REASONING ABOUT ACTION PRECONDITIONS WITH PROGRAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

One of the fundamental skills required for an agent acting in an environment to complete tasks is the ability to understand what actions are plausible at any given point. This work explores a novel use of code representations to reason about action preconditions for sequential decision making tasks. Code representations offer the flexibility to model procedural activities and associated constraints as well as the ability to execute and verify constraint satisfaction. Leveraging code representations, we decompose the problem of learning an agent policy for sequential decision making tasks into the sub-problems of *precondition inference* and *action prediction*. We show that these sub-problems can be formulated as code-completion problems and exploit pre-trained code understanding models to tackle them. We demonstrate that the proposed code representation coupled with our novel precondition-aware action prediction strategy outperforms prior policy learning approaches in a few-shot learning setting across task-oriented dialog and embodied textworld benchmarks.

1 INTRODUCTION

Sequential decision making agents are tasked with choosing an optimal action given a sequence of observations. A key capability for learning an optimal agent policy is understanding the plausibility of different actions. For instance, a dialog agent recommending restaurants needs to have basic information like location and type of food in order to look up its database for potential options. Understanding the necessary conditions for performing an action (e.g., location and type of food are necessary for the database lookup action) is referred to as *precondition inference* or *affordance learning* in the literature (Ahn et al., 2022; Sohn et al., 2020).

Policy learning approaches that treat the policy as a black box regression model that maps input observations to optimal actions offer limited transparency to the decision making process of the agent. For instance, few-shot prompted large language models have demonstrated strong capabilities for policy learning (Logeswaran et al., 2022; Huang et al., 2022a; Micheli & Fleuret, 2021; Ahn et al., 2022). Although some approaches have been proposed to improve the interpretability of these models by producing on-the-fly rationales for their predictions (e.g., in the form of chain-of-thought) (Huang et al., 2022b; Yao et al., 2022), it is non-trivial to verify whether these rationales are adequate as they are dynamically generated during action prediction. As a result, it is difficult to guarantee with certainty that model predictions are consistent with the rationales.

In this work we propose to use code as a representation to reason about preconditions. Code representations offers many advantages. Programs are a natural formalism to model event sequences and offer the flexibility to express dependency constraints between events, such as in the form of assertions (Liang et al., 2022; Singh et al., 2022). Verifying that a program meets the specifications dictated by the assertions amounts to simply executing the program and verifying that the program ran successfully. The ability to execute and verify constraint satisfaction is a key benefit of code representations compared to alternative representations such as natural language. Representing preconditions in the form of procedural statements in code further provides transparency, controllability and better generalization to unseen scenarios compared to alternative representations of affordance such as neural network functions. Finally, this also enables us to exploit strong priors captured by code understanding models for policy learning problems.

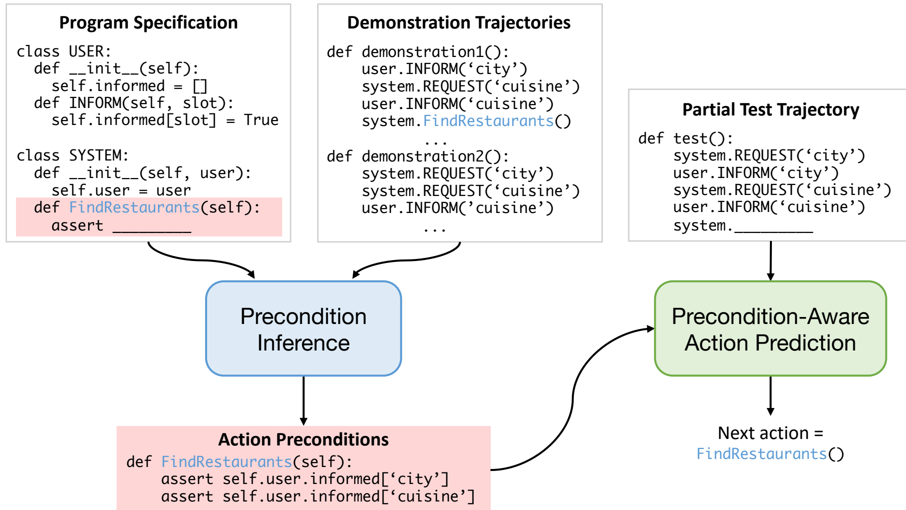

Figure 1: **Approach Overview**: We propose to use programs as a representation of the agent's trajectory to construct a policy. We decompose the policy learning problem into precondition inference and action prediction sub-problems and tackle them by formulating them as code-completion problems and leveraging pre-trained code models. We first infer action preconditions from demonstration programs. We then use inferred preconditions to predict the next action given a partial test program.

Armed with the code representation, we decompose the policy learning problem into precondition inference and action prediction sub-problems and formulate these sub-problems as code completion problems. We then leverage pre-trained code models to address these sub-problems. We propose a pipeline to generate precondition candidates from expert demonstrations. We also propose an action prediction mechanism that ensures predicted actions are consistent with the inferred preconditions. We present extensive analysis and ablations that show the impact of different components of our approach.

In summary, we make the following contributions in this work.

- We propose to use programs as a representation to reason about action preconditions.
- We show that pre-trained code generation models are capable of inferring action preconditions from expert demonstrations alone in a zero-shot manner.
- Combining action preconditions with a novel precondition-aware action prediction strategy, we demonstrate that the proposed framework leads to better agent policies compared to baselines on task-oriented dialog and embodied textworld benchmarks.

2 Problem Setting

We consider a sequential decision making setting where the agent receives a sequence of observations $o_i \in \mathcal{O}$ and performs an action $a_i \in \mathcal{A}$, where we assume discrete observation and action spaces $\mathcal{O}, \mathcal{A}$. The agent's trajectory can be represented as a sequence of observations and actions $\tau = (o_1, a_1, o_2, a_2, \ldots, o_n, a_n)$. We consider a few-shot learning setting where we are given demonstrations $\mathcal{D} = \{\tau^1, \ldots, \tau^n\}$ and we want to estimate a policy $\pi(a_t|\tau_{<t}, \mathcal{D})$ that predicts the next action given the history of observations and actions $\tau_{<t} = (o_{1:t}, a_{1:t-1})$. The goal of learning is to generalize to a set of test trajectories $\mathcal{D}^{\text{test}}$: maximize $\mathbb{E}_{\tau \in \mathcal{D}^{\text{test}}}[\log \pi(a_t|\tau_{<t}, \mathcal{D})]$.

