Program Synthesis Guided Reinforcement Learning for Partially Observed Environments

Yichen David Yang* MIT EECS & CSAIL
Jeevana Priya Inala Microsoft Research
Osbert Bastani University of Pennsylvania
Yewen Pu Autodesk Research
Armando Solar-Lezama MIT EECS & CSAIL
Martin Rinard MIT EECS & CSAIL

Abstract

A key challenge for reinforcement learning is solving long-horizon planning problems. Recent work has leveraged programs to guide reinforcement learning in these settings. However, these approaches impose a high manual burden on the user since they must provide a guiding program for every new task. Partially observed environments further complicate the programming task because the program must implement a strategy that correctly, and ideally optimally, handles every possible configuration of the hidden regions of the environment. We propose a new approach, *model predictive program synthesis (MPPS)*, that uses program synthesis to automatically generate the guiding programs. It trains a generative model to predict the unobserved portions of the world, and then synthesizes a program based on samples from this model in a way that is robust to its uncertainty. In our experiments, we show that our approach significantly outperforms non-program-guided approaches on a set of challenging benchmarks, including a 2D Minecraft-inspired environment where the agent must complete a complex sequence of subtasks to achieve its goal, and achieves a similar performance as using handcrafted programs to guide the agent. Our results demonstrate that our approach can obtain the benefits of program-guided reinforcement learning without requiring the user to provide a new guiding program for every new task.

1 Introduction

Reinforcement learning is a prominent technique for solving challenging planning and control problems [50, 4]. Despite significant recent progress, solving long-horizon problems remains a significant challenge due to the combinatorial explosion of possible strategies. One promising approach to addressing these issues is to leverage *programs* to guide the behavior of the agents [3, 62, 39]. The approaches in this paradigm typically involve three key elements:

- **Domain-specific language (DSL):** For a given domain, the user defines a set of *components* c that correspond to intermediate subgoals that are useful for that domain (e.g., "get wood" or "build bridge"), but leaves out how exactly to achieve these subgoals.
- **Task-specific program:** For every new task in the domain, the user provides a sequence of components (i.e. a program written in the DSL) that, if followed, enable the agent to achieve its goal in the task (e.g., ["get wood"; "build bridge"; "get gem"]).
- **Low-level neural policy:** For a given domain, the reinforcement learning algorithm learns an option [63] that implements each component (i.e., achieves the subgoal specified by that component). Typically a neural policy is learned as each option.

*Correspondence to yicheny@csail.mit.edu

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

Given a new task in a domain, the user provides a program in the DSL that describes a high-level strategy to solve that task. The agent then executes the program by deploying the sequence of learned options that correspond to the components in that program.

A key drawback of this approach is programming overhead: for every new task (a task consists of an instantiation of an environment and a goal), the user must analyze the environment, design a strategy to achieve the goal, and encode the strategy into a program, with a poorly written program producing a suboptimal agent. Furthermore, partially observed environments significantly complicate the programming task because the program must implement a strategy that correctly, and ideally optimally, handles every possible configuration of the hidden regions of the environment.

To address this challenge, we propose a new approach, *model predictive program synthesis (MPPS)*, that automatically synthesizes the guiding programs for program guided reinforcement learning.

MPPS works with a conditional generative model of the environment and a high level specification of the goal of the task to automatically synthesize a program that achieves the goal, with the synthesized program robust to uncertainty in the model. Because the automatically generated agent, and not the user, reasons about how to solve each new task, MPPS significantly reduces user burden. Given a goal specification ϕ, the agent uses the following three steps to choose its actions:

- **Hallucinator:** First, inspired by world-models [29], the agent keeps track of a conditional generative model g over possible realizations of the unobserved portions of the environment.
- **Synthesizer:** Next, the agent synthesizes a program p that achieves ϕ assuming the hallucinator g is accurate. Since world predictions are stochastic in nature, it samples multiple predicted worlds and computes the program that maximizes the probability of success.
- **Executor:** Finally, the agent executes the options corresponding to the components in the program $p = [c_1; ...; c_k]$ for a fixed number of steps N.

If ϕ is not satisfied after N steps, then the above process is repeated. Since the hallucinator now has more information (because the agent has explored more of the environment), the agent now has a better chance of achieving its goal. Importantly, the agent is implicitly encouraged to explore since it must do so to discover whether the current program can successfully achieve the goal ϕ.

We instantiate our approach in the context of a 2D Minecraft-inspired environment [3, 57, 62], which we call "craft," and a "box-world" environment [76]. We demonstrate that our approach significantly outperforms non-program-guided approaches, while achieving a similar performance as using handcrafted programs to guide the agent. In addition, we demonstrate that the policy we learn can be transferred to a continuous variant of the craft environment, where the agent is replaced by a MuJoCo [66] Ant. Thus, our approach can obtain the benefits of program-guided reinforcement learning without requiring the user to provide a new guiding program for every new task.[2]

Related work. In general, program guidance makes reinforcement learning more tractable in at least two ways: (i) it provides intermediate rewards and (ii) it reduces the size of the search space of the policy by decomposing the policy into separate components. Previous research in program guided reinforcement learning demonstrates the benefits of this approach to guide reinforcement learning in the craft environment [62]. This previous research requires the user to provide both a DSL for the domain and a program for every new task. Furthermore, their approach requires that the user includes conditional statements in the program to handle partial observability, which imposes an even greater burden on the user. In contrast, we only require the user to provide a specification encoding the goal for each new task, and automatically handle partial observability.

There has been work enabling users to write specifications in a high-level language based on temporal logic [39], with these specifications then translated into shaped rewards to guide learning. Furthermore, recent work has shown that even if the subgoal encoded by each component is omitted, the program (i.e., a sequence of symbols) can still aid learning [3]. Unlike our approach, this previous work requires the user to provide the guiding programs and does not handle partial observability.

More broadly, our work fits into the literature on combining high-level planning with reinforcement learning. In particular, there is a long literature on planning with options [63] (also known as *skills* [33]), including work on inferring options [61]. Most of these approaches focus on MDPs with discrete state and action spaces and fully observed environments. Recent work [1, 41, 40, 32, 79, 74,

[2]The code is available at: `https://github.com/yycdavid/program-synthesis-guided-RL`

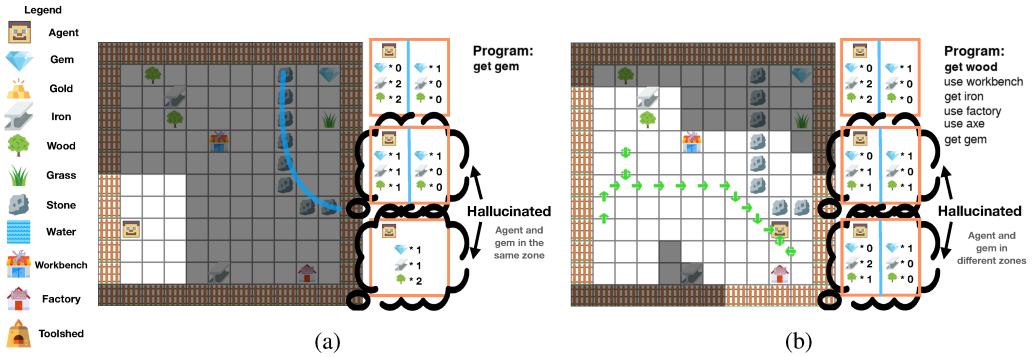


Figure 1: (a) The initial state of an example task for the craft environment. Bright regions are observed and dark ones are unobserved. This particular map has two *zones* separated by a stone boundary (blue line). The first zone contains the agent, 2 irons, and 2 woods; the second contains 1 grass and 1 gem (the goal). The agent represents the high-level structure of the map (e.g., resources in each zone) using state features. The ground truth features are in the top-right; we only show the counts of gems, irons, and woods in each zone and the zone containing the agent. The two thought bubbles below are features hallucinated by the agent based on the observed parts of the map. In both, the zone that the agent is in contains a gem, so the synthesized program is "get gem" (b) The state after the agent took 20 steps (green arrows), failed to obtain the gem, and is now re-synthesizing the program. Having explored more of the map, it predicts that the gem is in a different zone, indicated by its two hallucinations. As a result, it synthesizes a program that includes building and using an axe to break the stone, which leads to successful completion of the task.

77, 6, 64, 49] addresses the challenge of handling continuous state and action spaces by combining high-level planning with reinforcement learning to handle low-level control, but does not handle the challenge of partial observations, whereas our work tackles both challenges.

