of highly likely loop sketches. The feedback mechanism differs PBR from existing two-stage program synthesis [Feng et al. 2017a,b; Wang et al. 2017, 2020] that only enumerates (example-consistent) sketches without using the feedback from sketch completion to optimize the sketch generation process.

4.1 Low-level Loop Sketch Completion

PBR executes (synthesized) partial programs on the fly and uses the execution results to guide and accelerate loop sketch completion. The main ideas are that (1) it ranks partial programs by their reward performance and explores these programs following this order and (2) PBR *lazily* synthesizes conditional statements on an as-needed basis when the inferred axiomatic semantics of a partial program π and the actual states of π in execution are out-of-sync.

We illustrate our ideas using the `FourCorners` task taken from [Trivedi et al. 2021]. The goal of the agent is to put a marker at each corner of the environment depicted in Fig. 10. 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. The state abstraction and actions for this example in the DSL (Fig. 5) are:

$$\begin{aligned} \text{Object } o &\ \in \{\ \texttt{marker}\ \} \\ \text{State Abstraction } h &\ ::= \texttt{present}(o), \texttt{leftIsClear()}, \texttt{rightIsClear()}, \texttt{frontIsClear()} \\ \text{Control Action } c &\ ::= \texttt{put}(o), \texttt{pickUp}(o), \texttt{turnLeft()}, \texttt{turnRight()}, \texttt{move()} \end{aligned} \tag{7}$$

For this example, we assume that the loop sketch given by the high-level synthesizer is:

$$\textbf{while}\ (\texttt{not present(marker)})\ \{\ ??_{C_1}\ \};\ ??_{C_2} \tag{8}$$

$$\textsc{Action-Hoare}\quad \frac{\langle \sigma, c\rangle \Downarrow \sigma', r \qquad \varphi = \alpha(\sigma) \qquad \psi = \alpha(\sigma')}{\sigma,\ c \triangleright \{\varphi\}\ c\ \{\psi\},\ \sigma',\ r}$$

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

$$\textsc{Action-Monitor}\quad \frac{\varphi(\sigma) \qquad \langle \sigma, c\rangle \Downarrow \sigma', r \qquad \alpha(\sigma') \Rightarrow \psi}{\sigma,\ \{\varphi\}\ c\ \{\psi\} \triangleright \{\varphi\}\ c\ \{\psi\},\ \sigma',\ r}$$

$$\textsc{Action-Widen}\quad \frac{\varphi(\sigma) \qquad \langle \sigma, c\rangle \Downarrow \sigma', r \qquad \alpha(\sigma') \not\Rightarrow \psi}{\sigma,\ \{\varphi\}\ c\ \{\psi\} \triangleright \{\varphi\}\ c\ \{\psi \sqcup \alpha(\sigma')\},\ \sigma',\ r}$$

Fig. 9. Action statement synthesis rules. $\Downarrow$ defines the DSL operational semantics. α abstracts a concrete state to the abstract domain $\mathcal{A}$. r is the reward of executing an action c in a program state σ.

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

We present PBR's low-level loop sketch completion procedure using formalized synthesis rules:

$$\sigma,\ S \triangleright S',\ \sigma',\ r$$

where S is either a complete program or a partial program and σ is a state from which S is executed. For instance, S is a given loop sketch by the high-level synthesizer and σ is an initial state. As we will explain below, a synthesis rule may update S to yield a new program S'. It does so by evaluating S from σ following the DSL operational semantics $\langle \sigma, S\rangle \Downarrow \sigma', r$ in Fig. 8 where σ' is the resulting state of executing S and r is the reward obtained by the execution. We extend the DSL operational semantics to support partial programs: $\langle \sigma, ??_C\rangle \Downarrow (\bot, ??_C),\ 0$ where $??_C$ is a sketch in the partial program reached during execution that prevents it from further proceeding and $\bot$ denotes that a program cannot be executed beyond this point. 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 environment.

Action Synthesis Rules. Fig. 9 depicts the synthesis rules for actions. The Action-Hoare rule is applied to a newly enumerated action c during the synthesis process. We execute c from the current program state σ following the DSL operational semantics to yield the resulting state σ'. PBR lifts concrete states σ and σ' before and after executing c into its presumed precondition φ and postcondition ψ by state abstraction. Later, we show that PBR can use inferred preconditions and postconditions to *lazily* synthesize conditional statements on an as-needed basis to vastly reduce sketch completion search space. We express the precondition (resp. postcondition) of c based on σ (resp. σ') leveraging the state abstraction predicates H equipped within the DSL (Equation 7):

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

The abstraction function $\alpha(\sigma)$ constructs a Boolean conjunction of literals where each literal is a state abstraction predicate $h(\sigma)$ or its negation $\neg h(\sigma)$ evaluated on σ. The axiomatic semantics we infer during program execution is an under-approximation of the true semantics. The rule also collects the reward of executing c at state σ.

Example 1. Assume that the sketch in Equation. 8 is expanded to the partial program in Fig. 10. PBR abstracts the states before and after executing `move` (assuming that the agent is initially at the left bottom corner facing east).

The Action-Monitor rule monitors the execution of an action c associated with inferred precondition φ and postcondition ψ. The rule applies when c has been executed in the past, e.g. c is within some **while** loop. The rule validates if the pre state σ and the post state σ' of executing c are consistent with φ and ψ respectively.

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

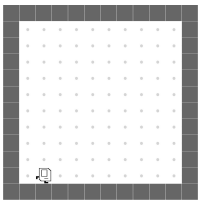


Fig. 10. Pre and postcondition inference of `while (not present(marker)) {move;}; ??`$_{C2}$ in FourCorners.

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

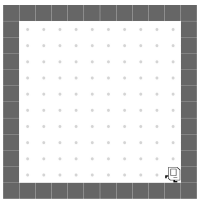


Fig. 11. Widen the postconditon of `move` to include an unseen state resulting from executing `move`.

The Action-Widen rule applies when the post state σ' of executing an action c does not satisfy the previously inferred postcondition ψ of c. The rule joins ψ with $\alpha(\sigma')$ in the lattice space of the (conjunctive) predicate abstraction domain formed by the state abstraction predicates.

Example 2. When PBR continues to execute the partial program in Fig. 11, it discovers that the inferred postconidtion of `move` is no longer valid when the agent touches the right wall in the state depicted in Fig. 11. The postcondition is widened to include this new state.

The Action-LazyBranch rule allows PBR to synthesize conditional statements on an as-needed basis. It applies when the state σ encountered before executing an action c does not satisfy the previously inferred precondition φ 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 before c, creating a branch point. The intuition is that since σ is new to the program, a new piece of statements represented by a sketch $??_S$ might be needed to handle it. The **if** condition b is a state abstraction predicate or its negation, which evaluates to True for the unseen state.

Example 3. When we continue to execute `move` from the state in Fig. 11, 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. 11 and the *actual* execution state of the program in Fig. 11 are out-of-sync. PBR 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. 12.

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

Fig. 12. PBR synthesizes a new conditional as the current running state in Fig. 11 before executing `move` does not meet the inferred precondition of `move`. Particularly, it does not satisfy the predicate `frontIsClear`.

Before delving into the synthesis rules for conditional statements, loop statements, and sequential statements, we present two rules that are common to all compound statements. The

rules infer the precondition and postcondition for a compound statement S by abstracting the states encountered before and after executing S using the state abstraction function α.

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

Conditional Statement Synthesis Rules. Fig. 13 depicts the synthesis rules for conditional statements. 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 S_1 has been completed with a `skip` command. Intuitively, S_1 and S_2 perform different actions depending on the Boolean condition b. This rule expects the control of either computation upon completion to return to a merge point with consistent environment states in terms of their postconditions. The `join` function used by the rule searches a merge point in S_2 by comparing the last statement s_k^1 of S_1 with each statement in S_2:

$$\begin{aligned}
&\texttt{join}(S_1, S_2) = \\
&\quad \texttt{assume}\ S_1 \ \equiv\ \underbrace{s_1^1;\ s_2^1;\ \cdots;\ s_{k-1}^1}_{S_1'};\ \{\varphi_k^1\}\ s_k^1\ \{\psi_k^1\};\ \texttt{skip} \\
&\quad \texttt{if}\ \exists\ j.\ S_2 \ \equiv\ \underbrace{s_1^2;\ s_2^2;\ \cdots;\ s_{j-1}^2}_{S_2'};\ \{\varphi_j^2\}\ s_j^2\ \{\psi_j^2\};\ \underbrace{s_{j+1};\ \cdots}_{S_2''} \bigwedge s_k^1 = s_j^2 \bigwedge \psi_k^1 \Rightarrow \psi_j^2 \\
&\quad \texttt{then return}\ \{\ \textbf{if}\ (b)\ S_1'\ \textbf{else}\ S_2';\ \{\varphi_k^1 \sqcup \varphi_j^2\}\ s_j^2\ \{\psi_j^2\};\ S_2''\ \}\ \texttt{else return}\ \bot
\end{aligned}$$

If s_k^1 is syntactically equivalent to an action s_j^2 as the j-th statement in S_2 meaning that the agent has conducted the same action in both branches and the postcondition of s_k^1 in S_1 implies that of s_j^2 meaning that the resulting state of taking s_k^1 has previously been seen, a merge point is found at j. At the merge point, we join the preconditions of s_k^1 and s_j^2 as it is executed by both branches. As an alternative approach, we can relax the requirement on $s_k^1 = s_j^2$ and generate a loop statement in the form of "**if** (b) S_1'; s_k^1 **else** S_2'; s_j^2; S_2''". Through empirical evaluation, we have observed that the chosen definition in `join` results in smaller programs and hence fastens the synthesis process.

Example 4. When synthesizing the body of the appended **if** statements in Fig. 12, assume that PBR has chosen to expand $??_S$ to a sequence of actions in Fig. 14. In fact, PBR prefers the action `put(marker)` because it uses reward signals to prioritize high-reward programs. `put(marker)` gets a 0.25 reward from the environment by putting a marker in the bottom right corner. PBR searches if a common point exists. Notice that in Fig. 14, `move` is such a common point for both - the postcondition of the (first) `move` in the **if** body implies the postcondition of the same (second) `move` action out of the scope of **if**. PBR derives a program in Fig. 15 by applying the If-Merge rule. Executing this program can successfully put a marker on each of the corners.

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

Fig. 15. PBR applies the If-Merge synthesis rule (Fig. 13) to join the branches of the conditional statement in Fig. 14.

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 $\bot$ to notify the low-level synthesizer to prune away this search

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

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

$$\textsc{If-Erase}\quad \frac{S_1 \equiv \{\varphi_1^1\}\ c_1^1\ \{\psi_1^1\};\ ??_S \qquad S_2 \equiv \{\varphi_1^2\}\ c_1^2\ \{\psi_1^2\};\ S_2' \qquad c_1^1 = c_1^2 \qquad \langle \sigma, c_1^1\rangle \Downarrow \sigma', r \qquad \sigma',\ S_2' \triangleright S_2'',\ \sigma'',\ r'}{\sigma,\ \textbf{if}\ (b)\ \{S_1\};\ S_2 \triangleright \{\varphi_1^2 \sqcup \varphi_1^1\}\ c_1^1\ \{\psi_1^1 \sqcup \psi_1^2\};\ S_2'',\ \sigma'',\ r + r'}$$

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

Fig. 13. Conditional statement synthesis rules.

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

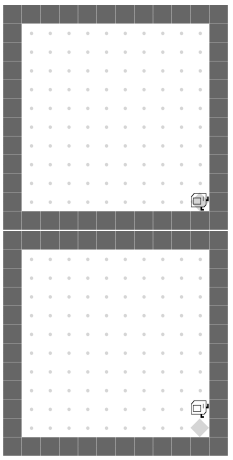


Fig. 14. PBR fills out the **if** statement body in Fig. 12. The pre and post state of `move` are depicted.

direction (Algorithm 1). Requiring matched postconditions at merge points (the `join` function) can be a too strong condition. For example, in rare situations, a program $\textbf{while}(b)\{\textbf{if}(b_1)\ S_1\ \textbf{else}\ S_2;\ S_3\}$ with inconsistent postconditions of S_1 and S_2 may be a valid task solution. When this happens, PBR can alternatively synthesize an equivalent program $\textbf{while}(b)\{\textbf{while}(\neg b_1 \wedge b)\{S_2;\ S_3\};\ \textbf{while}(b_1 \wedge b)\{S_1;\ S_3\}\}$, which does not need the conditional. Intuitively, the If-Merge-Fail rule simplifies conditional statement synthesis for low-level sketch completion at the cost of complicating high-level sketch generation. This design choice has proven to be effective in our experience, efficiently eliminating large sets of partial programs with useless conditional statements. It is particularly beneficial for robot control, where branch-merge behaviors are commonly observed at subgoals.

The If-Erase rule also applies to an $\textbf{if}\ (b)\ \{??_{S1}\}$ statement appended before a compound statement S_2 by the Action-LazyBranch rule. When the first statement to fill $??_{S1}$ is an action c_1^1

$$\textsc{While-Synth}\ \frac{b(\sigma) \qquad S \equiv s_1;\ s_2;\ \cdots;\ s_k;\ \texttt{skip} \qquad \sigma,\ S \triangleright S',\ \sigma',\ r \qquad \neg b(\sigma')}{\sigma,\ \textbf{while}\ (b)\ \{S\} \triangleright \bot,\ \bot,\ 0}$$

$$\textsc{While-Skip}\ \frac{\neg b(\sigma)}{\sigma,\ \textbf{while}\ (b)\ \{S\} \triangleright \textbf{while}\ (b)\ \{S\},\ \sigma,\ \mathbf{0}} \qquad \textsc{While-Sketch1}\ \frac{b(\sigma) \qquad \sigma,\ S \triangleright S',\ \sigma',\ r \qquad \sigma' = (\bot,\ \_)}{\sigma,\ \textbf{while}\ (b)\ \{S\} \triangleright \textbf{while}\ (b)\ \{S'\},\ \sigma',\ r}$$

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

Fig. 16. Loop statement synthesis rules.

and c_1^1 is equivalent to c_1^2 that is the first statement of S_2, the conditional is unnecessary as c_1^1 is executed on both branches regardless the Boolean condition b. The rule joins the preconditions and postconditions of the common action c_1^1 on both branches and removes the conditional.