3 Approach

We first discuss the code representation in Section 3.1. We then describe our approach to precondition inference and action prediction problems in Sections 3.2 and 3.3, respectively. See Figure 1 for an overview of our approach.

3.1 Representing Agent Trajectories as Programs

We represent the agent's interaction with the environment as a program. We represent every action and observation as one or more function calls that modify the state of a predefined set of variables v that capture a summary of the agent's experience (e.g., observations $o_{1:t}$ and actions $a_{1:t-1}$).[1] Such a representation exists since we assume discrete observation and action spaces (for instance, we can define a separate function for each string in $\mathcal{O}, \mathcal{A}$). Variables v capture information about the state of the environment and are useful for reasoning about the plausibility of performing an action at any given point. Assume we have defined a set of functions $\mathcal{F}_{\mathcal{O}}, \mathcal{F}_{\mathcal{A}}$ corresponding to observations $\mathcal{O}$ and actions $\mathcal{A}$ respectively. Given this representation of observations and actions, a trajectory τ can be viewed as a program which consists of a sequence of function calls.[2] We present an example program in Figure 1 and further examples in the Appendix.

3.2 Precondition Inference

The precondition g_a of an action a is a function $g_a(\tau_{<t}) \in \{0, 1\}$ that informs whether the action is plausible in a given context $\tau_{<t}$ (i.e., $g_a(\tau_{<t}) = 0$ represents the action is implausible and $g_a(\tau_{<t}) = 1$ represents the action is plausible). Equivalently, for the corresponding function in the program representation $f_a \in \mathcal{F}_{\mathcal{A}}$, we seek to identify assertion statements in terms of variables v (which represent a summary of $\tau_{<t}$) and the function arguments of f_a. (For example in Figure 1, the precondition for the action `FindRestaurants` is identified as `assert user.informed['city'] and user.informed['cuisine']`).

We predict the preconditions for each action independently of other actions and the process described below is repeated for each action $a \in \mathcal{A}$.[3] Our approach to precondition inference consists of the following steps: candidate generation, validation and ranking. We detail these steps next (See Appendix D for an illustration).

Candidate Generation. Given demonstration trajectories $\mathcal{D}$ we first generate candidate preconditions by prompting a pre-trained code generation model. The prompt consists of (i) a demonstration $\tau \in \mathcal{D}$ (ii) definitions of functions $\mathcal{F}_{\mathcal{O}}$ and (iii) definition of function f_a with the `assert` keyword in its body. The `assert` prefix forces the model to generate assertion statements (as opposed to arbitrary code). We vary the demonstration program and sample multiple precondition candidates for each demonstration to come up with a pool of plausible candidates $\mathcal{H}_a^{\text{initial}}$. The pre-trained model is expected to understand what are appropriate contexts in which a function can be used and use this understanding to come up with assertion statements.

Although pretrained models have strong priors about appropriate assert statements, the above process has some limitations. First, generated statements may not be meaningful due to syntax errors or other deficiencies in the generated expressions. Second, the candidates are obtained based on static analysis of the program alone and the model has to implicitly reason about execution and program state in order to predict accurate preconditions.

Candidate Validation. One of the key advantages of a program representation as opposed to alternative representations such as natural language is the ability to execute. We augment the above candidate generation approach with a verification approach where each candidate is vetted for validity and consistency with the data. Given a candidate assertion, we verify whether each of the demonstration programs can run successfully (e.g,. replacing the function body with a candidate assertion and executing a demonstration program). Candidates which led to execution failures are discarded. The remaining candidates $\mathcal{H}_a^{\text{valid}}$ are thus valid and consistent with the data.

Many of the assertions generated will not be useful in practice. For example, trivial assertions such as '`assert True`' do not convey any useful information about instances where the function is applicable. Although all the candidate assertions at this point are valid, they may be *sub-optimal* for the purpose

[1] For instance, the variable `informed` captures whether a slot preference was indicated by the user in Figure 1.

[2] We interchangeably refer to actions/observations as functions and trajectories as programs in the rest of the paper.

[3] Jointly reasoning about preconditions for all functions in $\mathcal{F}_{\mathcal{A}}$ can be interesting and is left to future work.

of constructing a policy. We seek assertions that help discriminate situations where the function is applicable. We propose a ranking mechanism to identify the most discriminative assertions.

Candidate Ranking. We seek to identify a small set of precondition/assertion candidates from $\mathcal{H}^{\text{valid}}$ useful for constructing an agent policy. We begin with the observation that if candidates h_1, h_2 are such that h_2 is satisfied whenever h_1 is satisfied (i.e., $h_1 \Rightarrow h_2$), h_1 is more desirable as it is more discriminative of the contexts where action a is applicable (and hence leads to a better policy). As we discuss in the experiments, this assumption leads to high precision solutions and compromises recall. We leave alternate ranking criteria to future work.

However, verifying the above property ($h_1 \Rightarrow h_2$) for given precondition candidates is generally intractable. We thus consider an approximation where we examine if the property is satisfied in scenarios that appear in the demonstration trajectories. Consider the precondition function $g_a(\,\cdot\,; h)$ which assumes h to be the precondition of action a. Define $C_a^h = \{(i,j) | g_a(\tau^i_{<j}; h) = 1, \tau^i \in \mathcal{D}\}$ to be the set of instances (i,j) where precondition h is satisfied at time-step j of demonstration trajectory τ^i. We use $C_a^{h_1} \subseteq C_a^{h_2}$ as a proxy to determine whether $h_1 \Rightarrow h_2$.

We define the optimal set of assertion statements as in Equation (1). When multiple equivalent candidates exist, we choose one random representative to retain in set $\mathcal{H}_a^{\text{opt}}$. Note that the conjunction of assertions in $\mathcal{H}_a^{\text{opt}}$ is equivalent to that of $\mathcal{H}_a^{\text{valid}}$. However, identifying a small set of assertion statements is beneficial both for interpretability and for providing as a prompt for models which have limited context lengths.

$$\mathcal{H}_a^{\text{opt}} = \{h \in \mathcal{H}_a^{\text{valid}} \mid \nexists h' \in \mathcal{H}_a^{\text{valid}} \text{ s.t. } C_a^{h'} \subset C_a^{h}\} \tag{1}$$

3.3 Precondition-aware Action Prediction

We pose action prediction as a code completion problem where given a partial agent trajectory, a code model is tasked with suggesting possible next actions (as functions from $\mathcal{F}_{\mathcal{A}}$, with appropriate arguments). We address this learning problem with few-shot prompting where the agent is provided demonstration trajectories as part of its prompt. In addition, we also provide information about preconditions as part of the prompt consisting of functions in $\mathcal{F}_{\mathcal{O}}, \mathcal{F}_{\mathcal{A}}$. Functions $f_a \in \mathcal{F}_{\mathcal{A}}$ consist of assertion statements $\mathcal{H}_a^{\text{opt}}$ (predicted using the process described in the previous section) in the body representing the preconditions for that action. In summary, the prompt for the code model, denoted by `query`, consists of (i) one or more demonstrations $\tau \in \mathcal{D}$, (ii) definitions of functions $\mathcal{F}_{\mathcal{O}}$, $\mathcal{F}_{\mathcal{A}}$ and (iii) query program consisting of past observations and actions (See Figure 1 for an illustration).

