Predicate Invention from Pixels via Pretrained Vision-Language Models

Ashay Athalye[1*], Nishanth Kumar[1*]. Tom Silver[2], Yichao Liang[3], Tomás Lozano-Pérez[1], Leslie Pack Kaelbling[1]

[1]MIT CSAIL, [2]Princeton University, [3]University of Cambridge

Abstract

Our aim is to learn to solve long-horizon decision-making problems in highly-variable, combinatorially-complex robotics domains given raw sensor input in the form of images. Previous work has shown that one way to achieve this aim is to learn a structured abstract transition model in the form of symbolic predicates and operators, and then plan within this model to solve novel tasks at test time. However, these learned models do not ground directly into pixels from just a handful of demonstrations. In this work, we propose to invent predicates that operate directly over input images by leveraging the capabilities of pretrained vision-language models (VLMs). Our key idea is that, given a set of demonstrations, a VLM can be used to propose a set of predicates that are potentially relevant for decision-making and then to determine the truth values of these predicates in both the given demonstrations and new image inputs. We build upon an existing framework for predicate invention, which generates feature-based predicates operating on object-centric states, to also generate visual predicates that operate on images. Experimentally, we show that our approach — `pix2pred` — is able to invent semantically meaningful predicates that enable generalization to novel, complex, and long-horizon tasks across two simulated robotic environments.

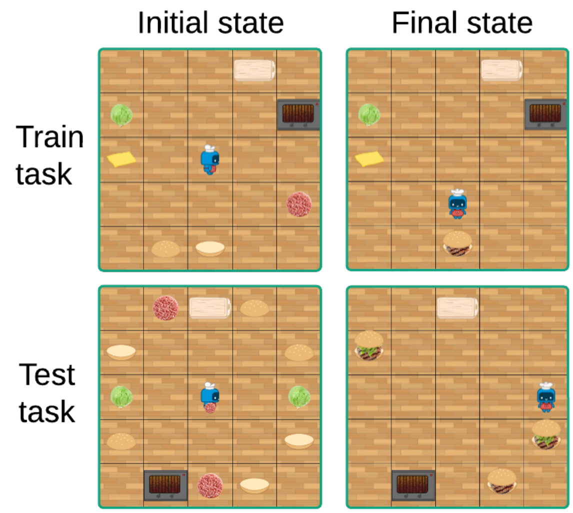

Figure 1: **Example train vs. test task in the "Burger" environment.** Training tasks involve creating a single burger with a single ingredient or stacking two ingredients. Tasks at test time involve more objects, different object arrangements, and novel compositions of goals as compared to those at training time. For example, one of the tasks at test time is to create three burgers – two with multiple ingredients – and where the robot starts out holding something (no training task starts out with the robot holding something).

1 Introduction

A core research objective in robotics is to develop a robot that can make long-term decisions from low-level sensory inputs to accomplish a very broad range of tasks. Recent work has shown that model-free imitation learning can solve complex tasks directly from pixels (Chi et al. 2023; Zhao et al. 2023). However, sequential and compositional generalization to novel tasks beyond the training distribution remain open challenges, especially when task horizons are long and demonstration datasets are small. In this work, we take an aggressively model-based approach to address these challenges. In particular, we use sparse demonstration data to learn *abstract world models* (Silver et al. 2023; Liang et al. 2024) and *plan* in those world models to achieve strong compositional and sequential generalization.

As a simple, pedagogical example, consider the "Burger" domain depicted in Figure 1. The agent is given demonstrations showing how to cook a patty, how to cut lettuce, how to stack ingredients, and how to make a sandwich with a single ingredient. It is then tasked with making multiple burgers, each with a cooked patty and chopped lettuce, from novel initial images where some patties may be cooked already, some lettuce chopped already, and where the robot might start out holding an object. This test task is challenging for model-free imitation learning because it is much longer-horizon than the demonstrated tasks and involves novel numbers and configurations of objects. However, if the agent can learn to recognize a small number of abstract state features, such as when a particular patty is `Cooked`, when a particular head of lettuce is `Chopped`, and whether or not the robot is currently holding an object, – as well as abstract transition rules involving these concepts: cooking a patty causes it to go from `Raw` to `Cooked`, chopping lettuce causes it to go from `Whole` to `Chopped`, a robot can only pick up an object if it is not `Holding` something else – it

*Equal contribution. Correspondence to {ashay, njk}@mit.edu

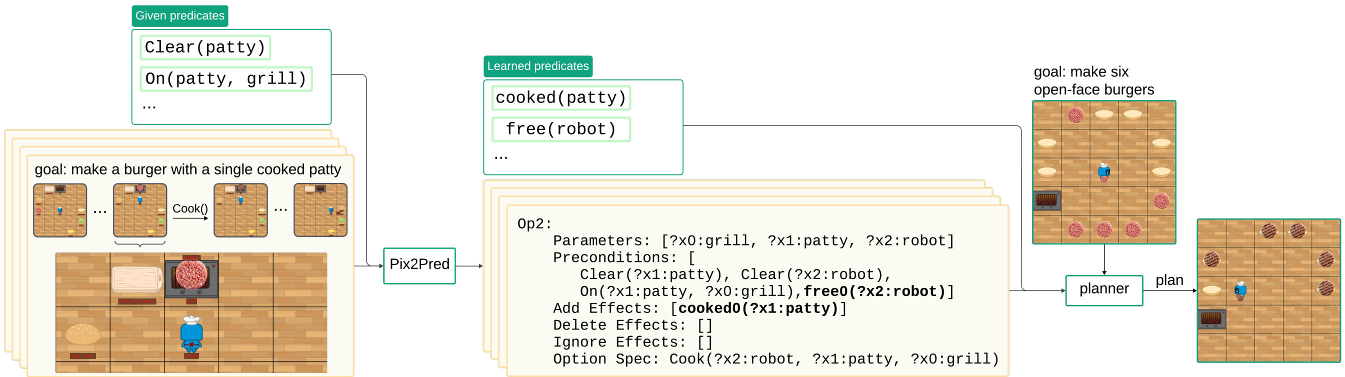

Figure 2: **`Pix2Pred` overview.** `pix2pred` are given an initial set of predicates, a set of controllers (temporally extended actions like *pick*, *cook*, etc.) and demonstrations of solving tasks in the form of goal-directed observation-action sequences. Our method learns more predicates and learns operators in terms of the learned and given predicates. Given a novel observation and goal at test time, we use a planner to generate a plan with our learned operators that achieves the goal. Goals in the train and test tasks are defined in terms of the given predicates; goals are written as text in the figure to simplify the figure.

can combine these concepts in many ways when planning and generalize substantially to new tasks.

We are specifically interested in learning PDDL models (Fox and Long 2003; McDermott et al. 1998) in the form of symbolic predicates (e.g., `Cooked`) and operators (e.g., cooking a patty causes it to go from `Raw` to `Cooked`) that can be leveraged by powerful classical planning algorithms (Helmert 2006). Such models feature several biases that are conducive to efficient planning and learning. Firstly, they are *relational*: world states and goals are characterized in terms of properties and relations between objects. Secondly, they are *object-oriented*: the number and properties of objects may vary across different problem instances, allowing the same model to support a wide range of specific tasks. Finally, they are *sparse*: each modeled action depends on and affects only a relatively small number of other objects, abstracting out irrelevant details and predicting core cause-effect interactions. Several recent works have demonstrated that such models can be learned efficiently to support generalization to tasks featuring new states, new goals, more objects, and longer time horizons than those demonstrated at training time (Silver et al. 2021, 2022, 2023; Asai and Fukunaga 2018; Ahmetoglu et al. 2022). However, missing from previous work is a method for learning predicates that *can be computed from raw visual observations* from *very few demonstrations*.

To address this gap, our first key idea is to use Vision-Language Models (VLMs) to *propose and ground* predicates. Given demonstrations, we leverage the commonsense knowledge of the VLM *propose* predicates with natural language names. Given an invented predicate and new states, we can again use the VLM to *ground* and evaluate the predicate directly on images. However, we find experimentally that the VLM's proposed set of predicates do not support efficient planning or generalization across training tasks. Given this, our second key idea is to *generate* a large pool of candidates and *subselect* predicates by *explicitly optimizing for efficient and effective planning*. We extend a recent method for program-synthesis-based predicate invention (Silver et al. 2023) that searches over sets of predicates via hill-climbing to minimize a fast-planning objective. After running this procedure to subselect predicates, we obtain a compact set of predicates that supports efficient and effective downstream decision-making.

In experiments, we evaluate the extent to which our method — `pix2pred` — is able to learn symbolic predicates and operators that generalize to novel problems. Across four tasks in two simulated environments, we compare to several related baselines and find that `pix2pred` consistently achieves the highest success rate on novel problem instances involving more objects, more complex goals, and longer horizons than those demonstrated during training

2 Related Work

Decision-Making with Foundation Models. Our work is strongly inspired by impressive recently-demonstrated capabilities of Large (Vision) Language Models (LLMs, VLMs) across a variety of challenging text and image tasks (OpenAI 2023; Team et al. 2023; Dubey et al. 2024). We build on a large body of work that leverages foundation models for decision-making in robotics tasks (Ichter et al. 2023; Singh et al. 2023; Liang et al. 2022; Huang et al. 2022; Curtis et al. 2024; Huang et al. 2023, 2024; Hu et al. 2023a; Duan et al. 2024; Yang et al. 2024) (see Hu et al. (2023b) for a recent survey). However, many of these approaches operate under a number of different restrictive assumptions. Some are confined to a particular task distribution (e.g., only pick-and-place) or seek to synthesize new relatively short-horizon skills (e.g., pouring a liquid into a cup) (Huang et al. 2023, 2024; Duan et al. 2024). Others assume skills are provided, and attempt to use foundation models to compose them to solve longer-horizon tasks (Ichter et al. 2023; Singh et al. 2023; Liang et al. 2022; Huang et al. 2022; Curtis et al. 2024; Hu et al. 2023a; Quartey et al. 2024; Kumar et al. 2024a). These approaches all use a foundation model directly for planning, which has been shown to perform significantly worse than dedicated planning algorithms that use an appropriate abstraction (Valmeekam et al. 2022; Kambhampati et al. 2024). By contrast, our approach leverages foundation models to construct abstractions that are fed into a planning system at inference time. We treat the work of Hu et al. (2023a) as representative of previous approaches that use foundation models to plan, and compare directly to variants of this approach in our experiments (Section 5).

Learning Abstractions for Planning. We build on a long line of work that learns abstractions for efficient planning (Bertsekas, Castanon et al. 1988; Jong and Stone 2005; Abel, Hershkowitz, and Littman 2016; Ugur and Piater 2015; Asai and Fukunaga 2018; Konidaris, Kaelbling, and Lozano-Pérez 2018; Silver et al. 2021; James, Rosman, and Konidaris 2022; Mao et al. 2022; Chitnis et al. 2022; Kumar et al. 2023; Silver et al. 2023; Han et al. 2024; Liu et al. 2024). Several of these works assume that predicates or operators are already given (Chitnis et al. 2021; Kumar et al. 2023; Mao et al. 2022) or focus on the problem of learning predicates and other abstractions from online interaction (James, Rosman, and Konidaris 2022; Liang et al. 2024), or assume access to dense natural language descriptions with each demonstration (Liu et al. 2024; Han et al. 2024). By contrast, we aim to invent predicates and operators in an offline fashion from image-based demonstrations with no supervision on the kinds or number of predicates to learn. Like us, "skills to symbols" (Konidaris, Kaelbling, and Lozano-Pérez 2018) performs predicate invention given skills, but operates on point cloud data and uses a bisimulation objective to select predicates that is different from the fast-planning objective that we use. Our work builds directly on the method from Silver et al. (2023), which learns predicates and operators that operate over a handcrafted, low-dimensional feature space. We extend this approach to learn predicates and operators that operate directly on images, and we demonstrate experimentally (Section 5) that this is necessary to solve tasks where attributes of objects important for decision-making cannot be easily extracted and encoded into a low-dimensional feature space.

3 Problem Setting and Background

We consider the problem of learning from demonstrations in deterministic, fully-observed environments. `pix2pred` takes as input a set of core object-types with meaningful natural-language names, an initial set of predicates, a set of parameterized controllers, and demonstrations of solving tasks in the form of goal-directed observation-action sequences, where goals are expressed in terms of the initial predicates and the sequences are segmented by controller. Each observation contains raw sensor data that includes one or more images characterizing the world state at that time, as well as data from sensors on the robot. We assume access to a perception function that can identify basic attributes of objects (e.g., position, dimensions, etc.) from the raw sensor data and can annotate objects of the given object-types on the images with unique names. Each controller execution in the demonstration is labeled with its name, the names of the objects it operates on in terms of names given by the perception function, and the continuous parameters it was executed with. From this, `pix2pred` learns predicates and operators (specifically in the form of a PDDL domain file (McDermott et al. 1998)), as well as samplers that produce the continuous parameters for the controllers. Below, we describe the precise structure of our demonstrations more formally. We also provide background on symbolic predicates, operators, samplers, and an existing framework for predicate invention that we build on.

3.1 Demonstrations

Our demonstrations are provided within a set of training *tasks* drawn from some distribution $\mathcal{T}$. We are ultimately interested in learning to solve any task T drawn from this distribution. All tasks occur within a common *environment*.

Environments and Tasks We model an environment as a tuple $\langle I, A, f, \Lambda, P\rangle$, where I is the observation space, A is the action space, and f is an unknown transition function, denoted $f : I \times A \to I$. An observation obs $\in I$ consists of a sequence of one or more images (obs^{img}) representing camera images from cameras operated by the robot, as well as a vector of continuous real-valued numbers (obs$^{\text{feat.}}$) representing raw sensor data associated with the robot, such as GPS and encoder data. The agent receives some observation obs_t at some timestep t, and can execute an action $a \in A$. The environment will then return a new observation $\text{obs}_{t+1} \leftarrow f(\text{obs}_t, a)$. We assume f is deterministic. Λ is a finite set of object types, where each *type* $\lambda \in \Lambda$ has a name (e.g., `robot`, `patty`, etc.) and a tuple of real-valued features representing basic attributes of the object (e.g., position for most objects, position and joint angles for the robot).

Within an environment, a *task* is a tuple $\langle \mathcal{O}, i_0, g\rangle$. $\mathcal{O}$ is a set of *objects*, each with a type from Λ. The environment's *perception function* $P : I \times \mathcal{O} \to \mathcal{X}$ processes each observation into a structured *state* used for downstream decision-making. These states have two parts: an *image-based* state, (x^{img}), and a *feature-based* state, (x^{feat}), concatenated together to form a state $x \in \mathcal{X}$. The perception function gives a unique name to each object in the observations and annotates the images with their names, as shown in Figure 2; this forms the image-based state. In practice this can be achieved by combining open-vocabulary object-detection and segmentation models (Ren et al. (2024)). The perception function also identifies the basic attributes of each object according to its type; this forms the feature-based state. In practice, this can be implemented with custom detectors for each property of each object, and is simple when the attributes are basic properties like pose and dimensions. We assume that objects are *fully-observable* to the perception function (i.e., there is no observation in which certain objects or types of objects are unobservable). i_0 is the *initial observation* of the task, and x_0 represents the initial state of a task produced by the perception function, where $x_o \leftarrow P(i_0, \mathcal{O})$. We will henceforth refer to a task as involving this initial state: $\langle \mathcal{O}, x_0, g\rangle$. We describe the goal g in a the next section.