Loop Statement Synthesis Rules. Fig. 16 depicts the synthesis rules for loops. The While-Synth rule applies to a loop statement **while** (b) $\{S\}$ to be executed from an environment state σ where the loop body S has just been completed by the synthesizer with a `skip` command. The rule expects the execution of S from σ to yield a state that satisfies b to ensure that S can be repetitively executed. Through this rule, PBR promotes the use of loops to explicitly capture repetitive behaviors for complex tasks and prunes away large sets of useless programs that execute the loop body S only once - if repetitive behavior is not needed, the task should have been solved with a simpler sketch without the loop structure. If σ happens to be the state that results in only one loop iteration before termination, PBR eventually generates a conditional statement to separate this corner case from the more general situation when the loop body is executed more than once. Although this rule risks generating infinite loops, PBR mitigates it by imposing a limit on the number of program execution steps during synthesis. Additionally, programs that do not demonstrate task progress receive low rewards and are therefore not favored by the synthesizer.

The While-Sketch1 rule monitors the execution of the first iteration of a loop and applies when the loop body S is partial or the evaluation of S results in a new sketch S' even S is complete. The latter happens when the program was synthesized from one initial state and is then evaluated on another initial state - the evaluation of S from the new initial state at some point produces a state inconsistent with the precondition inferred from past executions, triggering a new conditional statement appended to S by Action-LazyBranch.

The While-Sketch2 rule applies when the execution of the loop body S results in a new sketch (with some nonterminal) in any subsequent loop iteration beyond the first one. This happens when the execution of S encounters a new environment state at some point that does not meet the precondition inferred from past loop iterations. A new conditional statement is appended to S by the Action-LazyBranch rule to resolve it. The appended new control flow to S at a subsequent loop iteration does not affect prior loop iterations if executing S from exactly the same initial state σ. Fig. 11 and Fig. 12 exemplify how the rule is applied for loop body synthesis.

Sequential Statement Synthesis Rules. The Comp1 rule applies when the execution of S_1 results in a sketch S_1' (e.g. a conditional statement is appended if the execution from σ on S_1 encounters new states not permissible by previously inferred preconditions in S_1 or S_1 itself is a sketch). The final state of $S_1;\ S_2$ is $\bot$ as the program is incomplete. The Comp2 rule is straightforward. The final reward achieved by $S_1;\ S_2$ is the sum of the rewards by S_1 and S_2.

$$\textsc{Comp1}\ \frac{\sigma,\ S_1;\triangleright S_1',\ \sigma',\ r \qquad \sigma' = (\bot,\ \_)}{\sigma,\ S_1;\ S_2 \triangleright S_1';\ S_2,\ \sigma',\ r} \qquad \textsc{Comp2}\ \frac{\sigma,\ S_1;\triangleright S_1',\ \sigma',\ r_1 \qquad \sigma',\ S_2 \triangleright S_2',\ \sigma'',\ r_2}{\sigma,\ S_1;\ S_2 \triangleright S_1';\ S_2',\ \sigma'',\ r_1 + r_2}$$

Fig. 17. Composition statement synthesis rules.

Algorithm 1 Reward-guided Loop Sketch Completion.

1: **procedure** $\text{PBR}_{P_{ROG}}$ (e: `environment`, ε: `loop sketch`, Q: `search queue`, ξ: reward threshold)
2: $\quad\varepsilon_{best},\ r_{best},\ i \leftarrow \varepsilon,\ -\infty,\ 0$ ▷ i holds the current random seed
3: $\quad$**while** True **do**
4: $\quad\quad$**if** $i \geq N$ **then break** ▷ ϵ_{best} has full reward on all N seeds
5: $\quad\quad\sigma \leftarrow \textsc{reset}\ (e, i)$ ▷ Reset e with random seed i to an initial state.
6: $\quad\quad\varepsilon_{best},\ \sigma_f,\ \_ \leftarrow \textsc{reduce}\ (\sigma,\ \varepsilon_{best})$ ▷ Apply synthesis rules in Fig. 9, 13, 16, 17
7: $\quad\quad Q \leftarrow Q \cup \{\varepsilon_{best},\ \sigma_f,\ \textsc{eval}(\varepsilon_{best}),\ i\}$ ▷ Evaluate avg. reward of ε_{best} over all N seeds
8: $\quad\quad$**while** $Q \neq \emptyset \wedge$ (*max search budget not reached*) **do**
9: $\quad\quad\quad\varepsilon,\ \sigma_f,\ r,\ i \leftarrow \arg\max_{\{\varepsilon,\ \sigma,\ r,\ i\} \in Q} r - \lambda \cdot \textsc{cost}\ (\varepsilon)$ ▷ Dequeue a program ε
10: $\quad\quad\quad\varepsilon_{best},\ r_{best} \leftarrow \varepsilon,\ r$ **if** $r_{best} > r$ **otherwise** $\varepsilon_{best},\ r_{best}$
11: $\quad\quad\quad$**if** $\epsilon \neq \bot \wedge \sigma_f = (\bot, ??_{\mathbb{A}})$ **then** ▷ ε is a partial program with nonterminal $\mathbb{A}$
12: $\quad\quad\quad\quad$**for** *each produce rule* l `for` $\mathbb{A}$ **do**
13: $\quad\quad\quad\quad\quad\varepsilon' \leftarrow \textsc{expand}(\varepsilon, \mathbb{A}, l)$ ▷ Expand $\mathbb{A}$ using production rules
14: $\quad\quad\quad\quad\quad\sigma \leftarrow \textsc{reset}\ (e, i)$
15: $\quad\quad\quad\quad\quad\varepsilon',\ \sigma_f',\ r' \leftarrow \textsc{reduce}(\sigma,\ \varepsilon')$ ▷ Apply synthesis rules in Fig. 9, 13, 16, 17
16: $\quad\quad\quad\quad\quad$**if** $\sigma_f' \neq (\bot,\ \_) \ \wedge\ r' \geq \xi$ **then**
17: $\quad\quad\quad\quad\quad\quad\varepsilon_{best},\ i \leftarrow \epsilon',\ i + 1$ ▷ ε' achieves the full reward on seed i
18: $\quad\quad\quad\quad\quad\quad$**break** ▷ Next, search it on seed $i + 1$
19: $\quad\quad\quad\quad\quad$**else**
20: $\quad\quad\quad\quad\quad\quad Q \leftarrow Q \cup \{\varepsilon',\ \sigma_f',\ \textsc{eval}(\varepsilon'),\ i\}$ ▷ Eval avg. reward of ε' over all N seeds
21: $\quad\quad\quad$**if** (*max search budget reached*) **then** break
22: $\quad$**return** $\textsc{eval}(\varepsilon_{best})$ ▷ Evaluate avg. reward of ε_{best} over all N seeds

Theorem 4.1 (Soundness). *If* $\sigma,\ S \triangleright S',\ \sigma',\ r$, *then* $\langle \sigma, S' \rangle \Downarrow \sigma', r$

The Sketch Completion Algorithm. The synthesis algorithm PBR_{Prog} for sketch completion is depicted in Algorithm. 1. The algorithm takes as input a task environment e, a loop sketch ε, an initial program search queue Q, and a reward threshold ξ beyond which the task is considered solved (e.g. $\xi = 1.0$ for a binary sparse reward setting). The goal is to synthesize a robot-control program ε_{best} based on the sketch ε whose averaged reward performance over N (a hyper-parameter) random seeds meets ξ. We assume e can be reset to an arbitrary initial state given a random seed i (Line 5). On Line 6, the `REDUCE` function applies the synthesis rules to the current best program ε_{best}. Initially, Line 6 has no effect as there is not any statement in the initial sketch ε_{best}. However, at subsequent synthesis iterations, the `REDUCE` function could add new conditional statements when the execution of ε_{best} on a new initial state triggers PBR's on-demand conditional statement synthesis.

From Line 8 to Line 20, PBR_{Prog} maintains a priority queue Q of (partial) programs ϵ, together with the resulting states σ_f from executing ϵ from the initial state induced by random seed i, the reward r achieved by executing ϵ averaged over all the N random seeds, and the random seed i

used to search ϵ. At each step, the program ε with the best quality is dequeued from Q (Line 9). We measure the quality of ε based on both its structure cost $\textsc{cost}\ (\varepsilon)$ and its reward performance. Let each production rule of the DSL have a non-negative cost defined as $\texttt{cost}(l)$. The structural cost of a program ε is $\texttt{cost}(\varepsilon) = \sum_{l \in \mathcal{L}(\varepsilon)} \texttt{cost}(l)$, where $\mathcal{L}(\varepsilon)$ is the multiset of production rules used to construct ε. The algorithm introduces a hyper-parameter λ to control the trade-off between structure cost and reward performance. If ε is an incomplete program (Line 16), Algorithm 1 expands the nonterminal $\mathbb{A}$ reached during the execution of ε by enumerating all suitable production rules for $\mathbb{A}$ (Line 13). For each enumerated partial program ε', Algorithm 1 applies the synthesis rules to evaluate ε' based on the initial state induced by random seed i (Line 15). If ε' is a complete program that meets the reward threshold ξ, we break the inner loop to go on searching for ε' using a new initial state induced by the next random seed $i + 1$ (Line 18). Otherwise, ε' and its reward are enqueued for further processing (Line 20). The search process continues until a program achieves the maximum reward on all N random seeds, terminating the search (Line 4). Otherwise, the best reward obtained based on this loop sketch is returned.

4.2 High-Level Loop Sketch Generation

PBR generates high-level loop sketches that contain loop structures with loop bodies as "holes" to be completed by the low-level synthesizer (Sec. 4.1). We formulate loop sketch generation as tree search. A tree node represents a sketch permitted by the sketch grammar (same as Equation 5):

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

In the following, we use the terms "tree node" and "sketch" interchangeably. The root node corresponds to the empty loop sketch $??_S$. A tree edge (u, u') encodes a single-step application of a production rule in the sketch grammar to obtain the loop sketch u' by replacing a nonterminal $??_S$ in the sketch u. A loop sketch ε^* without any nonterminal $??_S$ is a leaf node. Sketches on leaf nodes only contain nonterminals $??_C$ (code block) that can be filled out by the low-level synthesizer PBR_{Prog} (Algorithm 1). Given an environment e, we measure the quality q of a sketch ε on a leaf node to solve e as:

$$q(e, \varepsilon^*) = \text{PBR}_{Prog}(e, \varepsilon^*, Q, \xi)$$

where the program search queue Q is initialized to $\emptyset$ and ξ is the task reward threshold. We further define the quality of a tree path based on the quality of the leaf node of the tree path.

Rather than enumerating tree paths to search for the best quality loop sketch, PBR uses the low-level synthesizer to provide feedback to guide high-level sketch generation. Specifically, PBR's loop sketch synthesis algorithm PBR_{Sket} is based on a variant of Monte Carlo Tree Search (MCTS). The PBR_{Sket} algorithm repeats four steps at each search iteration:

- Selection: The search tree is recursively traversed from the root guided by the score of each tree node (defined below) until reaching a child node u that either still has one or more unexplored $??_S$ nonterminals or is already a leaf node.
- Expansion: If the sketch u has unexplored $??_S$ nonterminals, for its leftmost nonterminal $??_S$, create a child node u' which is obtained by applying either the sketch production rule $??_S \ ::= ??_C;\ \textbf{while}\ b\ \{??_S\};\ ??_S$ or $??_S \ ::= \ ??_C$ to expand u.
- Simulation: If the sketch u is already a leaf node, we set $\varepsilon^* = u$. Otherwise, ε^* is obtained by iteratively expanding any remaining nonterminal $??_S$ in its child node u' until a leaf node is reached. PBR then measures the quality of ε^* as $q(e, \varepsilon^*) = \text{PBR}_{Prog}(e, \varepsilon^*, Q, \xi)$.
- Backpropagation: The reward feedback $q(e, \varepsilon^*)$ from the low-level synthesizer in the simulation step 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) [Kocsis and Szepesvári 2006] as follows: $UCT_u = (\overline{R_u} + 2C_p\sqrt{\frac{2\ln N}{N_u}})$ where $\overline{R_u}$ is the averaged reward received in simulations branching from node u, N is the number of visits of the parent of u during tree search, N_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 Appendix D.

Given a task environment e, the high-level sketch generation algorithm PBR_{Sket} repeatedly constructs a search tree following the above 4 steps until reaching a leaf node ε^* on which $\text{PBR}_{Prog}(e, \varepsilon^*, Q, \xi)$ can synthesize a complete program whose reward is beyond the reward threshold ξ. During the course of the search, a leaf node ε^* may be sampled multiple times by Monte Carlo Tree Search (MCTS) if it leads to a higher reward. We maintain the search queue Q for this node and allow low-level sketch completion each time to resume the search from Q.

4.3 Curriculum Synthesis

In robot control, agents are often faced with a sequence of tasks drawn from a task distribution, 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 to enable knowledge reusability across tasks via reusable skills. PBR 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. PBR adds programs synthesized to solve simpler tasks as new skills in the form of callable procedures to the DSL in Fig. 5. 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. 18. Lifting a synthesized program to a new skill.

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 4b can be abstracted to a procedure skill_j (where the specific name is not of significance), shown in Fig. 18. Adding this skill to the DSL, another robot-control program can be synthesized to iteratively invoke this procedure in a loop to clean each floor of a multi-floor house. As the set of robot skills grows in the DSL, the size of the environment action space increases, which introduces some overhead. However, since the program search space becomes more efficiently compressed by these new skills, curriculum synthesis expedites the generation of more complex programs, which would be challenging to synthesize from scratch given the computational constraints.

4.4 PBR Neurosymbolic Programming

State abstraction plays a crucial role in PBR, however, defining abstraction predicates over visual inputs in the form of RGB images for vision-based robot-control tasks is challenging. As a common practice in the RL community, we assume the existence of a simulator modeling the environment of a real robot. Leveraging the simulator, PBR synthesizes state-based robot-control programs using available state information from the simulator. Subsequently, PBR utilizes imitation learning in simulation to transfer these state-based programs into vision-based neurosymbolic programs, where state abstraction predicates such as `present(key)` are learned as neural net primitives for high-level scene abstractions (e.g. segmentation and classification).