As with precondition inference, reasoning about the next action based on static analysis of the preconditions alone is not ideal. We thus augment the action generation approach above with a validation process where predicted action candidates are verified using execution. Next action candidates are sampled from the model until an action consistent with the preconditions is found or a maximum number of attempts is exceeded. The first attempt uses greedy sampling and subsequent attempts resort to random sampling. If random sampling does not yield an action consistent with preconditions, we default to the greedy action. We sample an action by generating tokens until a newline token is encountered. See Algorithm 1 of the Appendix for a pseudocode description of the algorithm.

4 Experiments

We attempt to answer the following key questions in our evaluation: (i) Is it possible to extract information about action preconditions from demonstrations of agent behavior? (ii) Is such inferred precondition information useful for building better agent policies? (iii) Do program representations enable us to perform these tasks better compared to natural language representation?

We use the python programming language as the code representation in our experiments due to it's simplicity and popularity, as well as the recent release of code models that primarily focus on python. We use the CodeGen 2B (Nijkamp et al., 2022) and StarCoder 16B (Li et al., 2023) open-source pre-trained code models in our experiments. Specifically, we use the versions of these models which were further pre-trained on python code after initial pre-training on many programming languages.

Action	**Ground-truth precondition**	**Predicted precondition**	**Prec**	**Rec**	**F_1**
`INFORM(slot)`	`user.requested_slot[slot]`	`slot in user.requested_slot`	1	1	1
`REQUEST(slot)`	`not user.informed_slot[slot]`	`user.informed_slot[slot] == False`	1	1	1
`GOODBYE()`	`user.no_more`	`user.no_more`	1	1	1
`OFFER(slot)`	`query_success`	`query_success == True`	1	1	1
`INFORM_COUNT()`	`query_success`	`query_success == True`	1	1	1
`OFFER_INTENT(intent)`	`user.selected`	`query_success == True`	0.49	1	0.66

`CONFIRM(slot)`	`user.informed_slot['to_location']`
	`user.informed_slot['from_location']`
	`user.informed_slot['leaving_date']`
	`user.informed_slot['travelers']`
		`user.selected`
		`slot != 'travelers' or`
		`user.informed_slot[slot]`
			0.96	0.7	0.81

`NOTIFY_SUCCESS()`	`query_success`	`user.selected` `not user.no_more`	1	0.44	0.62
`REQ_MORE()`	`user.selected or user.no_more`	`user.selected`	1	1	1

`FindBus()`	`user.informed_slot['to_location']`
	`user.informed_slot['from_location']`
	`user.informed_slot['leaving_date']`
		`not user.no_more`
		`user.informed_intent['FindBus']`
		`user.informed_slot['leaving_date']`
		`user.informed_slot['from_location']`
			0.99	0.89	0.94

`BuyBusTicket()`	`user.affirmed`	`user.affirmed`	1	1	1

Table 1: Ground-truth and predicted preconditions for actions in the *Buses* domain of the SGD benchmark. Precision, Recall and F-score metrics for the predictions are shown in the last three columns. The `'self'` prefix is omitted from variables for brevity.

4.1 Benchmarks

Task Oriented Dialog Benchmark. The decision making component of a task-oriented dialog system is a dialog manager which takes as input a sequence of utterances represented as dialog acts and predicts the next action. In this setting, a user and a system (agent) take turns to speak and user utterances constitute the agent's observations. We use the SGD dataset (Rastogi et al., 2020) in our experiments. There are 11 user acts and 11 system acts defined in the dataset, each of which takes either no argument, a single slot argument or a single intent argument. We define analogous functions $\mathcal{F}_{\mathcal{O}}$ and $\mathcal{F}_{\mathcal{A}}$ corresponding to each of these acts. We take an object-oriented approach and group these functions respectively under a user class and a system class. We define variables v corresponding to each user action that records whether the action was performed (e.g., `informed_slot[]`, `requested_slot[]` in Figure 1). We experiment with 10 domains (schemas) from the SGD dataset. 10 instances are used as demonstrations and 50 instances for testing. We manually define the ground-truth preconditions for each system action based on prior knowledge about the dialog acts. Note that these are only used for evaluation and no supervision is provided to the model about ground-truth preconditions.

Embodied Textworld. We experiment with the ALFworld embodied textworld benchmark (Shridhar et al., 2020) which involves an agent interacting with an environment to perform object interaction tasks. The observations and actions are natural language statements and the agent is expected to perform a task specified using a natural language instruction (e.g., *move the keys to the table*). There are 9 types of actions which include interaction and navigation actions. Each of these action types take an object argument and/or a receptacle argument and we define analogous functions $\mathcal{F}_{\mathcal{A}}$. In addition, we define auxiliary functions $\mathcal{F}_{\mathcal{O}}$ for adding/removing an object from the agent's inventory and updating the set of objects visible to the agent. We define three variables v that summarize the agent's experience: the set of visible objects, the agent's inventory and states of objects in the environment (e.g., open/closed). The dataset has 6 task types. We use 2 instances from each task type as demonstrations and the standard test set of the benchmark for testing. The benchmark provides ground-truth preconditions for the actions in a PDDL (Planning Domain Definition Language) representation, which we convert to python assertions for evaluation. We provide the complete program specification for these benchmarks in Appendix A.