Initial Predicate Set and Goals In order to specify the goal g of a task, as well as provide a basis for decision-making and learning, we provide an initial set of predicates

Ψ_{init} as part of the demonstrations. Each *predicate* ψ can be thought of as a template for a binary classifier over states. Specifically, a predicate is characterized by a name, an ordered list of types $(\lambda_1, \ldots, \lambda_m)$ and a classifier function $c_\psi : \mathcal{X} \times \mathcal{O}^m \rightarrow \{\text{true}, \text{false}\}$ parameterized by typed object arguments. A *ground atom* $\underline{\psi}$ consists of a predicate ψ and objects $(o_1, \ldots, o_m)$. Each ground atom induces a particular binary state classifier $c_{\underline{\psi}} : \mathcal{X} \rightarrow \{\text{true}, \text{false}\}$, where $c_{\underline{\psi}}(x) \triangleq c_\psi(x, (o_1, \ldots, o_m))$. For example, the ground atom `Cooked(patty0)` would have the value "true" if the object `patty0` is cooked.[1]

We assume the initial predicates Ψ_{init} are sufficient for representing task goals, and may additionally contain concepts useful across all tasks in the environment, but are insufficient on their own in that additional predicates will be necessary to capture all concepts critical for solving tasks. Specifically, the goal g of a task is a set of ground atoms involving predicates in Ψ_{init} and objects in $\mathcal{O}$. A goal g is said to *hold* in a state x if for all ground atoms $\underline{\psi} \in g$, the classifier $c_{\underline{\psi}}(x)$ returns true. We use the term "goal predicates" for predicates that are used in defining the goal. We provide a discussion about the necessity of predicate invention despite having these predicates in Appendix A.4.

Actions as Hybrid Controllers The environment's action space A is defined by a finite set of *controllers* $\mathcal{C}$. Each controller $C((\lambda_1, \ldots, \lambda_v), \Theta) \in \mathcal{C}$ has a semantically-meaningful name, as well as optional discrete typed parameters $(\lambda_1, \ldots, \lambda_v)$ and continuous parameters Θ. For instance, a controller `Pick` for picking up an object might have one discrete parameter of type `item` and a Θ that is a placeholder for a specific grasp. An *action* $a \in \mathcal{A}$ in a task is a controller $C \in \mathcal{C}$ with discrete and continuous arguments fully specified: $a = C((o_1, \ldots o_v), \theta)$.

Final Demonstration Structure We provide the agent with a demonstration set $\mathcal{D}$. Each demonstration consists of: (1) a training task $T \sim \mathcal{T}$, (2) a near-optimal *plan* that solves the task, and (3) the *state sequence* of that plan with values for all initial predicate ground atoms.

Recall that a task is a tuple $\langle \mathcal{O}, x_0, g \rangle$. A *plan* $\pi = (a_1, \ldots, a_m)$ is a sequence of actions $a \in \mathcal{A}$ such that successive execution of the actions in the environment, starting from x_0, results in a final state x_m where g holds. A *state sequence* of a plan is the set of states $(x_1, \ldots, x_{m+1})$ encountered as that plan is executed in an environment. An example of this input is shown in Figure 2.

Given a handful (less than 15) of these demonstrations, the agent's objective is to invent additional predicates and a set of operator definitions in terms of the initial and learned predicates that enables the agent to plan and efficiently solve a set of held-out tesk tasks drawn from $\mathcal{T}$. Test tasks involve more objects, different object configurations, novel compositions of goals, and longer-horizons than the training tasks solved in the demonstrations.

[1]It would have this value if its classifier was accurate. We discuss what happens if this classifier is inaccurate in Section 4.3.

3.2 Operators and Planning

Predicates Ψ induce an abstract state space S_Ψ of the task's underlying continuous state space $\mathcal{X}$. Operators help specify an abstract transition model over S_Ψ.

More specifically, given a particular state x_t, as well as predicates Ψ, we can compute the corresponding *abstract state* (denoted by s_t) by selecting all the ground atoms that are "true" in x_t. Formally, $s_t = \textsc{Abstract}(x_t, \Psi) \triangleq \{\underline{\psi} : c_{\underline{\psi}}(x_t) = \text{true}, \forall \psi \in \Psi\}$. Then, each operator ω is a tuple $\omega = \langle \overline{v}, P, E^+, E^-, \mathcal{C} \rangle$. Here, $\overline{v}$ are typed variables representing the operator's arguments, P, E^+ and E^- are sets of predicates representing operator preconditions, add effects and delete effects respectively, and $\mathcal{C}$ is a controller associated with the operator. Note importantly that the discrete parameters of $\mathcal{C}$ must be a subset of $\overline{v}$. Specifying objects of the correct type as arguments induces a *ground operator* $\underline{\omega} = \langle \underline{P}, \underline{E^+}, \underline{E^-}, \underline{\mathcal{C}} \rangle$. Ground operators define a (partial) abstract transition function: given $\underline{\omega} = \langle \underline{P}, \underline{E^+}, \underline{E^-}, \underline{\mathcal{C}} \rangle$, if $\underline{P} \subseteq s$, then the successor abstract state s' is $(s \setminus \underline{E^-}) \cup \underline{E^+}$ where $\setminus$ is set difference.

A ground operator specifies a corresponding ground controller $\underline{\mathcal{C}}$. To extract an action a from this ground controller, we must specify values for the parameters Θ associated with controller $\mathcal{C}$. Following previous work (Silver et al. 2023; Kumar et al. 2023; Chitnis et al. 2022), we leverage continuous parameter *samplers* to do this.

Given these components, we can perform planning to solve test tasks. We assume access to a planner that takes a set of predicates, operators, and samplers, as well as a task $T = (\mathcal{O}, x_0, g)$ associated with an environment, and outputs a plan that achieves the goal at the symbolic level, if such a plan exists. We implement such a planner via off-the-shelf classical planning algorithms (Helmert 2006), with details in Appendix A.1, though we note that our predicate invention framework is compatible with other planning strategies.

3.3 Predicate Invention Problem

Given demonstrations $\mathcal{D}$ structured as described in Section 3.1, our objective is to *learn* additional predicates (Ψ), operators (Ω), and samplers (Σ) that enable aggressive generalization to novel test tasks from the task distribution $\mathcal{T}$ by calling a planner with these learned components. We directly leverage an existing approach to learn neural network samplers from our demonstration data (Appendix A.3). We are mainly interested in learning predicates and operators.

Predicate Invention via Hill-Climbing We build on the framework introduced by Silver et al. (2023) for predicate invention. Their method first generates an initial pool of candidate predicates Ψ_{pool} from a grammar and then selects the subset of predicates from this pool that maximizes an objective function designed to measure planning efficiency over the demonstrations, where the selection is done via a hill-climbing optimization. Predicates subselected from the pool Ψ_{pool} are added to the initial set Ψ_{init} to form a final set of predicates Ψ. After, operators are learned in terms of the predicates Ψ. Intuitively, optimizing their particular objective fuction yields the predicate set (and corresponding operator set) that produces plans that most closely align with the demonstrations on the training tasks.

Importantly, this previous work makes two strong assumptions that do not hold in our setting: (1) the state space $\mathcal{X}$ is composed entirely of continuous object features (i.e., $\mathcal{X}$ only consists of x^{feat}), and (2) learned predicate classifiers are not noisy. We relax these assumptions and enable predicates that operate over images.

4 Inventing Predicates from Pixels

Our main interest is to invent predicates *from pixels* (i.e., predicates that operate directly over one or more images in our image-based state x^{img}). We wish to do this from a handful of structured demonstrations described in Section 3.1. These demonstrations do not possess any direct supervision for additional predicates (i.e., the demonstrations provide no indication of the number or structure of any predicates we should invent). The hill-climbing framework for predicate invention (Section 3.3) is able to learn predicates in such a setting, but requires a pool of initial candidate predicates to subselect from. To define such an initial pool, we must specify two things: *what* the predicates should be (i.e., what visual concepts are relevant for decision-making), and *how* should they be implemented (i.e., for every predicate ψ in the pool, how can we define a corresponding classifier function c_ψ).

We propose to use a VLM to both *generate* an initial pool of predicates that might be relevant to decision-making, and *implement* the classifier function for each predicate in the pool. Recent work has demonstrated that VLMs are able to answer a variety of common-sense natural language questions given one or more images (Antol et al. 2015; Kim et al. 2024; Majumdar et al. 2024). The core intuition motivating our approach is that predicate generation and implementation correspond to two different types of questions. Predicate generation can be accomplished by giving the VLM access to our demonstrations, and asking "what visual concepts are changing between false and true after each action execution?". Predicate classifier implementation can be accomplished by giving the VLM access to a particular image and asking it "is this particular ground predicate true or false in the current state?". In the remainder of this section, we explain how exactly we implement this procedure and how we modify and leverage the hill-climbing predicate invention framework described in Section 3.3 to yield a compact subset of predicates optimized for planning.[2]

4.1 Implementing Visual Predicates with a VLM

Recall that each state $x \in \mathcal{X}$ is composed of two parts: a feature-based state x^{feat} and an image-based state x^{img}. Visual predicates operate exclusively over x^{img}. Specifically, a visual predicate is a tuple of a name, a sequence of m types, and a classifier function: $\psi_{\text{vis}} = \langle \text{name}, (\lambda_1, \dots, \lambda_m), c_\psi \rangle$. Here, the classifier function returns a boolean given an image-based state and object arguments $c_\psi : \mathcal{X}^{\text{img}} \times \mathcal{O}^m \to \{\text{true}, \text{false}\}$.

[2]See Figure 8 for a detailed overview of our entire pipeline.

We propose to implement the classifier function c_ψ by querying a VLM. Specifically, we create a prompt string txt_ψ with the predicate's name and arguments to be filled in by the predicate's typed object arguments $(\lambda_1, \dots, \lambda_m)$ (e.g. `Cooked(?obj1)`).[3] When the predicate is ground, we substitute the specific objects used to ground it into the prompt (e.g. `Cooked(patty1)`). We implement the classifier function by simply passing the prompt with this ground atom string into a VLM with some additional instruction, asking it to output either "true", "false", or "unknown" (which we take to be equivalent to "false").

While this idea is straightforward, we found that several additional modifications to the text prompt txt_ψ and images x^{img} were critical to obtain accurate labels for ground atoms from the VLM. We discuss these in detail and provide examples of our prompts in Appendix A.5.

4.2 Proposing an Initial Pool of Visual Predicates

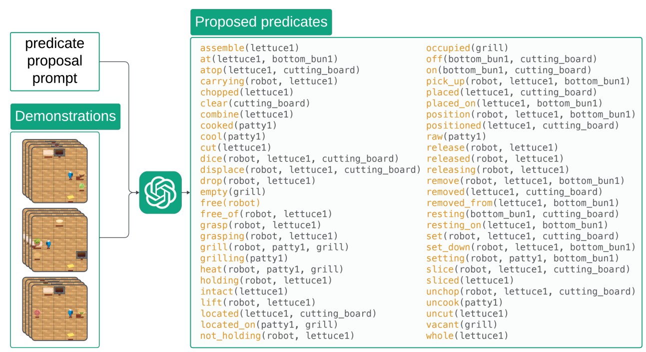

Figure 3: **VLM predicate proposal example.** We prompt (Appendix A.5) a VLM to propose a large pool of predicates that describe important concepts in the given demonstrations. Technically, the VLM outputs ground atoms as strings, and we parse the predicate (visualized in orange) from each ground atom. Shown here are proposals generated from demonstrations in the task distribution "Combo Burger" within the "Burger" environment (Section 5). The VLM proposes 563 predicates and we filter these into 136 "valid" predicates; the figure shows 56 of these.

We generate a pool candidate of visual predicates by prompting a VLM on each demonstration $d \in \mathcal{D}$. For a demonstration $d \in \mathcal{D}$ of length $k = len(d)$, we extract the image-based state at each timestep $x_t^{\text{img}}, t \in [0, k-1]$ and add a text heading to each image corresponding to which timestep t in the demonstration it belongs to. We then pass all these images in sequence, along with the actions executed in between them $(a_0, a_1, \dots, a_{k_d - 1})$ directly to a VLM. We prompt the VLM to output a set of proposals for ground atoms based on these inputs.

Given a set of ground atoms proposed for each demonstration, we then parse this set to discover atoms that are syntactically incorrect. In particular, we remove atoms that include object names that are not part of the demonstration d (for instance, the VLM might propose an atom

[3]We create the prompt string txt_ψ with the predicate's name but other choices are possible. We could ask the VLM, "Is `patty1` cooked?" instead of "What is the value of `Cooked(patty1)?`"

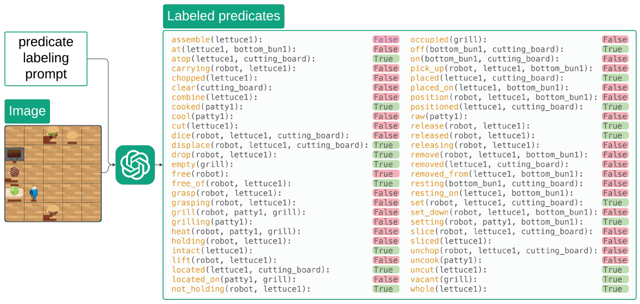

Figure 4: **VLM predicate labeling example.** We prompt (Appendix A.5) a VLM to determine the truth values of ground atoms (each represented as a string) in the given image, labeling each as either true, false, or unknown. Shown here are a subset of the labels generated for a particular state from a task in the task distribution"Combo Burger" within the "Burger" environment (Section 5).

`ontop(patty1, pot1)`, even though there is no object named `pot1`). We then *lift* each of these atoms into a visual predicate by creating typed variables for each of the objects identified in each proposed ground atom (for instance, an atom `cooked(patty1)` would be lifted to `cooked(?p: patty)`). Finally, we remove any duplicate lifted atoms and add only unique predicates to our pool. We obtain a final pool of predicates $\Psi^{\text{vis}}_{\text{pool}}$ by running this procedure on each demonstration.

4.3 Predicate Invention via Subset Selection From Noisy Data

Given the procedure described in Section 4.2, we can generate an initial pool of visual predicates $\Psi^{\text{vis}}_{\text{pool}}$. We combine these with feature-based predicates (generated via a programmatic grammar as described in Silver et al. (2023)) to create an overall pool $\Psi_{\text{pool}} \leftarrow \Psi^{\text{vis}}_{\text{pool}} \cup \Psi^{\text{feat}}_{\text{pool}}$. We wish to run hill-climbing optimization as described in Section 3.3 to subselect a small set of relevant predicates that are optimized for efficient and effective planning over our training tasks.

Unfortunately, we cannot directly apply the particular hill-climbing optimization from previous work because our visual predicates are noisy. Visual predicates might not be consistent across states in the demonstrations due to occasional hallucination or mislabeling on the part of the VLM. This noise creates outliers in the abstract state space transition data and causes the operator learning algorithm to overfit and create a large number of unnecessary operators to model these outlier transitions. These extra operators make planning inefficient by increasing the branching factor of the planner's search algorithm. They also cause the hill-climbing predicate subselection – which interally performs operator learning on every subset of predicates it scores – to overfit and select a large number of unnecessary predicates. These overfit predicates can appear in the definitions of all operators – not just the operators created to model the outlier transition data – because operators are learned to be chainable (add effects of an earlier operator satisfy the preconditions of a later operator). The overfit predicates and overfit operators fail to generalize outside of the training distribution. We combat this problem by modifying the operator learning approach to be more tolerant to noise [4], and by regularizing the hill-climbing optimization itself via early stopping to discourage overfitting.