Formally, for a synthesized state-based program π, we obtain a neurosymbolic program π^{vis} by replacing each state abstraction predicate h in π with a neural net NN^h_θ where θ is the set of training weights. NN^h_θ may be pretrained models to fasten the training. PBR performs DAgger-style imitation learning [Ross et al. 2011] of the decision result of h (teacher) on states from trajectories obtained by executing π^{vis} which in turn invokes the neural perception NN^h_θ (student). Specifically, the student policy π^{vis} is continuously executed in a vision-based simulation environment collecting trajectories stored in a replay buffer D_{vis}. The student NN^h_θ is trained to maximize:

$$\mathcal{L}(\theta) = \mathop{\mathbb{E}}_{O,\tau\sim D_{vis}} \Big[\sum_{s_t \in \tau} \log NN^h_\theta\big(h(O, s_t) \mid sim(s_t)\big) \Big]$$

where (O, τ) is a pair of object information o and state-based execution trace $\tau = (s_0, s_1, \ldots, s_T)$ obtained from the simulator, $sim(s_t)$ renders a state to an RGB image input, and $h(o, s_t)$ provides the ground-truth label for $NN^h_\theta(\cdot \mid sim(s_t))$. Intuitively, for vision-based robot-control tasks, the program π^{vis} serves as a symbolic prior over the neural perception modules NN^h_θ, which are trained modularly using existing deep learning techniques. Once trained, the execution of π^{vis} in the real-world environment does not depend upon any state information.

5 EXPERIMENTS

Implementation. To ensure program termination, in our implementation, we support the "return" statement as a special action for the agent to take to complete program execution when the goal is reached. PBR also adds a penalty to programs that do not terminate within the time horizon of a control task during program search.

Evaluation. We present a comprehensive empirical study of PBR, showcasing its capability to solve long-horizon robotic tasks with sparse reward signals. We demonstrate the effectiveness of PBR's synthesis strategy, which includes decomposed search space for loops, on-demand synthesis of conditional statements, and curriculum synthesis of new skills. In particular, 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 PBR algorithm?
- **(RQ2)** Does *curriculum synthesis* enable PBR to expedite the generation of large programs needed for complex tasks, which would be challenging to synthesize?
- **(RQ3)** How effective is PBR at learning programs for robotics tasks in *continuous state spaces*?
- **(RQ4)** How effective is PBR at learning *neurosybmolic* programs for *vision-based* robotics tasks?

Main Baselines. Throughout the evaluation, we consider two important DRL baselines.

- **DRL:** A deep reinforcement learning baseline that takes raw environment states as controller input for producing an action for the agent to execute. The RL algorithm is proximal policy optimization (PPO) [Schulman et al. 2017a], a state-of-the-art DRL algorithm. This baseline uses the ability of deep neural nets to learn useful (implicit) state abstractions to allow the agents to generalize between similar states.
- **DRL-abs:** A recurrent reinforcement learning baseline based on PPO that takes abstract states as controller input. This baseline allows for a fair comparison to PBR as both approaches utilize the same abstract state information. Specifically, all returned values of abstraction predicates e.g. { `frontIsClear() == true, present(marker) == fasle, ...` } are concatenated as a binary vector, which is then fed to the recurrent neural net policy as its input.

We introduce additional baselines when relevant below. We exclude hierarchical planning [Silver et al. 2021] as a baseline since it requires abstract transition models to determine the state transitions for each control action within the abstract state space. These transition models need to be

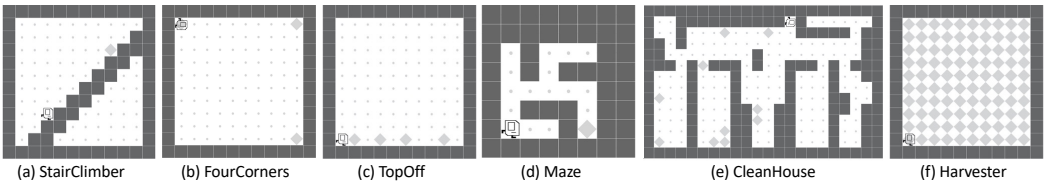


Fig. 19. Tasks of the "karel The Robot"

manually defined or learned from demonstrations, whereas PBR is a model-free learning approach. Furthermore, the controllers generated by hierarchical planning are typically environment-specific and lack the ability to handle task variations, which is a goal that PBR strives to achieve.

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 [Pattis 1981], taken from [Trivedi et al. 2021], to evaluate the capability of loop sketch synthesis and on-demand conditional statement synthesis. In these environments visualized in Figure 19, 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. 5) with the state abstraction predicates and actions defined in Equation 7. `CleanHouse` and `FourCorners` are running examples in Sec. 1 and Sec. 4.1. 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 placed until the agent either misses a marker or puts a marker in undesired positions (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. The reward is -1 if the agent attempts to place an extra marker and 0 otherwise. On `Harvester`, the goal is to pick up all the markers and the reward is determined by the ratio of successfully collected markers.
Hyperparameters To control the trade-off between structural costs and rewards, we set λ as 0.04 in Algorithm 1 for the low-level synthesizer PBR_{Prog}. Our ablation study on λ in Sec. C.2 shows that the particular value of λ does not significantly impact PBR's performance.
Additional Baselines Other than, DRL and DRL-abs, we evaluated PBR against the following baselines and ablations:

- EnumS: top-down enumeration which synthesizes and evaluates complete programs in order of increasing complexity measured by program structural cost (defined in Sec. 4.1).
- EnumR: a variant of top-down enumeration that executes partial programs to rank programs based on both rewards and structural costs normalized with a hyperparameter λ (same as in PBR). We optimize the λ by a serial of experiments and find $\lambda = 0.1$ to work the best.
- LEAPS [Trivedi et al. 2021]: A neural program synthesizer that learns a smooth program embedding space from a large set of pairs of sampled programs and their execution behaviors and then searches over the embedding space to synthesize a program with the best reward using stochastic search.
- PBR_{Sket}+EnumR: This ablation replaces the low-level PBR_{Prog} algorithm in PBR with the enumerative search method EnumR (see above) that completes a look sketch by top-down enumeration guided by both structure costs and reward signals.

Table 1. Comparing the mean reward performance (standard deviation) of PBR against the baselines on the Karel domain, evaluated over 1000 random seeds. The maximum reward value (a.k.a. reward threshold) is 1.0.

	StairClimber	FourCorners	TopOff	Maze	CleanHouse	Harvester
EnumS	0.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	0.02(0.00)	0.56(0.00)
EnumR	0.80(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	0.02(0.00)	0.56(0.00)
DRL	0.00(0.00)	0.31(0.07)	0.00(0.00)	0.90(0.00)	0.31(0.01)	0.91(0.04)
DRL-abs	0.99(0.00)	0.10(0.12)	0.01(0.00)	**1.00**(0.00)	0.39(0.00)	0.33(0.17)
LEAPS	-0.17(0.44)	0.25(0.00)	0.45(0.50)	0.18(0.38)	0.12(0.09)	0.33(0.00)
PBR_{Sket}+EnumR	0.00(0.00)	**1.00**(0.00)	0.40(0.49)	0.00(0.00)	0.00(0.00)	0.36(0.00)
EnumS+PBR_{Prog}	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)
PBR	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)

Table 2. Comparing the mean time cost (standard deviation) of PBR against the baselines on the Karel domain. Time costs are in seconds. >2hrs indicates a program with the full reward is not found.

	StairClimber	FourCorners	TopOff	Maze	CleanHouse	Harvester
EnumS	>2hrs	1170.82 (29.79)	1609.06(764.56)	3611.09(437.55)	>2hrs	>2hrs
EnumR	>2hrs	1187.31 (20.14)	184.21 (5.87)	2980.31(690.46)	>2hrs	>2hrs
PBR_{Sket}+EnumR	>2hrs	221.79(83.61)	>2hrs	>2hrs	>2hrs	>2hrs
EnumS+PBR_{Prog}	**20.13**(7.11)	3.76(3.97)	3.10(3.49)	135.04(106.61)	890.12(406.01)	1262.79(393.78)
PBR	22.02(7.89)	**3.52**(3.84)	**2.85**(3.35)	**124.83**(85.25)	**537.67**(145.05)	**171.38**(103.21)

- EnumS+PBR_{Prog}: This ablation replaces the high-level PBR_{Sket} algorithm in PBR with the enumerative search method EnumS (see above) that uses top-down search to enumerate loop sketches in order of increasing complexity with respect to loop sketch structural costs[1].

Results For each benchmark, PBR synthesizes a program based on 5 random initial environment states (Fig. 3). In evaluation, we test the program and report the average reward it achieves over 1000 random initial states. For a fair comparison, baseline models and the ablations are also evaluated on the same 1000 initial states. Table. 1 and Table. 2 depict the reward performance and execution time of all the tools. As depicted in Table. 1, PBR is able to synthesize a program that yields the maximum reward for all the tasks. *We have included the synthesized programs and their program execution trajectories in Sec. F of the supplementary materials.* The enumeration baselines perform significantly worse than PBR on all tasks, demonstrating the effectiveness of PBR's hierarchical loop synthesis and on-demand conditional statement synthesis. Among the tasks, `CleanHouse` has the most complex layout and requires an agent to turn into *all* rooms to pick up markers when present, while `Harvester` requires the agent to pick up markers on *every* spot and turn around the grid world *repetitively* at boarders. PBR synthesizes programs with loops to capture these behaviors. The DRL and DRL-abs methods fail to learn generalizable models that reliably and completely solve these two environments in all instances and also fail on tasks that require logical reasoning e.g. TopOff. Although LEAPS can potentially learn loop programs with stochastic search, `Harvester` and `CleanHouse` requires synthesizing complex state-conditioned loops and conditional statements, which are difficult to be sampled.

The first ablation PBR_{Sket}+EnumR evaluates the effectiveness of the low-level synthesizer in PBR - does on-demand conditional statement synthesis reduce the program search space? From

[1]EnumR cannot be used because rewards cannot be assigned to loop sketches that are not located on the leaf node of a loop sketch search tree.

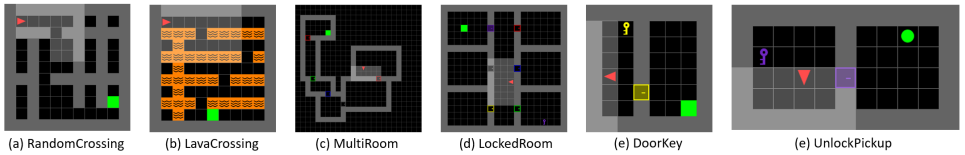


Fig. 20. Tasks of the MiniGrid Environments.

Table. 1 and Table. 2, PBR solves all the tasks in a few minutes. PBR without this strategy times out on 4 benchmarks. The second ablation EnumS+PBR$_{Prog}$ evaluates the effectiveness of the high-level synthesizer in PBR - does the Monte Carlo tree search strategy defined in Sec. 4.2 prioritize high-quality loop sketches? Table. 1 and Table. 2 show that PBR outperforms EnumS+PBR$_{Prog}$ by a significant margin on `CleanHouse` and `Harvester`. Both programs involve complex state-conditioned loop structures. The high-level synthesizer (PBR$_{Sket}$) effectively uses the feedback from the low-level synthesizer as a heuristic to guide the generation of highly likely loop sketches.

We additionally evaluate PBR on DoorKey depicted in Fig. 2. The challenge lies in that the initial agent positions and marker positions are randomly set. With the limited state abstraction predicates (Equation 7) in the DSL, our attempts to manually write a program that is correct for all potential initial states were unsuccessful. PBR synthesizes a program evaluated with 1.0 reward over 1000 random seeds. The DRL and DRL-abs baselines struggle with exploring the environments and can only reach the marker in the first room but fail to reach the goal in the second room (0.5 reward).

5.2 RQ2: Curriculum Synthesis

We evaluate how PBR can expedite program synthesis for a stream of tasks with various complexity using MiniGrid [Chevalier-Boisvert et al. 2018], a collection of grid world environments with goal-oriented tasks, which are widely used to evaluate state-of-the-art reinforcement learning algorithms. The tasks involve solving different mazes and interacting with various objects such as goals (green squares), doors, keys, boxes, and walls. In each environment, a positive reward of 1.0 is given only when the robot navigates to the (green) goal region.

$$\begin{aligned} \text{Objects } o &\in \{\texttt{goal}, \texttt{door}, \texttt{key}, \texttt{box}\} \\ \text{Skills } h &::= \texttt{present}(o), \texttt{leftIs(o)}, \texttt{rightIs(o)}, \texttt{leftIsClear()}, \texttt{rightIsClear()}, \texttt{frontIsClear()} \\ \text{Actions } c \in C &::= \texttt{put}(o), \texttt{pickUp}(o), \texttt{turnLeft()}, \texttt{turnRight()}, \texttt{move()}, \texttt{toggle()} \end{aligned} \tag{9}$$

The action space is significantly more complicated than that of Karel as the agent needs to manipulate various objects e.g. toggle to open a locked door with a picked-up key. The agent can detect if an object o is directly in its front or to its left or right within its visible area (gray region). These environments feature random layouts, agent initial positions, and goal positions.

We focus on 6 tasks depicted in Fig. 20. **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 connected rooms with closed doors to reach a goal. **LockedRoom**: the agent has to find a key in a room to unlock another room containing the goal. **DoorKey**: similar to LockedRoom, but the key is initially in the same room as the agent. **UnlockPickup**: the agent has to pick up a box placed behind a locked door in another room. These environments are notoriously challenging for classical RL algorithms due to their sparse reward nature [Chevalier-Boisvert et al. 2018]. Our DRL baseline here is the PPO agent [Schulman et al. 2017a] provided within the MiniGrid library. It gets

close to **zero** reward on **MultiRoom** and **LockedRoom**. To our knowledge, no RL algorithms have achieved the maximum reward for these tasks.

We depict PBR's learning performance in Fig. 21. As these environments increase in complexity as the number of objects to manipulate increases, PBR adds programs synthesized for a simpler environment as a new skill that can be reused to constitute sophisticated programs for more complex environments. For example, PBR adds the goal-reaching program synthesized for the third environment **Multiroom** as a new skill $\text{skill}_3(obj)$ to the DSL. This skill can then be reused as a new control action in the form of callable procedures such as $\text{skill}_3(key)$ and $\text{skill}_3(door)$. When synthesizing a program in **LockedRoom**, the agent can call $\text{skill}_3(key)$ to grab the key if it faces a locked door. It can then return to the locked door via $\text{skill}_3(door)$ to open it. New skills effectively compress the search space for the agent to explore long-horizon tasks. Using curriculum synthesis, PBR achieves the **maximum 1.0** reward on all the 6 environments evaluated over 5000 random seeds. Without curriculum synthesis, PBR failed to synthesize programs with the full reward. The synthesized programs feature multiple sequential loops with deeply nested conditionals making them impossible to be synthesized by EnumS or EnumR. We empirically find that the synthesized programs generalize to larger rooms in each of the environments. *We have included the synthesized programs in Sec. G of the supplementary materials.*

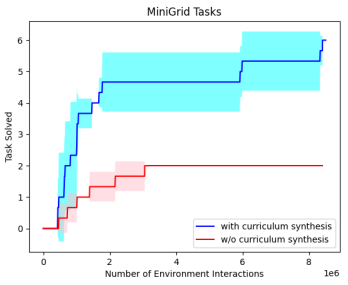


Fig. 21. PBR's learning performance for solving the MiniGrid tasks, evaluated over 5 trails each tested with 5000 random seeds.