Classical STRIPS planning [24] cannot handle uncertainty in the realization of the environment. Replanning [60] can be used to handle small changes to an initially known environment, but cannot handle environments that are initially completely unknown. There has been work on hierarchical planning in POMDPs [11, 67], but this research does not incorporate predicate abstractions (i.e., state features) that can be used, for example, to handle continuous state and action spaces. Given multiple possible environments, generalized planning [27, 36, 59, 34] can be used to compute a plan that is valid for all of them. However, in our setting, oftentimes no such plan exists. We instead synthesize a plan that is valid in a maximal number of hallucinated environments. There is also prior work on planning in partially observable environments [9, 18]. Unlike our approach, these approaches assume that the effective state space is small, which enables them to compile the problem into a concrete POMDP which can be efficiently solved using POMDP algorithms. We leverage program synthesis [58] with the world models approach [29] to address these issues; generally speaking, our solver-aided plan synthesis approach is more flexible than existing planning algorithms that target narrower problem settings.

Finally, there has broadly been recent interest in using program synthesis to learn programmatic policies that are more interpretable [71, 72, 38], verifiable [8, 70, 2], and generalizable [37]. In contrast, we are not directly synthesizing the policy, but a program to guide the policy. Appendix C discusses additional related work in a broader context.

2 Motivating Example

Figure 1a shows a 2D Minecraft-inspired crafting game. In this grid world, the agent can navigate and collect resources (e.g., wood), build tools (e.g., a bridge) at workshops using collected resources, and use the tools to traverse obstacles (e.g., use a bridge to cross water). The agent can only observe the 5×5 grid around its current position; since the environment is static, any previously observed cells remain visible. A single task consists of a randomly generated map (i.e., the environment) and goal (i.e., obtain a certain resource or build a certain tool). We consider the meta-learning setting [25]:

we have a set of training tasks for learning the policy, and our goal is to have a policy that works well on new tasks occurring in the future.

DSL. A premise of our approach is a user-provided DSL consisting of components useful for the domain. Figure 2a shows the DSL for the craft environment. For each component, the user also specifies what the component is expected to achieve as a logical predicate. To deal with high-dimensional state spaces, the logical predicates are expressed over features $\alpha(s)$ of the state—e.g., the logical predicate for "get wood" is

$$\forall i, j \,.\, (z^- = i \wedge z^+ = j) \Rightarrow (b^-_{i,j} = \text{connected}) \wedge (\rho^+_{j,\text{wood}} = \rho^-_{j,\text{wood}} - 1) \wedge (\iota^+_{\text{wood}} = \iota^-_{\text{wood}} + 1).$$

This predicate is over two sets of features: (i) features $\alpha(s^-)$, denoted by a $-$, of the initial state s^- (i.e., where execution of the component starts), and (ii) features $\alpha(s^+)$, denoted by a $+$, of the final state s^+ (i.e., where the subgoal is achieved and execution of the component terminates). The first feature is the categorical feature z that indicates the zone containing the agent. In particular, we divide the map into zones that are regions separated by obstacles such as water and stone—e.g., the map in Figure 1a has two zones: (i) the region containing the agent, and (ii) the region blocked off by stones. Now, the feature $b_{i,j}$ indicates whether zones i and j are connected, $\rho_{i,r}$ denotes the count of resource r in zone i, and ι_r denotes the count of resource r in the agent's inventory.

Thus, this formula says that (i) the agent goes from zone i to j, (ii) i and j are connected, (iii) the count of wood in the agent's inventory increases by one, and (iv) the count of wood in zone j decreases by one. Appendix A.1 describes the full set of components we use.

Approach. Before solving any new tasks, for each component c, we use reinforcement learning to train an option $\tilde{c}$ that attempts to achieve the subgoal encoded by c. Given a new task, the user specifies the goal of the task as a logical predicate ϕ. Encoding the goal is typically simple; for example, the goal of the task in Figure 1a is getting gem, which is encoded as $\phi := \iota_{\text{gem}} \geq 1$. Then the agent attempts to solve the task as follows.

First, based on the observations so far, the agent uses a hallucinator g to predict multiple potential worlds, each of which represents a possible realization of the full map. Rather than predicting concrete states, it suffices to predict the state features. For instance, Figure 1a shows two samples of the world predicted by g; here, the only values it predicts are the number of zones in the map, the type of the boundary between the zones, and the counts of the resources and workshops in each zone. In this example, the first predicted world contains two zones, and the second contains one zone. Note that in both predicted worlds, there is a gem located in same zone as the agent.

Next, the agent synthesizes a program p that achieves the goal ϕ in a maximal number of predicted worlds. The synthesized program in Figure 1a is a single component "get gem," which refers to searching the current zone (or zones already connected with the current zone) for a gem. Note that this program achieves the goal for the predicted worlds shown in Figure 1a.

Finally, the agent executes the program $p = [c_1; ...; c_k]$ for a fixed number N of steps. In particular, it executes the policy π_τ of option $\tilde{c}_\tau = (\pi_\tau, \beta_\tau)$ corresponding to c_τ until the termination condition β_τ holds, upon which it switches to executing $\pi_{\tau+1}$. In our example, there is only one component "get gem," so it executes the policy for this component until the agent finds a gem.

In this case, the agent fails to achieve its goal ϕ since there is no gem in its current zone. Thus, it repeats the above process. Since it now has more observations, g more accurately predicts the world—e.g., Figure 1b shows the intermediate step when the agent re-plans. Note that it now correctly predicts that the only gem is in the second zone. Thus, the newly synthesized program is

$$p = [\underbrace{\text{get wood}; \text{use workbench}; \text{get iron}; \text{use factory};}_{\text{for building axe}} \text{ use axe}; \text{get gem}].$$

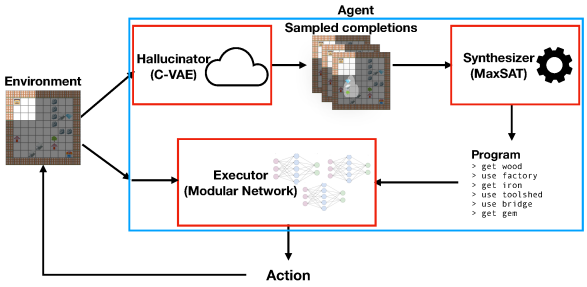
That is, it builds an axe to break the stone so it can get to the zone containing the gem. Finally, the agent executes this new program, which successfully finds the gem.

3 Problem Formulation

POMDP. We consider a partially observed Markov decision process (POMDP) with states $\mathcal{S} \subseteq \mathbb{R}^n$, actions $\mathcal{A} \subseteq \mathbb{R}^m$, observations $\mathcal{O} \subseteq \mathbb{R}^q$, initial state distribution $\mathcal{P}_0$, observation function $h : \mathcal{S} \to \mathcal{O}$,

$$
\begin{array}{rcl}
C & := & \text{get } R \mid \text{use } T \mid \text{use } W \\
R & := & \text{wood} \mid \text{iron} \mid \text{grass} \mid \text{gold} \mid \text{gem} \\
T & := & \text{bridge} \mid \text{axe} \mid \text{ladder} \\
W & := & \text{factory} \mid \text{workbench} \mid \text{toolshed}
\end{array}
$$

(a)



(b)

Figure 2: (a) DSL of components for the craft environment; the three kinds of components are get resource (R), use tool (T), and use workshop (W). (b) Architecture of our agent (the blue box).

and transition function $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Given initial state $s_0 \sim \mathcal{P}_0$, policy $\pi : \mathcal{O} \to \mathcal{A}$, and time horizon $T \in \mathbb{N}$, the generated trajectory is $(s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$, where $o_t = h(s_t)$, $a_t = \pi(o_t)$, and $s_{t+1} = f(s_t, a_t)$. We assume the state includes the unobserved parts of the environment—e.g., in the craft environment, it represents both the entire map and the agent's current position and inventory.

We consider a meta-learning setting, where we have a set of sampled training tasks (world configurations and goal) and a set of test tasks. Our goal is to learn a policy using the training set that achieves good performance on the test set.

User-provided components. We consider programs $p = [c_1; ...; c_k]$ composed of *components* $c_\tau \in C$. We assume the user provides the set of components C that are useful for the domain. Importantly, these components only need to be provided once for a domain; they are shared across all tasks in this domain. Each component is specified as a logical predicate that encodes the intended behavior of that component. More precisely, c is a logical predicate over s^- and s^+, where s^- denotes the initial state before executing c and s^+ denotes the final state after executing c. For instance, the component

$$c \equiv (s^- = s_0 \Rightarrow s^+ = s_1) \wedge (s^- = s_2 \Rightarrow s^+ = s_3)$$

says that if the POMDP is currently in state s_0, then c should transition it to s_1, and if it is currently in state s_2, then c should transition it to s_3. Rather than defining c over the concrete states, we can define it over features $\alpha(s^-)$ and $\alpha(s^+)$ of the states in order to handle high-dimensional state spaces.

User-provided goal specification. The goal of each task is specified with a logical predicate ϕ over the final state; as with components, ϕ may be specified over features $\alpha(s)$ instead of concrete states. Our objective is to design an agent that can achieve any given specification ϕ (i.e., act in the POMDP to reach a state that satisfies ϕ) as quickly as possible.