5 Experiments

Our experiments are designed to answer the following questions.

Q1. How well does `pix2pred` generalize to novel, more complex goals when compared to an imitation approach that doesn't use a planner?

Q2. How critical is it to perform explicit score-based optimization for subselection of predicates?

Q3. How critical is it to invent both visual and feature-based predicates?

Environments. We now describe our experimental environments and tasks, with details in Appendix A.6. We implement 4 task distributions across 2 environments. Each environment was introduced by previous work and lightly adapted for our setting.

- *Kitchen:* A simulated robotic environment featuring a robotic arm positioned in front of a kitchen that includes a stove, a microwave, and controllable lights (Gupta et al. 2019). The task is to boil water in a kettle by turning on a specific stove burner and pushing the kettle onto it. The task has a short horizon, requiring only two high-level skills in sequence to achieve the goal. Demonstrations show the robot solving the task using a particular burner from various kettle initial positions, and at test time the robot must use a different burner on the same stove.
- *Burger:* A simulated 2D grid world environment (based on the Robotouille environment introduced by (Wang et al. 2023)) featuring a robot that must prepare and assemble burgers from various components. Preparation includes chopping lettuce on a cutting board and cooking patties on a grill. We experiment with three task distributions within this environment, with each distribution featuring its own set of training and testing tasks: (1) *Bigger Burger* involves producing a burger with extra patties after being shown how to create a burger with a single patty, (2) *More Burger Stacks* involves producing multiple burgers after being shown how to create a single burger, and (3) *Combo Burger* involves producing burgers with multiple ingredients after being shown how to create burgers with a single ingredient. The tasks have a long horizon. For example, test tasks in *Bigger Burger*, *More Burger Stacks*, and *Combo Burger* can require a sequence of 16, 29, and 30 controllers to achieve the goal, respectively.

Approaches. We evaluate our method against several ablations and baselines from the literature, including those that attempt to solve test tasks directly without learning any abstractions.

- *Ours*: Inventing visual and feature-based predicates with `pix2pred`, learning operators in terms of these predicates, and planning with these operators at test time.

[4] Modifications are described in detail in Appendix A.2

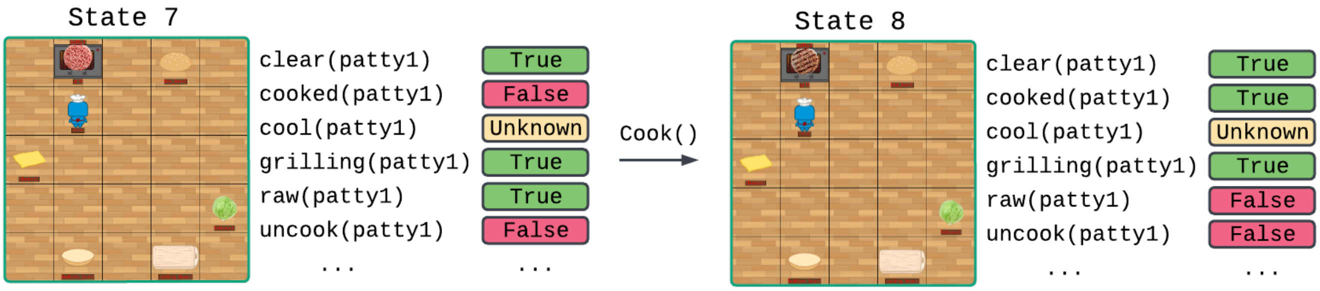

Figure 5: **Labeling of ground atoms with visual predicates involving the patty object type in Burger**. This figure depicts the visual state before and after the *cook* skill in demonstration #0 for the task distribution "Combo Burger". It also shows the labels assigned by the VLM to ground atoms with visual predicates involving the patty object in these states. The VLM proposes a wide variety of predicates involving the patty: `clear`, `cooked`, `cool`, `grilling`, `raw`, and `uncook`. Of these, `cooked(?p:patty)` is most relevant to achieving the goal while also being accurately labeled. Our approach automatically selects this predicate from the pool. See Figure 9 in the appendix for an analogous situation with the *chop* skill.

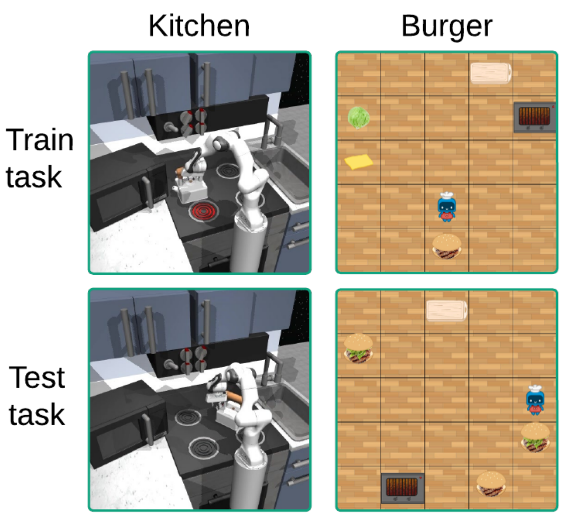

Figure 6: **Environments.** Top row: train task examples. Bottom row: test task examples. See Section 5 for descriptions of each environment.

- *VLM subselect*: An ablation of `pix2pred` where we have the VLM directly subselect a compact set of visual predicates instead of delegating subselection to the hill-climbing optimization. We still learn feature-based predicates via hill-climbing in this case.
- *No subselect*: An ablation of `pix2pred` where we perform no hill-climbing, but rather select all predicates in Ψ_{pool}.
- *No invent*: An ablation of `pix2pred` where we do not propose or subselect any additional predicates beyond the provided ones in Ψ_{init}.
- *No visual*: An ablation of `pix2pred` where we only invent feature-based predicates.
- *No feature*: An ablation of `pix2pred` where we only invent visual predicates.
- *VLM feat. pred*: A baseline inspired by recent work from Han et al. (2024). Here, a VLM writes predicates as functions over the feature-based state instead of being able to invent its own novel visual concepts.
- *ViLa*: A baseline that does not learn abstractions but uses a VLM to plan on test tasks directly (Hu et al. 2023a). We test two variants of this: *ViLa-pure* which is the original approach as implemented by Hu et al. (2023a) adapted to our setting, and *ViLa-fewshot*, which additionally provides the demonstrations as part of the few-shot prompt to give the VLM some context of what kinds of plans it should return for testing tasks.

Experimental Setup. Each seed varies the number and locations of objects in the train and test tasks, and we average results over 5 random seeds for each task distribution. For each seed, we sample 10 test tasks. We use GPT-4o (OpenAI 2023) as the VLM for all approaches. For our tasks *Burger*, as mentioned in section 4.3, we early stop the predicate subselection optimization when the hill-climbing objective value goes below 2000 (a value we found by running `pix2pred` on a validation set). We evaluate each approach by providing the test task and allowing the approach to make and execute exactly one single plan.

Results and Analysis. Figure 7 shows our success rate plots. `pix2pred` outperforms previous approaches on 3 out of 4 of the task distributions by a wide margin. In particular, `pix2pred` seems to generalize to more complex tasks much better than using few-shot learning with a VLM (i.e., ViLa) (**Q1**). In the short-horizon kitchen task that requires 2 actions, we find that ViLa with demonstrations provided performs comparably to `pix2pred`. However, the performance of this baseline drops-off significantly in the longer-horizon burger tasks that require more than 15 actions. In particular, we found that ViLa seems to solve testing tasks by pattern-matching aspects of the training demonstrations and is unable to solve test tasks that are unlike any of the demonstrations. For instance, the agent never starts out holding an object in any of the training demonstrations, but it does so in about 50% of the testing task instances in the Burger environment. We observed that ViLa does not correctly put down the held object before attempting to pick up other objects, whereas `pix2pred` learns a predicate corresponding to the agent's hands being empty and correctly selects this as a precondition for picking up any other object. We find that `pix2pred` only fails due to noise in VLM labeling: the VLM infrequently labels the initial state incorrectly such that a critical precondition is false, which prevents our plan-

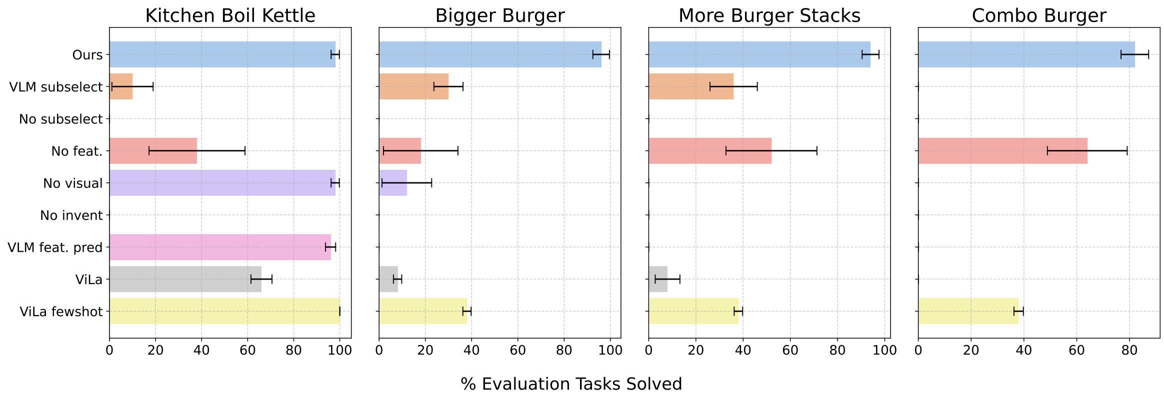

Figure 7: **`pix2pred` versus baselines**. Percent of evaluation tasks solved per task across both simulated environments. All results are averaged over 5 seeds. Black bars denote standard deviations.

ner from making a valid plan.

We also find that explicit optimization via hill-climbing leads to much better performance on testing tasks when compared to simply selecting predicates using a VLM (**Q2**). This can be seen from the fact that the VLM subselect baseline performs significantly worse than our main approach across all 4 task distributions. Qualitatively, we notice that this baseline tends to select far more predicates than necessary: in the burger domain, it selects predicates `Cooked(?p1: patty)` and `Chopped(?l1: lettuce)` (necessary for decision-making), but also extraneous predicates like `Raw(?p1: patty)`, `Available(?p1: patty)`, and `Whole(?l1: lettuce)` that the VLM labels quite inconsistently. This results in a complex set of operators that overfit strongly to the training tasks and do not generalize well. Additionally, we see that the "no subselect" baseline fails completely at all tasks, indicating that some subselection is critical; learning operators in terms of the more than 500 predicates in the proposal pool leads to extremely overfit operators.

We find that both feature-based and visual predicates are important in most of our tasks (**Q3**). The "no invent" baseline fails completely in all the tasks because the initial predicates are insufficient. The "no visual." baseline performs somewhat well in the Kitchen task, but fails in the Burger tasks where visual predicates, specifically predicates corresponding to `Cooked` and `Chopped`, are necessary, since the feature-based state does not contain sufficient information to classify these concepts. Interestingly, the predicate corresponding to whether the robot's hand is empty can be classified more accurately in Burger tasks if features of the robot's hand are used vs. attempting to recognize that concept visually. This leads to our approach doing better than the "no feat." baseline in the Burger tasks, since our approach is not only limited to purely visual predicates and can invent whichever predicate has higher classification accuracy.

6 Limitations and Conclusion

We proposed `pix2pred`, a method for inventing symbolic predicates that operate over raw images. We found that our approach, from just a handful of demonstrations (3-12), is able to invent semantically meaningful predicates that afford efficient planning and generalization to novel tasks across a range of domains and problems.

There are several noteworthy limitations of our present work. Firstly, VLM hallucinations adversely impact predicate proposal and atom labelling – especially with small amounts of demonstration data – which can inhibit our approach's ability to learn useful predicates and operators. Secondly, the hill-climbing framework we use for predicate invention can be extremely slow, especially as the number of demonstrations and size of the initial predicate pool grows. Finally, our approach assumes input demonstrations are segmented in terms of a provided set of parameterized controllers with names that correspond to their function.

One interesting future direction is to expand the structure of predicates a VLM is allowed to produce, perhaps by allowing it to write arbitrarily-long text descriptions or even code to define new predicates (Liang et al. 2024). Another is to improve the underlying hill-climbing optimization procedure to be significantly more efficient and tolerant to noise, enabling `pix2pred` to scale to much larger domains that require potentially many tens or hundreds of predicates. Eventually we'd like to extend `pix2pred` to automatically segment demonstrations and learn controllers so that we can learn to plan directly from the simplest low-level demonstration data.

References

Abel, D.; Hershkowitz, D.; and Littman, M. 2016. Near optimal behavior via approximate state abstraction. In *International Conference on Machine Learning (ICML)*.

Ahmetoglu, A.; Seker, M. Y.; Piater, J.; Oztop, E.; and Ugur, E. 2022. Deepsym: Deep symbol generation and rule learning for planning from unsupervised robot interaction. *Journal of Artificial Intelligence Research (JAIR)*.

Antol, S.; Agrawal, A.; Lu, J.; Mitchell, M.; Batra, D.; Zitnick, C. L.; and Parikh, D. 2015. Vqa: Visual question answering. In *IEEE International Conference on Computer Vision (ICCV)*.

Asai, M.; and Fukunaga, A. 2018. Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary. In *AAAI Conference on Artificial Intelligence (AAAI)*.

Bertsekas, D. P.; Castanon, D. A.; et al. 1988. Adaptive aggregation methods for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*.

Bonet, B.; and Geffner, H. 2019. Learning first-order symbolic representations for planning from the structure of the state space. *arXiv preprint arXiv:1909.05546*.

Chi, C.; Feng, S.; Du, Y.; Xu, Z.; Cousineau, E.; Burchfiel, B.; and Song, S. 2023. Diffusion Policy: Visuomotor Policy Learning via Action Diffusion. In *Robotics: Science and Systems (RSS)*.

Chitnis, R.; Silver, T.; Tenenbaum, J.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. GLIB: Efficient Exploration for Relational Model-Based Reinforcement Learning via Goal-Literal Babbling. In *AAAI*.

Chitnis, R.; Silver, T.; Tenenbaum, J. B.; Lozano-Pérez, T.; and Kaelbling, L. P. 2022. Learning Neuro-Symbolic Relational Transition Models for Bilevel Planning. In *AAAI CLeaR Workshop*.

Curtis, A.; Kumar, N.; Cao, J.; Lozano-Pérez, T.; and Kaelbling, L. P. 2024. Trust the PRoC3S: Solving Long-Horizon Robotics Problems with LLMs and Constraint Satisfaction. In *Conference on Robot Learning (CoRL)*.

Curtis, A.; Silver, T.; Tenenbaum, J. B.; Lozano-Perez, T.; and Kaelbling, L. P. 2021. Discovering State and Action Abstractions for Generalized Task and Motion Planning. *arXiv preprint arXiv:2109.11082*.

Duan, J.; Yuan, W.; Pumacay, W.; Wang, Y. R.; Ehsani, K.; Fox, D.; and Krishna, R. 2024. Manipulate-Anything: Automating Real-World Robots using Vision-Language Models. arXiv:2406.18915.

Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Yang, A.; Fan, A.; Goyal, A.; Hartshorn, A.; Yang, A.; Mitra, A.; Sravankumar, A.; Korenev, A.; Hinsvark, A.; Rao, A.; Zhang, A.; Rodriguez, A.; Gregerson, A.; Spataru, A.; Roziere, B.; Biron, B.; Tang, B.; Chern, B.; Caucheteux, C.; Nayak, C.; Bi, C.; Marra, C.; McConnell, C.; Keller, C.; Touret, C.; Wu, C.; Wong, C.; Ferrer, C. C.; Nikolaidis, C.; Allonsius, D.; Song, D.; Pintz, D.; Livshits, D.; Esiobu, D.; Choudhary, D.; Mahajan, D.; Garcia-Olano, D.; Perino, D.; Hupkes, D.; Lakomkin, E.; AlBadawy, E.; Lobanova, E.; Dinan, E.; Smith, E. M.; Radenovic, F.; Zhang, F.; Synnaeve, G.; Lee, G.; Anderson, G. L.; Nail, G.; Mialon, G.; Pang, G.; Cucurell, G.; Nguyen, H.; Korevaar, H.; Xu, H.; Touvron, H.; Zarov, I.; Ibarra, I. A.; Kloumann, I.; Misra, I.; Evtimov, I.; Copet, J.; Lee, J.; Geffert, J.; Vranes, J.; Park, J.; Mahadeokar, J.; Shah, J.; van der Linde, J.; Billock, J.; Hong, J.; Lee, J.; Fu, J.; Chi, J.; Huang, J.; Liu, J.; Wang, J.; Yu, J.; Bitton, J.; Spisak, J.; Park, J.; Rocca, J.; Johnstun, J.; Saxe, J.; Jia, J.; Alwala, K. V.; Upasani, K.; Plawiak, K.; Li, K.; Heafield, K.; Stone, K.; El-Arini, K.; Iyer, K.; Malik, K.; Chiu, K.; Bhalla, K.; Rantala-Yeary, L.; van der Maaten, L.; Chen, L.; Tan, L.; Jenkins, L.; Martin, L.; Madaan, L.; Malo, L.; Blecher, L.; Landzaat, L.; de Oliveira, L.; Muzzi, M.; Pasupuleti, M.; Singh, M.; Paluri, M.; Kardas, M.; Oldham, M.; Rita, M.; Pavlova, M.; Kambadur, M.; Lewis, M.; Si, M.; Singh, M. K.; Hassan, M.; Goyal, N.; Torabi, N.; Bashlykov, N.; Bogoychev, N.; Chatterji, N.; Duchenne, O.; Çelebi, O.; Alrassy, P.; Zhang, P.; Li, P.; Vasic, P.; Weng, P.; Bhargava, P.; Dubal, P.; Krishnan, P.; Koura, P. S.; Xu, P.; He, Q.; Dong, Q.; Srinivasan, R.; Ganapathy, R.; Calderer, R.; Cabral, R. S.; Stojnic, R.; Raileanu, R.; Girdhar, R.; Patel, R.; Sauvestre, R.; Polidoro, R.; Sumbaly, R.; Taylor, R.; Silva, R.; Hou, R.; Wang, R.; Hosseini, S.; Chennabasappa, S.; Singh, S.; Bell, S.; Kim, S. S.; Edunov, S.; Nie, S.; Narang, S.; Raparthy, S.; Shen, S.; Wan, S.; Bhosale, S.; Zhang, S.; Vandenhende, S.; Batra, S.; Whitman, S.; Sootla, S.; Collot, S.; Gururangan, S.; Borodinsky, S.; Herman, T.; Fowler, T.; Sheasha, T.; Georgiou, T.; Scialom, T.; Speckbacher, T.; Mihaylov, T.; Xiao, T.; Karn, U.; Goswami, V.; Gupta, V.; Ramanathan, V.; Kerkez, V.; Gonguet, V.; Do, V.; Vogeti, V.; Petrovic, V.; Chu, W.; Xiong, W.; Fu, W.; Meers, W.; Martinet, X.; Wang, X.; Tan, X. E.; Xie, X.; Jia, X.; Wang, X.; Goldschlag, Y.; Gaur, Y.; Babaei, Y.; Wen, Y.; Song, Y.; Zhang, Y.; Li, Y.; Mao, Y.; Coudert, Z. D.; Yan, Z.; Chen, Z.; Papakipos, Z.; Singh, A.; Grattafiori, A.; Jain, A.; Kelsey, A.; Shajnfeld, A.; Gangidi, A.; Victoria, A.; Goldstand, A.; Menon, A.; Sharma, A.; Boesenberg, A.; Vaughan, A.; Baevski, A.; Feinstein, A.; Kallet, A.; Sangani, A.; Yunus, A.; Lupu, A.; Alvarado, A.; Caples, A.; Gu, A.; Ho, A.; Poulton, A.; Ryan, A.; Ramchandani, A.; Franco, A.; Saraf, A.; Chowdhury, A.; Gabriel, A.; Bharambe, A.; Eisenman, A.; Yazdan, A.; James, B.; Maurer, B.; Leonhardi, B.; Huang, B.; Loyd, B.; Paola, B. D.; Paranjape, B.; Liu, B.; Wu, B.; Ni, B.; Hancock, B.; Wasti, B.; Spence, B.; Stojkovic, B.; Gamido, B.; Montalvo, B.; Parker, C.; Burton, C.; Mejia, C.; Wang, C.; Kim, C.; Zhou, C.; Hu, C.; Chu, C.-H.; Cai, C.; Tindal, C.; Feichtenhofer, C.; Civin, D.; Beaty, D.; Kreymer, D.; Li, D.; Wyatt, D.; Adkins, D.; Xu, D.; Testuggine, D.; David, D.; Parikh, D.; Liskovich, D.; Foss, D.; Wang, D.; Le, D.; Holland, D.; Dowling, E.; Jamil, E.; Montgomery, E.; Presani, E.; Hahn, E.; Wood, E.; Brinkman, E.; Arcaute, E.; Dunbar, E.; Smothers, E.; Sun, F.; Kreuk, F.; Tian, F.; Ozgenel, F.; Caggioni, F.; Guzmán, F.; Kanayet, F.; Seide, F.; Florez, G. M.; Schwarz, G.; Badeer, G.; Swee, G.; Halpern, G.; Thattai, G.; Herman, G.; Sizov, G.; Guangyi; Zhang; Lakshminarayanan, G.; Shojanazeri, H.; Zou, H.; Wang, H.; Zha, H.; Habeeb, H.; Rudolph, H.; Suk, H.; Aspegren, H.; Goldman, H.; Damlaj, I.; Molybog, I.; Tufanov, I.; Veliche, I.-E.; Gat, I.; Weissman, J.; Geboski, J.; Kohli, J.;

Asher, J.; Gaya, J.-B.; Marcus, J.; Tang, J.; Chan, J.; Zhen, J.; Reizenstein, J.; Teboul, J.; Zhong, J.; Jin, J.; Yang, J.; Cummings, J.; Carvill, J.; Shepard, J.; McPhie, J.; Torres, J.; Ginsburg, J.; Wang, J.; Wu, K.; U, K. H.; Saxena, K.; Prasad, K.; Khandelwal, K.; Zand, K.; Matosich, K.; Veeraraghavan, K.; Michelena, K.; Li, K.; Huang, K.; Chawla, K.; Lakhotia, K.; Huang, K.; Chen, L.; Garg, L.; A, L.; Silva, L.; Bell, L.; Zhang, L.; Guo, L.; Yu, L.; Moshkovich, L.; Wehrstedt, L.; Khabsa, M.; Avalani, M.; Bhatt, M.; Tsimpoukelli, M.; Mankus, M.; Hasson, M.; Lennie, M.; Reso, M.; Groshev, M.; Naumov, M.; Lathi, M.; Keneally, M.; Seltzer, M. L.; Valko, M.; Restrepo, M.; Patel, M.; Vyatskov, M.; Samvelyan, M.; Clark, M.; Macey, M.; Wang, M.; Hermoso, M. J.; Metanat, M.; Rastegari, M.; Bansal, M.; Santhanam, N.; Parks, N.; White, N.; Bawa, N.; Singhal, N.; Egebo, N.; Usunier, N.; Laptev, N. P.; Dong, N.; Zhang, N.; Cheng, N.; Chernoguz, O.; Hart, O.; Salpekar, O.; Kalinli, O.; Kent, P.; Parekh, P.; Saab, P.; Balaji, P.; Rittner, P.; Bontrager, P.; Roux, P.; Dollar, P.; Zvyagina, P.; Ratanchandani, P.; Yuvraj, P.; Liang, Q.; Alao, R.; Rodriguez, R.; Ayub, R.; Murthy, R.; Nayani, R.; Mitra, R.; Li, R.; Hogan, R.; Battey, R.; Wang, R.; Maheswari, R.; Howes, R.; Rinott, R.; Bondu, S. J.; Datta, S.; Chugh, S.; Hunt, S.; Dhillon, S.; Sidorov, S.; Pan, S.; Verma, S.; Yamamoto, S.; Ramaswamy, S.; Lindsay, S.; Lindsay, S.; Feng, S.; Lin, S.; Zha, S. C.; Shankar, S.; Zhang, S.; Zhang, S.; Wang, S.; Agarwal, S.; Sajuyigbe, S.; Chintala, S.; Max, S.; Chen, S.; Kehoe, S.; Satterfield, S.; Govindaprasad, S.; Gupta, S.; Cho, S.; Virk, S.; Subramanian, S.; Choudhury, S.; Goldman, S.; Remez, T.; Glaser, T.; Best, T.; Kohler, T.; Robinson, T.; Li, T.; Zhang, T.; Matthews, T.; Chou, T.; Shaked, T.; Vontimitta, V.; Ajayi, V.; Montanez, V.; Mohan, V.; Kumar, V. S.; Mangla, V.; Albiero, V.; Ionescu, V.; Poenaru, V.; Mihailescu, V. T.; Ivanov, V.; Li, W.; Wang, W.; Jiang, W.; Bouaziz, W.; Constable, W.; Tang, X.; Wang, X.; Wu, X.; Wang, X.; Xia, X.; Wu, X.; Gao, X.; Chen, Y.; Hu, Y.; Jia, Y.; Qi, Y.; Li, Y.; Zhang, Y.; Zhang, Y.; Adi, Y.; Nam, Y.; Yu; Wang; Hao, Y.; Qian, Y.; He, Y.; Rait, Z.; DeVito, Z.; Rosnbrick, Z.; Wen, Z.; Yang, Z.; and Zhao, Z. 2024. The Llama 3 Herd of Models. arXiv:2407.21783.

Fox, M.; and Long, D. 2003. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*.

Garrett, C. R.; Chitnis, R.; Holladay, R.; Kim, B.; Silver, T.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems*.

Gupta, A.; Kumar, V.; Lynch, C.; Levine, S.; and Hausman, K. 2019. Relay Policy Learning: Solving Long-Horizon Tasks via Imitation and Reinforcement Learning. arXiv:1910.11956.

Han, M.; Zhu, Y.; Zhu, S.-C.; Wu, Y. N.; and Zhu, Y. 2024. InterPreT: Interactive Predicate Learning from Language Feedback for Generalizable Task Planning. arXiv:2405.19758.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)*.

Hu, Y.; Lin, F.; Zhang, T.; Yi, L.; and Gao, Y. 2023a. Look Before You Leap: Unveiling the Power of GPT-4V in Robotic Vision-Language Planning. arXiv:2311.17842.

Hu, Y.; Xie, Q.; Jain, V.; Francis, J.; Patrikar, J.; Keetha, N.; Kim, S.; Xie, Y.; Zhang, T.; Fang, H.-S.; Zhao, S.; Omidshafiei, S.; Kim, D.-K.; akbar Agha-mohammadi, A.; Sycara, K.; Johnson-Roberson, M.; Batra, D.; Wang, X.; Scherer, S.; Wang, C.; Kira, Z.; Xia, F.; and Bisk, Y. 2023b. Toward General-Purpose Robots via Foundation Models: A Survey and Meta-Analysis.

Huang, W.; Wang, C.; Li, Y.; Zhang, R.; and Fei-Fei, L. 2024. ReKep: Spatio-Temporal Reasoning of Relational Keypoint Constraints for Robotic Manipulation. In *Conference on Robot Learning (CoRL)*.

Huang, W.; Wang, C.; Zhang, R.; Li, Y.; Wu, J.; and Fei-Fei, L. 2023. VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models. *arXiv preprint arXiv:2307.05973*.

Huang, W.; Xia, F.; Xiao, T.; Chan, H.; Liang, J.; Florence, P.; Zeng, A.; Tompson, J.; Mordatch, I.; Chebotar, Y.; et al. 2022. Inner Monologue: Embodied Reasoning through Planning with Language Models. *arXiv preprint arXiv:2207.05608*.

Ichter, B.; Brohan, A.; Chebotar, Y.; Finn, C.; Hausman, K.; Herzog, A.; Ho, D.; Ibarz, J.; Irpan, A.; Jang, E.; Julian, R.; Kalashnikov, D.; Levine, S.; Lu, Y.; Parada, C.; Rao, K.; Sermanet, P.; Toshev, A. T.; Vanhoucke, V.; Xia, F.; Xiao, T.; Xu, P.; Yan, M.; Brown, N.; Ahn, M.; Cortes, O.; Sievers, N.; Tan, C.; Xu, S.; Reyes, D.; Rettinghouse, J.; Quiambao, J.; Pastor, P.; Luu, L.; Lee, K.-H.; Kuang, Y.; Jesmonth, S.; Joshi, N. J.; Jeffrey, K.; Ruano, R. J.; Hsu, J.; Gopalakrishnan, K.; David, B.; Zeng, A.; and Fu, C. K. 2023. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. In *Conference on Robot Learning (CoRL)*.

James, S.; Rosman, B.; and Konidaris, G. 2022. Autonomous learning of object-centric abstractions for high-level planning. In *International Conference on Learning Representations (ICLR)*.

Jong, N. K.; and Stone, P. 2005. State Abstraction Discovery from Irrelevant State Variables. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Kambhampati, S.; Valmeekam, K.; Guan, L.; Stechly, K.; Verma, M.; Bhambri, S.; Saldyt, L.; and Murthy, A. 2024. LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. In *arXiv preprint*.

Kim, W.; Choi, C.; Lee, W.; and Rhee, W. 2024. An image grid can be worth a video: Zero-shot video question answering using a vlm. *arXiv preprint arXiv:2403.18406*.

Konidaris, G.; Kaelbling, L. P.; and Lozano-Pérez, T. 2018. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research (JAIR)*.

Kumar, N.; McClinton, W.; Chitnis, R.; Silver, T.; Lozano-Pérez, T.; and Kaelbling, L. P. 2023. Learning Efficient Abstract Planning Models that Choose What to Predict. In *Conference on Robot Learning (CoRL)*.

Kumar, N.; Ramos, F.; Fox, D.; and Garrett, C. R. 2024a. Open-World Task and Motion Planning via Vision-Language Model Inferred Constraints. In *CoRL Workshop on Language and Robot Learning: Language as an Interface*.

Kumar, N.; Silver, T.; McClinton, W.; Zhao, L.; Proulx, S.; Lozano-Pérez, T.; Kaelbling, L. P.; and Barry, J. 2024b. Practice Makes Perfect: Planning to Learn Skill Parameter Policies. In *Robotics: Science and Systems (RSS)*.

Liang, J.; Huang, W.; Xia, F.; Xu, P.; Hausman, K.; Ichter, B.; Florence, P.; and Zeng, A. 2022. Code as Policies: Language Model Programs for Embodied Control. In *arXiv preprint*.