5.3 RQ3: Robotics Tasks in Continuous State Space

We evaluate PBR using the following *continuous*, long horizon robotics tasks with weak rewards.

Highway Driving. We consider the task of a self-driving car navigating a multilane highway depicted in Fig. 22 (left) adapted from [Leurent 2018], aiming to drive at high speed in the rightmost lane while avoiding collisions with other vehicles. In the DSL, we define state abstraction predicates such as `frontIsClear()` : $\neg\exists c$: `car. ttc_front(self,`c`)` $< 3s$ to check if the predicted time-to-collision of observed vehicles on the same lane as the ego-vehicle is less than a threshold based on their current speed and position. Additionally, we have `leftIsClear()` : $\neg\exists c$: `car. ttc_left(self,`c`)` $< 3s$ which checks 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 directly borrowed from [Leurent 2018] which add a layer of speed and steering controllers on top of the continuous low-level control so that the ego-vehicle can automatically follow the road at a desired velocity. These skills change the target lane and speed that are used as setpoints for the low-level controllers. The reward is 1.0 at every timestep only if the ego-car drives on the rightmost lane with the highest permissible speed. The horizon is 50.

Fetch Environments. The Fetch-Pick & Place task is for a manipulator to pick up a block and place it on a target position on top of a table or in mid-air as depicted in Fig. 22 (middle), taken from [Plappert et al. 2018]. The robot used is a 7-DoF Fetch Mobile Manipulator with a two-fingered parallel gripper. We consider state abstraction predicates `Closing` (indicates if the gripper is closed), `NearObj` (indicates if the gripper of the arm 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`

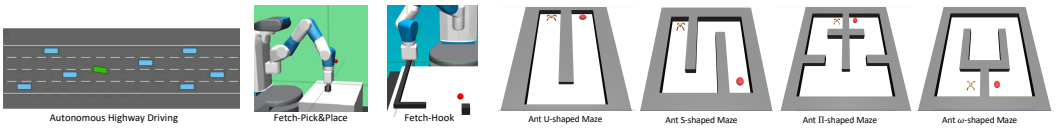


Fig. 22. Evaluating PBR in robot tasks with continuous state and action space. Task from left to right: Autonomous Highway Driving, Fetch Pick & Place, Fetch Hook, and Ant Navigation environments.

(indicates if the block is in the goal region $g \in \mathbb{R}^3$). These predicates are borrowed from [Jothimurugan et al. 2021]. The continuous action space is $\mathbb{R}^4$, where the first 3 components encode the target gripper position and the last encodes the target gripper width. Accordingly, the skills in our DSL are defined to include `openGripp()`, `closeGripp()`, `moveUp()`, `moveDown()`, `move(`g`)` that moves the gripper to the goal region g. The reward function assigns 0.5 if the block is successfully grasped, 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.

Ant Navigation. Our environments navigate the MuJoCo quadruped ant [Todorov et al. 2012] to reach a goal position (shown in the red sphere) in the 4 mazes depicted in Fig. 22 (right). The objective is to synthesize a *single* general program capable of reaching the goal regions in all 4 mazes. The DSL is equipped with the state abstraction predicates and skills in Equation 7. We adopt the same assumption used in [Duan et al. 2016]: the quadrupted ant receives range sensor readings about nearby obstacles as well as its goal (when visible). The predicates `frontIsClear()`, `leftIsClear()`, `rightIsClear()`, and `present(goal)` analyze 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 pretrained as neural net models `UP`, `DOWN`, `LEFT` and `RIGHT` by a deep RL algorithm SAC [Haarnoja et al. 2018]. 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 models correspondingly. Detailed descriptions of the predicates and skill modules are included in Appendix H. A positive reward is given only when the robot reaches the goal region. We define success as the program terminates within 600 control steps (i.e. the number of calls to `move`) and the ant is within 0.5 units of its goal position.

Results. Table. 3 compares PBR and the baselines on the highway autonomous driving environment. There is no particular goal in this environment. PBR learns a program that ensures safe avoidance of neighboring cars, facilitates lane changes to the right whenever possible, and maintains the highest permissible speed when the front is clear. In our observation, the DRL and DRL-abs baselines struggle to strike a balance between risk and speed. They tend to be either overly cautious, always slowing down (DRL-abs), or excessively daring, resulting in crashes (DRL).

Table. 3 depicts the goal-reaching success rate of PBR and the baselines on the other robot learning tasks, averaged over 5 trials are shown below and each trail evaluates the learned program or model 1000 times. PBR is able to yield the highest success rate for all the tasks. Curriculum synthesis enhances the efficiency of PBR in generating robot-control programs for the Hook environment. In this 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. Similarly, in the Ant navigation tasks, the program synthesized for Ant-U includes the behavior of turning left when facing a wall. When this program is evaluated in the Ant-S environment, PBR adapts by adding a conditional statement to instruct the ant to turn right for goal reaching. The learned program generalizes to the rest of the two environments. The synthesized program is depicted in Fig. 34 in Appendix H.

Table 3. Comparing the mean performance of PBR against the baselines over the robot learning tasks in Fig. 22, evaluated over 1000 random seeds for 5 trails. On Highway we show the accumulated rewards (as there is no goal in this environment) and we show the goal-reaching success rates for the other benchmarks.

	Highway (rewards)	Pick&Place	Hook	Ant-U	Ant-S	Ant-Π	Ant-ω
PBR	19	100%	100%	97.9%	98.5%	86.4%	93.9%
DRL	-0.02	27%	0%	0%	0%	0%	0%
DRL-abs	6.98	47%	16%	0%	0%	0%	0%

5.4 RQ4: Neurosymbolic Programming for Vision-based Tasks

We applied PBR's neurosymbolic programming technique (Sec. 4.4) to the following challenging vision-based robot-control environments.

Minecraft World. The Minecraft world is a complex 3D vedio game environment built with the Malmo platform [Johnson et al. 2016]. The agent takes visual inputs as 240x320 RGB images. It needs to navigate a randomly created multi-room cave to find a treasure box. The rooms are separated by randomly generated obstacles including **Lava** that the agent has to cross through a bridge, **Tunnel** that the agent has to find for entering a different room, and **Door** that has to be toggle (opened) by the agent in order go through the tunnel behind it. The agent gets a 1.0 reward only when the treasure box is reached within 60 control steps. Our DSL to synthesize a program for Minecraft is equipped with the state abstraction predicates and control actions defined in Equation 9 for obstacle detection and navigation. We apply PBR to synthesize a symbolic program in the Minecraft simulator utilizing state information. The synthesized program is depicted in Fig. 36 in Appendix I, which precisely specifies how a cave should be systematically traversed. We then replace the state abstraction predicates (e.g. `frontIsClear`) with deep neural networks that predict binary labels based on raw image observations. Details of the perception model architecture and data collection/labeling are described in Appendix I. For fine-tuning the pretrained neural perception modules, we execute this program in the game environment over 20 random seeds to collect misclassified observations.

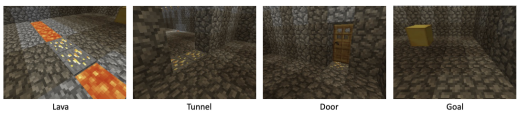


Fig. 23. Example scenes of the objects in the Minecraft game environment.

We compare the success rate of the neurosymbolic program over 500 random seeds in the Minecraft environment with the baselines. The success rates are shown below:

	DRL	DRL-abs	PBR	PBR-1-Finetune	PBR-2-Finetune
Minecraft	0%	40%	52.6%	78.6%	**96.4%**

The DRL agent takes visual inputs and directly predicts control actions from Equation 9. It can never make its way into the second room and fails in all cases (more details are given in Appendix I). The synthesized program succeeds in 52.6% episodes with pretrained perception modules. After finetuning the perception modules using misclassified observations collected from 20 runs, PBR achieves 78.6% success rate and obtains 96.4% success rate after finetuning for the second time. We additionally compared our method with a curriculum RL algorithm [Matiisen et al. 2019] in Appendix I and show that our technique outperforms it by a large margin.

Embodied-AI Household Environments. We have applied our neurosymbolic programming technique to finding specific types of objects in the virtually realistic AI2-THOR household environments [Kolve et al. 2017] as introduced in Sec. 1. The DSL is equipped with the perceptions and

control actions in Equation 7. The perception modules were trained as neural networks from **9000** depth images and **6000** RGB images from the AI2-THOR environment for obstacle detection and object identification. On the navigation tasks from RoboTHOR [Deitke et al. 2020], our neurosymbolic program outperforms a state-of-the-art RL model trained from **200 million** environment interactions by **44%**. We present the detailed results in Appendix J.

6 RELATED WORK

Two-staged Program Synthesis Our work is related to many prior works on decomposing a program synthesis problem into two phases such as [Feng et al. 2017a,b; Wang et al. 2017, 2020]. In the first phase, only example-consistent program sketches are enumerated, and the missing pieces in a sketch are synthesized in the second phase. Instead, PBR is a two-level *hierarchical* program synthesis framework. The high-level synthesizer leverages the feedback on sketch quality from low-level sketch completion to guide its sketch generation process. The feedback mechanism differs PBR with existing two-staged program synthesis techniques.

Execution-guided Program Synthesis Existing work extends neural program synthesis by incorporating execution traces to enable program generation beyond syntax-based approaches. Zohar and Wolf [2018] employs a neural network to process intermediate values of a program and prioritize search directions for a small DSL without conditionals and loops. Ellis et al. [2019] exposes execution results of programs encountered during a bottom-up search to a neural predictor. The predicted production rule probabilities are then used to prioritize more likely programs. Similarly, Chen et al. [2019] leverages intermediate values of partial programs that are consumed by a neural encoder-decoder model to guide top-down program search. PBR differs in the sense that it generalizes program states encountered in execution to presumed pre- and postconditions to enable on-demand conditional statement synthesis.

Programmatic Reinforcement Learning Existing methods mostly train a programmatic policy to imitate a pretrained RL oracle. They synthesize policies by enumerating a range of templates, including decision trees [Bastani et al. 2018; Silver et al. 2020], finite state machines [Inala et al. 2020a], or program sketches [Verma et al. 2019, 2018]. Other approaches optimize programmatic policies directly through RL methods using reward signals. To curb discrete program semantics, they leverage differentiable programs [Qiu and Zhu 2022] or a smooth program embedding space [Trivedi et al. 2021]. These methods are limited by the capabilities of existing RL algorithms and struggle with long-horizon sparse-reward tasks. Yang et al. [2021] uses a MaxSAT solver to synthesize straight-line programs to guide a policy toward a goal. However, the straight-line programs lack generalization to task variations.

We provide a more comprehensive discussion of related work on general program synthesis and reinforcement learning in broader contexts in Appendix A.

7 CONCLUSION

We present PBR, a programming-by-reward paradigm, that leverages program synthesis to address the exploration challenges in deep reinforcement learning for long-horizon robotics tasks with sparse rewards. By developing a novel synthesis algorithm with decomposed search space for loops, on-demand synthesis of conditional statements, and curriculum synthesis of robot skills as reusable procedures, PBR effectively compresses the search space for tackling long-horizon, multi-stage, and procedural robotics tasks. The experimental results demonstrate the superior performance of PBR in robot learning compared to program synthesis and deep RL baselines, showcasing its potential for advancing the field of robotics applications.

REFERENCES

Joshua Achiam. 2018. Spinning Up in Deep Reinforcement Learning. (2018).

Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. 2020. Neurosymbolic Reinforcement Learning with Formally Verified Exploration. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/448d5eda79895153938a8431919f4c9f-Abstract.html

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, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, Rina Dechter, Michael J. Kearns, and Richard S. Sutton (Eds.). AAAI Press / The MIT Press, 119–125. http://www.aaai.org/Library/AAAI/2002/aaai02-019.php

Pierre-Luc Bacon, Jean Harb, and Doina Precup. 2017. The Option-Critic Architecture. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, Satinder P. Singh and Shaul Markovitch (Eds.). AAAI Press, 1726–1734. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14858

Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 227:1–227:29. https://doi.org/10.1145/3428295

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/e6d8545daa42d5ced125a4bf747b3688-Paper.pdf

Elliot Chane-Sane, Cordelia Schmid, and Ivan Laptev. 2021. Goal-Conditioned Reinforcement Learning with Imagined Subgoals. In *ICML*.

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, and Yisong Yue. 2021. Neurosymbolic Programming. *Found. Trends Program. Lang.* 7, 3 (2021), 158–243. https://doi.org/10.1561/2500000049

Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021. Web question answering with neurosymbolic program synthesis. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 328–343. https://doi.org/10.1145/3453483.3454047

Xinyun Chen, Chang Liu, and Dawn Song. 2019. Execution-Guided Neural Program Synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=H1gfOiAqYm

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

Matt Deitke, Winson Han, Alvaro Herrasti, Aniruddha Kembhavi, Eric Kolve, Roozbeh Mottaghi, Jordi Salvador, Dustin Schwenk, Eli VanderBilt, Matthew Wallingford, Luca Weihs, Mark Yatskar, and Ali Farhadi. 2020. RoboTHOR: An Open Simulation-to-Real Embodied AI Platform. In *CVPR*.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

Thomas G. Dietterich. 1999. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *CoRR* cs.LG/9905014 (1999). https://arxiv.org/abs/cs/9905014

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 - Volume 48* (New York, NY, USA) *(ICML'16)*. JMLR.org, 1329–1338.