4.2 Precondition Inference

Metrics. Recall that a precondition $g_a(\tau_{<t}) \in \{0,1\}$ informs whether an action $a \in \mathcal{A}$ is plausible given context $\tau_{<t} = (o_{1:t}, a_{1:t-1})$. Define $C_a = \{(i,j) | g_a(\tau^i_{<j}) = 1, \tau^i \in \mathcal{D}^{\text{test}}\}$ to be the set of instances (i,j) where the precondition of action a is satisfied at time-step j of test trajectory

Model	SGD			Alfworld		
	Prec	Rec	F_1	Prec	Rec	F_1
Prompt.(2B)	0.73	0.75	0.74	0.59	**0.89**	0.71
Prompt.(16B)	0.73	0.77	0.75	0.69	0.68	0.68
Ours(2B)	0.91	0.69	0.78	**1.0**	0.85	**0.92**
Ours(16B)	**0.92**	**0.78**	**0.84**	**1.0**	0.81	0.90

Table 2: Precondition Inference performance on the SGD and Alfworld benchmarks. Metrics are precision, recall and F_1 score (All metrics higher ↑ the better). The simple prompting baseline and our approach are both evaluated with the CodeGen 2B model and StarCoder 16B model.

$\tau^i \in \mathcal{D}^{\text{test}}$. We define precision and recall metrics for a particular action a as in Equation (2), where $C_a^{\text{pred}}, C_a^{\text{gt}}$ respectively correspond to the predicted and ground-truth preconditions. The F_1 score is defined as the harmonic mean of precision and recall. These metrics are macro-averaged across all actions $a \in \mathcal{A}$ to obtain the final metrics.

$$\text{Prec} = \frac{|C_a^{\text{pred}} \cap C_a^{\text{gt}}|}{|C_a^{\text{pred}}|}, \text{Rec} = \frac{|C_a^{\text{pred}} \cap C_a^{\text{gt}}|}{|C_a^{\text{gt}}|} \tag{2}$$

Baseline. We consider a simple prompting baseline where a code model is prompted with a partial program specification and predicts assertion statements with greedy decoding.

Results. Table 2 shows the precondition inference performance of code generation models. The precision for our models are high since the ranking criteria we adopted encourages high precision solutions. For example, the ground truth precondition for the `INFORM(slot)` action in **??** is `requested_slot[slot]`.[4] However, the predicted precondition is `(requested_slot[slot] and query_success==True)`. The models pick up the fact that `query_success==True` in all instances the `INFORM(slot)` action appeared in the demonstration trajectories. In this case precision is 1.0 since the ground truth precondition will be satisfied whenever the predicted precondition is satisfied. However, the recall is lower (0.57) since the predicted and ground-truth preconditions do not agree whenever `query_success! =True`.

Table 1 presents qualitative prediction results for all action types in the *Buses* domain of the SGD benchmarks. Note that our models receive no supervision about preconditions and predict these candidates by only observing expert demonstrations. We present qualitative prediction results for the Alfworld benchmark in Appendix C. Next, we analyze how these inferred action precondition can help build better agent policies.

4.3 Precondition-aware Agent Policy

Metrics. In the SGD benchmark, we evaluate policy performance against actions in the demonstrations. Since the agent needs to predict multiple actions in a turn, we compute F_1 score treating demonstration actions as ground-truth. For the embodied textworld task, a simulation environment is available, and we define the **success rate (SR)** metric which measures how often the agent successfully completed the given task. We also define **precondition compatibility**, which measures how often the predicted action is compatible with (i.e., does not violate) the ground-truth preconditions.

Baselines. We consider the following baseline policy approaches. First, we consider a *few-shot prompting baselines* that imitates the demonstration trajectories to predict the next action. Second, we consider a prompting baseline which is provided the ground-truth preconditions in addition to the demonstration trajectories. We call this the *precondition prompting* baseline. We compare our approach (*ours*) which uses predicted preconditions along with the proposed precondition-aware action prediction strategy against these baselines.

[4] `'self'` and `'user'` omitted for brevity

Backbone	Language guidance	Approach	SGD		Alfworld	
			F_1	Cmp.	SR	Cmp.
LLAMA 13B	No	Few-shot prompting	0.77	0.88	0.06	0.10
		Precondition Prompting	0.78	0.78	0.06	0.27
		Ours	0.80	0.96	0.10	0.50
StarCoder 16B	No	Few-shot prompting	0.84	0.92	0.06	0.56
		Precondition Prompting	0.84	0.94	0.15	0.62
		Ours	0.85	0.97	0.23	0.96
StarCoder 16B	React Prompts	Few-shot prompting	-	-	0.44	0.81
		Ours	-	-	0.48	0.96

Table 3: Policy performance on SGD and Alfworld benchmarks. The performance metrics are F_1 score, task success rate (SR) and precondition compatibility (Cmp.) (All metrics higher ↑ the better).

In addition to the regular prompting strategy, we also consider the effect of adding chain-of-thought natural language guidance inspired by React prompts (Yao et al., 2022). We design 'think' prompts similar to the original work and insert them as code comments. Before each action, the agent generates an optional code comment ('think' step) and predicts an action.

We use both LLM and CodeLM models for the experiments. LLAMA-13B (Touvron et al., 2023) is used as the LLM in our experiments.[5] The StarCoder 16B model is used as the CodeLM unless specified otherwise. We use preconditions predicted by the StarCoder 16B model in all experiments.

Results. We present the main results in Table 3. For fair comparison, we limit the number of demonstrations to 4 for the SGD benchmark, which is the maximum number of demonstrations the LLM baseline (LLAMA-13B) can accommodate. First, we observe that code models generally perform better than LLMs. LLMs particularly struggle on Alfworld (6% success rate). Our proposed approach which combines few-shot prompting with knowledge about preconditions performs better than both LLM and code baselines on both benchmarks, with a significant improvement on Alfworld (6% to 23%). In particular, we observe that the proposed precondition-based reasoning approach helps both LLMs and code models generate actions that are more accurate and consistent with preconditions.

We observe that the React code comments significantly help improve policy performance. Our approach improves the success rate of the React agent from 44% to 48% and its precondition compatibility from 81% to 96%. This shows the synergistic potential of our approach and recent advances in prompting for policy learning problems such as generating natural language rationales before predicting actions.

These results shows that explicitly reasoning about preconditions helps build better policies. Although few-shot prompting enables language/code models to perform tasks with limited supervision, they are generally limited by the number of demonstrations that can fit in the context window. Condensing the information in multiple trajectories in a small set of rules (e.g., preconditions) can help overcome this limitation, and this idea may further be applicable to other tasks in general.

4.4 Ablations and Analysis

We perform a series of ablations to understand the impact of different components in our pipeline.

How to incorporate precondition information? In Figure 2 we analyze the impact of (i) including precondition information in the prompt and (ii) the precondition-aware action sampling strategy. Overall, while that inclusion of precondition information in the model prompt generally helps, our proposed precondition-aware action prediction strategy yields more consistent and significant improvements (e.g., 0.64 to 0.68 with predicted preconditions and 0.73 with ground-truth

[5] LLAMA performed best across the open-source LLMs we tried.