Liang, Y.; Kumar, N.; Tang, H.; Weller, A.; Tenenbaum, J. B.; Silver, T.; Henriques, J. F.; and Ellis, K. 2024. VisualPredicator: Learning Abstract World Models with Neuro-Symbolic Predicates for Robot Planning. arXiv:2410.23156.

Liu, W.; Nie, N.; Zhang, R.; Mao, J.; and Wu, J. 2024. BLADE: Learning Compositional Behaviors from Demonstration and Language. In *Conference on Robot Learning (CoRL)*.

Majumdar, A.; Ajay, A.; Zhang, X.; Putta, P.; Yenamandra, S.; Henaff, M.; Silwal, S.; Mcvay, P.; Maksymets, O.; Arnaud, S.; et al. 2024. Openeqa: Embodied question answering in the era of foundation models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Mao, J.; Lozano-Pérez, T.; Tenenbaum, J. B.; and Kaelbling, L. P. 2022. PDSketch: Integrated Domain Programming, Learning, and Planning. In Oh, A. H.; Agarwal, A.; Belgrave, D.; and Cho, K., eds., *Advances in Neural Information Processing Systems (NeurIPS)*.

McDermott, D.; Ghallab, M.; Howe, A. E.; Knoblock, C. A.; Ram, A.; Veloso, M. M.; Weld, D. S.; and Wilkins, D. E. 1998. PDDL-the planning domain definition language.

OpenAI. 2023. GPT-4V(ision) System Card. Accessed: 2024-08-22.

Quartey, B.; Rosen, E.; Tellex, S.; and Konidaris, G. 2024. Verifiably Following Complex Robot Instructions with Foundation Models. arXiv:2402.11498.

Ren, T.; Liu, S.; Zeng, A.; Lin, J.; Li, K.; Cao, H.; Chen, J.; Huang, X.; Chen, Y.; Yan, F.; Zeng, Z.; Zhang, H.; Li, F.; Yang, J.; Li, H.; Jiang, Q.; and Zhang, L. 2024. Grounded SAM: Assembling Open-World Models for Diverse Visual Tasks. arXiv:2401.14159.

Silver, T.; Chitnis, R.; Kumar, N.; McClinton, W.; Lozano-Pérez, T.; Kaelbling, L.; and Tenenbaum, J. B. 2023. Predicate invention for bilevel planning. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*.

Silver, T.; Chitnis, R.; Tenenbaum, J.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Learning Symbolic Operators for Task and Motion Planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Silver, T.; Hariprasad, V.; Shuttleworth, R. S.; Kumar, N.; Lozano-Pérez, T.; and Kaelbling, L. P. 2022. PDDL planning with pretrained large language models. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*.

Singh, I.; Blukis, V.; Mousavian, A.; Goyal, A.; Xu, D.; Tremblay, J.; Fox, D.; Thomason, J.; and Garg, A. 2023. ProgPrompt: Generating Situated Robot Task Plans using Large Language Models. In *IEEE International Conference on Robotics and Automation (ICRA)*.

Team, G.; Anil, R.; Borgeaud, S.; Wu, Y.; Alayrac, J.-B.; Yu, J.; Soricut, R.; Schalkwyk, J.; Dai, A. M.; Hauth, A.; et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Ugur, E.; and Piater, J. 2015. Bottom-up learning of object categories, action effects and logical rules: From continuous manipulative exploration to symbolic planning. In *IEEE International Conference on Robotics and Automation (ICRA)*.

Valmeekam, K.; Olmo, A.; Sreedharan, S.; and Kambhampati, S. 2022. Large language models still can't plan (a benchmark for LLMs on planning and reasoning about change). In *NeurIPS 2022 Foundation Models for Decision Making Workshop*.

Wang, H.; Gonzalez-Pumariega, G.; Sharma, Y.; and Choudhury, S. 2023. Demo2Code: From Summarizing Demonstrations to Synthesizing Code via Extended Chain-of-Thought. arXiv:2305.16744.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q. V.; Zhou, D.; et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems (NeurIPS)*.

Yang, J.; Zhang, H.; Li, F.; Zou, X.; Li, C.; and Gao, J. 2023. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v. *arXiv preprint arXiv:2310.11441*.

Yang, Z.; Garrett, C.; Fox, D.; Lozano-Pérez, T.; and Kaelbling, L. P. 2024. Guiding Long-Horizon Task and Motion Planning with Vision Language Models. arXiv:2410.02193.

Zhao, T. Z.; Kumar, V.; Levine, S.; and Finn, C. 2023. Learning fine-grained bimanual manipulation with low-cost hardware. In *Robotics: Science and Systems (RSS)*.

A Appendix

A.1 Planner Implementation Details

Algorithm 1: Planning and Execution pseudocode

Input: Task $\langle \mathcal{O}, x_0, g \rangle$, predicates Ψ, operators Ω, samplers Σ.
$s_0 \leftarrow \text{ABSTRACT}(x_0, \Psi)$
Call classical planner to generate an abstract plan: $\text{plan}^{\text{abs}} = \underline{\omega_0}, \underline{\omega_1}, ..., \underline{\omega_m} \leftarrow \text{Planner}(s_0, \mathcal{O}, \Psi, \Omega, g)$
Extract controller sequence with discrete args filled in: skeleton $= \underline{\mathcal{C}_0}, \underline{\mathcal{C}_1}, ..., \underline{\mathcal{C}_m}$
For $i = 0, \ldots, m$:
  Sample $\theta \sim \sigma_{\omega_i}$, where θ represents the continuous parameters of $\mathcal{C}_i$ and σ_{ω_i} is the sampler associated with ω_i
  Use θ to fully ground controller $\underline{\mathcal{C}_i}$ into an action a_i.
  execute action a_i and obtain the following state $x_{i+1} \leftarrow f(x_i, a_i)$

Algorithm 1 shows the pseudocode for the planning and execution strategy we implement following recent work (Kumar et al. 2024b). Given an initial state x_0 for a task, we simply evaluate the classifiers of the given predicates Ψ to convert the state into an abstract state s_0. We then call a classical planner with the task object set $\mathcal{O}$ and operators Ω to compute an abstract plan that achieves the goal (if one exists). We extract the ground controller sequence for this abstract plan, and then simply greedily execute each controller sequentially by calling the sampler associated with each operator.

We note that more sophisticated planning strategies are possible. In particular, provided a photorealistic simulation environment for the transition function f, we could leverage many task and motion planning (Garrett et al. 2021), such as bilevel planning (Silver et al. 2023; Kumar et al. 2023; Chitnis et al. 2022).

A.2 Operator Learning

We adapt the "cluster and intersect" operator learning strategy from previous work (Chitnis et al. 2022) to handle noise in predicate values inherent to our setting. Specifically, we learn a set of operators Ω from our demonstrations $\mathcal{D}$ and predicates Ψ in four steps. Of these, the first three steps are largely taken directly from previous work: we introduce a modification to the third step, as well as the final step to handle noise.

1. *Partitioning*: Each demonstration can be expressed as a sequence of transitions $\{(x, a, x')\}$, with $x, x' \in \mathcal{X}$ and $a \in \mathcal{A}$. Recall that each action a is a controller with particular discrete and continuous arguments specified; let $\underline{\mathcal{C}}$ denote the corresponding controller with the same discrete object argument values, but continuous parameter values left unspecified. First, we use Ψ to ABSTRACT all states x, x' in the demonstrations $\mathcal{D}$, creating a dataset of transitions $\{(s, a, s')\}$ with $s, s' \in \mathcal{S}_\Psi$. Next, we partition these transitions via the following equivalence relation: $(s_1, a_1, s_1') \equiv (s_2, a_2, s_2')$ if the effects and controllers *unify*, that is, if there exists a mapping between the objects such that $\underline{\mathcal{C}_1}$, $(s_1 - s_1')$, and $(s_1' - s_1)$ are equivalent to $\underline{\mathcal{C}_2}$, $(s_2 - s_2')$, and $(s_2' - s_2)$ respectively. After this step, we have effectively 'clustered' all transitions in $\mathcal{D}$ together: we can associate each transition $\{(s, a, s')\}$ with a particular equivalence class.
2. *Arguments and Effects induction:* For each equivalence class created in the previous step, we create $\overline{v}$ by selecting an arbitrary transition (s, a, s') and replacing each object that appears in the controller $\underline{\mathcal{C}}$ or effects with a variable of the same type. This further induces a substitution $\delta : \overline{v} \to \mathcal{O}$ for the objects $\mathcal{O}$ in this transition. Given this, the E^+, and E^- can then be created by applying δ to $(s' - s)$, and $(s - s')$ respectively. By construction, for all other transitions τ in the same equivalence class, there exists an injective substitution δ_τ under which the controller arguments and effects are equivalent to the newly created E^+, and E^-.
3. *Precondition learning:* The only remaining component required to turn each equivalence class into an operator is the operator preconditions. For this, we perform an intersection over all abstract states in each equivalence class (Bonet and Geffner 2019; Curtis et al. 2021). Recall that an abstract state is simply the collection of ground atoms that are 'true', thus taking an intersection amounts to finding the set of atoms that are *always* true across every initial state of every transition (s, a, s') in the equivalence class. However, since some of our predicate classifiers might be noisy they might not *always* hold. Thus, we take a 'soft' intersection: we take any atom as a precondition that is true across more than a specific percentage (set by a hyperparameter $h_{\text{pre\_frac}}$) of transitions in the equivalence class. More specifically, $P \leftarrow \bigcap_{\tau=(s,\cdot,\cdot)} \delta_\tau^{-1}(s)$ if $\frac{|\delta_\tau^{-1}(s)|}{|\mathcal{D}|} \geq h_{\text{pre\_frac}}$, where $\delta_\tau^{-1}(s)$ substitutes all occurrences of the objects in s with the parameters in $\overline{v}$ following an inversion of δ_τ, and discards any atoms involving objects that are not in the image of δ_τ. $|\delta_\tau^{-1}(s)|$ denotes the number of transitions in $\mathcal{D}$ in which the lifted atom $|\delta_\tau^{-1}(s)|$ holds, and $|\mathcal{D}|$ denotes the total number of transitions in $\mathcal{D}$. In our experiments, we set $h_{\text{pre\_frac}}$ to 0.8.
4. *Pruning low-data operators:* Noise in the atom values often causes there to be equivalence classes with very few data points, since transitions that have been affected by noise do not unify with other transitions in our dataset. These lead to operators that are overfit to those particular noisy transitions, which are undesirable. We combat this via simply discarding learned operators that have data below a certain fraction (denoted by hyperparameter $h_{\text{data\_frac}}$) of the total transitions associated with a particular controller $\mathcal{C}$. In particular, let $|\tau_\mathcal{C}|$ denote the number of transitions in $\mathcal{D}$ where the action a involves a particular controller $\mathcal{C}$. For any operator ω, let $|\tau_\omega|$ denote the number of datapoints associated with the equivalence class used to construct that operator. We only keep an operator ω if $\frac{|\tau_\omega|}{|\tau_\mathcal{C}|} \geq h_{\text{data\_frac}}$. In our experiments, we set

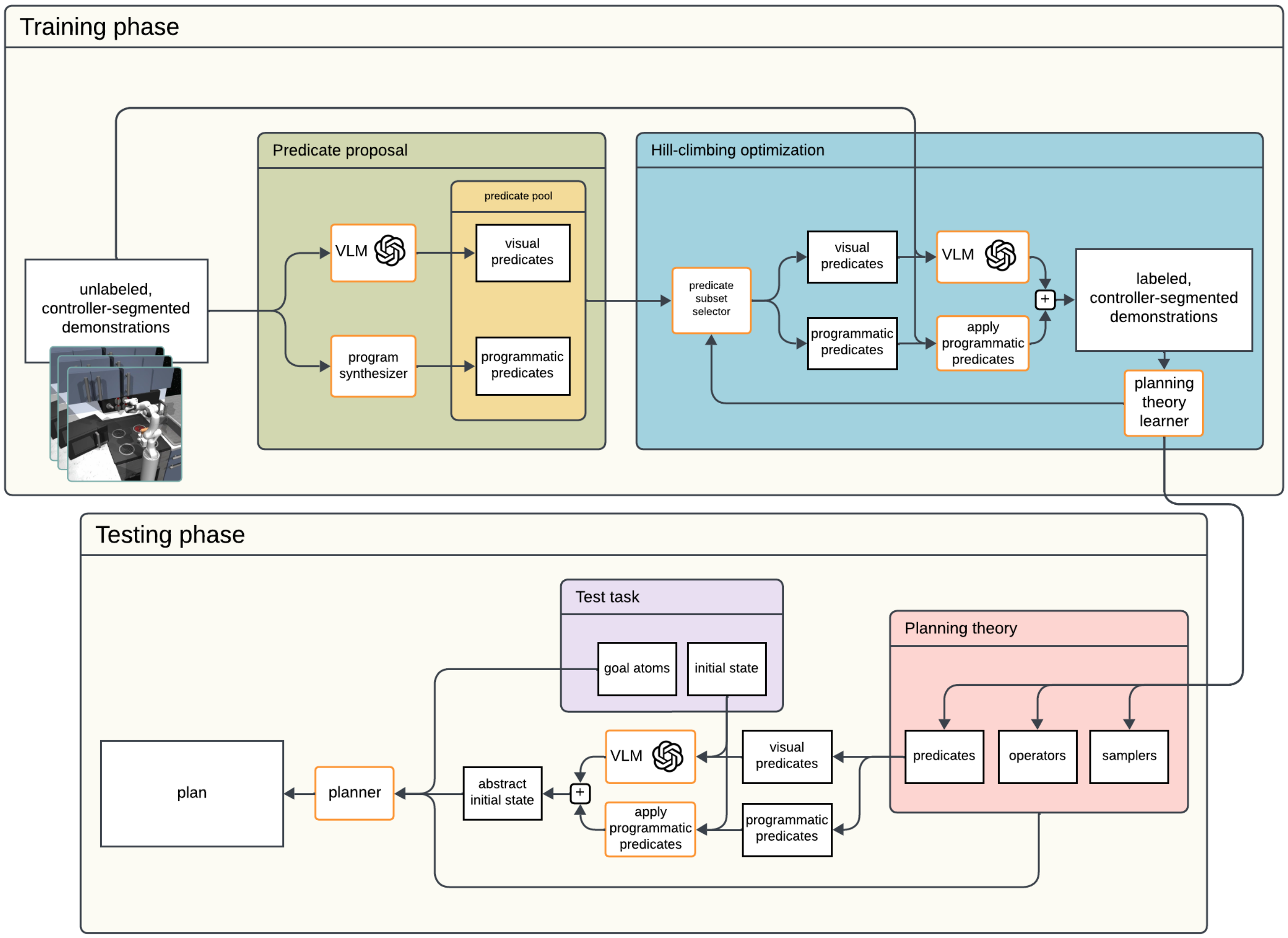

Figure 8: **`Pix2Pred` Detailed Overview.** At training time, we learn a planning theory of predicates, operators, and samplers from demonstrations. We start by proposing a large pool of candidate predicates: we prompt a VLM to generate "visual predicates" (e.g. `'cooked(?p:patty)'`) that operate on images in the image-based state, and we use a programmatic grammar to generate "feature-based predicates" (like `patty.z < 0.5`) that operate on features of the feature-based state (that is computed by a perception function). Next, we apply the hill-climbing optimization from (Silver et al. 2023) to subselect a small set of predicates optimized for efficient and effective planning over the training tasks. As part of this, we label the truth values of ground atoms with visual predicates by asking the VLM for their values; we label those with feature-based predicates via their underlying programmatic classifier. After selecting a final set of predicates, we learn operators and samplers using variants of methods from (Chitnis et al. 2022). At testing time, we label ground atoms in the initial state to get an abstract initial state and use this abstract state with our operators and samples to plan to achieve the goal.