4 Model Predictive Program Synthesis

We describe here the architecture of our agent, depicted in Figure 2b. It is composed of three parts: the *hallucinator* g, which predicts possible worlds; the *synthesizer*, which generates a program p that maximizes the probability of success according to worlds sampled from g; and the *executor*, which follows p to act in the POMDP. These parts are run once every N steps to generate a program p to execute for the subsequent N steps, until the user-provided specification ϕ is achieved.

Hallucinator. First, the hallucinator is a conditional generative model trained to predict the unobserved parts of the environment given the observations. To be precise, the hallucinator g encodes a distribution $g(s \mid o)$, which is trained to approximate the actual distribution $P(s \mid o)$. Then, at each iteration (i.e., once every N steps), our agent samples m worlds $\hat{s}_1, ..., \hat{s}_m \sim g(\cdot \mid o)$. Our technique can work with any type of conditional generative model as the hallucinator; in our experiments, we use a conditional variational auto-encoder (CVAE) [56].

When using state features, we can have g directly predict the features; this approach works since the synthesizer only needs to know the values of the features to generate a program (see below).

Synthesizer. The synthesizer computes a program that maximizes the probability of satisfying ϕ:

$$p^* = \arg\max_p \mathbb{E}_{P(s|o)} \mathbb{1}[p \text{ solves } \phi \text{ for } s] \approx \arg\max_p \frac{1}{m} \sum_{j=1}^{m} \mathbb{1}[p \text{ solves } \phi \text{ for } \hat{s}_j], \tag{1}$$

where the $\hat{s}_j$ are samples from g. The objective (1) can be expressed as a MaxSAT problem [48]. In particular, suppose for now that we are searching over programs $p = [c_1; ...; c_k]$ of fixed length k. Then, consider the constrained optimization problem

$$\arg\max_{\xi_1,...,\xi_k} \frac{1}{m} \sum_{j=1}^{m} \exists s_1^-, s_1^+, ..., s_k^-, s_k^+ \; . \; \psi_j, \tag{2}$$

where ξ_τ and s_τ^δ (for $\tau \in \{1, ..., k\}$ and $\delta \in \{-, +\}$) are the optimization variables. Here, $\xi_1, ..., \xi_k$ encodes the program $p = [c_1; ...; c_k]$, and ψ_j encodes the constraints that p solves ϕ for world $\hat{s}_j$—i.e.,

$$\psi_j \equiv \psi_{j,\text{start}} \wedge \left[\bigwedge_{\tau=1}^{k} \psi_{j,\tau}\right] \wedge \left[\bigwedge_{\tau=1}^{k-1} \psi'_{j,\tau}\right] \wedge \psi_{j,\text{end}},$$

where (i) $\psi_{j,\text{start}} \equiv (s_1^- = \hat{s}_j)$ encodes that the initial state is $\hat{s}_j$, (ii) $\psi_{j,\tau} \equiv \big((\xi_\tau = c) \Rightarrow c(s_\tau^-, s_\tau^+)\big)$ encodes that if the the τth component of p is $c_\tau = c$, then the transition from s_τ^- to s_τ^+ on step τ satisfies $c(s_\tau^-, s_\tau^+)$, (iii) $\psi'_{j,\tau} \equiv (s_\tau^+ = s_{\tau+1}^-)$ encodes that the final state of the τth step equals the initial state the $(\tau+1)$th step, and (iv) $\psi_{j,\text{end}} \equiv \phi(s_j^+)$ encodes that the final state of the last component should satisfy the user-provided goal ϕ. We use a MaxSAT solver to solve (2) [16]. Given a solution $\xi_1 = c_1, ..., \xi_k = c_k$, the synthesizer returns the corresponding program $p = [c_1; ...; c_k]$.

We incrementally search for longer and longer programs, starting from $k = 1$ and incrementing k until either we find a program that achieves at least a minimum objective value, or we reach a maximum program length k_{max}, at which point we use the best program found so far.

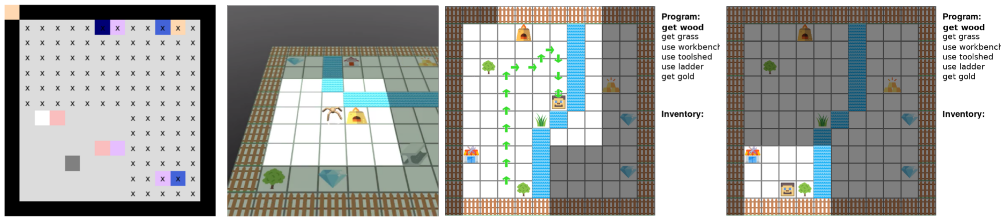
Executor. For each user-provided component $c \in C$, we use reinforcement learning to learn an option $\tilde{c} = (\pi, \beta)$ that executes the component, where $\pi : \mathcal{O} \to \mathcal{A}$ is a policy and $\beta : \mathcal{O} \to \{0, 1\}$ is a termination condition. The executor runs the synthesized program $p = [c_1; ...; c_k]$ by deploying each corresponding option $\tilde{c}_\tau = (\pi_\tau, \beta_\tau)$ in sequence, starting from $\tau = 1$. In particular, it uses action $a_t = \pi_\tau(o_t)$ at each time step t, where o_t is the observation on that step, until $\beta_\tau(o_t) = 1$, at which point it increments $\tau \leftarrow \tau + 1$. It continues until either it has completed running the program ($\beta_k(o_t) = 1$), or after N steps. In the former case, by construction, the goal ϕ has been achieved, so the agent terminates. In the latter case, the agent iteratively reruns the hallucinator and the synthesizer based on the current observation to get a new program. At this point, the hallucinator likely has additional information about the environment, so the new program has a greater chance of success.

5 Learning Algorithm

Next, we describe our algorithm for learning the parameters of models used by our agent. In particular, there are two parts that need to be learned: (i) the parameters of the hallucinator g and (ii) the options $\tilde{c}$ based on the user-provided components c.

Hallucinator. The goal is to train the hallucinator $g(s \mid o)$ to approximate the actual distribution $P(s \mid o)$ of the state s given the observation o. We obtain samples (o_t, s_t) from the training tasks using rollouts from a random agent and train $g_\theta(s \mid o)$ using supervised learning. In our experiments, we take g_θ to be a CVAE and train it using the evidence lower bound (ELBo) on the log likelihood [46].

Executor. Our framework uses reinforcement learning to learn options $\tilde{c}$ that implement the user-provided components c; these options can be shared across multiple tasks. We use neural module networks [3] as the model for the executor policy; but in general our approach can also work with other types of models. In particular, we take $\tilde{c} = (\pi, \beta)$, where $\pi : \mathcal{O} \to \mathcal{A}$ is a neural module and $\beta : \mathcal{O} \to \{0, 1\}$ checks when to terminate execution. First, β is constructed directly from c—i.e., it returns whether c is satisfied based on the current observation o. Next, we train π on the training tasks, which consist of randomly generated initial states s and goal specifications ϕ. Just for training, we use the ground truth program p synthesized based on the fully observed environment; this approach



(a) (b) (c) (d)

Figure 3: (a) The box-world environment. The grey pixel denotes the agent. The goal is to get the white key. The unobserved parts of the map is marked with "x". The key currently held by the agent is shown in the top-left corner. In this map, the number of boxes in the path to the goal is 4, and it contains 1 distractor branch. (b) The ant-craft environment. The policy needs to control the ant to perform the crafting tasks. (c,d) Comparison of behaviors between the optimistic approach (left) and our MPPS approach (right), in a task where the goal is to get gold. (c) The state when the optimistic approach first synthesizes the correct program instead of the (incorrect) one "get gold". It only does so after observing all the squares in its current zone. (d) The initial state of our MPPS strategy. It directly synthesizes the correct program, since the hallucinator knows the gold is most likely in the other zone based on the observations. Thus, the agent completes the task much more quickly.

avoids the need to run the synthesizer repeatedly during training. Given p, we sample a rollout $\{(o_1, a_1, r_1), ..., (o_T, a_T, r_T)\}$ by running the current options $c_\tau = (\pi_\tau, \beta_\tau)$ according to the order specified by p (where π_τ is randomly initialized). We give the agent a reward $\tilde{r}$ at each step when it achieves the subgoal of the component c_τ, as well as a final reward when it achieves the final goal ϕ. Then, we use actor-critic reinforcement learning [47] to update π. Finally, we use curriculum learning to speed up training—i.e., we train using tasks that can be solved with shorter programs first [3].