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 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 9165–9174. https://proceedings.neurips.cc/paper/2019/hash/50d2d2262762648589b1943078712aa6-Abstract.html

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017a. 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, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 422–436. https://doi.org/10.1145/3062341.3062351

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017b. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 599–612. https://doi.org/10.1145/3009837.3009851

Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. 2017. Differentiable Programs with Neural Libraries. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 1213–1222. http://proceedings.mlr.press/v70/gaunt17a.html

Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. TerpreT: A Probabilistic Programming Language for Program Induction. *CoRR* abs/1608.04428 (2016). arXiv:1608.04428 http://arxiv.org/abs/1608.04428

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

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*. PMLR, 1861–1870.

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*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 6765–6774. https://proceedings.neurips.cc/paper/2017/hash/32fdab6559cdfa4f167f8c31b9199643-Abstract.html

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask R-CNN. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 2980–2988. https://doi.org/10.1109/ICCV.2017.322

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016a. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016b. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

Rodrigo Toro Icarte, Ethan Waldie, Toryn Q. Klassen, Richard Anthony Valenzano, Margarita P. Castro, and Sheila A. McIlraith. 2019. Learning Reward Machines for Partially Observable Reinforcement Learning. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 15497–15508. https://proceedings.neurips.cc/paper/2019/hash/532435c44bec236b471a47a88d63513d-Abstract.html

Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. 2020a. Synthesizing Programmatic Policies that Inductively Generalize. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=S1l8oANFDH

Jeevana Priya Inala, Yichen Yang, James Paulos, Yewen Pu, Osbert Bastani, Vijay Kumar, Martin C. Rinard, and Armando Solar-Lezama. 2020b. Neurosymbolic Transformers for Multi-Agent Communication. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/9d740bd0f36aaa312c8d504e28c42163-Abstract.html

Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. 2016. The Malmo Platform for Artificial Intelligence Experimentation.. In *Ijcai*. Citeseer, 4246–4247.

Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. 2021. Compositional Reinforcement Learning from Logical Specifications. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 10026–10039. https://proceedings.neurips.cc/paper/2021/hash/531db99cb00833bcd414459069dc7387-Abstract.html

Apoorv Khandelwal, Luca Weihs, Roozbeh Mottaghi, and Aniruddha Kembhavi. 2022. Simple but Effective: CLIP Embeddings for Embodied AI. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*. IEEE, 14809–14818. https://doi.org/10.1109/CVPR52688.2022.01441

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR (Poster)*. http://arxiv.org/abs/1412.6980

Levente Kocsis and Csaba Szepesvári. 2006. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4212)*, Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou (Eds.). Springer, 282–293. https://doi.org/10.1007/11871842_29

Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. 2017. AI2-THOR: An Interactive 3D Environment for Visual AI. *arXiv* (2017).

John R. Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Stat. Comput. (UK* 4 (1994), 191–198.

John Langford. 2010. Efficient Exploration in Reinforcement Learning. In *Encyclopedia of Machine Learning*, Claude Sammut and Geoffrey I. Webb (Eds.). Springer, 309–311. https://doi.org/10.1007/978-0-387-30164-8_244

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 436–449. https://doi.org/10.1145/3192366.3192410

Youngwoon Lee, Shao-Hua Sun, Sriram Somasundaram, Edward S. Hu, and Joseph J. Lim. 2019. Composing Complex Skills by Learning Transition Policies. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=rygrBhC5tQ

Youngwoon Lee, Jingyun Yang, and Joseph J. Lim. 2020. Learning to Coordinate Manipulation Skills via Skill Behavior Diversification. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=ryxB2lBtvH

Edouard Leurent. 2018. An Environment for Autonomous Driving Decision-Making. https://github.com/eleurent/highway-env.

Tambet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. 2019. Teacher–student curriculum learning. *IEEE transactions on neural networks and learning systems* 31, 9 (2019), 3732–3740.

Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. 2018. Data-Efficient Hierarchical Reinforcement Learning. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 3307–3317. https://proceedings.neurips.cc/paper/2018/hash/e6384711491713d29bc63fc5eeb5ba4f-Abstract.html

Soroush Nasiriany, Vitchyr Pong, Steven Lin, and Sergey Levine. 2019. Planning with Goal-Conditioned Policies. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 14814–14825. https://proceedings.neurips.cc/paper/2019/hash/c8cc6e90ccbff44c9cee23611711cdc4-Abstract.html

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

Richard E. Pattis. 1981. *Karel the Robot: A Gentle Introduction to the Art of Programming* (1st ed.). John Wiley & Sons, Inc., USA.

Xue Bin Peng, Michael Chang, Grace Zhang, Pieter Abbeel, and Sergey Levine. 2019. MCP: Learning Composable Hierarchical Control with Multiplicative Compositional Policies. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 3681–3692. https://proceedings.neurips.cc/paper/2019/hash/95192c98732387165bf8e396c0f2dad2-Abstract.html

Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. 2018. Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research. arXiv:arXiv:1802.09464

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 107–126. https://doi.org/10.1145/2814270.2814310

Wenjie Qiu and He Zhu. 2022. Programmatic Reinforcement Learning without Oracles. In *International Conference on Learning Representations*. https://openreview.net/forum?id=6Tk2noBdvxt

Ahmed Hussain Qureshi, Jacob J. Johnson, Yuzhe Qin, Taylor Henderson, Byron Boots, and Michael C. Yip. 2020. Composing Task-Agnostic Policies with Deep Reinforcement Learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=H1ezFREtwH

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR,

8748–8763. http://proceedings.mlr.press/v139/radford21a.html

Stéphane Ross, Geoffrey J. Gordon, and Drew Bagnell. 2011. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011 (JMLR Proceedings, Vol. 15)*, Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík (Eds.). JMLR.org, 627–635. http://proceedings.mlr.press/v15/ross11a/ross11a.pdf

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. *SIGPLAN Not.* 48, 4 (mar 2013), 305–316. https://doi.org/10.1145/2499368.2451150

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017a. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017b. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

Ameesh Shah, Eric Zhan, Jennifer J. Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. 2020. Learning Differentiable Programs with Admissible Neural Heuristics. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/342285bb2a8cadef22f667eeb6a63732-Abstract.html

Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: component-based synthesis with control structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 73:1–73:29. https://doi.org/10.1145/3290386

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, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 10251–10258. https://aaai.org/ojs/index.php/AAAI/article/view/6587

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, Prague, Czech Republic, September 27 - Oct. 1, 2021*. IEEE, 3182–3189. https://doi.org/10.1109/IROS51168.2021.9635941

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

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*. IEEE, 5026–5033.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. 2021. Learning to Synthesize Programs as Interpretable and Generalizable Policies. In *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (Eds.). https://openreview.net/forum?id=wP9twkexC3V

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. 2018. HOUDINI: Lifelong Learning as Program Synthesis. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 8701–8712. https://proceedings.neurips.cc/paper/2018/hash/edc27f139c3b4e4bb29d1cdbc45663f9-Abstract.html

Abhinav Verma, Hoang Minh Le, Yisong Yue, and Swarat Chaudhuri. 2019. Imitation-Projected Programmatic Reinforcement Learning. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 15726–15737. https://proceedings.neurips.cc/paper/2019/hash/5a44a53b7d26bb1e54c05222f186dcfb-Abstract.html

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, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 5052–5061. http://proceedings.mlr.press/v80/verma18a.html

Alexander Vezhnevets, Volodymyr Mnih, Simon Osindero, Alex Graves, Oriol Vinyals, John P. Agapiou, and Koray Kavukcuoglu. 2016. Strategic Attentive Writer for Learning Macro-Actions. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 3486–3494.

https://proceedings.neurips.cc/paper/2016/hash/c4492cbe90fbdbf88a5aec486aa81ed5-Abstract.html

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, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 452–466. https://doi.org/10.1145/3062341.3062365

Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration using Datalog Program Synthesis. *Proc. VLDB Endow.* 13, 7 (2020), 1006–1019. https://doi.org/10.14778/3384345.3384350

Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. 2020. DD-PPO: Learning Near-Perfect PointGoal Navigators from 2.5 Billion Frames. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=H1gX8C4YPr

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 *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (Eds.). https://openreview.net/forum?id=QwNLVId9Df

Amit Zohar and Lior Wolf. 2018. Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 2098–2107. https://proceedings.neurips.cc/paper/2018/hash/390e982518a50e280d8e2b535462ec1f-Abstract.html

A MORE RELATED WORK

Symbolic Program Synthesis. Symbolic Program synthesis involves combinatorial search for programs that match a specification. Many different search methods have been explored within program synthesis, including search within a version-space algebra [Gulwani 2011], bottom-up enumerative search [Udupa et al. 2013], stochastic search [Schkufza et al. 2013], genetic programming [Koza 1994], or reducing the synthesis problem to logical satisfiability [Solar-Lezama et al. 2006]. EUPhony [Lee et al. 2018] biases the program search space by extending a regular CFG to a PHOG (probabilistic higher order grammar) in which a production rule is assigned a probability and parametrized by a context. The program synthesis procedure is accelerated by A^* search by casting rule probability to weight on a program deviation graph. FlashMeta [Polozov and Gulwani 2015] is a meta-framework that facilitates top-down program synthesis by using witness functions that captures the inverse semantics of operators in a language. Our synthesis procedure PBR differs as it solves a novel setting where user intent is specified using a reward function as opposed to input and output examples. The use of Monte Carlo tree search in PBR guides the search toward a space of programs. While PBR also considers program semantics for search space construction, it lazily synthesizes control flow structures on an as-needed basis. In terms of guided program search, PBR prioritizes partial programs that look promising by their reward. Similarly, FrAngel [Shi et al. 2019] guides the search to randomly combine fragments from partial solutions that satisfy any subset of the input-output examples. Probe [Barke et al. 2020] uses the syntactic information from partial solutions to update the weights of probabilistic grammar productions. However, these approaches cannot be used to guide the exploration of high-level program sketches and are less effective for RL environments with sparse reward signals.

Neurosymbolic Program Synthesis. Neurosymbolic programming emerges as a promising approach to combine learning and inference for programming systems with hybrid program/neural architectures [Chaudhuri et al. 2021]. WebQA [Chen et al. 2021] develops an optimal neurosymbolic synthesis technique for web question answering that combines pre-trained neural modules for natural language processing with programming constructs for string processing and tree traversal. TERPRET [Gaunt et al. 2017, 2016] allows gradient descent to backpropagate through differentiable programs to jointly learn neural net weights and program structures defined by a hand-written program template. The Houdini framework [Valkov et al. 2018] synthesizes neurosymbolic programs by composing trainable neural network modules within a functional program using type-directed enumeration. NEAR [Shah et al. 2020] completes missing expressions in a partial program by neural networks to approximate the best performance of any concrete program derivable from the partial program and uses it as a heuristic to guide program search. Adapting the heuristics developed in these existing techniques for programming by example to programming by reward is challenging, especially for programs that are non-differentiable.

Programmatic Reinforcement Learning. Our work is closely related to recent advance on exploring domain-specific programs as an interpretable representation for RL. Existing programmatic RL methods mostly synthesize programs to imitate pretrained RL models using a teacher-student learning paradigm. They synthesize programmatic RL models by enumerating a set of templates in the form of either decision trees [Bastani et al. 2018; Silver et al. 2020], finite state machines [Inala et al. 2020a], or program sketches [Verma et al. 2019, 2018]. Decision tree controllers are in general easier to learn but difficult to generalize to novel problems (e.g. Table. 1), while finite state machines have strong generalizing behavior but scale quadratically with the number of states. There also exist work [Icarte et al. 2019] that learns automata-based reward machines (from simulation traces) acted as external memory to stateless RL models. PBR alternatively synthesizes stateful programmatic

	StairClimber	FourCorners	TopOff	Maze	Harvester
LEAPS	0.00(0.00)	0.25(0.00)	0.00(0.00)	0.00(0.00)	0.00(0.00)
VIPER	0.00(0.00)	0.40(0.42)	0.03(0.00)	0.10(0.12)	0.04(0.00)
DRL-PPO	0.00(0.00)	0.00(0.00)	0.01(0.01)	0.00(0.00)	0.00(0.00)
PBR	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)

Table 4. Comparing the mean reward performance (standard deviation) of PBR against its baselines on the Karel domain, evaluated over 10 random seeds in 100×100 Karel state space. For all of our benchmarks, the maximum reward value (a.k.a. reward threshold) is 1.0.

RL models without having to resort to external memory. Anderson et al. [2020]; Inala et al. [2020b]; Verma et al. [2019, 2018] synthesizes simple stateless programmatic models to mimic pertrained deep neural net controllers using stochastic search. Yang et al. [2021] uses a MaxSAT solver to synthesize straight-line programs to guide a policy to reach a goal. Qiu and Zhu [2022] conducts bilevel optimization to jointly learn the structure and parameters of a loop-free programmatic RL model. Our approach differs as we (1) focus on applications where obtaining near-optimal neural net controllers as oracles is impossible and (2) consider stateful programs that can flexibly compose behaviors through state-conditioned loops and nested conditional statements, which enables our method to address long-horizon RL tasks with complex environment conditions and generalize to novel RL environments.

Compositional Reinforcement Learning. To improve RL efficiency, a line of research that transfers past skills into new skills for solving new complex problems has emerged. Lee et al. [2019, 2020] learn high-level polices to select the previously trained primitive policies. The macro-action models [Vezhnevets et al. 2016] similarly learn a high-level planner to combine actions sequences from primitives. These methods activate only one policy at each timestep. Other works have explored simultaneously composing multiple task-agnostic primitive skills by learning to assign weights for action composition [Peng et al. 2019; Qureshi et al. 2020]. Hierarchical reinforcement learning, such as option-critic [Bacon et al. 2017], is also a promising approach to solve complex tasks. However, option-critic is prone to inefficient task decomposition. Nachum et al. [2018] address this issue by automatically decomposing a complex task into subtasks and solving them by optimizing the subtask objectives. Our technique extends beyond existing work to efficiently solve long-horizon tasks with weak reward signals. Through the comparison between PBR and the DRL baseline in Sec. 5, we demonstrate the effectiveness of leveraging state abstraction and the novel programming-by-reward synthesis paradigm.

B GENERALIZATION EXPERIMENTS ON KAREL

We investigate whether programs synthesized by PBR and the baselines generalize to novel scenarios. We expand all Karel tasks except `CleanHouse` to the 100×100 grid size to investigate how well programs synthesized from their origin environment in Fig. 19 could generalize to a larger state space. We exclude `CleanHouse` as no baseline has a reasonable level of performance on it[2].

We sample 1000 random initial states for each task to evaluate PBR programs and the baselines programs and demonstrate the results in Table 4. Only PBR show generalizability. The rewards achieved by PBR programs remain 1.0 for all the tasks.