$h_{\text{data\_frac}} = 0.05$.

A.3 Sampler Learning

To enable execution, we must also learn samplers for proposing continuous controller parameters (Algorithm 1). Adapting prior work (Silver et al. 2023; Kumar et al. 2023; Chitnis et al. 2022), we train one sampler per operator, defined as:

$$\sigma(x, o_1, \ldots, o_k) = s_\sigma(x[o_1] \oplus \cdots \oplus x[o_k]),$$

where $x[o]$ is the feature vector for o, $\oplus$ denotes concatenation, and s_σ is a learned model. Treating this as supervised learning, we use operator-specific datasets τ_ω, where each of these datasets is composed of all transitions from the equivalence class used to induce this operator (Appendix A.2). Each transition $(s_i, a_{i+1}, s_{i+1}) \in \tau_\omega$ maps operator arguments $\overline{v}$ to objects via $\delta : \overline{v} \to \mathcal{O}_\tau$. Using this mapping, the input for training is $x[\delta(v_1)] \oplus \cdots \oplus x[\delta(v_k)]$, where $(v_1, \ldots, v_k) = \overline{v}$, and the output is the continuous parameter vector θ, which are the continuous parameters used for a_{i+1}.

Each sampler is parameterized by two neural networks. The first predicts a Gaussian distribution over θ, regressing its mean and covariance. The second network classifies $(x[o_1] \oplus \cdots \oplus x[o_k], \theta)$ as valid or invalid, enabling rejection sampling from the Gaussian. Negative examples are transitions outside τ_ω but using the same controller.

A.4 More on Goals

The agent may sometimes invent visual predicates that can be interpreted as component of the task goal. For example, the goal of a task may be to make a burger with a cooked patty inside it. Here, the goal predicate would be an indicator for if this is achieved or not, perhaps called `BurgerComplete(bottom_bun, patty, top_bun)`, in which case `Cooked(patty)` is an implicit component of that goal, in that `BurgerComplete`'s classifier implicitly checks that the patty is cooked. This is in contrast with a situation where the goal of a task is to fill a cup of water, and where the cup starts out upside down so that you can't pour into it. Here, `Upright(cup)` may not be a component of `HasWater(cup)`, in that the classifier for `HasWater` may not be explicitly checking if the cup is upright. In the former case, one could argue that there is no usefulness to inventing `Cooked` if we already have the goal predicate `BurgerComplete`, as we should be able to extract this concept from inside the goal predicate. But this may not be possible depending on how the classifier for the goal predicate was implemented – we may not have access to its internals. And even if we did, it may not be realistically possible to generate a dataset of burgers with and without cooked patties and label them with `BurgerComplete`'s classifier to obtain a dataset to train a classifier for `Cooked`. Furthermore, in our experimental setup, we would not necessarily need the classifier for `BurgerComplete` at all: at training time, any ground atoms associated with this predicate could be manually labeled in the demonstrations (easy to do when there are few goals per task), and at test time, because we execute one single plan without execution monitoring and replanning, we only need to check if the agent has solved the task once when the agent has completed its plan, which we could also do manually (analogously, the head chef at a restaurant does a manual check on the final burgers produced by the cooking staff).

A.5 Additional Prompting Details

Here, we provide additional details how we prompt a VLM for both labeling and proposal of atoms.

Atom Labeling. Recall from Section 4.1 that the goal of atom labeling is to obtain the truth value of a particular ground atom $\underline{\psi}(o_0, \ldots, o_l)$ given an image-based state at timestep t in a trajectory x_t^{img}. We do this by prompting a VLM with a text prompt, as well as a string representation of the atom (e.g. `Cooked(patty1)`) and asking it to output the truth value of the atom. Since we usually want to query for the values of many atoms at once, we provide all atoms in one single query and ask a VLM to label their truth values simultaneously.

We use a different prompt depending on whether timestep $t = 0$ or $t > 0$. For $t = 0$, we simply provide the initial image-based state x_0^{img} and a prompt (shown below) that asks the model to label all the values of atoms listed in the prompt with either "True', "False" or "Unknown".

> You are a vision system for a robot. Your job is to output the values of the following predicates based on the provided visual scene. For each predicate, output True, False, or Unknown if the relevant objects are not in the scene or the value of the predicate simply cannot be determined. Output each predicate value as a bulleted list with each predicate and value on a different line. For each output value, provide an explanation as to why you labelled this predicate as having this particular value. Use the format: <predicate>: <truth_value>. <explanation>.
> Predicates:

We then specify the ground atom strings whose values we'd like to query. For instance, in the "Bigger Burger" task, these might be:

```
on_table(patty1)
on_table(patty2)
on_table(top_bun1)
on_table(top_bun2)
prepared(patty1)
prepared(patty2)
raw(patty1)
raw(patty2)
uncooked(patty1)
uncooked(patty2)
```

We found two aspects of this were critical to labeling accuracy. Firstly, chain-of-thought prompting (Wei et al. 2022) was extremely useful: our prompt asks the VLM to explicitly provide reasoning for its choices in addition to labeling the truth value. Secondly, recall that the image-based state x^{img} includes segmentation of object names rendered onto the image itself (as depicted in Figure 2). This is a form of set-of-marks prompting (Yang et al. 2023) and is critical to the VLM being able to identify objects being referred to in the text prompt.

For all states x_t in a trajectory that are not the initial state (i.e., $t > 0$), we prompt the VLM with the following information: (1) the image-based state from the current and previous timesteps (i.e. x_t^{img} and x_{t-1}^{img}), (2) the action executed to get to the current state (i.e. a_{t-1}), and (3) the response of the VLM from the previous timestep.

We modify the prompt to the VLM to be as follows (here we show some of the predicates queried and a particular action for a certain timestep in a trajectory for a Burger task):

> You are a vision system for a robot. You are provided with two images corresponding to the states before and after a particular skill is executed. You are given a list of predicates below, and you are given the values of these predicates in the image before the skill is executed. Your job is to output the values of the following predicates in the image after the skill is executed. Pay careful attention to the visual changes between the two images

to figure out which predicates change and which predicates do not change. For the predicates that change, list these separately at the end of your response. Note that in some scenes, there might be no changes. First, output a description of what changes you expect to happen based on the skill that was just run, explicitly noting the skill that was run. Second, output a description of what visual changes you see happen between the before and after images, looking specifically at the objects involved in the skill's arguments, noting what objects these are. Next, output each predicate value in the after image as a bulleted list with each predicate and value on a different line. For each predicate value, provide an explanation as to why you labeled this predicate as having this particular value. Use the format: <predicate>: <truth_value>. <explanation>.

Your response should have three sections. Here is an outline of what your response should look like:

[START OUTLINE]

Expected changes based on the executed skill

[insert your analysis on the expected changes you will see based on the skill that was executed]

Visual changes observed between the images

[insert your analysis on the visual changes observed between the images]

Predicate values in the after image

[insert your bulleted list of '* ¡predicate¿: ¡truth value¿. ¡explanation¿']

[END OUTLINE]

Predicates: available(robot)
busy(robot)
cooked(patty1)
empty(bottom_bun1)
empty(grill)
empty(robot)
empty_grill(grill)
free(robot)

Skill executed between states: Place[robot:robot, top_bun1:top_bun, patty1:patty]

We also provide the response of the VLM from a previous timestep, an example of which is shown below.

Predicate values in the first scene, before the skill was executed: # Expected changes based on the executed skill

The skill executed is "Pick[robot:robot, top_bun1:top_bun]". We expect the robot to pick up the top bun. The top bun should no longer be on the ground or table and should be in the robot's possession.

Visual changes observed between the images

In the before image, the top bun is on the ground. In the after image, the robot is holding the top bun, indicating that the robot has successfully picked it up.

Predicate values in the after image

* available(robot): Unknown. The previous value was unknown, and the pick action does not provide information about availability.
* busy(robot): Unknown. The previous value was unknown, and the pick action does not provide information about busyness.
* cooked(patty1): True. The patty remains cooked, as it was cooked before the action. Previously, the value was true.
* empty(bottom_bun1): False. The bottom bun remains unchanged. Previously, the value was false.
* empty(grill): True. The grill remains empty, as the action does not involve the grill. Previously, the value was true.
* empty(robot): False. The robot is now holding the top bun. Previously, the value was true.
* empty_grill(grill): True. The grill remains empty, as the action does not involve the grill. Previously, the value was true.
* free(robot): Unknown. The previous value was unknown, and the pick action does not provide information about freedom. ...

We found that prompting the VLM to describe differences between two scenes in a chain-of-thought fashion led to much more accurate output than simply labeling atoms based on a single scene.

There are two additional prompting techniques we found helpful and important for labeling accuracy: (1) asking the VLM to "double-check" its label output based on the explanations it provided, and (2) augmenting the image-based state to include a close-up crop of the image for the objects the robot is currently interacting with. For this double-checking, we provide the VLM with the following prompt:

Sometimes your reasoning about the value of a predicate at the current timestep uses an incorrect value of that predicate in the previous timestep. Below, I give you give you the values of the predicates at the previous timestep once again. Please check your reasoning and provide a corrected version of your previous answer, if it needs correcting. Regardless of whether or not it needs correctly, your reply should be formatted exactly the same as the previous answer.

We use double-checking only at training time to maximize labeling accuracy before hill-climbing. We augment images with additional object crops only for tasks in the Burger domain.

Atom Proposal Recall from Section 4.1 that the objective of atom proposal is to generate an initial pool of visual predicates given a set of demonstrations $\mathcal{D}$. We prompt a VLM to propose ground atoms on each demonstration $d \in D$, and then aggregate these and "lift" them into predicates. Importantly, it is crucial to have diversity in the initial predicate pool: our approach will subselect and remove irrelevant predicates, but will fail if the initial pool does not have a

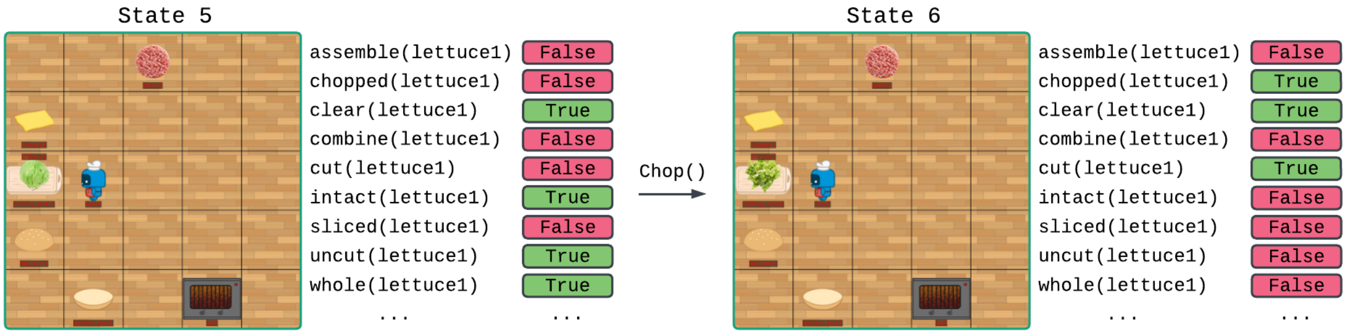

Figure 9: **Labeling of ground atoms with visual predicates involving the lettuce object type in Burger**. This figure depicts the visual state before and after the *chop* skill in demonstration #1 for the task distribution "Combo Burger". It also shows the labels assigned by the VLM to ground atoms with visual predicates involving the lettuce object in these states. The VLM proposes a wide variety of predicates involving the patty: `assemble`, `chopped`, `clear`, `combine`, `cut`, `intact`, `sliced`, `uncut`, and `whole`. Of these, `chopped(?l:lettuce)` and `cut(?l:lettuce)` are the most relevant to achieving the goal while also being accurately labeled. Our approach automatically selects the former predicate from the pool.

predicate that is important for decision-making across tasks from the task distribution $\mathcal{T}$. To promote diversity in the pool, we explicitly prompt the VLM to list all ground atoms that might be relevant to decision-making, and also to output at least one synonym and antonym of every ground atom it proposes (e.g. if it proposes 'cooked(patty1)', it might also propose 'grilled(patty1)' and 'uncooked(patty1)').

Below, we provide our full prompt for atom proposal:

> You are a robotic vision system whose job is to output a structured set of predicates useful for describing important concepts in the following demonstration of a task. You will be provided with a list of actions used during the task, as well as images of states before and after every action execution. Please provide predicates in terms of the following objects: {objs}. For each predicate, output it in the following format: predicate_name(obj1, obj2, obj3...). Start by generating predicates that change before and after each action. After this, generate any other predicates that perhaps do not change but are still important to describing the demonstration shown. For each predicate you generate, also generate some predicates that are synonyms and antonyms so that any predicate that is even tangentially relevant to the demonstrations is generated.

Once again, we found that chain-of-thought prompting was critical to good proposals. We also found that ensuring all the image-based states have segmented objects with names rendered on them was important to get a good set of proposals.

We then provide a text representation of the action sequence executed in the demonstration trajectory. An example in a Burger task is:

> Skills executed in trajectory:
> Pick[robot:robot, patty1:patty]
> Place[robot:robot, patty1:patty, grill:grill]
> Cook[robot:robot, patty1:patty, grill:grill]
> Pick[robot:robot, patty1:patty]
> Place[robot:robot, patty1:patty, bottom_bun1:bottom_bun]
> Pick[robot:robot, top_bun1:top_bun]
> Place[robot:robot, top_bun1:top_bun, patty1:patty]

An example response from the VLM in a Burger task is shown below.

> ### Predicates for Each Action
> 1. **Pick[robot:robot, patty1:patty]**
> - Before: on(patty1, ground)
> - After: holding(robot, patty1)
> - Synonyms: grasp(robot, patty1)
> - Antonyms: release(robot, patty1)
>
> 2. **Place[robot:robot, patty1:patty, grill:grill]**
> - Before: holding(robot, patty1)
> - After: on(patty1, grill)
> - Synonyms: position(patty1, grill)
> - Antonyms: off(patty1, grill)
>
> 3. **Cook[robot:robot, patty1:patty, grill:grill]**
> - Before: raw(patty1)
> - After: cooked(patty1)
> - Synonyms: grill(patty1)
> - Antonyms: uncooked(patty1)
>
> 4. **Pick[robot:robot, patty1:patty]**
> - Before: on(patty1, grill)
> - After: holding(robot, patty1)
> - Synonyms: grasp(robot, patty1)
> - Antonyms: release(robot, patty1)
>
> 5. **Place[robot:robot, patty1:patty, bottom_bun1:bottom_bun]**
> - Before: holding(robot, patty1)

- After: on(patty1, bottom_bun1)
- Synonyms: position(patty1, bottom_bun1)
- Antonyms: off(patty1, bottom_bun1)

6. **Pick[robot:robot, top_bun1:top_bun]**
- Before: on(top_bun1, ground)
- After: holding(robot, top_bun1)
- Synonyms: grasp(robot, top_bun1)
- Antonyms: release(robot, top_bun1)

7. **Place[robot:robot, top_bun1:top_bun, patty1:patty]**
- Before: holding(robot, top_bun1)
- After: on(top_bun1, patty1)
- Synonyms: position(top_bun1, patty1)
- Antonyms: off(top_bun1, patty1)

Other Important Predicates

- **Initial State:**
- on(patty1, ground)
- on(bottom_bun1, ground)
- on(top_bun1, ground)
- empty(robot)

- **Final State:**
- on(top_bun1, patty1)
- on(patty1, bottom_bun1)
- empty(robot)

- **Synonyms:**
- empty(robot), free(robot)
- on(obj1, obj2). above(obj1, obj2)

- **Antonyms:** - empty(robot), holding(robot, obj)
- on(obj1, obj2), off(obj1, obj2)

A.6 Additional Experimental Details

Additional Environment Details. Here, we describe in detail the initial predicates, training demonstrations, and test tasks for each task in each of our experimental environments. Note that the controllers listed below have discrete object parameters (indicated by the '?') as well as continuous parameters where applicable θ (shown within []).