6 Experiments

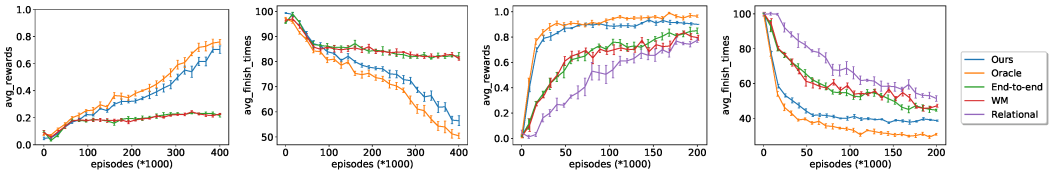
We empirically show that our approach significantly outperforms prior approaches that do not leverage programs, and furthermore achieves similar performance as an oracle given the ground truth program.

6.1 Benchmarks

2D-craft. We consider a 2D Minecraft-inspired game [3] (Figure 1a). A map is a 10×10 grid, where each grid cell is either empty or contains a resource (e.g., wood), obstacle (e.g., water), or workshop. Each task consists of a randomly sampled map, initial position, and goal (one of 10 possibilities, either getting a resource or building a tool), which typically require the agent to achieve several intermediate subgoals. In contrast to prior work, our agent does not initially observe the entire map; instead, they can only observe cells within two units. Since the environment is static, any previously observed cells remain visible. The actions are discrete: moving in one of the four directions, picking up a resource, using a workshop, or using a tool. The maximum episode length is $T = 100$.

Box-world. Next, we consider box-world [76], which requires abstract reasoning. It is a 12×12 grid world with locks and boxes (Figure 3a). The agent is given a key to get started, and its goal is to unlock a white box. Each lock locks a box in the adjacent cell containing a key. Lock and boxes are colored; the key needed to open a lock is in the box of the same color. The actions are to move in one of the four directions; the agent opens a lock and obtains the key simply by walking over it. We assume that the agent can unlock multiple locks with each key. The agent can only observe grid cells within a distance of 3 (as well as the previously observed cells). Each task consists of a randomly sampled map and initial position, where the number of boxes in the path to the goal is randomly chosen between 1 to 4, and the number of "distractor branches" (i.e., boxes that the agent can open but does not help them reach the goal) is also randomly chosen between 1 to 4.

More details about the environments are described in Appendix B.1



(a) (b) (c) (d)

Figure 4: (a,b) Training curves for the 2D-craft environment. (c,d) Training curves for the box-world environment. (a,c) The average reward on the test set over the course of training; the agent gets a reward of 1 if it successfully finishes the task within the time horizon, and 0 otherwise. (b,d) The average number of steps taken to complete the tasks in the test set. We run all the training with 5 different random seeds, and report the mean and standard error of each metric. We show our approach ("Ours"), program guided agent ("Oracle"), end-to-end neural policy ("End-to-end"), world models ("WM"), and relational reinforcement learning ("Relational"). For our approach, we include the episodes used for training the hallucinator in the starting parts of the training curve; since the number of episodes used for hallucinator training is substantially smaller than the number of episodes for executor training, the parts for hallucinator training are hardly noticeable.

Table 1: Average rewards and average completion times on the test set for each approach at the end of training. We report the mean and standard error (in parentheses) over 5 random seeds for training.

	2D-craft		**Box-world**		**Ant-craft**	
	Reward	**Finish step**	**Reward**	**Finish step**	**Reward**	**Finish step**
End-to-end	0.22 (0.01)	82.3 (1.3)	0.85 (0.02)	44.7 (0.6)	0.12 (0.03)	93.1 (2.2)
World models [29]	0.23 (0.01)	81.2 (0.7)	0.80 (0.02)	47.2 (0.9)	0.13 (0.01)	91.3 (1.2)
Relational [76]	-	-	0.77 (0.02)	51.3 (1.6)	-	-
Ours	0.70 (0.03)	56.4 (2.0)	0.90 (0.00)	38.6 (0.4)	0.40 (0.01)	79.2 (1.7)
Oracle	0.76 (0.02)	50.4 (1.1)	0.97 (0.01)	30.8 (0.5)	0.43 (0.02)	77.2 (1.6)

6.2 Baselines

End-to-end. A set of DNN policies that solves the tasks end-to-end. It uses one DNN policy per type of goal, i.e. one network will be used to solve all tasks with the goal of "get gem", another network for tasks with the goal of "build bridge". This baseline is trained using the same actor-critic algorithm and curriculum learning strategy as described in Section 5.

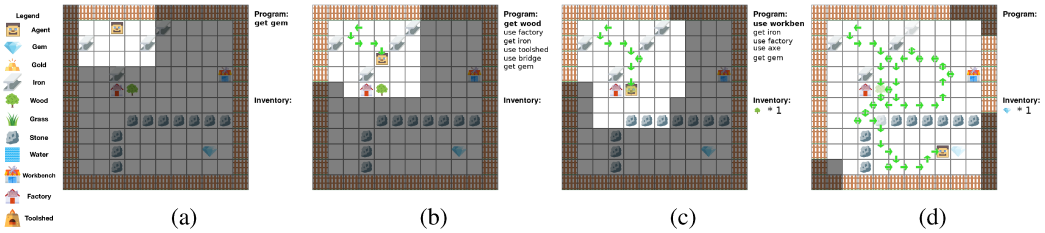
World models [29]. This approach handles partial observability by using a generative model to predict the future. It trains a VAE model that encodes the current observation o_t into a latent vector z_t, and trains a recurrent model to predict z_{t+1} based on $z_1, ..., z_t$. Then, it trains a policy using the latent vectors from the VAE model and the recurrent model as inputs.

Relational reinforcement learning [76]. For box-world, we also compare with this approach, which uses a relational module based on the multi-head attention mechanism [69] for the policy network to facilitate relational reasoning.

Oracle. Finally, we compare to an oracle, which is our approach but given the ground truth program (i.e., guaranteed to achieve ϕ). This can be seen as the program-guided agent approach [62]. This baseline is an oracle since it strictly requires more information as input from the user.

6.3 Implementation Details

2D-craft. For our approach, we use a CVAE hallucinator, with MLP (with 200 hidden units) encoder/decoder, trained on 20K (s, o) pairs collected by a random agent. We use the Z3 [16] solver to solve the MaxSAT problems. We use $m = 3$ hallucinated environments, $N = 20$ steps before replanning in our main experiments, and $N = 5$ in the example behaviors we show for better demonstrations. We use the same actor (resp., critic) network architecture for the policies across all

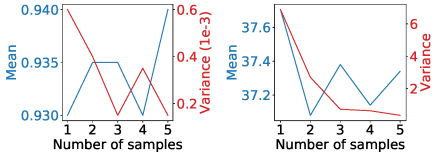


(a) (b) (c) (d)

Figure 5: Example behavior of our policy in a task with the goal of getting gem. (a) The start state. The agent initially hallucinates that there is a gem in the same zone, thus starts with a simple program "get gem". (b) After several steps, the agent observes a wood and a factory. Hallucinating based on these new observations, the agent synthesizes a new program that builds a bridge to cross some water and get gem. This is a reasonable guess since wood, iron and factory are part of the recipe to build a bridge, therefore the presence of them hints that the solution might be via building a bridge. (c) After the agent finishes the "get wood" component, it observes that there are stones in the map, for which bridge cannot be used. Hallucinating based on these new observations, the agent synthesizes a new program that builds an axe to cross the stone. This is a correct program for this task. (d) The final state. The agent executes the program and successfully gets the gem.

Table 2: Comparison to optimistic synthesis and random hallucination strategies on the 2D-craft environment.

	Avg. reward	**Avg. finish step**
Ours	**0.70 (0.03)**	**56.4 (2.0)**
Optimistic	0.42 (0.02)	70.2 (1.2)
Random	0.48 (0.02)	72.6 (0.9)



(e) Reward (f) Finish time

Figure 6: Effect of varying the number of samples m on our approach, evaluated on the box-world over 5 random seeds. Mean and variance of (a) the average reward, (b) the average finishing time on the test tasks.

approaches—i.e., an MLP with 128 (resp., 32) hidden units. We train the policies of each approach on 400K episodes over randomly sampled training tasks, and evaluate on a test set of 50 tasks. [3]

Box-world. Following [76], we use a one-layer CNN with 32 kernels of size 3×3 to preprocess the map across all approaches. For our approach, we have a component for each color where the subgoal is to get the key of that color; see Appendix A.2 for details. For the hallucinator, we use the same architecture as in the craft environment but with 300 hidden units, and trained with 100K (s, o) pairs. For the synthesizer, we use $m = 3$ and $N = 10$. We train the policies for each approach on 200K episodes, and evaluate on a test set containing 40 tasks.

6.4 Results

Table 1 (left two columns) shows the performance of each approach at the end of training. Figure 4 shows the training curves. Our approach significantly outperforms the non-program-guided baselines, both in terms of fraction of tasks solved and in time taken to solve them; it also converges faster, demonstrating that program guidance makes learning significantly more tractable. Our approach also performs comparably to the oracle, delivering comparable performance with significantly less user burden. Figure 5 shows the behavior of our policy in an example task in the 2D-craft environment; see Appendix E for more examples.