[2]Our `CleanHouse` program synthesized by PBR (Fig. 4b) generalizes to the grid size of 100×100.

	StairClimber	FourCorners	TopOff	Maze	CleanHouse	Harvester
Sket+EnumR	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	0.00(0.00)	1.00(0.00)
Sket+PBR$_{Prog}$	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)
PBR	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)	**1.00**(0.00)

Table 5. Comparing the mean reward performance (standard deviation) of PBR against its ablated version on the Karel domain, evaluated over 10 random seeds. For all of our benchmarks, the maximum reward value (a.k.a. reward threshold) is 1.0.

	StairClimber	FourCorners	TopOff	Maze	CleanHouse	Harvester
Sket+EnumR	142.15(1.38)	24.50(0.90)	>2hrs	3274.66(3668.57)	>2hrs	685.35(14.08)
Sket+PBR$_{Prog}$	6.10(1.28)	0.38(0.04)	0.44(0.07)	51.99(49.73)	31.13(0.30)	21.46(0.04)
PBR	22.02(7.89)	**3.52**(3.84)	**2.85**(3.35)	**124.83**(85.25)	**537.67**(145.05)	**171.38**(103.21)

Table 6. Comparing the mean time cost (standard deviation) of PBR against its ablated version on the Karel domain. Time costs are in seconds. >2hrs indicates a program with the full reward is not found.

C ADDITIONAL ABLATION STUDY

C.1 Ablation on PBR's Design Choices

We consider the following additional ablations of our method:

- Sket+EnumR: We evaluate the necessity of the low-level synthesizer PBR$_{Prog}$ by comparing its efficiency with the enumerative search method EnumR (see above) without lazy control flow structure synthesis. Note that loop sketch is provided in this method for comparison.
- Sket+PBR$_{Prog}$: We evaluate the strength of the low-level synthesizer in isolation with respect to a given loop sketch.

The results are depicted in Table. 5 and Table. 6. Recall that PBR is a hierarchical synthesis algorithm in which the the high-level synthesizer PBR$_{Sket}$ searches over loop sketches while the low-level synthesizer PBR$_{Prog}$ (Algorithm. 1) searches over a high-level loop sketch for a complete program.

The two ablations Sket+EnumR and Sket+PBR$_{Prog}$ evaluate the effectiveness of the low-level synthesizer in PBR - does lazily synthesizing control flow structures on an as-needed basis improve synthesis performance, assuming a handwritten, ground truth loop sketch is provided a priori? Table. 5 and Table. 6 confirm that lazy control flow structure synthesis is significantly more efficient. For example, on `Harvester`, PBR synthesizes the desired program in 21 seconds. The enumeration-based approach takes 685 seconds to find the same program.

C.2 Ablation on Hyperparameters

We evaluate the impact of the structure cost weight λ on the performance of PBR in Table 7. In general, PBR is stable with different choice of the values of λ. On `FourCorners`, `TopOff`, and `Harvester`, when λ increases, PBR takes longer time to synthesize the desired program. Especially, on `Harvester`, when only structural costs are considered, PBR cannot find a solution within 2 hours. Compared with `StairClimber` and `Maze`, these three tasks enjoy richer reward signals. For example, putting a marker at every corner on `FourCorners` gets a reward of 0.25. Thus, rewards play a significant role to guide program synthesis. Increasing the impact on structure costs weakens the guidance from reward signals in the synthesis procedure.

λ	StairClimber	FourCorners	TopOff	Maze	CleanHouse	Harvester
0.02	19.55(7.01)	3.68(3.99)	2.87(3.36)	124.35(86.83)	>2hrs	179.36(54.19)
0.04	22.02(7.89)	3.52(3.84)	2.85(3.35)	124.83(85.25)	537.67(145.05)	171.38(103.21)
0.1	19.73(7.05)	3.52(3.84)	2.98(3.62)	123.29(86.43)	457.75(138.18)	>2hrs
0.2	19.73(7.05)	3.71(3.68)	4.49(5.98)	123.57(87.27)	385.19(179.07)	>2hrs
Cost Only	21.63(7.75)	9.5(3.36)	4.42(6.03)	122.77(87.56)	409.79(189.94)	>2hrs

Table 7. Comparing the mean time cost (standard deviation) of PBR against various choice of λ. Time costs are in seconds. >2hrs indicates a program with the full reward is not found.

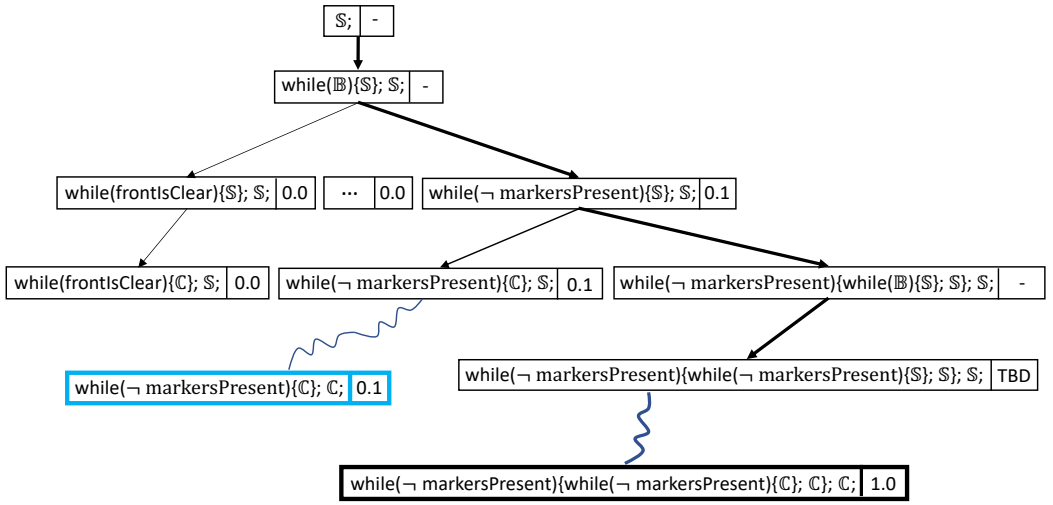


Fig. 24. Illustration of the MCTS-based Sketch Generation. Each tree node on the search tree maintains the average reward of this node. In the simulation step, browsing randomly the production rules for expanding the $\mathbb{S}$ nonterminals in the chosen child node n' to reach a leaf node for low-level sketch completion is, however, not efficient. This would often result in deeply nested or sequential loops. We replace the random simulation strategy in standard MCTS by a heuristic that simply replaces all the S nonterminals in n' using the production rule $\mathbb{S}$ ::= $\mathbb{C}$ to reach the nearest goal node to n'. The precision of the rewards on tree nodes improves with the growth of the tree.

D ILLUSTRATION OF THE MCTS-BASED SKETCH GENERATION

PBR is a novel *hierarchical* program synthesis procedure. At the high level, PBR generates "skeletal" programs, which represents a *loop sketch* that only contains sequential or nested while loop structures with the loop bodies as "holes" *yet to be determined*. For the `Cleanhouse` example, a loop sketch could be

$$\textbf{while}\ (\neg\ \texttt{MarkerPresent()})\ \{\ \mathbb{C}_1\ \};\mathbb{C}_2 \tag{10}$$

where $\mathbb{C}$ represents a missing hole. At the low level, we synthesize the missing pieces in a loop sketch. Notably, as visualized in Fig. 3, the low-level synthesizer provides feedback to guide the top-level synthesizer. In our method, the feedback is the best reward achieved by the low-level synthesizer on the high-level loop sketch. The high-level synthesizer, implemented as a variant of

Monte Carlo Tree Search, uses the feedback to prioritize a search path for generating the next loop sketch for low-level sketch completion.

For example, as depicted in Fig. 24, for `Cleanhouse`, PBR discovers a program on top of the loop sketch in Equation. 10 that can clean at least one marker out of the total markers in a house (the blue box in Fig. 24) and get a reward 0.1. The other single-loop sketches with a different loop condition cannot lead to a nonzero-reward program. The high-level synthesizer prioritizes loop sketches based on this condition and may expand the loop sketch in Equation. 10 to the following (the blackbox in Fig. 24) for iteratively picking up all the markers:

$$\textbf{while}\ (\neg\ \texttt{MarkerPresent()})\ \{\textbf{while}\ (\neg\ \texttt{MarkerPresent()})\ \{\ \mathbb{C}_1\ \};\ \mathbb{C}_2\};\ \mathbb{C}_3$$

E THE DRL-PPO ALGORITHM

We apply deep reinforcement learning with the PPO policy gradient algorithm as the baseline for discrete environments in our experiment. In this section, we introduce the framework as well as the training details of DRL-PPO. Karel states are represented as $H \times W \times 16$ array and MiniGrid states are represented as $H \times W \times 3$ arrays. Therefore, Convolutional Neural Network (CNN) is utilized as the state encoder to process observations from environment. Such network consists of two convolutional layers, the first of which is defined as 32 filters, kernal size 2 and stride 1; the second of which is defined as 32 filters, kernal size 4 and stride 1. ReLu activation layer is added after each convolutional layer. We pass the output of CNN to a linear layer and obtain a 512-dimension vector. Actor-critic framework is used for DRL-PPO baseline. Both actor and critic are constructed as Multi-Layer Perception (MLP) neural network with 2 hidden layers whose hidden size is 64 and share state encoder to get embedded observation vector as input.

The implementation of RL training for the MiniGrid domain is based on the train-ac RL algorithm[3] and we apply the default hyperparameters of train-ac in our experiment. As for the Karel domain, we implemented the model based on OpenAI SpinningUp [Achiam 2018], the hyperparameters setting of it is as below:

Karel Task	Clip Ratio	Entropy Reg
StairClimber	0.1	0.01
FourCorners	0.2	0.01
TopOff	0.2	0.0
Maze	0.05	0.001
CleanHouse	0.1	0.01
Harvester	0.2	0.01

For each Karel task, we train the network for 2000 epochs and set environment steps for each train epoch as 2000. Moreover, both the actor and the critic are trained for 40 steps in each train epoch with a learning rate 0.001.

F ROLLOUTS IN KAREL DOMAINS

We show the best program found for each Karel task by PBR in this section and visualize a rollout of each of these programs.

Figure 25 shows the success program for the StairClimber task. As the initial position of the agent is facing right and locates exactly above a stair (or wall), after executing the action sequence in the while body the agent would go up for one stair and face right again. As a result, after a few iteration of the while loop, the agent is guaranteed to reach the goal.

[3]https://github.com/lcswillems/rl-starter-files

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

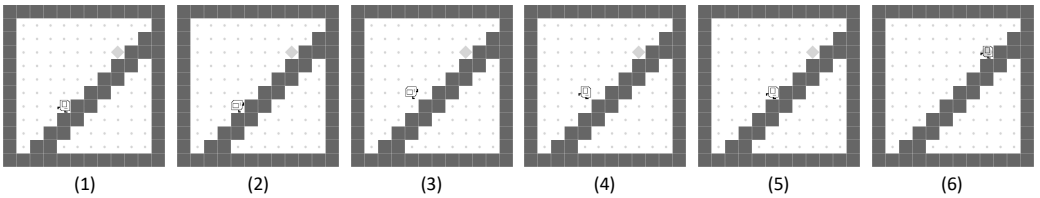


Fig. 25. Execution of the synthesized program on StairClimber.

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

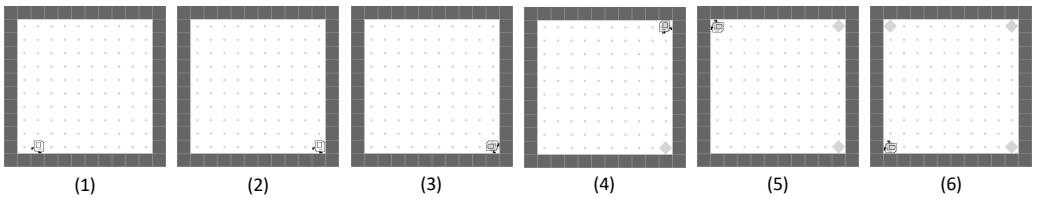


Fig. 26. Execution of the synthesized program on FourCorners.

Figure 26 represents the best program PBR synthesized for the FourCorners task. In the initial state, the agent is located in the second square of bottom row and faces right. It keeps moving forward until touching a wall in front of it to reach right bottom corner. Then, based on the if statement, the agent puts a marker in the corner and turns left to face a new corner. By executing the above actions repeatedly, the agent is guaranteed to reach all the corners and put a marker for each one.

Figure 27 shows a solution found by PBR for TopOff. The program is simple which allows the agent to move along the bottom row and put a marker wherever there exists a marker.

Figure 28 shows a success program synthesized by PBR for Maze. The agent travels to its right and turns back when it reaches dead end. Then it enters the second right-most path where the marker locates. By the end, it goes back to the initial position, faces left, and moves to the square above the initial position.

Figure 29 represents one success program PBR found for CleanHouse. The action sequence of inner while loop is similar to the program synthesized for Maze except that the program enters the

```
while (frontIsClear()) {
    if (present(marker)){
        put(marker);
    }
    move();
};
```

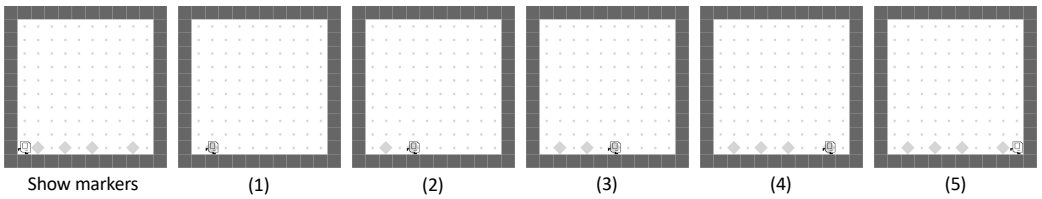


Fig. 27. Execution of Synthesized Program on TopOff task.