Model size	Prec. prompt	Prec. sample	1-shot		10-shot	
			$F_1 \uparrow$	Cmp.↑	$F_1 \uparrow$	Cmp.↑
2B	✗	✗	0.51	0.73	-	-
2B	✓	✗	0.52	0.75	-	-
2B	✓	✓	0.58	**0.92**	-	-
16B	✗	✗	0.64	0.78	0.89	0.97
16B	✓	✗	0.65	0.77	0.90	0.97
16B	✓	✓	**0.68**	0.90	**0.91**	**0.99**

Figure 2: Ablation to study the effect of (a) providing preconditions as part of the prompt (column 2) and (b) precondition-aware action prediction strategy (column 3) in the SGD benchmark. The 2B model can only accommodate upto 4 demonstrations and hence is not evaluated in the 10-shot setting. The performance metrics are F_1 score and precondition compatibility (Cmp.).

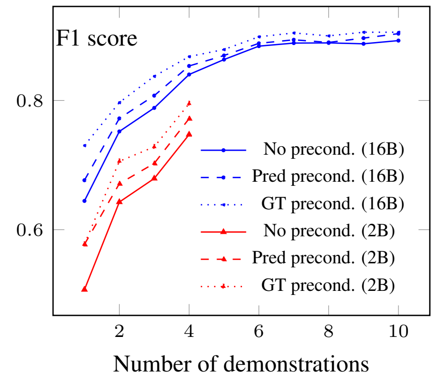

Figure 3: Ablation showing policy performance (F1 score) in the SGD benchmark when varying (a) the number of demonstrations (from 1 to 10), (b) model scale (CodeGen 2B vs StarCoder 16B) and (c) precondition information available to the policy model (None/Predicted/Ground-truth).

preconditions in the 1-shot setting). In addition, it helps predict actions that are more consistent with their preconditions (e.g., precondition compatibility improves to > 0.9 for both 2B and 16B models).

Amount of Supervision. Figure 3 shows model performance for varying amounts of supervision. Due to it's maximum context size of 2048 tokens, the CodeGen 2B model can only accommodate upto 4 demonstrations. Knowledge about preconditions is particularly helpful when the number of demonstrations is small. Even as we increase the number of demonstrations, models continue to benefit from explicitly reasoning about preconditions. Furthermore, performance with ground-truth preconditions shows that improvements in quality of preconditions lead to improvements in policy performance.

Model scale. We observe that the small model benefits more from precondition knowledge compared to the big model regardless of the amount of supervision. Enhancing the reasoning capabilities of small models is important as big models demand higher costs and computation. Leveraging precondition information and execution-based verification is a promising strategy to enhance small models.

5 Related Work

Programs as Policies. There exist prior work that advocate viewing robot policies as code/programs (Liang et al., 2022; Singh et al., 2022). This is in contrast to most prior work that reason about plans almost entirely in natural language. Similar to natural language based prompting, LLMs are prompted with examples of programs corresponding to example tasks and are required to generate programs for a query task. The programs can be rich and composed of functions supported by the target robot API or third party library functions. Code comments help break down high-level task into subtasks and assertions are used to take environment feedback into account and provide an error recovery mechanism. These prior work assume that precondition information is specified as part of the prompt. In contrast, our work attempts to discover such information from action trajectories.

Reasoning with Verification. Prior work has attempted to augment capabilities of language models with programs and execution. Liu et al. (2022) combine LLMs with a physics simulator to answer physics questions. Given a query, a text-to-code model trained with supervised learning generates a program in order to perform a simulation. The simulator performs the simulation and produces an output, which is fed as additional information to the LLM to generate a response. Gao et al. (2022) interleave natural language chain-of-thought statements with program statements to perform calculations for arithmetic reasoning problems. Analogous to these work, we find that the ability to verify via execution improves the performance of LMs, particularly small models.

Reasoning via Prompting. Chain-of-thought prompting (Wei et al., 2022) has emerged as a powerful technique for getting language models to perform step-by-step reasoning. These ideas have also been applied to planning problems (Huang et al., 2022b; Yao et al., 2022) where agents are taught how to come up with rationales for predicted actions with few-shot demonstrations. Compared to chain-of-thought rationales, which are dynamically generated on the fly and are problem-specific, we seek to identify a set of rules that capture action preconditions. This provides more controllability over the action generation process and the rules can also be vetted/edited to achieve desired behavior.

Structured Prediction with Programs. Programs have been used as a representation for structured prediction tasks in NLP (Wang et al., 2022; Madaan et al., 2022; Zhang et al., 2023). Code models have been used by these work for modeling procedural real-world activities, also called the script generation problem. They find that code models have strong capability to reason about event sequences with minimal supervision compared to language models.

Subtask Graph Framework. Subtask Graphs are a modeling framework proposed by Sohn et al. (2018; 2020) to learn subtask preconditions from demonstrations. Preconditions are modeled as boolean expressions involving a pre-defined set of boolean subtask variables which represent whether a subtask has been completed or not. An Inductive Logic Programming (ILP) algorithm is used to identify the optimal boolean expression. This framework was further extended to model real-world procedural activities in Jang et al. (2023); Logeswaran et al. (2023). In contrast to the use of boolean expressions as the class of functions to model preconditions, the program representation we consider has the flexibility to represent a broader set of scenarios. We draw inspiration from these works to formulate and evaluate the precondition inference component in our approach.

6 Conclusion

This work presented a novel approach to reason about action preconditions using programs for learning agent policies in a sequential decision making setting. We proposed to use programs as a representation of the agent's observations and actions and showed that precondition inference and action prediction can be formulated as code-completion problems. By leveraging the strong priors of pre-trained code models, we proposed a novel approach to infer action preconditions from demonstration trajectories without any additional supervision. With the predicted preconditions, our precondition-aware action prediction strategy enables the agent to predict actions that are consistent with the preconditions and lead to better task completion compared to baselines. Our study opens an exciting new direction to reason about action preconditions by leveraging code models.

Limitations

One of the limitations of our work is that coming up with a program specification requires domain knowledge. For instance, we define variables such as the agent's inventory and set of visible objects in the embodied textworld benchmark and variables that capture which slots were requested/informed in the dialog benchmark. It would be interesting to generalize our approach to broader scenarios by identifying the key variables of interest with less manual intervention. It would be interesting to further explore the use of preconditions to more actively direct the agent towards task completion.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pp. 9118–9147. PMLR, 2022a.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022b.