Effect of the learned hallucinator. The hallucinator is a key in our approach to handle partial observations. Here we study the benefit of the learned hallucinator to our approach. First, we test a naive strategy for handling partial observations: the agent first randomly explores the map until the current zone is fully observed, then it synthesizes a program and follows it. This strategy only

[3] In our experiments, we train the hallucinator and the executor separately; but in general, one can also interleave the training of the two.

achieves an average reward of $0.024(\pm 0.004)$ in 2D-craft, showing that our benchmarks require effective techniques for handling partial observations. We compare to two ablations without a learned hallucinator: (i) an *optimistic* synthesizer that synthesizes the shortest possible program making best-case assumptions about the unobserved parts of the map, and (ii) a *random* hallucinator that randomly samples completions of the world (See Appendix B.3 for more details). Table 2 shows the results on the 2D-craft environment. As can be seen, our approach significantly outperforms both alternatives. Figure 3c & 3d shows the difference in behavior between our approach and the optimistic strategy; by using a learned hallucinator, our approach is able to leverage the current observations effectively and synthesize a correct program sooner.

Effect of the number of hallucinator samples. We vary the number of hallucinator samples m on box-world. Figure 6 shows the results on the test set over 5 random seeds. As can be seen, varying m does not significantly affect the mean performance, but increasing m significantly reduces variance. Thus, increasing m makes the policy more robust to the uncertainty in the hallucinator. This fact shows the benefit of using multiple samples and MaxSAT synthesis.

Transfer to MuJoCo Ant. To demonstrate that our approach can be adapted to handle continuous control tasks, we consider a variant of 2D-craft where the agent is replaced by a MuJoCo ant [53] (Figure 3b). We consider a simplified setup where we only model the movements of the ant; the ant automatically picks up resources in the grid cell it currently occupies. We focus on transfer learning from 2D-craft. In particular, we pretrain a goal-reaching policy for the ant using soft actor-critic [30]: given a random goal position, this policy moves the ant to that position. The actions output by each approach are translated into a goal position used as input to this goal-reaching policy. We initialize each policy with the corresponding model for 2D-craft and fine-tune it on ant-craft for 40K episodes. Table 1 (rightmost column) shows the results. Our approach significantly outperforms the non-program-guided baselines, both in terms of fraction of tasks solved and time taken to solve them. This demonstrates that our approach is also effective on tasks involving continuous control under a transfer learning setup.

7 Conclusion

We propose an approach that automatically synthesizes programs to guide reinforcement learning for complex long-horizon tasks. Our model predictive program synthesis (MPPS) approach handles partially observed environments by leveraging an approach inspired by world models, where it learns a generative model over the remainder of the world conditioned on the observations, and then synthesizes a guiding program that accounts for the uncertainty in this model. Our experiments demonstrate that MPPS significantly outperforms non-program-guided approaches, while performing comparably to an oracle given a ground truth guiding program. Our results highlight that MPPS can deliver the benefits of program-guided reinforcement learning without requiring the user to provide a guiding program for every new task.

One limitation of our approach is that, as with existing program guided approaches, the user must provide a set of components for each domain. This process only needs to be completed once for each domain since the components can be reused across tasks; nevertheless, automatically inferring these components is an important direction for future work. Finally, we do not foresee any negative societal impacts or ethical concerns for our work (outside of generic risks in improving robotics capabilities).

Acknowledgments and Disclosure of Funding

We gratefully acknowledge support from DARPA HR001120C0015, NSF CCF-1917852, NSF CCF-1910769, and ARO W911NF-20-1-0080. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense, the Army Research Office, or the U.S. Government. We thank the anonymous reviewers for their insightful and helpful comments.

References

[1] David Abel, Nate Umbanhowar, Khimya Khetarpal, Dilip Arumugam, Doina Precup, and Michael Littman. Value preserving state-action abstractions. In *International Conference on Artificial Intelligence and Statistics*, pages 1639–1650. PMLR, 2020.

[2] Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. Neurosymbolic reinforcement learning with formally verified exploration. *arXiv preprint arXiv:2009.12612*, 2020.

[3] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 166–175, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/andreas17a.html.

[4] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

[6] Akhil Bagaria and George Konidaris. Option discovery using deep skill chaining. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=B1gqipNYwH.

[7] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs, 2017.

[8] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Advances in neural information processing systems*, pages 2494–2504, 2018.

[9] Blai Bonet. High-level planning and control with incomplete information using pomdp's. 1998.

[10] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=H1Xw62kRZ.

[11] Laurent Charlin, Pascal Poupart, and Romy Shioda. Automated hierarchy discovery for planning in partially observable environments. *Advances in Neural Information Processing Systems*, 19: 225, 2007.

[12] Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. Web question answering with neurosymbolic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 328–343, 2021.

[13] Xinyun Chen, Chang Liu, and Dawn Song. Towards synthesizing complex programs from input-output examples, 2018.

[14] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1gfOiAqYm.

[15] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using deduction-guided reinforcement learning. In *International Conference on Computer Aided Verification*, pages 587–610. Springer, 2020.

[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992.

[17] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 990–998. JMLR.org, 2017.

[18] Denise Draper, S. Hanks, and Daniel S. Weld. Probabilistic planning with information gathering and contingent execution. In *AIPS*, 1994.

[19] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Unsupervised learning by program synthesis. In *Advances in neural information processing systems*, pages 973–981, 2015.

[20] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *Advances in neural information processing systems*, pages 6059–6068, 2018.

[21] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems*, pages 9169–9178, 2019.

[22] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 835–850, 2021.

[23] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices*, 53(4):420–435, 2018.

[24] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[25] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135. PMLR, 2017.

[26] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated task and motion planning. *CoRR*, abs/2010.01083, 2020. URL `https://arxiv.org/abs/2010.01083`.

[27] Edward Groshev, Maxwell Goldstein, Aviv Tamar, Siddharth Srivastava, and Pieter Abbeel. Learning generalized reactive policies using deep neural networks, 2018.

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

[29] David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018. URL `http://arxiv.org/abs/1803.10122`.

[30] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL `http://proceedings.mlr.press/v80/haarnoja18b.html`.

[31] Shivam Handa and Martin Rinard. Inductive program synthesis over noisy data. *CoRR*, abs/2009.10272, 2020. URL `https://arxiv.org/abs/2009.10272`.

[32] Mohammadhosein Hasanbeig, Natasha Yogananda Jeppu, Alessandro Abate, Tom Melham, and Daniel Kroening. Deepsynth: Program synthesis for automatic task segmentation in deep reinforcement learning. *CoRR*, abs/1911.10244, 2019. URL `http://arxiv.org/abs/1911.10244`.

[33] Karol Hausman, Jost Tobias Springenberg, Ziyu Wang, Nicolas Heess, and Martin Riedmiller. Learning an embedding space for transferable robot skills. In *International Conference on Learning Representations*, 2018.

[34] Yuxiao Hu and Giuseppe De Giacomo. Generalized planning: Synthesizing plans that work for multiple environments. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, page 918–923. AAAI Press, 2011. ISBN 9781577355144.

[35] Jiani Huang, Calvin Smith, Osbert Bastani, Rishabh Singh, Aws Albarghouthi, and Mayur Naik. Generating programmatic referring expressions via program synthesis. In *International Conference on Machine Learning*, pages 4495–4506. PMLR, 2020.

[36] León Illanes and Sheila A. McIlraith. Generalized planning via abstraction: Arbitrary numbers of objects. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):7610–7618, Jul. 2019. doi: 10.1609/aaai.v33i01.33017610. URL `https://ojs.aaai.org/index.php/AAAI/article/view/4754`.

[37] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020.

[38] Jeevana Priya Inala, Yichen Yang, James Paulos, Yewen Pu, Osbert Bastani, Vijay Kumar, Martin Rinard, and Armando Solar-Lezama. Neurosymbolic transformers for multi-agent communication. In *NeurIPS*, 2020.

[39] Kishor Jothimurugan, Rajeev Alur, and Osbert Bastani. A composable specification language for reinforcement learning tasks. In *NeurIPS*, 2019.

[40] Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. Compositional reinforcement learning from logical specifications. *arXiv preprint arXiv:2106.13906*, 2021.

[41] Kishor Jothimurugan, Osbert Bastani, and Rajeev Alur. Abstract value iteration for hierarchical reinforcement learning. In *AISTATS*, 2021.

[42] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical task and motion planning in the now. In *2011 IEEE International Conference on Robotics and Automation*, pages 1470–1477, 2011. doi: 10.1109/ICRA.2011.5980391.

[43] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Integrated task and motion planning in belief space. *The International Journal of Robotics Research*, 32(9-10):1194–1227, 2013. doi: 10.1177/0278364913484072.

[44] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=rywDjg-RW`.

[45] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[46] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[47] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12, pages 1008–1014. MIT Press, 2000. URL `https://proceedings.neurips.cc/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf`.