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

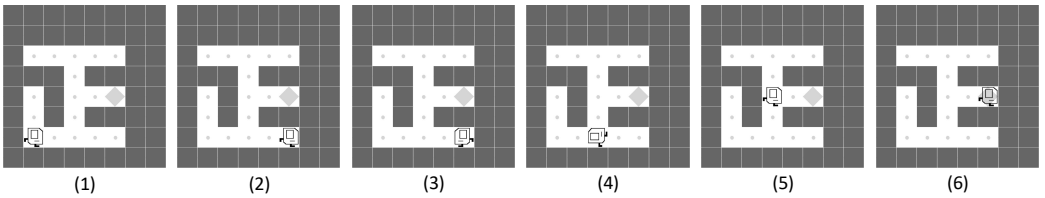


Fig. 28. Execution of the synthesized program on Maze.

left-most path first if there is a fork. When no marker is presented, the agent travels along the wall. After the agent discover a marker, the program exists the inner while loop and picks the presented marker. This action results in no marker present again and the program continues the outer while loop.

Figure 30 shows a success program synthesized for Harvester. The agent picks all the markers around the boarder at first based on the inner while loop. Then it attempts to pick markers along a column next to its position. When it touches a wall, the agent turns around to find another column.

G SYNTHESIZED PROGRAMS FOR MINIGRID

We show the synthesized programs for the MiniGrid environments in Fig. 32, and Fig. 33.

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

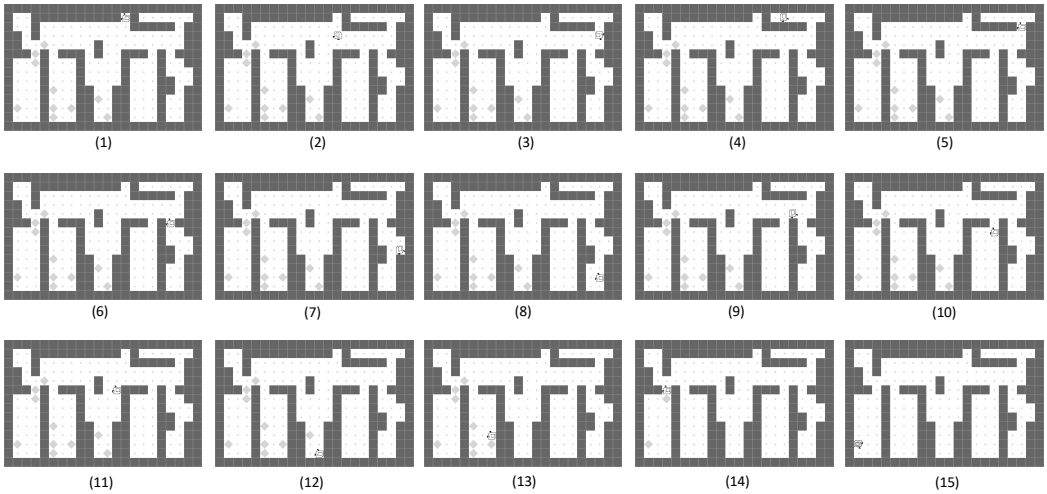


Fig. 29. Execution of synthesized program on CleanHouse.

	FindObj	MultiRoom	LockedRoom
PBR	701.70(181.71)	706.69(181.764)	1606.5(95.89)

Table 8. The mean time cost (standard deviation) in seconds of PBR for MiniGrid

H MORE DETAILS AND RESULTS OF THE CONTINUOUS CONTROL ENVIRONMENTS

H.1 Synthesized Neurosymbolic Program by PBR for Ant Navigation

We show the synthesized program by PBR for ant navigation in Fig. 34. Fig. 35 demonstrates example success trajectories of each ant navigation task. The maximum number of steps in an episode is 600. In the figure, the green triangles denote the initial positions, the red stars denote the goal positions, and the blue lines represent the trajectories.

First, the execution of the program with the neural modules leverages both the instructions in the program and the pre- and post-conditions of the instructions. For example, `move` under the condition `rightIsClear` in Fig. 34 would continue to apply the same primitive policy to move along a cardinal direction until the right of the ant is no longer `rightIsClear` as specified in the inferred post-condition of `move`. Second, after making a turn by `turnLeft` or `turnRight`, the ant may collide with obstacles or stuck at the corner then run into unstable states. The reason is that

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

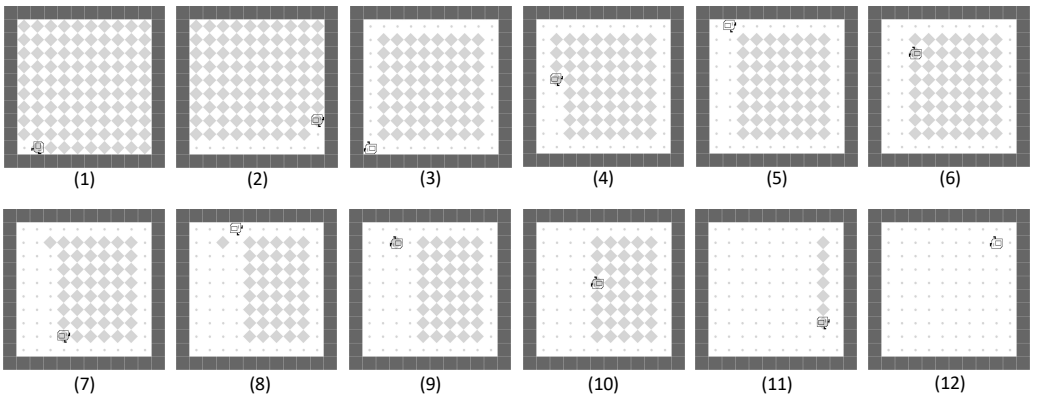


Fig. 30. Execution of the synthesized program on Harvester.

a turn usually happens when the ant crosses the boundary between multiple areas where sharp corners are present, and after a turn, the ant changes the direction of `move` and its associated primitive skill immediately. This unattended and immediate transition leads to failure trajectories. To alleviate this, we use a simple and intuitive transition strategy that gradually changes the trend of the move. Let $N(N \geq 1)$ be the total number of transition steps, and $i(1 \leq i \leq N-1)$ be the index of transition steps, the action `move` in transitions can be interpreted as $(\frac{i}{N} \times S_{new} + \frac{N-i}{N} \times S_{prev})$, where S_{new} denotes the primitive skill used in the new direction, and S_{prev} is the primitive skill of the previous direction. The experiment shows, with the transition strategy, the ant could make a smooth turn and traverse the maze efficiently.

We adopt the same assumption that used in [Duan et al. 2016]: the MuJoCo quadrupted ant has the direct sensory information of its surrounding environment. Specifically, the ant is aware of the maximum distance it can travel without colliding with any obstacles in any direction. Given the reference moving direction, we can compute the results for the `frontIsClear`, `leftIsClear`, and `rightIsClear` functions. Besides, the ant can realize the existence of the goal when it is visible and close enough (we set the visibility threshold to be 4.0 unit). Thus, the ant knows the relative angles between its current moving direction to the direction it facing towards the goal. Based on this, the

```
while (frontIsClear()) {
    if (present(closedDoor)) {
        turnRight();
    }
    move();
};
turnRight();
while (not rightIs(goal)) {
    if (present(goal)) {
        return();
    }
    if (leftIsClear()) {
        turnLeft()
        if (frontIsCloasedDoor()) {
            toggle();
        }
    }
    if (not frontIsClear()) {
        turnRight();
    }
    move();
};
turnRight();
while (not present(goal)) {
    move();
};
```

Fig. 31. Synthesized Program for FindObj.

	U-shaped Maze	S-shaped Maze	Π-shaped Maze	ω-shaped Maze
HIRO [Nachum et al. 2018]	0	0	0	0
LEAP [Nasiriany et al. 2019]	0.963	0.633	0.602	0.185
RIS [Chane-Sane et al. 2021]	0.985	0.993	0.925	0.955
PBR	0.979	0.985	0.864	0.939

Table 9. Success rate of tasks in of Ant Navigation Environment. The Results are averaged over 1000 evaluations. The maximum episode length for U-shaped Maze, S-shaped Maze, ω-shaped Maze is 600, and this number is set to 800 for Π-shaped Maze. HIRO, LEAP, and RIS models are trained on each individual maze for 1e6 environment steps.

`pickUp(goal)` function could compute the weights of primitives skills so that the weighted sum of primitives can always point toward the goal directly.

H.2 Comparison with RIS

Without fine-tuning the primitive skills models, we evaluate our program in all four tasks 1000 times and compare the average success rate with Reinforcement Learning with Imagined Subgoals (RIS) [Chane-Sane et al. 2021]. The results are shown in Table 9. From the table, we can see that the PBR's results are competitive with current state-of-the-art results from RIS, yet our method yields intuitive and explainable programs. What is more, PBR is significantly more sample-efficient than RIS by reusing primitive skills across all tasks and generalizes zero-shot.

PBR has following additional advantages over RIS:

```
while (frontIsClear()) {
    if (present(closedDoor)) {
        turnRight();
    }
    move();
};
turnRight();
while (not rightIs(goal)) {
    if (present(goal)) {
        return();
    }
    if (present(closedDoor)) {
        turnRight();
    }
    if (leftIsClear()) {
        turnLeft()
        if (frontIsCloasedDoor()) {
            toggle();
        }
    }
    if (not frontIsClear()) {
        turnRight();
    }
    move();
};
turnRight();
while (not present(goal)) {
    move();
};
```

Fig. 32. Synthesized Program for MultiRoom.

- **Primitive reusability**: we may reuse any primitive skills in any programs without re-training or fine-tuning.
- **Program generalizability**: the synthesized program shown in Fig. 34 is a generalized program that works in all four ant navigation tasks, while RIS requires training from scratch in every new domain.
- **Program interpretability**: PBR yields intuitive and interpretable programs that involves common program structures and control flows, e.g., sequential flow, while loop, if-else statement, etc. This helps developers understand how these tasks are solved at a logic level, and the program itself can even be served as a prototype for solving more complex problems.

H.3 Comparison with classic RL algorithms

We further evaluate whether these ant navigation tasks can be solved by classic reinforcement learning algorithms. To match the experimental settings, besides observations introduced by the standard MuJoCo Ant, we additionally include basic sensory information: the goal position, and the maximum distance it can travel in four absolute directions (positive/negative x-axis and positive/negative y-axis) in the modified agent's observation space. The agent is expected to output actions to control the ant to achieve the goal in the maze. Results shows, none of Soft Actor-Critic (SAC) [Haarnoja et al. 2018] or Proximal Policy Optimization (PPO) [Schulman et al. 2017a] can train an agent to achieve the goal for once in any environment, not mentioning a universal agent can solve all of them. The reason is due to the problematic reward signals. In these tasks, sparse

```
while (frontIsClear()) {
    if (present(closedDoor)) {
        turnRight();
    }
    move();
};
turnRight();
while (not rightIs(goal)) {
    if (present(goal)) {
        return();
    }
    if (leftIsClear()) {
        turnLeft()
        if (present(closedDoor)) {
            toggle();
        }
    }
    if (not frontIsClear()) {
        if (present(lockedDoor)) {
            get(key);
            pickUp(key);
            get(lockedDoor);
            toggle();
        }
        else {
            turnRight();
        }
    }
    move();
};
turnRight();
while (not present(goal)) {
    move();
};
```

Fig. 33. Synthesized program for LockedRoom.

```
while (not present(goal)) {
    if (rightIsClear()) {
        turnRight();
        move();
    } else {
        if (not frontIsClear()) {
            turnLeft();
            move();
        } else {
            move();
        }
    }
};
pickUp(goal);
```

Fig. 34. Synthesized program for Ant Navigation Environment.

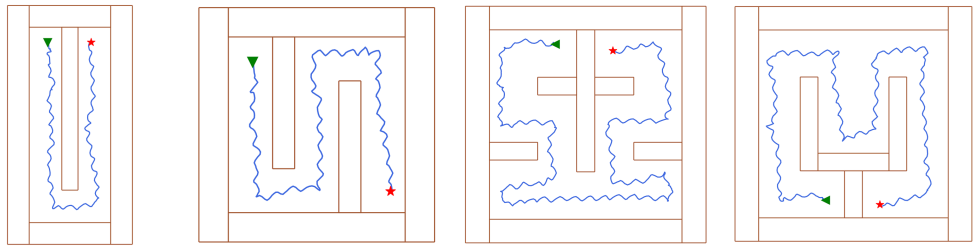


Fig. 35. Example successful trajectories of ant navigation tasks.

rewards are unable to encourage exploration effectively, while dense distance-based rewards are misleading.

I MORE DETAILS AND RESULTS OF THE VIDEO GAME ENVIRONMENTS

I.1 Synthesized Symbolic Minecraft Programs by PBR

PBR synthesizes the following symbolic program (Fig. 36) in the simulated environment in Fig. 23. The program is a symbolic prior over the perception modules e.g. `rightIsClear`, `frontIsClear`, and `left(goal)` trained using deep learning techniques in the realistic 3D game environment. We discuss the details of perception module training in Sec. I.3.

```
while (not leftIs(goal)) {
    if (rightIsClear()) {
        turnRight();
    }
    if (not frontIsClear()) {
        if (present(door)) {
            toggle();
        }
        else {
            turnLeft();
        }
    }
    move();
};
turnLeft();
move();
```

Fig. 36. Synthesized program for Minecraft Environment.

In the Minecraft environment, besides `move`, `turnLeft` and `turnRight`, an additional action `toggle` is included for the agent to open a closed door. By the program, the agent traverses around the cave. Based on the perception function "present(door)", when its front is not clear, the agent detects whether its front is a door or a wall. If its front is a door, the agent would execute the action "toggle" to open it. If front is a wall, the agent would turn left to continue to traverse the cave.

I.2 Perception Models

The Minecraft agent makes control decisions based on visual inputs as 240x320 RGB images. The agent is equipped with 6 perception functions, `frontIsClear`, `leftIsClear`, `rightIsClear`,

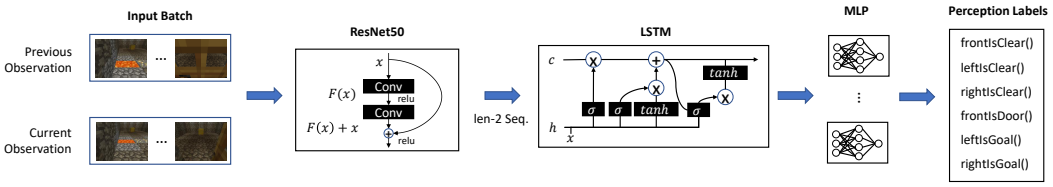


Fig. 37. The Perception Model with ResNet and LSTM.