Yunseok Jang, Sungryull Sohn, Lajanugen Logeswaran, Tiange Luo, Moontae Lee, and Honglak Lee. Multimodal subtask graph generation from instructional videos. *arXiv preprint arXiv:2302.08672*, 2023.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! 2023.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.

Ruibo Liu, Jason Wei, Shixiang Shane Gu, Te-Yen Wu, Soroush Vosoughi, Claire Cui, Denny Zhou, and Andrew M Dai. Mind's eye: Grounded language model reasoning through simulation. *arXiv preprint arXiv:2210.05359*, 2022.

Lajanugen Logeswaran, Yao Fu, Moontae Lee, and Honglak Lee. Few-shot subgoal planning with language models. *arXiv preprint arXiv:2205.14288*, 2022.

Lajanugen Logeswaran, Sungryull Sohn, Yunseok Jang, Moontae Lee, and Honglak Lee. Unsupervised task graph generation from instructional video transcripts. *arXiv preprint arXiv:2302.09173*, 2023.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*, 2022.

Vincent Micheli and François Fleuret. Language models are few-shot butlers. *arXiv preprint arXiv:2104.07972*, 2021.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Abhinav Rastogi, Xiaoxue Zang, Srinivas Sunkara, Raghav Gupta, and Pranav Khaitan. Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 8689–8696, 2020.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

Sungryull Sohn, Junhyuk Oh, and Honglak Lee. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. *Advances in neural information processing systems*, 31, 2018.

Sungryull Sohn, Hyunjae Woo, Jongwook Choi, and Honglak Lee. Meta reinforcement learning with autonomous inference of subtask dependencies. *arXiv preprint arXiv:2001.00248*, 2020.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Xingyao Wang, Sha Li, and Heng Ji. Code4struct: Code generation for few-shot structured prediction from natural language. *arXiv preprint arXiv:2210.12810*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Li Zhang, Hainiu Xu, Yue Yang, Shuyan Zhou, Weiqiu You, Manni Arora, and Chris Callison-Burch. Causal reasoning of entities and events in procedural texts. *arXiv preprint arXiv:2301.10896*, 2023.

A Program Specification

We present the program specification for the *Restaurants* domain in the SGD benchmark in Figure 4 and the Alfworld benchmark in Figure 5.

```python
from collections import defaultdict

class USER:

  def __init__(self):
    self.informed_intent = defaultdict(lambda:
     False)
    self.informed_slot = defaultdict(lambda:
     False)
    self.requested_slot = defaultdict(lambda:
     False)

    self.no_more = False
    self.selected = False
    self.affirmed = False
    self.affirm_intent = False
    self.negate_intent = False
    self.request_alternatives = False

  def INFORM_INTENT(self, intent):
    self.informed_intent[intent] = True

  def NEGATE_INTENT(self):
    self.negate_intent = True

  def AFFIRM_INTENT(self):
    self.affirm_intent = True

  def REQUEST_ALTS(self):
    self.request_alternatives = True

  def INFORM(self, slot):
    self.informed_slot[slot] = True

  def REQUEST(self, slot):
    self.requested_slot[slot] = True

  def GOODBYE(self):
    self.no_more = True

  def THANK_YOU(self):
    self.no_more = True

  def SELECT(self):
    self.selected = True

  def AFFIRM(self):
    self.affirmed = True

  def NEGATE(self):
    self.affirmed = False
```

```python
class SYSTEM:

  def __init__(self, user):
    self.user = user
    self.query_success = None

  def INFORM(self, slot):
    assert self.user.requested_slot[slot]

  def REQUEST(self, slot):
    assert not self.user.informed_slot[slot]

  def GOODBYE(self):
    assert self.user.no_more

  def FindRestaurants(self):
    assert self.user.informed_slot['city']
    assert self.user.informed_slot['cuisine']
    assert self.user.informed_intent['
     FindRestaurants']

  def ReserveRestaurant(self):
    assert self.user.selected or self.user.
     affirmed

  def OFFER(self, slot):
    assert self.query_success or self.user.
     affirmed

  def INFORM_COUNT(self):
    assert self.query_success

  def OFFER_INTENT(self, intent):
    assert self.user.selected

  def CONFIRM(self, slot):
    assert self.user.informed_slot['time']
    assert self.user.informed_slot['city']
    assert self.user.selected or self.user.
     informed_slot['restaurant_name']

  def NOTIFY_SUCCESS(self):
    assert self.query_success

  def NOTIFY_FAILURE(self):
    assert not self.query_success

  def REQ_MORE(self):
    assert self.user.selected or self.user.
     no_more or not self.query_success

  def set_query_status(self, status):
    self.query_success = status
```

Figure 4: Program specification of the *Restaurants* domain in the SGD benchmark. Note that the assertion statements are assumed to be unknown and only used for evaluation.

```python
from collections import defaultdict

class Environment:

  def __init__(self):
    self.object_states = defaultdict(lambda: False)

  def set_property(self, obj, property, value):
    self.object_states[(obj, property)] = value

  def get_property(self, obj, property):
    return self.object_states[(obj, property)]

class Agent:

  def __init__(self, env):
    self.env = env
    self.inventory = None
    self.visible_objects = set()

  def add_inventory(self, obj):
    self.inventory = obj

  def remove_inventory(self, obj):
    self.inventory = None

  def update_visible_objects(self, *args):
    self.visible_objects.update(list(args))

  def is_visible(self, obj):
    return obj in self.visible_objects

  def goto(self, recep):
    assert self.is_visible(recep)

  def open(self, recep):
    assert self.is_visible(recep)
    assert self.env.get_property(recep, 'open') == False

  def close(self, recep):
    assert self.is_visible(recep)
    assert self.env.get_property(recep, 'open') == True

  def take(self, obj, recep):
    assert self.is_visible(obj)
    assert self.is_visible(recep)
    assert self.inventory == None, 'Can only hold one object at a time'

  def put(self, obj, recep):
    assert self.inventory == obj, 'Need to be holding the object'
    assert self.is_visible(recep)

  def clean(self, obj, recep):
    assert self.is_visible(recep)
    assert self.inventory == obj, 'Need to be holding the object'
    assert 'sink' in recep

  def heat(self, obj, recep):
    assert self.is_visible(recep)
    assert self.inventory == obj, 'Need to be holding the object'
    assert 'microwave' in recep

  def cool(self, obj, recep):
    assert self.is_visible(recep)
    assert self.inventory == obj, 'Need to be holding the object'
    assert 'fridge' in recep

  def toggle(self, obj):
    assert self.is_visible(obj)
```

Figure 5: Program specification of the Alfworld benchmark. Note that the assertion statements are assumed to be unknown and only used for evaluation.