[48] M W Krentel. The complexity of optimization problems. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, page 69–76, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 0897911938. doi: 10.1145/12130.12138. URL `https://doi.org/10.1145/12130.12138`.

[49] Alexander C. Li, Carlos Florensa, Ignasi Clavera, and Pieter Abbeel. Sub-policy adaptation for hierarchical reinforcement learning, 2020.

[50] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[51] Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B Tenenbaum, and Armando Solar-Lezama. Representing partial programs with blended abstract semantics. *arXiv preprint arXiv:2012.12964*, 2020.

[52] Camille Phiquepal and Marc Toussaint. Combined task and motion planning under partial observability: An optimization-based approach. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 9000–9006, 2019. doi: 10.1109/ICRA.2019.8793260.

[53] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[54] Ameesh Shah, Eric Zhan, Jennifer J. Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. Learning differentiable programs with admissible neural heuristics. *CoRR*, abs/2007.12101, 2020. URL `https://arxiv.org/abs/2007.12101`.

[55] David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring lisp programs from examples. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'75, page 260–267, San Francisco, CA, USA, 1975. Morgan Kaufmann Publishers Inc.

[56] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28, pages 3483–3491. Curran Associates, Inc., 2015. URL `https://proceedings.neurips.cc/paper/2015/file/8d55a249e6baa5c06772297520da2051-Paper.pdf`.

[57] Sungryull Sohn, Junhyuk Oh, and Honglak Lee. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 7156–7166, Red Hook, NY, USA, 2018. Curran Associates Inc.

[58] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, USA, 2008.

[59] Siddharth Srivastava. Foundations and applications of generalized planning. *AI Commun.*, 24 (4):349–351, December 2011. ISSN 0921-7126.

[60] Anthony Stentz et al. The focussed dˆ* algorithm for real-time replanning. In *IJCAI*, volume 95, pages 1652–1659, 1995.

[61] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.

[62] Shao-Hua Sun, Te-Lin Wu, and Joseph J. Lim. Program guided agent. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=BkxUvnEYDH`.

[63] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2): 181–211, 1999.

[64] Chen Tessler, Shahar Givony, Tom Zahavy, Daniel J. Mankowitz, and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft, 2016.

[65] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T Freeman, Joshua B Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. *arXiv preprint arXiv:1901.02875*, 2019.

[66] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

[67] Marc Toussaint, Laurent Charlin, and Pascal Poupart. Hierarchical pomdp controller optimization by likelihood maximization. In *UAI*, volume 24, pages 562–570, 2008.

[68] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. Houdini: Lifelong learning as program synthesis. In *Advances in Neural Information Processing Systems*, pages 8687–8698, 2018.

[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[70] Abhinav Verma. Verifiable and interpretable reinforcement learning through program synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9902–9903, 2019.

[71] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5045–5054. PMLR, 2018.

[72] Abhinav Verma, Hoang M Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *NeurIPS*, 2019.

[73] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[74] Markus Wulfmeier, Dushyant Rao, Roland Hafner, Thomas Lampe, Abbas Abdolmaleki, Tim Hertweck, Michael Neunert, Dhruva Tirumala, Noah Siegel, Nicolas Heess, and Martin Riedmiller. Data-efficient hindsight off-policy option learning, 2021.

[75] Halley Young, Osbert Bastani, and Mayur Naik. Learning neurosymbolic generative models via program synthesis. In *ICML*, 2019.

[76] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, Murray Shanahan, Victoria Langston, Razvan Pascanu, Matthew Botvinick, Oriol Vinyals, and Peter Battaglia. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=HkxaFoC9KQ`.

[77] Jesse Zhang, Haonan Yu, and Wei Xu. Hierarchical reinforcement learning by discovering intrinsic options, 2021.

[78] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William E. Byrd, Matthew Might, Raquel Urtasun, and Richard S. Zemel. Neural guided constraint logic programming for program synthesis. *CoRR*, abs/1809.02840, 2018. URL `http://arxiv.org/abs/1809.02840`.

[79] Shangtong Zhang and Shimon Whiteson. Dac: The double actor-critic architecture for learning options, 2019.

A Components for Environments

A.1 Components for the Craft Environment

In this section, we describe the components (i.e., logical formulae encoding pre/post-conditions for each option) that we use for the craft environment. First, recall that the domain-specific language that encodes the set of components for the craft environment is

$$\begin{array}{rcl} C & := & \text{get } R \mid \text{use } T \mid \text{use } W \\ R & := & \text{wood} \mid \text{iron} \mid \text{grass} \mid \text{gold} \mid \text{gem} \\ T & := & \text{bridge} \mid \text{axe} \mid \text{ladder} \\ W & := & \text{factory} \mid \text{workbench} \mid \text{toolshed} \end{array}$$

Also, the set of possible artifacts (objects that can be made in some workshop using resources or other artifacts) in the craft environment is

$$A = \{\ \text{bridge},\ \text{axe},\ \text{plank},\ \text{stick},\ \text{ladder}\ \}.$$

We define the following features:

- **Zone:** $z = i$ indicates the agent is in zone i
- **Boundary:** $b_{i,j} = b$ indicates how zones i and j are connected, where

$$b \in \{\text{connected}, \text{water}, \text{stone}, \text{not adjacent}\}$$

- **Resource:** $\rho_{i,r} = n$ indicates that there are n units of resource r in zone i
- **Workshop:** $\omega_{i,r} = b$, where $b \in \{\text{true}, \text{false}\}$, indicates whether there exists a workshop r in zone i
- **Inventory:** $\iota_r = n$ indicates that there are n objects r (either a resource or an artifact) in the agent's inventory

We use $z^-, b^-, \rho^-, \omega^-, \iota^-$ and $z^+, b^+, \rho^+, \omega^+, \iota^+$ to denote the initial state and the final state for a component, respectively. Now, the logical formulae for each component are defined as follows.

(1) "get r" (for any resource $r \in R$). First, we have the following component telling the agent to obtain a specific resource r:

$$\begin{aligned} \forall i, j\ .\ (z^- = i \wedge z^+ = j) \Rightarrow (b^-_{i,j} = \text{connected}) \\ \wedge\ (\rho^+_{j,r} = \rho^-_{j,r} - 1) \wedge (\iota^+_r = \iota^-_r + 1) \wedge \mathcal{Q}. \end{aligned}$$

Here, $\mathcal{Q}$ refers to the conditions that the other fields of the abstract state stay the same—i.e.,

$$\begin{aligned} (b^+ = b^-) \wedge (\omega^+ = \omega^-) \wedge (\iota^+_{\setminus r} = \iota^-_{\setminus r}) \\ \wedge\ (\rho^+_{\setminus (j,r)} = \rho^-_{\setminus (j,r)}), \end{aligned}$$

where $\iota_{\setminus r}$ means all the other fields in ι except ι_r, and similarly for $\rho_{\setminus (j,r)}$. In particular $\mathcal{Q}$ addresses the *frame problem* from classical planning.

(2) "use r" (for any workshop $r \in W$). Next, we have a component telling the agent to use a workshop to create an artifact. To do so, we introduce a set of auxiliary features to denote the number of artifacts made in this component: $m_o = n$ indicates that n units of artifact o is made. The set of artifacts that can be made at workshop r is denoted as A_r, and the number of units of ingredient q needed to make 1 unit of artifact o is denoted as $k_{o,q}$, where $q \in R \cup A$; note that $\{A_r\}$ and $\{k_{o,q}\}$ come from the rule of the game.

Then, the logical formula for "use r" is

$$\begin{aligned}
&\forall i, j \,.\, (z^- = i \wedge z^+ = j) \Rightarrow (b^-_{i,j} = \text{connected}) \\
&\quad \wedge (w_{j,r} = \text{true}) \wedge \left(\sum_{o \in A_r} m_o \geq 1 \right) \wedge \left(\sum_{o \notin A_r} m_o = 0 \right) \\
&\quad \wedge \left(\forall q \in R,\ \iota^+_q = \iota^-_q - \sum_{o \in A_r} k_{o,q} m_o \right) \\
&\quad \wedge \left(\forall q \in A,\ \iota^+_q = \iota^-_q - \sum_{o \in A_r} k_{o,q} m_o + m_q \right) \\
&\quad \wedge \left(\forall o \in A_r,\ \neg \left(\bigwedge_q \iota^+_q \geq k_{o,q} \right) \right) \\
&\quad \wedge \mathcal{Q},
\end{aligned}$$

where

$$\mathcal{Q} = (b^+ = b^-) \wedge (\omega^+ = \omega^-) \wedge (\rho^+ = \rho^-).$$

This formula reflects the game setting that when the agent uses a workshop, it will make artifacts until the ingredients in the inventory are depleted.