`present(door)`, `leftIs(goal)`, `rightIs(goal)` for obstacle, door and goal detection. We train a deep neural network as shown in Fig. 37 that takes raw image observations as input and predicts one of the 6 perception results. The perception modules are pretrained over 4605 realistic scenes with 1000 data points for validation. To predict the perception label of an observation, we take both the previous and current observations as input. Resnet50 [He et al. 2016a] pretrained with ImageNet [Deng et al. 2009] is leveraged to encode two consecutive observations separately and LSTM [Hochreiter and Schmidhuber 1997] is then used to combine the two encoded variables. The final output of LSTM is passed to 6 MLPs for the prediction of perception labels respectively. During training, Resnet 50 is freezed and we update parameters of LSTM and MLPs with the Adam optimizer.

I.3 Train Details

In order to control the agent to traverse in Minecraft cave, we train the perception models to predict perception labels required in the synthesized program. During pretraining, we collect 4605 groups (past and current) of observation images as the train set and 1000 groups of images as the validation set. We use Adam as the optimizer and set the learning rate to be 0.001 which is divided by 10 for each 50 epochs. Moreover, we use batch size 100 to train the model for 200 epochs. To achieve better accuracy, we utilize synthesized program as policy and simulate the agent in real world with 20 different seeds to collect misclassified observations for finetuning. We repeat the finetuning process for 2 times, and collect 66 and 210 groups of observation images respectively. For each finetuning, we group the collected images with the images used for pretraining to further train the model for 50 more epochs with learning rate as 0.0001. The details of data collection are shown as follows:

- To collect a dataset for pretraining, we prepare several Minecraft environments with various random seeds. Then the agent is randomly placed in the environments and execute a random action with the purpose to collect its past and current observations. In order to obtain the ground truth perception label, we let the agent to move forward, left and right for one block and open the front door (if any) to detect clear directions, the goal position and door positions.
- To collect a dataset for finetuning, we simulate the synthesized program with the trained perception models for at most 60 steps. After each step, we obtain the ground truth perception labels based on the past and the current observation and compare them with the predicted perception labels. If the perception models make incorrect prediction, we collect the past and current observations at this step in the finetuning dataset.

I.4 More Results

We compare the reward performance of our neurosymbolic program model with Proximal Policy Optimization (PPO) [Schulman et al. 2017b] and Teacher Student Curriculum Learning (TSCL) [Matiisen et al. 2019] in Table. 10. We follow TSCL to define the reward:

	PPO	TSCL	PBR-0-F	PBR-1-F	PBR-2-F
Minecraft	-1045.00	730.00	48.73	567.87	**924.28**

Table 10. Result of Minecraft Environment.

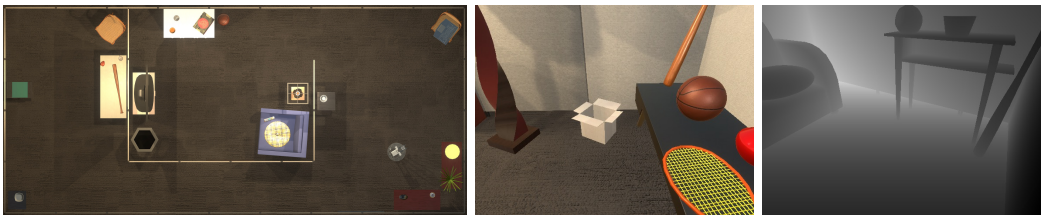


Fig. 38. An example scene in RoboTHOR ObjNav (left) and RGB (middle) / depth (right) image of the robot.

- If the agent fails to find the goal, the reward it gets yields to -1000.
- If the agent successfully reaches the goal, it receives 1000 as reward.
- For each `move` execution, a punish of reward -0.1 is given.

Directly applying PPO in the Minecraft environment makes no progress in reaching the goal and gets -1045 as its reward. With the help of curriculum learning, TSCL achieves a reward 730 by learning to reach the goal in simple cases (e.g. Minecraft with only 1 room) firstly and then training on more complex cases step by step. As comparison, PBR with 2 finetune times gets the best reward, 924.38. Notice that although PBR without finetuning results in lower reward compared with TSCL, the synthesized program succeeds in 52.6% episodes over 500 random rollouts. The perception models give wrong prediction in the other episodes. PBR achieves 78.6% success rate after finetuning for the first time and 96.4% success rate after finetuning for the second time.

J MORE DETAILS AND RESULTS OF THE EMBODIED AI HOUSEHOLD ENVIRONMENTS

Environment Setting. We have applied our neurosymbolic programming technique to household environments introduced in Sec. 1. We focused on the RoboTHOR ObjectNav benchmark [Deitke et al. 2020] in the AI2-THOR [Kolve et al. 2017] interactive environments for embodied AI. In this task, a home-assisted LoCoBot robot[4] uses ego-centric 480 × 600 RGB-D camera inputs (Fig. 38) to navigate to specific types of objects (e.g., apple and basketball) in households with various floor plans within 500 control steps. The initial position and orientation and goal object type are randomly generated in each trial. The benchmark requires the robot explicitly execute the `stop` command to terminate a trial and the trail is successful when finally the desired object is visible within 1 meter to the robot. We require the synthesized program terminates itself when the stopping condition is satisified. We synthesize a neurosymbolic program for ObjectNav in our DSL equipped with the perceptions and control actions in Equation 7.

Perception and object detection. As the agent has to take visual inputs as RGB-D images, the perception modules are deep neural networks with raw image observations as inputs. Unlike the Minecraft environment where the agent make 90 degree turns, the rotation angle of the robot in the AI2-THOR household environments is set to 30 degrees (clockwise or counter-clockwise), and each step moves 0.25 meter. We define `leftIsClear` (resp. `rightIsClear`) as any direction in two steps

[4]Locobot: an open source low cost robot. http://www.locobot.org/

Table 11. Success rate of tasks in Embodied AI Household Environment. Results are averaged for 3 runs.

	Apple	BaseballBat	BaseketBall	HousePlant	Laptop
Embodied-Clip	0.51	0.27	0.55	0.46	0.33
PBR	0.59	0.47	0.70	0.70	0.59

of `turnLeft` (resp. `turnRight`) is clear for the robot to `move` two steps. These perception modules were pretrained over 9000 depth images sampled from realistic scenes in 75 AI2-THOR virtual environments. Details of the training setup and data collection are described in Appendix J.1. The other perception function `present`(o) needs to understand the semantics of its view to identify if an object o is visible. To this end, we finetune a pre-trained Mask R-CNN [He et al. 2017] object detection model to detect target objects over 6000 samples RGB images covering all object types presented in the AI2-THOR virtual environments. `present`(o) applies the object detector to the current ego-centric RGB image and once o is detected with a high confidence (0.5 confidence score), it correlates the object mask with the current depth image to estimate if the distance between the robot and o is less than 1 meter (Appendix J.2). We apply PBR to synthesize a symbolic program in the discrete simulation environment shown in Fig. 4a that approximates the realistic AI2-THOR household environments with the initial robot and target object positions randomly placed everywhere. The reward function is the same to that of Minecraft.

Results. The synthesized program is shown in Fig. 39. We selected five representative object types (`Apple`, `BaseballBat`, `BasketBall`, `HousePlant`, and `Laptop`) as the target object types. We sample 100 AI2-THOR realistic navigation environments for each target type and compare the success rates for these tasks by our synthesized neurosymbolic program against a modern RL method Embodied-CLIP [Khandelwal et al. 2022]. The model is equipped with a pretrained state-of-the-art visual encoder CLIP [Radford et al. 2021] for processing perceptual inputs and was trained by a state-of-the-art deep reinforcement learning algorithm (DD-PPO [Wijmans et al. 2020]) extensively using 200 million environment interactions. The results are shown in Table 11. On these navigation tasks from the RoboTHOR benchmarks, the neurosymbolic program significantly outperforms Embodied-Clip despite being trained with order-of-magnitude less environment interactions.

In the AI2-THOR environments, other than `move`, `turnLeft`, `turnRight`, and a special action `stop` that terminates the current trajectory, the robot additionally has two primitive actions `lookUp` and `lookDown`. For simplicity, we use a pre-set viewing angle that fits well for navigation tasks, so no `lookUp` or `lookDown` are involved in our DSL.

J.1 Perception

To construct the perception model, we acquire three predicate functions: `frontIsClear`, `leftIsClear` and `rightIsClear` to predict whether it is clear in the robot's *abstract* directions. Specifically, each function contains a predicate model which processes the encoded depth image of the current view and outputs a real value that represents the maximum number of steps in each direction. After comparing with thresholds, these numbers are converted to *true* or *false* final predictions.

In all 75 scenes, we teleport the locobot to 120 random valid positions with random orientation and save the depth image as the input. We collect $75 \times 120 = 9000$ data points in total for training. We define s_θ ($-60 \leq \theta \leq 60$) as the maximum number of steps the robot can move in θ degree direction, where positive values of θ in degree denote clockwise rotation and negative values denote counterclockwise. We collect $s_f = s_0$, $s_l = max(s_{-60}, s_{-30})$, $s_r = max(s_{+60}, s_{+30})$ to represent maximum steps in *front, left,* and *right* directions, respectively. Given ground-truth depth images

```
while (frontIsClear())
    move();
turnLeft();
while (not present(target_object)) {
    if (rightIsClear()) {
        turnRight();
        move();
    } else {
        if (not frontIsClear()) {
            turnLeft();
            move();
        }
        else move();
    }
};
```

Fig. 39. Synthesized program for EAI Household Environment.

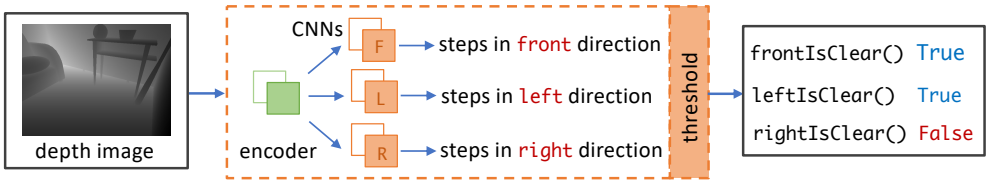


Fig. 40. Architecture of the perception model, fully connected layers are omitted. We set the `Isclear` threshold to 2.5 steps for `front` and `left` direction, and 1.5 step for `right` direction.

```
class DepthImageEncoder:
    model = Sequential(
        Conv2d(in_channels=1, out_channels=3, kernel_size=3, stride=1),
        BatchNorm2d(3),
        ReLU(),
        MaxPool2d(kernel_size=3),
    )
```

Fig. 41. Network architecture of the depth image encoder.

and their corresponding max number of steps in each *directions* as labels, we train a perception model that contains one depth image encoder and three predicate models. Each predicate model is responsible for predicting the maximum number of steps of moves. At this point, the predicate models can predict real numbers given depth images, yet the expected output is the binary `True` or `False` prediction. We then defined that, once a predicted value is larger than a threshold (e.g, 2 steps), the function gives positive output that it *is clear* in that direction.

We use a simple network architecture to construct our perception model (see Fig. 40). The architecture of the universal depth image encoder and predicate models are listed in Fig. 41 and Fig. 42, respectively. During training time, we use Adam optimizer [Kingma and Ba 2015] with the learning rate 0.004, set the batch size to 64, and we train the model which contains one depth image encoder and three predicate models jointly for 100 epochs. After the training completes, the final overall success prediction rate of the perception is 95.35%.

```
class PredicateModel:
    model = Sequential(
        Conv2d(in_channels=3, out_channels=3, kernel_size=3, stride=2, padding=1),
        BatchNorm2d(3),
        ReLU(),
        MaxPool2d(kernel_size=2),
        Flatten(),
        Linear(in_features=6360, out_features=2048),
        ReLU(),
        Linear(in_features=2048, out_features=5),
    )
```

Fig. 42. Network architecture of a predicate model.

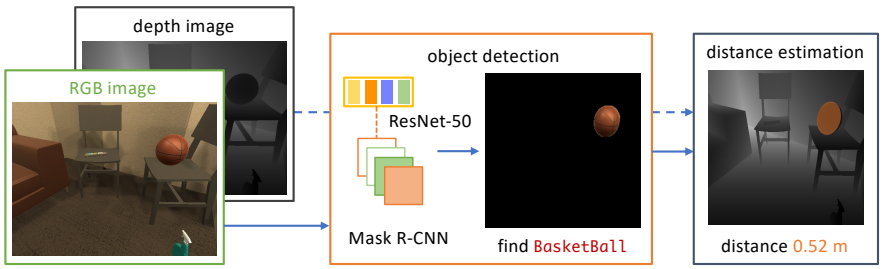


Fig. 43. Implementation of `present(target_object)` function. In the example, the function identifies the target object `BasketBall` and estimates the distance is less than 1.0 meter.

J.2 Object detection

To identify if a target object is present in an RGB image and estimate the distance to the object, we implement the function `present(target_object)` to predict whether the desired object is visible in the view and is close enough to the robot. We choose Mask R-CNN [He et al. 2017] as the object detector. For simplicity, we use torchvision [Paszke et al. 2019] implementation of pre-trained Mask R-CNN model with ResNet-50 [He et al. 2016b] backbone as the starting point for fine-tuning.

We collect the RGB image dataset in the same way as we collect depth images by randomly teleporting the robot. The major difference is that we ensure that each RGB image contains at least one visible target object in the robot's view, and we collect 500 images for each type of target object. The total number of these RGB images is 6000. We summarize the training and inference hyperparameters in Table. 12. During inference time, when a target object is identified in the RGB view of the robot, Mask R-CNN not only indicates the existence of the target object, but also predicts the segmentation mask of the predicted region. We then interpolate the depth image to match the size of the output mask, and then filter the depth information of the target object from the ground-truth depth image (see Fig. 43). We use a simple heuristic that takes the average distance of every pixel's distance information as the overall distance to the robot. Finally, if the target object is found and the averaged distance is smaller than the required distance (1.0 meter), the `present(target_object)` function will yield `True` result so that the robot can execute a `stop` action to terminate the current trajectory.

Training hyperparameters	
Batch size	10
Optimizer	SGD
Momentum	0.9
Weight decay	0.0005
Learning rate	0.005
Number of epochs	30
Inference hyperparameters	
Region Proposal Network score threshold	0.50
Box score threshold	0.05
Segmentation mask threshold	0.50

Table 12. Training and inference of hyperparameters used in the object detection model. We set other hyperparameters of the Mask R-CNN model to the default values in the torchvision implementation [Paszke et al. 2019].