B EXAMPLE TRAJECTORIES

Figure 6 shows an example trajectory from the *Restaurants* domain in the SGD benchmark and the corresponding program representation. Figures 7 and 8 show an example trajectory from the *pick and place* task of the Alfworld benchmark in its original text representation and the corresponding program representation.

```
USER:   (INFORM_INTENT, FindRestaurants)
SYSTEM: (REQUEST, city)
USER:   (INFORM, city)
SYSTEM: (REQUEST, cuisine)
USER:   (INFORM, cuisine)
SYSTEM: (FindRestaurants)
        (Query successful)
        (OFFER, restaurant_name)
        (OFFER, city)
        (INFORM_COUNT)
USER:   (REQUEST_ALTS)
SYSTEM: (OFFER, restaurant_name)
        (OFFER, city)
USER:   (REQUEST, has_live_music)
SYSTEM: (INFORM, has_live_music)
USER:   (INFORM_INTENT, ReserveRestaurant)
        (SELECT)
SYSTEM: (REQUEST, time)
USER:   (INFORM, time)
SYSTEM: (CONFIRM, restaurant_name)
        (CONFIRM, city)
        (CONFIRM, time)
        (CONFIRM, party_size)
        (CONFIRM, date)
USER:   (INFORM, date)
        (NEGATE)
SYSTEM: (CONFIRM, time)
        (CONFIRM, date)
USER:   (INFORM, time)
        (NEGATE)
SYSTEM: (CONFIRM, time)
        (CONFIRM, date)
USER:   (AFFIRM)
SYSTEM: (ReserveRestaurant)
        (Query successful)
        (NOTIFY_SUCCESS)
USER:   (REQUEST, serves_alcohol)
SYSTEM: (INFORM, serves_alcohol)
USER:   (THANK_YOU)
        (GOODBYE)
SYSTEM: (GOODBYE)
```

```
user.INFORM_INTENT('FindRestaurants')
system.REQUEST('city')
user.INFORM('city')
system.REQUEST('cuisine')
user.INFORM('cuisine')
system.FindRestaurants()
system.set_query_status(True)
system.OFFER('restaurant_name')
system.OFFER('city')
system.INFORM_COUNT()
user.REQUEST_ALTS()
system.OFFER('restaurant_name')
system.OFFER('city')
user.REQUEST('has_live_music')
system.INFORM('has_live_music')
user.INFORM_INTENT('ReserveRestaurant')
user.SELECT()
system.REQUEST('time')
user.INFORM('time')
system.CONFIRM('restaurant_name')
system.CONFIRM('city')
system.CONFIRM('time')
system.CONFIRM('party_size')
system.CONFIRM('date')
user.INFORM('date')
user.NEGATE()
system.CONFIRM('time')
system.CONFIRM('date')
user.INFORM('time')
user.NEGATE()
system.CONFIRM('time')
system.CONFIRM('date')
user.AFFIRM()
system.ReserveRestaurant()
system.set_query_status(True)
system.NOTIFY_SUCCESS()
user.REQUEST('serves_alcohol')
system.INFORM('serves_alcohol')
user.THANK_YOU()
user.GOODBYE()
system.GOODBYE()
```

Figure 6: Example trajectory from the *Restaurants* domain of the SGD benchmark. The original data representation is shown on the left and the corresponding program representation is shown on the right.

```
You are in the middle of a room. Looking quickly around you, you see a countertop 1, a drawer
     1, a drawer 2, a drawer 3, a drawer 4, a drawer 5, a drawer 6, a garbagecan 1, a
     handtowelholder 1, a handtowelholder 2, a sinkbasin 1, a toilet 1, a toiletpaperhanger 1,
      a towelholder 1, and a towelholder 2.
Your task is to: put some spraybottle on garbagecan
> go to drawer 4
The drawer 4 is closed.
> open drawer 4
You open the drawer 4. The drawer 4 is open. In it, you see nothing.
> close drawer 4
You close the drawer 4.
> go to drawer 2
The drawer 2 is closed.
> open drawer 2
You open the drawer 2. The drawer 2 is open. In it, you see nothing.
> close drawer 2
You close the drawer 2.
> go to garbagecan 1
On the garbagecan 1, you see nothing.
> go to drawer 3
The drawer 3 is closed.
> open drawer 3
You open the drawer 3. The drawer 3 is open. In it, you see nothing.
> close drawer 3
You close the drawer 3.
> go to countertop 1
On the countertop 1, you see a sink 1, a soapbottle 1, and a toiletpaper 1.
> go to toilet 1
On the toilet 1, you see a cloth 1, and a spraybottle 1.
> take spraybottle 1 from toilet 1
You pick up the spraybottle 1 from the toilet 1.
> go to garbagecan 1
On the garbagecan 1, you see nothing.
> put spraybottle 1 in/on garbagecan 1
You put the spraybottle 1 in/on the garbagecan 1.
```

Figure 7: Example trajectory from the *pick and place* task of the Alfworld benchmark in its original text representation.

```
def put_some_spraybottle_on_garbagecan():
    # put some spraybottle on garbagecan.
    env = Environment()
    agent = Agent(env)
    agent.update_visible_objects('countertop 1', 'drawer 1', 'drawer 2', 'drawer 3', 'drawer 4
     ', 'drawer 5', 'drawer 6', 'garbagecan 1', 'handtowelholder 1', 'handtowelholder 2', '
     sinkbasin 1', 'toilet 1', 'toiletpaperhanger 1', 'towelholder 1', 'towelholder 2')
    agent.goto('drawer 4')
    env.set_property('drawer 4', 'open', False)
    agent.open('drawer 4')
    env.set_property('drawer 4', 'open', True)
    agent.close('drawer 4')
    env.set_property('drawer 4', 'open', False)
    agent.goto('drawer 2')
    env.set_property('drawer 2', 'open', False)
    agent.open('drawer 2')
    env.set_property('drawer 2', 'open', True)
    agent.close('drawer 2')
    env.set_property('drawer 2', 'open', False)
    agent.goto('garbagecan 1')
    agent.update_visible_objects('garbagecan 1')
    agent.goto('drawer 3')
    env.set_property('drawer 3', 'open', False)
    agent.open('drawer 3')
    env.set_property('drawer 3', 'open', True)
    agent.close('drawer 3')
    env.set_property('drawer 3', 'open', False)
    agent.goto('countertop 1')
    agent.update_visible_objects('countertop 1', 'sink 1', 'soapbottle 1', 'toiletpaper 1')
    agent.goto('toilet 1')
    agent.update_visible_objects('toilet 1', 'cloth 1', 'spraybottle 1')
    agent.take('spraybottle 1', 'toilet 1')
    agent.add_inventory('spraybottle 1')
    agent.goto('garbagecan 1')
    agent.update_visible_objects('garbagecan 1')
    agent.put('spraybottle 1', 'garbagecan 1')
    agent.remove_inventory('spraybottle 1')
```

Figure 8: Example trajectory from the *pick and place* task of the Alfworld benchmark in our program representation.