(3) "use r" ($r =$ bridge/axe/ladder). Next, we have the following component for telling the agent to use a tool. The formula for this component encodes the logic of zone connectivity. In particular, it is

$$\begin{aligned}
&\forall i, j \,.\, (z^- = i \wedge z^+ = j) \Rightarrow (b^-_{i,j} = \text{water/stone}) \\
&\quad \wedge (b^+_{i,j} = \text{connected}) \wedge (\iota^+_r = \iota^-_r - 1) \\
&\quad \wedge \Big(\forall i', j',\ (b^+_{i',j'} = \text{connected}) \Rightarrow \\
&\qquad\qquad \big((b^-_{i',j'} = \text{connected}) \vee \mathcal{X} \big) \Big) \\
&\quad \wedge \Big(\forall i', j',\ (b^+_{i',j'} \neq \text{connected}) \Rightarrow (b^+_{i',j'} = b^-_{i',j'}) \Big) \\
&\quad \wedge \mathcal{Q},
\end{aligned}$$

where

$$\begin{aligned}
\mathcal{X} &= (b^-_{i',i} = \text{connected} \vee b^-_{i',j} = \text{connected}) \\
&\quad \wedge (b^-_{j',i} = \text{connected} \vee b^-_{j',j} = \text{connected}) \\
\mathcal{Q} &= (\omega^+ = \omega^-) \wedge (\rho^+ = \rho^-) \wedge (\iota^+_{\setminus r} = \iota^-_{\setminus r}).
\end{aligned}$$

A.2 Components for Box World

In this section, we describe the components for the box world. They are all of the form "get k", where $k \in K$ is a color in the set of possible colors in the box world. First, we define the following features:

- **Box**: $b_{k_1,k_2} = n$ indicates that there are n boxes with key color k_1 and lock color k_2 in the map
- **Loose key**: $\ell_k = b$, where $b \in \{\text{true}, \text{false}\}$, indicates whether there exists a loose key of color k in the map
- **Agent's key**: $\iota_k = b$, where $b \in \{\text{true}, \text{false}\}$, indicates whether the agent holds a key of color k

As in the craft environment, we use b^-, ℓ^-, ι^- and b^+, ℓ^+, ι^+ to denote the initial state and the final state for a component, respectively. Since the configurations of the map in the box world can only contain at most one loose key, we add a cardinality constraint $\text{Card}(\ell) \leq 1$, where $\text{Card}(\cdot)$ counts the number of features that are true.

Then, the logical formula defining the component "get k" is

$$\mathcal{X} \vee \mathcal{Y},$$

where

$$\begin{aligned}
\mathcal{X} &= \ell_k^- \wedge \iota_k^+ \wedge (\text{Card}(l^+) = 0) \wedge (b^+ = b^-) \\
\mathcal{Y} &= (\text{Card}(\iota^-) = 1) \wedge \iota_k^+ \wedge \neg \iota_k^- \wedge (l^+ = l^-) \wedge \\
&\qquad \Big(\forall k_1 \,.\, \iota_{k_1}^- \Rightarrow \Big((b_{k,k_1}^+ = b_{k,k_1}^- - 1) \wedge (b_{\setminus(k,k_1)}^+ = b_{\setminus(k,k_1)}^-)\Big)\Big)
\end{aligned}$$

In particular, $\mathcal{X}$ encodes the desired behavior when the agent picks up a loose key k, and $\mathcal{Y}$ encodes the desired behavior when the agent unlocks a box to get key k.

B Experimental Details

B.1 Benchmarks

2D-craft. In this domain, a map is a 10×10 grid, where each grid cell is either empty or contains a resource (e.g., wood), obstacle (e.g., water), or workshop. The agent can only observe cells within the distance of 2 units. Since the environment is static, any previously observed cells remain visible. We follow the same approach as in prior work [3] to encode and preprocess the observations: each grid cell is first encoded using a one-hot encoding representing its content (with an entry for unobserved cells); then the preprocessing step extracts the 5×5 grid around the current position of the agent as the fine-scale features, and also an aggregated 5×5 grid of coarse-scale features which is aggregated over a 25×25 region from the original map (after padding) via max pooling. The flattened version of these features are the inputs to the policy networks in our approach and the baselines. More details can be found in [3] and its code repository. The test set we use contains tasks with 10 types of goals: get wood, get iron, get grass, get gold, get gem, build plank, build stick, build bridge, build axe, and build ladder. To make the test set more challenging, we include more (15 tasks) from the two hardest goals: get gold and get gem. These goals involve potentially longer horizons to achieve. The rest of the goals are in equal proportion. All our results are averaged over the test set (averaged across different types of goals). This setup follows prior work [3, 62].

For the MLP model architectures, we follow the prior work that originally introduced 2D-craft [3]; in particular, we adopt their model architecture for the actor and critic networks in both our approach and the baselines. We train our hallucinator to operate on state features (e.g. the counts of gems); it takes the state features of the observation as input and predicts the state features of the full map.

Box-world. In this domain, a map is a 12×12 grid with locks and boxes. The agent can only observe cells within the distance of 3 units. As in 2D-craft, since the environment is static, any previously observed cells remain visible. For encoding the observations, each grid cell is encoded using a one-hot encoding representing its content (with an entry for unobserved cells). Following [76], we use a one-layer CNN with 32 kernels of size 3×3 to preprocess the map across all approaches before feeding into the policy networks. The test set contains 40 tasks with the number of boxes in the path to the goal varying between 1 to 4; these difficulty levels are in equal proportion.

Ant-craft. This domain is the same as 2D-craft, except that the agent is replaced with a MuJoCo ant [53], a simulated four-legged robot. We consider a simplified setup where we only model the movements of the ant; the ant directly picks up resources, use tools, and use workshops when it is at the appropriate grid cell (e.g., we do not model the mechanics of grabbing).

B.2 Training

We train our models on an NVIDIA GeForce GTX 1080 Ti GPU. The actor-critic training of our approach takes around a day on 2D-craft (400K episodes), 12 hours on box-world (200K episodes), and a day for fine-tuning ant-craft (40K episodes). We use the Adam optimizer [45] with a learning rate of 0.002. We use a batch size of 10 episodes.

B.3 Ablations

Here, we provide more detail on the two ablations without a learned hallucinator.

Optimistic synthesizer. The optimistic synthesizer considers the unobserved parts of the world to be in any possible configuration. If a program can achieve the goal under any one of these configurations, this program is considered to be correct. The optimistic synthesizer chooses the shortest program considered to be correct in this optimistic sense. For example, if the goal of the task is "get gem", and there is some unobserved grid cells in the current zone, then an optimistic synthesizer will always synthesize the simplest program "get gem". This baseline also demonstrates the importance of using a hallucinator, instead of a heuristic such as pure optimism.

Random hallucinator. The random hallucinator randomly predicts the configuration of the unobserved parts of the world. In our experiments, the hallucinator directly predicts the abstract state features, so the random hallucinator simply predicts random values for each entry of the state features (e.g., number of wood in zone 1) under the condition that it does not conflict with existing observations (e.g., predicting number of wood in zone 1 to be 1 when there are already 2 woods observed in zone 1). The purpose of this ablation is to demonstrate the importance of using a learned hallucinator.

C Additional Related Work

Program synthesis. There has been a long line of work on program synthesis, which targets the problem of how to automatically synthesize a program that satisfies a given specification [55, 58, 28, 73, 31]. More broadly, recent work has explored learning neural network models to predict the program [17, 10, 14, 13, 5], as well as using neural models to guide synthesis [44, 54, 78, 7, 23, 21, 15, 51, 22]. There has also been work leveraging program synthesis to improve performance in image and natural language domains [19, 20, 68, 75, 65, 35, 12]. In contrast, our work uses program synthesis to guide reinforcement learning.

Task and motion planning (TAMP). TAMP is a hierarchical planning approach that uses high-level task planning and low-level motion planning [42, 26]. TAMP by itself does not handle partial observability; recent work has proposed extensions to address this challenge. For instance, [52] learns a full symbolic program to handle all possible cases—this program tends to be very complex (with many branches) and hence hard to learn. In contrast, our approach learns a simple straight line program that is most likely to solve the task and then replans if needed. Furthermore, [52] only handles discrete partial observations, whereas our approach does not have this restriction. Next, [43] performs planning in the belief space, which is more similar to our strategy. However, they make the significantly stronger assumption that a structured representation of belief space is available; in particular, they assume a probability distribution over the abstract state space is provided. In general, such a distribution can be difficult to obtain—most deep generative models are unable to explicitly provide the distribution over abstract states; instead, they provide either samples (e.g., GANs and VAEs) or probabilities of given states (e.g., normalizing flows; VAEs can provide a lower bound). As a consequence, it would be difficult to apply this approach to our environments.