- *Kitchen:*
  - Given Predicates:
    - `KettleBoiling(?k: kettle, ?kn: knob, ?b burner)`: is true only if ?kn is turned on, and the kettle ?k is ontop of ?b, and if ?kn and ?b are 'linked' (i.e., ?kn is the knob that causes burner ?b to glow red with heat).
  - Skills:
    - `TurnOnKnob(?g: gripper, ?kn: knob, [push_dir])`: Moves the gripper ?g to a fixed location near the knob, and pushes with angle "push_dir" with respect to the horizontal axis of the gripper for a fixed number of timesteps in an effort to flick the corresponding knob on.
    - `PushKettleOntoBurner(?g: gripper, ?k: kettle, ?b: burner, [push_x. push_y, push_z])`: Moves the gripper behind the current location of kettle ?k and then pushes along the 3D vector [push_x. push_y, push_z] for a fixed number of timesteps.
  - Training demonstrations: we provide 3 demonstrations that execute `TurnOnKnob` and then `PushKettleOntoBurner` in sequence to achieve `KettleBoiling(kettle1, knob2, burner2)`, where burner2 and knob2 are in the back left on the stove. The kettle starts out on the front left of the stove.
  - Test tasks: Given the kettle starts out on the front right burner, we task the agent with moving it to the back right burner. In this case, plans are only two steps, and actually simply replaying the demonstration plans will work.
- *Burger:* The three different task distributions we implement in this domain share common skills. We list these before listing the task-distribution-specific initial predicates, demonstrations, and goals.
  General info:
  - A "burger" consists of a top bun above a bottom bun, with one or more items in between.
  - Every type has a row/column/z attribute. The z attribute changes when an object is picked up or placed onto other objects. The grill and cutting board are of type `object`; the patty, lettuce, cheese, bottom bun, and top bun are of type `item`, which is a subtype of type `object`; the robot is of type `robot`, and has additional attributes "fingers" that indicates how open its gripper is, and "dir" that indicates the direction it is facing.

  Skills:
  - `Pick(?r: robot, ?i: item, [])`. Moves the robot `?r` to a cell adjacent to item `?i` and picks it up if the robot isn't currently holding anything.
  - `Place(?r: robot, ?i: item, ?o: object, [])`. Moves the robot `?r` to a cell adjacent to object `?o` and places item `?i` atop object `?o` if it is holding item `?i`.
  - `Cook(?r: robot, ?p: patty, ?g: grill)`. Given the robot is adjacent to grill `?g` and patty `?p` is atop the grill, cooks the patty so it appears grilled.
  - `Chop(?r: robot, ?l: lettuce, ?c: cutting_board)`. Given the robot is adjacent to cutting board `?c` and lettuce `?l` is atop the cutting board, chops the lettuce with a knife that's on the cutting board so it appears chopped.
  - **Bigger Burger**
    - Given predicates:
      - `On(?o1: object, ?o2: object)`: turns true when `?o1` is atop `?o2`.

- `OnGround(?o1: object)`: turns true when `?o1` is atop a cell `?o2` that forms the ground.
- `Clear(?o: object)`: is true only when there is no object atop the object `?o`.
- `Holding(?r: robot, ?i: item)`: turns true only when the robot `?r` is holding the item `?i`.
- `SomewhereAboveAndPrepped(?p: patty, ?b: bottom_bun)`: turns true when `?p` is somewhere above `?b` and `?p` is cooked.
- `RightAboveAndPrepped(?p: patty, ?c: cutting_board)`: turns true when `?p` is right above `?c` and `?p` is cooked.
- `RightAboveAndPrepped(?p: patty, ?g: grill)`: turns true when `?p` is right above `?g` and `?p` is cooked.
- `RightAboveAndPrepped(?p1: patty, ?p2: patty)`: turns true when `?p1` is right above `?p2` and `?p1` is cooked.

- Training demonstrations: We provide 4 demonstrations that make one burger with a single cooked patty, 4 demonstrations that cook two patties and stack them on the cutting board, and 4 demonstrations that cook two patties and stack them on the grill.
- Test tasks: There are 10 test tasks. In 5 of these test tasks, we ask the agent to make a burger with 2 cooked patties in it. In the other 5 test tasks, we ask the agent to make a burger with 2 cooked patties in it in addition to making a single open-face burger with a cooked patty, and the agent starts out holding a raw patty.

- **More Burger Stacks**
  - Given predicates:
    - `On(?o1: object, ?o2: object)`: turns true when `?o1` is atop `?o2`.
    - `OnGround(?o1: object)`: turns true when `?o1` is atop a cell `?o2` that forms the ground.
    - `Clear(?o: object)`: is true only when there is no object atop the object `?o`.
    - `Holding(?r: robot, ?i: item)`: turns true only when the robot `?r` is holding the item `?i`.
    - `SomewhereAboveAndPrepped(?p: patty, ?b: bottom_bun)`: turns true when `?p` is somewhere above `?b` and `?p` is cooked.
  - Training demonstrations: We provide 1 demonstration that makes two burgers, each with a single cooked patty, and 11 demonstrations that make one burger with a single cooked patty.
  - Test tasks: There are 10 test tasks. In 5 of these test tasks, we ask the agent to make 5 "open-face" burgers – burgers that consist of a cooked patty on a bottom bun. In the other 5 test tasks, we ask the agent to make 6 open-face burgers, and the agent starts out holding a raw patty.
- **Combo Burger**
  - Given predicates:
    - `On(?o1: object, ?o2: object)`: turns true when `?o1` is atop `?o2`.
    - `OnGround(?o1: object)`: turns true when `?o1` is atop a cell `?o2` that forms the ground.
    - `Clear(?o: object)`: is true only when there is no object atop the object `?o`.
    - `Holding(?r: robot, ?i: item)`: turns true only when the robot `?r` is holding the item `?i`.
    - `SomewhereAboveAndPrepped(?p: patty, ?b: bottom_bun)`: turns true when `?p` is somewhere above `?b` and `?p` is cooked.
    - `SomewhereAboveAndPrepped(?l: lettuce, ?b: bottom_bun)`: turns true when `?l` is somewhere above `?b` and `?l` is choppeed.
    - `SomewhereAboveAndPrepped(?l: lettuce, ?p: patty)`: turns true when `?l` is somewhere above `?p` and `?l` is chopped.
  - Training demonstrations: We provide 3 demonstrations that make one burger with a single cooked patty, 3 demonstrations that make one burger with a single chopped lettuce, 3 demonstrations that place a raw patty on the cutting board, chop lettuce, and place the chopped lettuce on the patty, and 3 demonstrations that place a raw patty on the grill, chop lettuce, and place the chopped lettuce on the patty.
  - Test tasks: There are 10 test tasks. In 5 of these test tasks, we ask the agent to make two burgers, each with chopped lettuce on a cooked patty. In the other 5 test tasks, we ask the agent to make two burgers, each with chopped lettuce on a cooked patty, in addition to making another burger with a single cooked patty, and the agent starts out holding a raw patty.

A.7 Learned Predicate and Operator Examples

Figures 10, 11, 12, and show our learned predicates and operators in our "Kitchen", "Bigger Burger", "Burger More Stacks" and "Combo Burger" tasks respectively.

Subselected Predicates:

- NOT-[[0:surface].z<=[idx_0]1.59]

Operators:

STRIPS-Op0:
  Parameters: [?x0:surface, ?x1:gripper, ?x2:knob]
  Preconditions: [KnobAndBurnerLinked(?x2:knob, ?x0:surface)]
  Add Effects: [NOT-[[0:surface].z<=[idx_0]1.59](?x0:surface)]
  Delete Effects: []
  Ignore Effects: []
  Controller: MoveAndTurnOnKnob(?x1:gripper, ?x2:knob)[params]
STRIPS-Op1:
  Parameters: [?x0:surface, ?x1:gripper, ?x2:kettle, ?x3:knob]
  Preconditions: [KnobAndBurnerLinked(?x3:knob, ?x0:surface),
NOT-[[0:surface].z<=[idx_0]1.59](?x0:surface)]
  Add Effects: [KettleBoiling(?x2:kettle, ?x0:surface, ?x3:knob)]
  Delete Effects: []
  Ignore Effects: []
  Controller: PushKettleOntoBurner(?x1:gripper, ?x2:kettle, ?x0:surface)[params]

Figure 10: Learned predicates and operators for the “Kitchen” task.

Subselected Predicates:

- cooked0(?x: patty)
- [[0:robot].fingers<=[idx_0]0.5]

Operators:

STRIPS-Op0:
Parameters: [?x0:patty, ?x1:robot]
Preconditions: [Clear(?x0:patty), Clear(?x1:robot), OnGround(?x0:patty), [[0:robot].fingers<=[idx_0]0.5](?x1:robot)]
Add Effects: [Holding(?x1:robot, ?x0:patty)]
Delete Effects: [Clear(?x0:patty), OnGround(?x0:patty), [[0:robot].fingers<=[idx_0]0.5](?x1:robot)]
Ignore Effects: []
Option Spec: Pick(?x1:robot, ?x0:patty)
STRIPS-Op1:
Parameters: [?x0:grill, ?x1:patty, ?x2:robot]
Preconditions: [Clear(?x0:grill), Clear(?x2:robot), Holding(?x2:robot, ?x1:patty)]
Add Effects: [Clear(?x1:patty), On(?x1:patty, ?x0:grill), [[0:robot].fingers<=[idx_0]0.5](?x2:robot)]
Delete Effects: [Clear(?x0:grill), Holding(?x2:robot, ?x1:patty)]
Ignore Effects: []
Option Spec: Place(?x2:robot, ?x1:patty, ?x0:grill)
STRIPS-Op2:
Parameters: [?x0:grill, ?x1:patty, ?x2:robot]
Preconditions: [Clear(?x1:patty), Clear(?x2:robot), On(?x1:patty, ?x0:grill), [[0:robot].fingers<=[idx_0]0.5](?x2:robot)]
Add Effects: [RightAboveAndPrepped(?x0:grill, ?x1:patty), cooked0(?x1:patty)]
Delete Effects: []
Ignore Effects: []
Option Spec: Cook(?x2:robot, ?x1:patty, ?x0:grill)
STRIPS-Op3:
Parameters: [?x0:grill, ?x1:patty, ?x2:robot]
Preconditions: [Clear(?x1:patty), Clear(?x2:robot), RightAboveAndPrepped(?x0:grill, ?x1:patty), On(?x1:patty, ?x0:grill), [[0:robot].fingers<=[idx_0]0.5](?x2:robot), cooked0(?x1:patty)]
Add Effects: [Clear(?x0:grill), Holding(?x2:robot, ?x1:patty)]
Delete Effects: [Clear(?x1:patty), RightAboveAndPrepped(?x0:grill, ?x1:patty), On(?x1:patty, ?x0:grill), [[0:robot].fingers<=[idx_0]0.5](?x2:robot)]
Ignore Effects: []
Option Spec: Pick(?x2:robot, ?x1:patty)
STRIPS-Op4:
Parameters: [?x0:bottom_bun, ?x1:patty, ?x2:robot]
Preconditions: [Clear(?x0:bottom_bun), Clear(?x2:robot), Holding(?x2:robot, ?x1:patty), OnGround(?x0:bottom_bun), cooked0(?x1:patty)]
Add Effects: [Clear(?x1:patty), SomewhereAboveAndPrepped(?x0:bottom_bun, ?x1:patty), On(?x1:patty, ?x0:bottom_bun), [[0:robot].fingers<=[idx_0]0.5](?x2:robot)]
Delete Effects: [Clear(?x0:bottom_bun), Holding(?x2:robot, ?x1:patty)]
Ignore Effects: []
Option Spec: Place(?x2:robot, ?x1:patty, ?x0:bottom_bun)
STRIPS-Op5:
Parameters: [?x0:robot, ?x1:top_bun]
Preconditions: [Clear(?x0:robot), Clear(?x1:top_bun), OnGround(?x1:top_bun), [[0:robot].fingers<=[idx_0]0.5](?x0:robot)]
Add Effects: [Holding(?x0:robot, ?x1:top_bun)]
Delete Effects: [Clear(?x1:top_bun), OnGround(?x1:top_bun), [[0:robot].fingers<=[idx_0]0.5](?x0:robot)]
Ignore Effects: []
Option Spec: Pick(?x0:robot, ?x1:top_bun)
STRIPS-Op6:
Parameters: [?x0:patty, ?x1:robot, ?x2:top_bun]
Preconditions: [Clear(?x0:patty), Clear(?x1:robot), Holding(?x1:robot, ?x2:top_bun), cooked0(?x0:patty)]
Add Effects: [Clear(?x2:top_bun), On(?x2:top_bun, ?x0:patty), [[0:robot].fingers<=[idx_0]0.5](?x1:robot)]
Delete Effects: [Clear(?x0:patty), Holding(?x1:robot, ?x2:top_bun)]
Ignore Effects: []
Option Spec: Place(?x1:robot, ?x2:top_bun, ?x0:patty)
STRIPS-Op7:
Parameters: [?x0:cutting_board, ?x1:patty, ?x2:robot]
Preconditions: [Clear(?x0:cutting_board), Clear(?x2:robot), Holding(?x2:robot, ?x1:patty), cooked0(?x1:patty)]
Add Effects: [Clear(?x1:patty), RightAboveAndPrepped(?x0:cutting_board, ?x1:patty), On(?x1:patty, ?x0:cutting_board), [[0:robot].fingers<=[idx_0]0.5](?x2:robot)]
Delete Effects: [Clear(?x0:cutting_board), Holding(?x2:robot, ?x1:patty)]
Ignore Effects: []
Option Spec: Place(?x2:robot, ?x1:patty, ?x0:cutting_board)
STRIPS-Op8:
Parameters: [?x0:patty, ?x1:patty, ?x2:robot]
Preconditions: [Clear(?x0:patty), Clear(?x2:robot), Holding(?x2:robot, ?x1:patty), cooked0(?x0:patty), cooked0(?x1:patty)]
Add Effects: [Clear(?x1:patty), RightAboveAndPrepped(?x0:patty, ?x1:patty), On(?x1:patty, ?x0:patty), [[0:robot].fingers<=[idx_0]0.5](?x2:robot)]
Delete Effects: [Clear(?x0:patty), Holding(?x2:robot, ?x1:patty)]
Ignore Effects: []
Option Spec: Place(?x2:robot, ?x1:patty, ?x0:patty)
STRIPS-Op9:
Parameters: [?x0:cutting_board, ?x1:patty, ?x2:robot]
Preconditions: [Clear(?x1:patty), Clear(?x2:robot), RightAboveAndPrepped(?x0:cutting_board, ?x1:patty), On(?x1:patty, ?x0:cutting_board), [[0:robot].fingers<=[idx_0]0.5](?x2:robot), cooked0(?x1:patty)]
Add Effects: [Clear(?x0:cutting_board), Holding(?x2:robot, ?x1:patty)]
Delete Effects: [Clear(?x1:patty), RightAboveAndPrepped(?x0:cutting_board, ?x1:patty), On(?x1:patty, ?x0:cutting_board), [[0:robot].fingers<=[idx_0]0.5](?x2:robot)]
Ignore Effects: []
Option Spec: Pick(?x2:robot, ?x1:patty)

Figure 11: Learned predicates and operators for the "More Burger Stacks" task.