C Qualitative Prediction Results for Alfworld

Table 4 shows precondition prediction results for the Alfworld benchmark.

Action	**Prediction**	**Ground-truth**
goto(recep)	is_visible(recep)	is_visible(recep)
open(recep)	recep in visible_objects env.get_property(recep, 'open') is False	is_visible(recep) env.get_property(recep, 'open') == False
close(recep)	env.get_property(recep, 'open')	is_visible(recep) env.get_property(recep, 'open') == True
take(obj, recep)	is_visible(obj) inventory is None	is_visible(obj) is_visible(recep) inventory == None
put(obj, recep)	obj in inventory	inventory == obj is_visible(recep)
clean(obj, recep)	obj in inventory	is_visible(recep) inventory == obj 'sink' in recep
heat(obj, recep)	inventory != None env.get_property(obj, 'heat') == False	is_visible(recep) inventory == obj 'microwave' in recep
cool(obj, recep)	obj in inventory env.get_property(obj, 'cool') == False	is_visible(recep) inventory == obj 'fridge' in recep
toggle(obj)	is_visible(obj) inventory is not None	is_visible(obj)

Table 4: Ground-truth and predicted preconditions for actions in the Alfworld benchmark.

D Precondition Inference

Figure 9 illustrates the pipeline for action precondition generation.

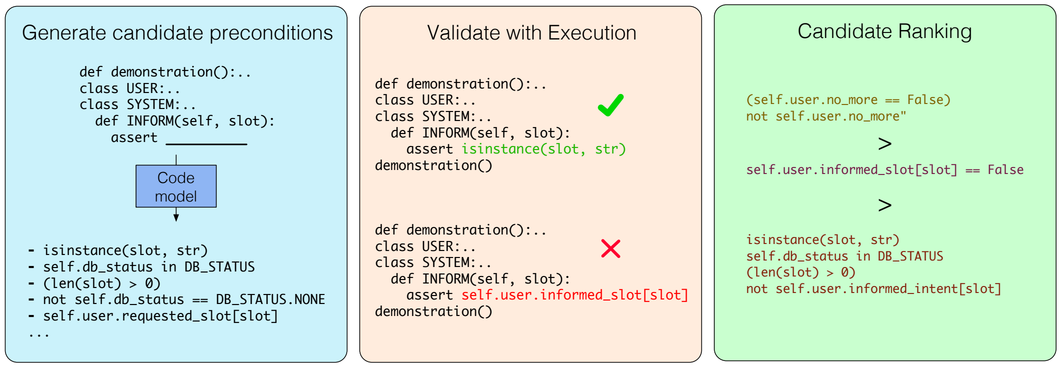

Figure 9: **Precondition Inference Overview**: We first generate multiple precondition candidates by prompting a code model with demonstration programs. We then validate these candidates based on the demonstrations via execution, and rank them to identify the most promising candidates.

E Precondition-aware Action Prediction

Algorithm 1 presents the pseudocode for our precondition-aware action sampling strategy.

Algorithm 1 Sample Next Action with Verification

Inputs: `query`, `max_attempts`
`verified` $\leftarrow$ *False*
`attempts` $\leftarrow 0$
`prediction` $\leftarrow$ GreedySample(`query`)
while not `verified` and `attempts` < `max_attempts` **do**
    if `attempts` > 0 **then**
        `prediction` $\leftarrow$ RandomSample(`query`)
    `program` $\leftarrow$ `query` + `prediction`
    `verified` $\leftarrow$ VerifyProgram(`program`)
    `attempts` $\leftarrow$ `attempts` + 1
if not `verified` **then**
    `prediction` $\leftarrow$ GreedySample(`query`)
Outputs: `prediction`, `verified`

F PRECONDITION INFERENCE EXAMPLE

Below we present an example showing the intermediate outputs of the precondition inference pipeline for action `INFORM` in the *Restaurants* domain of the SGD benchmark. The `INFORM` action takes a slot argument and the function syntax is `INFORM(self, slot)` (See the full program specification in Appendix A).

- Generate candidate preconditions $\mathcal{H}_a^{\text{initial}}$ by prompting a code model with demonstrations

```
not self.user.requested_slot[slot], 'Requested slot'
self.query_success!= None
(self.query_success == True)
(self.query_success)
self.query_success is not None
(self.user.informed_slot[slot])
self.query_success is None
(slot in self.user.informed_slot)
slot not in self.user.requested_slot.keys()
self.user.affirmed == True
self.user.affirmed
(hasattr(slot, '__name__'))
isinstance(slot, str)
slot!= 'date'
self.query_success in (True, False)
slot!='serves_alcohol'
self.user.requested_slot[slot]
```

- Identify valid candidates $\mathcal{H}_a^{\text{valid}}$ based on execution against the demonstration programs

```
self.query_success!= None
(self.query_success == True)
(self.query_success)
self.query_success is not None
isinstance(slot, str)
slot!= 'date'
self.query_success in (True, False)
self.user.requested_slot[slot]
```

- Identify candidates that are functionally equivalent

```
# Cluster 0
self.query_success!= None
self.query_success is not None
self.query_success in (True, False)
# Cluster 1
(self.query_success == True)
(self.query_success)
# Cluster 2
isinstance(slot, str)
slot!= 'date'
# Cluster 3
self.user.requested_slot[slot]
```

- Identify the precondition clusters that satisfy Equation (1)

```
# Cluster 1 (subsumes both cluster 0 and 2)
(self.query_success == True)
(self.query_success)
# Cluster 3
self.user.requested_slot[slot]
```

- Choose single representative (randomly) from each cluster to construct $\mathcal{H}_a^{\text{opt}}$

```
(self.query_success == True)
self.user.requested_slot[slot]
```