D Additional Analysis

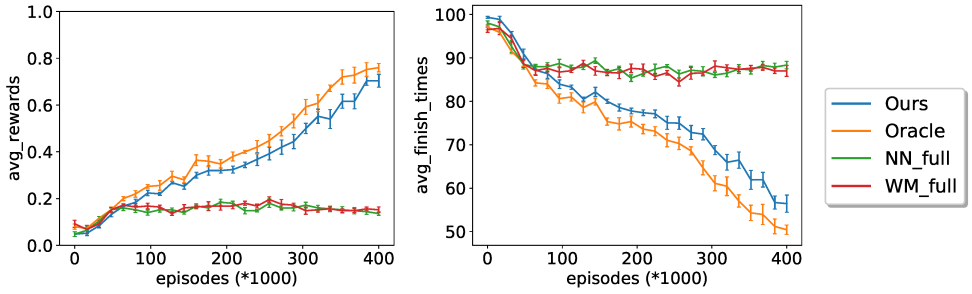
D.1 Stand-alone evaluations

Hallucinator. We perform additional experiments that measure the prediction accuracy of our trained hallucinator for 2D-craft. We measure accuracy in two ways. The first is the percentage of cases where the predicted state features match the ground truth state features in every entry of the state feature (e.g. the number of zones is an entry, the number of wood in zone 1 is an entry). We call this the "whole" accuracy. The second is the percentage of entries that are correctly predicted, treating each entry of the state feature separately. We call this the "individual" accuracy. We measure accuracy on the test set at different number of steps into the episode. The results are shown in Table 3. As can be seen, the learned hallucinator can correctly predict many entries of the state features, but rarely predicts the whole state features perfectly. This result is due to the intrinsic randomness in the distribution $P(s \mid o)$. Note that accuracy increases with the number of steps into the episodes since the agent has explored more of the map later in the episodes.

Executor. We measure the success rate of the learned executor in our approach at achieving a given component. We evaluate on the test set of 2D-craft environment, focusing on components from the oracle programs. The success rate is 93.8% (so the failure rate is 6.2%). The most common failure cases are that the agent gets stuck in some local region of the map. Note that since the program for

Table 3: Standalone accuracy of the hallucinator

Step	Whole acc.	Individual acc.
0	0.0%	70.9%
20	4.5%	82.9%
40	4.8%	85.5%



(a) (b)

Figure 7: Training curves for the 2D-craft environment, comparing our approach with baselines trained on fully observed environments. (a) The average reward on the test set over the course of training. (b) The average number of steps taken to complete the tasks on the test set. We run all the training with 5 different random seeds, and report the mean and standard error of each metric. We show our approach ("Ours"), program guided agent ("Oracle"), end-to-end neural policy trained on fully observed maps ("NN-full"), and world models trained on fully observed maps ("WM-full").

each task typically includes more than one component, this 6.2% failure rate will result in >6.2% failure rate in completing the tasks.

D.2 Baselines trained with fully observed maps

In our experiments, we use the programs synthesized from the fully observed maps for training the executor in our approach. This approach avoids repeatedly running the MaxSAT synthesizer during training, which helps speed up training. To ensure this additional information is not responsible for the performance of our approach compared with the non-program-guided baselines, we perform an additional experiment that trains the baselines in fully observed environments. Figure 7 shows results for the 2D-craft environment. As can be seen, our approach continues to significantly outperform the non-program-guided baselines. These results show that providing fully observed map information during training is not the reason our approach outperforms the baselines.

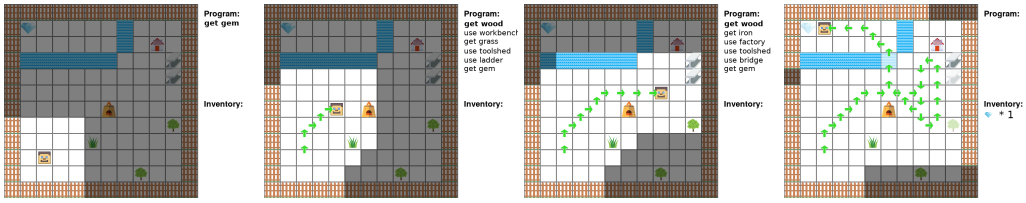
D.3 Non deterministic environment

We perform an additional experiment to study how our approach works when the environment is non-deterministic. We create a non-deterministic version of 2D-craft, where each action has 20% chance of failing (when a move action fails, the agent move to a random direction; when a use action fails, the action becomes a no-op). Table 4 shows the results. As can be seen, all the approaches take a longer time to solve tasks in these non-deterministic environments, but our approach continues to significantly outperform the non-program-guided baselines and perform comparably to the oracle. For the non-program guided baselines, the ratio of test tasks successfully solved does not change significantly, likely because they fail to solve the challenging tasks even when the environment is deterministic.

E Additional Examples

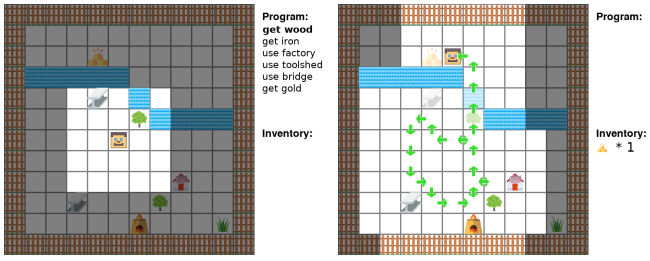
Table 4: Performance on the test set for the non-deterministic version of 2D-craft

	Avg. reward	**Avg. finish step**
End-to-end	0.22 (0.02)	83.3 (1.8)
World models	0.20 (0.01)	83.6 (0.7)
Ours	0.47 (0.03)	73.1 (1.2)
Oracle	0.50 (0.03)	69.9 (1.6)



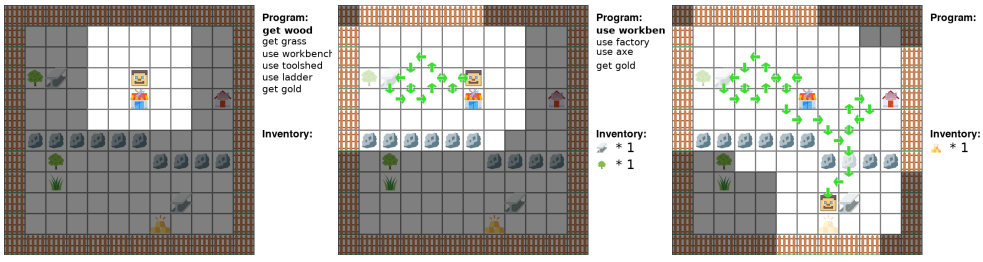
(a) (b) (c) (d)

Figure 8: Example behavior of our policy in a task with the goal of getting gem. (a) The start state. The agent initially hallucinates that there is a gem in the same zone, thus starts with a simple program "get gem". (b) After several steps, the agent observes a grass and a toolshed. Hallucinating based on these new observations, the agent synthesizes a new program that builds a ladder to get gem (which requires grass and toolshed). (c) After several more steps, the agent observes some water and iron. It re-synthesizes a new program that builds a bridge to cross water. This is a correct program for this task. (d) The final state. The agent executes the program and successfully get the gem.



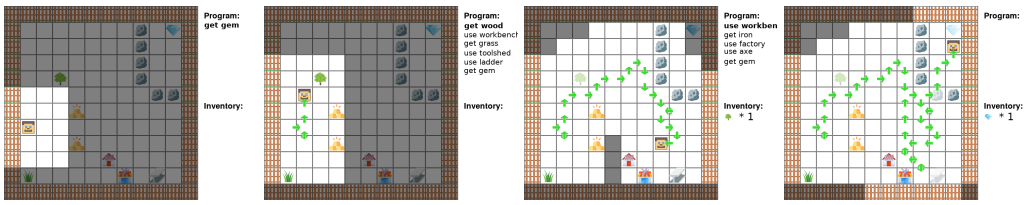
(a) (b)

Figure 9: Example behavior of our policy in a task with the goal of getting gold. (a) The start state. By hallucinating based on the current observations, the agent correctly synthesizes a program that builds and uses a bridge to get to the other zone and get gold. (b) The final state.



(a) (b) (c)

Figure 10: Example behavior of our policy in a task with the goal of getting gold. (a) The start state. Based on its hallucinations, the agent synthesizes a program that builds and uses a ladder to get a gold in the other zone. However, there is not enough resources and facilities to make a ladder in this map. (b) The intermediate state when the agent re-synthesizes a new program. With more observations, the agent changes the program to building and using an axe instead, which is a feasible solution in this map. (c) The final state.



(a) (b) (c) (d)

Figure 11: Example behavior of our policy in a task with the goal of getting gem. (a) The start state. The agent starts with a simple program "get gem". (b) After several steps, the agent observes a grass and a wood. Hallucinating based on these new observations, the agent synthesizes a new program that builds a ladder to get gem (which requires grass and wood). (c) During its search for workbench, the agent observes all the resources for building an axe. Therefore, it re-synthesizes a new program that builds a axe to cross the stone boundary. This is a correct program for this task. (d) The final state. The agent executes the program and successfully get the gem.