Subselected Predicates:

- cooked0(?x: patty)
- empty_hands(?x: robot)

Operators:

```
STRIPS-Op0:
   Parameters: [?x0:patty, ?x1:robot]
   Preconditions: [Clear(?x0:patty), Clear(?x1:robot), OnGround(?x0:patty), empty_hands0(?x1:robot)]
   Add Effects: [Holding(?x1:robot, ?x0:patty)]
   Delete Effects: [Clear(?x0:patty), OnGround(?x0:patty), empty_hands0(?x1:robot)]
   Ignore Effects: []
   Controller: Pick(?x1:robot, ?x0:patty)
STRIPS-Op1:
   Parameters: [?x0:grill, ?x1:patty, ?x2:robot]
   Preconditions: [Clear(?x0:grill), Clear(?x2:robot), Holding(?x2:robot, ?x1:patty)]
   Add Effects: [Clear(?x1:patty), On(?x1:patty, ?x0:grill), empty_hands0(?x2:robot)]
   Delete Effects: [Clear(?x0:grill), Holding(?x2:robot, ?x1:patty)]
   Ignore Effects: []
   Controller: Place(?x2:robot, ?x1:patty, ?x0:grill)
STRIPS-Op2:
   Parameters: [?x0:grill, ?x1:patty, ?x2:robot]
   Preconditions: [Clear(?x1:patty), Clear(?x2:robot), On(?x1:patty, ?x0:grill), empty_hands0(?x2:robot)]
   Add Effects: [cooked0(?x1:patty)]
   Delete Effects: []
   Ignore Effects: []
   Controller: Cook(?x2:robot, ?x1:patty, ?x0:grill)
STRIPS-Op3:
   Parameters: [?x0:grill, ?x1:patty, ?x2:robot]
   Preconditions: [Clear(?x1:patty), Clear(?x2:robot), On(?x1:patty, ?x0:grill), cooked0(?x1:patty), empty_hands0(?x2:robot)]
   Add Effects: [Clear(?x0:grill), Holding(?x2:robot, ?x1:patty)]
   Delete Effects: [Clear(?x1:patty), On(?x1:patty, ?x0:grill), empty_hands0(?x2:robot)]
   Ignore Effects: []
   Controller: Pick(?x2:robot, ?x1:patty)
STRIPS-Op4:
   Parameters: [?x0:bottom_bun, ?x1:patty, ?x2:robot]
   Preconditions: [Clear(?x0:bottom_bun), Clear(?x2:robot), Holding(?x2:robot, ?x1:patty), OnGround(?x0:bottom_bun),
cooked0(?x1:patty)]
   Add Effects: [Clear(?x1:patty), SomewhereAboveAndPrepped(?x0:bottom_bun, ?x1:patty), On(?x1:patty, ?x0:bottom_bun),
empty_hands0(?x2:robot)]
   Delete Effects: [Clear(?x0:bottom_bun), Holding(?x2:robot, ?x1:patty)]
   Ignore Effects: []
   Controller: Place(?x2:robot, ?x1:patty, ?x0:bottom_bun)
STRIPS-Op5:
   Parameters: [?x0:robot, ?x1:top_bun]
   Preconditions: [Clear(?x0:robot), Clear(?x1:top_bun), OnGround(?x1:top_bun), empty_hands0(?x0:robot)]
   Add Effects: [Holding(?x0:robot, ?x1:top_bun)]
   Delete Effects: [Clear(?x1:top_bun), OnGround(?x1:top_bun), empty_hands0(?x0:robot)]
   Ignore Effects: []
   Controller: Pick(?x0:robot, ?x1:top_bun)
STRIPS-Op6:
   Parameters: [?x0:patty, ?x1:robot, ?x2:top_bun]
   Preconditions: [Clear(?x0:patty), Clear(?x1:robot), Holding(?x1:robot, ?x2:top_bun), cooked0(?x0:patty)]
   Add Effects: [Clear(?x2:top_bun), On(?x2:top_bun, ?x0:patty), empty_hands0(?x1:robot)]
   Delete Effects: [Clear(?x0:patty), Holding(?x1:robot, ?x2:top_bun)]
   Ignore Effects: []
   Controller: Place(?x1:robot, ?x2:top_bun, ?x0:patty)
```

Figure 12: Learned predicates and operators for the “More Burger Stacks” task.

Subselected Predicates:

- cooked0(?x: patty)
- clear5(?r: robot)
- chopped0(?l: lettuce)

Operators:

```
NSRT-Op0:
    Parameters: [?x0:patty, ?x1:robot]
    Preconditions: [Clear(?x0:patty), Clear(?x1:robot),
OnGround(?x0:patty), clear5(?x1:robot)]
    Add Effects: [Holding(?x1:robot, ?x0:patty)]
    Delete Effects: [Clear(?x0:patty), OnGround(?x0:patty),
clear5(?x1:robot)]
    Ignore Effects: []
    Option Spec: Pick(?x1:robot, ?x0:patty)
NSRT-Op1:
    Parameters: [?x0:grill, ?x1:patty, ?x2:robot]
    Preconditions: [Clear(?x0:grill), Clear(?x2:robot),
Holding(?x2:robot, ?x1:patty)]
    Add Effects: [Clear(?x1:patty), On(?x1:patty, ?x0:grill),
clear5(?x2:robot)]
    Delete Effects: [Clear(?x0:grill), Holding(?x2:robot, ?x1:patty)]
    Ignore Effects: []
    Option Spec: Place(?x2:robot, ?x1:patty, ?x0:grill)
NSRT-Op2:
    Parameters: [?x0:grill, ?x1:patty, ?x2:robot]
    Preconditions: [Clear(?x1:patty), Clear(?x2:robot),
On(?x1:patty, ?x0:grill), clear5(?x2:robot)]
    Add Effects: [cooked0(?x1:patty)]
    Delete Effects: []
    Ignore Effects: []
    Option Spec: Cook(?x2:robot, ?x1:patty, ?x0:grill)
NSRT-Op3:
    Parameters: [?x0:grill, ?x1:patty, ?x2:robot]
    Preconditions: [Clear(?x1:patty), Clear(?x2:robot),
On(?x1:patty, ?x0:grill), clear5(?x2:robot), cooked0(?x1:patty)]
    Add Effects: [Clear(?x0:grill), Holding(?x2:robot, ?x1:patty)]
    Delete Effects: [Clear(?x1:patty), On(?x1:patty, ?x0:grill),
clear5(?x2:robot)]
    Ignore Effects: []
    Option Spec: Pick(?x2:robot, ?x1:patty)
NSRT-Op4:
    Parameters: [?x0:bottom_bun, ?x1:patty, ?x2:robot]
    Preconditions: [Clear(?x0:bottom_bun), Clear(?x2:robot),
Holding(?x2:robot, ?x1:patty), OnGround(?x0:bottom_bun),
cooked0(?x1:patty)]
    Add Effects: [Clear(?x1:patty),
SomewhereAboveAndPrepped(?x0:bottom_bun, ?x1:patty),
On(?x1:patty, ?x0:bottom_bun), clear5(?x2:robot)]
    Delete Effects: [Clear(?x0:bottom_bun), Holding(?x2:robot,
?x1:patty)]
    Ignore Effects: []
    Option Spec: Place(?x2:robot, ?x1:patty, ?x0:bottom_bun)
NSRT-Op5:
    Parameters: [?x0:robot, ?x1:top_bun]
    Preconditions: [Clear(?x0:robot), Clear(?x1:top_bun),
OnGround(?x1:top_bun), clear5(?x0:robot)]
    Add Effects: [Holding(?x0:robot, ?x1:top_bun)]
    Delete Effects: [Clear(?x1:top_bun), OnGround(?x1:top_bun),
clear5(?x0:robot)]
    Ignore Effects: []
    Option Spec: Pick(?x0:robot, ?x1:top_bun)
NSRT-Op6:
    Parameters: [?x0:patty, ?x1:robot, ?x2:top_bun]
    Preconditions: [Clear(?x0:patty), Clear(?x1:robot),
Holding(?x1:robot, ?x2:top_bun), cooked0(?x0:patty)]
    Add Effects: [Clear(?x2:top_bun), On(?x2:top_bun, ?x0:patty),
clear5(?x1:robot)]
    Delete Effects: [Clear(?x0:patty), Holding(?x1:robot,
?x2:top_bun)]
    Ignore Effects: []
    Option Spec: Place(?x1:robot, ?x2:top_bun, ?x0:patty)
NSRT-Op7:
    Parameters: [?x0:lettuce, ?x1:robot]
    Preconditions: [Clear(?x0:lettuce), Clear(?x1:robot),
OnGround(?x0:lettuce), clear5(?x1:robot)]
    Add Effects: [Holding(?x1:robot, ?x0:lettuce)]
    Delete Effects: [Clear(?x0:lettuce), OnGround(?x0:lettuce),
clear5(?x1:robot)]
    Ignore Effects: []
    Option Spec: Pick(?x1:robot, ?x0:lettuce)
NSRT-Op8:
    Parameters: [?x0:cutting_board, ?x1:lettuce, ?x2:robot]
    Preconditions: [Clear(?x0:cutting_board), Clear(?x2:robot),
Holding(?x2:robot, ?x1:lettuce)]
    Add Effects: [Clear(?x1:lettuce), On(?x1:lettuce,
?x0:cutting_board), clear5(?x2:robot)]
    Delete Effects: [Clear(?x0:cutting_board), Holding(?x2:robot,
?x1:lettuce)]
    Ignore Effects: []
    Option Spec: Place(?x2:robot, ?x1:lettuce, ?x0:cutting_board)
NSRT-Op9:
    Parameters: [?x0:cutting_board, ?x1:lettuce, ?x2:robot]
    Preconditions: [Clear(?x1:lettuce), Clear(?x2:robot),
On(?x1:lettuce, ?x0:cutting_board), clear5(?x2:robot)]
    Add Effects: [chopped0(?x1:lettuce)]
    Delete Effects: []
    Ignore Effects: []
    Option Spec: Chop(?x2:robot, ?x1:lettuce,
?x0:cutting_board)
NSRT-Op10:
    Parameters: [?x0:cutting_board, ?x1:lettuce, ?x2:robot]
    Preconditions: [Clear(?x1:lettuce), Clear(?x2:robot),
On(?x1:lettuce, ?x0:cutting_board), chopped0(?x1:lettuce),
clear5(?x2:robot)]
    Add Effects: [Clear(?x0:cutting_board), Holding(?x2:robot,
?x1:lettuce)]
    Delete Effects: [Clear(?x1:lettuce), On(?x1:lettuce,
?x0:cutting_board), clear5(?x2:robot)]
    Ignore Effects: []
    Option Spec: Pick(?x2:robot, ?x1:lettuce)
NSRT-Op11:
    Parameters: [?x0:bottom_bun, ?x1:lettuce, ?x2:robot]
    Preconditions: [Clear(?x0:bottom_bun), Clear(?x2:robot),
Holding(?x2:robot, ?x1:lettuce), OnGround(?x0:bottom_bun),
chopped0(?x1:lettuce)]
    Add Effects: [Clear(?x1:lettuce),
SomewhereAboveAndPrepped(?x0:bottom_bun, ?x1:lettuce),
On(?x1:lettuce, ?x0:bottom_bun), clear5(?x2:robot)]
    Delete Effects: [Clear(?x0:bottom_bun), Holding(?x2:robot,
?x1:lettuce)]
    Ignore Effects: []
    Option Spec: Place(?x2:robot, ?x1:lettuce, ?x0:bottom_bun)
NSRT-Op12:
    Parameters: [?x0:lettuce, ?x1:robot, ?x2:top_bun]
    Preconditions: [Clear(?x0:lettuce), Clear(?x1:robot),
Holding(?x1:robot, ?x2:top_bun), chopped0(?x0:lettuce)]
    Add Effects: [Clear(?x2:top_bun), On(?x2:top_bun,
?x0:lettuce), clear5(?x1:robot)]
    Delete Effects: [Clear(?x0:lettuce), Holding(?x1:robot,
?x2:top_bun)]
    Ignore Effects: []
    Option Spec: Place(?x1:robot, ?x2:top_bun, ?x0:lettuce)
NSRT-Op13:
    Parameters: [?x0:grill, ?x1:lettuce, ?x2:robot]
    Preconditions: [Clear(?x0:grill), Clear(?x2:robot),
Holding(?x2:robot, ?x1:lettuce), chopped0(?x1:lettuce)]
    Add Effects: [Clear(?x1:lettuce), On(?x1:lettuce, ?x0:grill),
clear5(?x2:robot)]
    Delete Effects: [Clear(?x0:grill), Holding(?x2:robot,
?x1:lettuce)]
    Ignore Effects: []
    Option Spec: Place(?x2:robot, ?x1:lettuce, ?x0:grill)
NSRT-Op14:
    Parameters: [?x0:cutting_board, ?x1:patty, ?x2:robot]
    Preconditions: [Clear(?x0:cutting_board), Clear(?x2:robot),
Holding(?x2:robot, ?x1:patty)]
    Add Effects: [Clear(?x1:patty), On(?x1:patty,
?x0:cutting_board), clear5(?x2:robot)]
    Delete Effects: [Clear(?x0:cutting_board),
Holding(?x2:robot, ?x1:patty)]
    Ignore Effects: []
    Option Spec: Place(?x2:robot, ?x1:patty, ?x0:cutting_board)
NSRT-Op15:
    Parameters: [?x0:grill, ?x1:lettuce, ?x2:robot]
    Preconditions: [Clear(?x1:lettuce), Clear(?x2:robot),
On(?x1:lettuce, ?x0:grill), chopped0(?x1:lettuce),
clear5(?x2:robot)]
    Add Effects: [Clear(?x0:grill), Holding(?x2:robot,
?x1:lettuce)]
    Delete Effects: [Clear(?x1:lettuce), On(?x1:lettuce, ?x0:grill),
clear5(?x2:robot)]
    Ignore Effects: []
    Option Spec: Pick(?x2:robot, ?x1:lettuce)
NSRT-Op16:
    Parameters: [?x0:lettuce, ?x1:patty, ?x2:robot]
    Preconditions: [Clear(?x1:patty), Clear(?x2:robot),
Holding(?x2:robot, ?x0:lettuce), chopped0(?x0:lettuce)]
    Add Effects: [Clear(?x0:lettuce),
SomewhereAboveAndPrepped(?x1:patty, ?x0:lettuce),
On(?x0:lettuce, ?x1:patty), clear5(?x2:robot)]
    Delete Effects: [Clear(?x1:patty), Holding(?x2:robot,
?x0:lettuce)]
    Ignore Effects: []
    Option Spec: Place(?x2:robot, ?x0:lettuce, ?x1:patty)
```

Figure 13: Learned predicates and operators for the “More Burger Stacks” task.