Program Synthesis for Robot Learning from Demonstrations

NOAH PATTON, The University of Texas at Austin, USA
KIA RAHMANI, The University of Texas at Austin, USA
MEGHANA MISSULA, The University of Texas at Austin, USA
JOYDEEP BISWAS, The University of Texas at Austin, USA
IŞIL DILLIG, The University of Texas at Austin, USA

Abstract. This paper presents a new synthesis-based approach for solving the Learning from Demonstration (LfD) problem in robotics. Given a set of user demonstrations, the goal of *programmatic LfD* is to learn a policy in a programming language that can be used to control a robot's behavior. We address this problem through a novel program synthesis algorithm that leverages two key ideas: First, to perform fast and effective generalization from user demonstrations, our synthesis algorithm views these demonstrations as *strings* over a finite alphabet and abstracts programs in our DSL as *regular expressions* over the same alphabet. This regex abstraction facilitates synthesis by helping infer useful program sketches and pruning infeasible parts of the search space. Second, to deal with the large number of object types in the environment, our method leverages a Large Language Model (LLM) to guide search. We have implemented our approach in a tool called PROLEX and present the results of a comprehensive experimental evaluation on 120 benchmarks involving 40 unique tasks in three different environments. We show that, given a 120 second time limit, PROLEX can find a program consistent with the demonstrations in 80% of the cases. Furthermore, for 81% of the tasks for which a solution is returned, PROLEX is able to find the ground truth program with just one demonstration. To put these results in perspective, we conduct a comparison against two baselines and show that both perform much worse.

1 INTRODUCTION

Learning From Demonstration (LfD) is an attractive paradigm for teaching robots how to perform novel tasks in end-user environments [Argall et al. 2009]. While most classical approaches to LfD are based on black-box behavior cloning [Ho and Ermon 2016; Ly and Akhloufi 2021], recent work has argued for treating LfD as a programmatic policy synthesis problem [Holtz et al. 2020a; Xin et al. 2023]. In particular, *programmatic LfD* represents the space of robot policies in a domain-specific language (DSL) and learns a DSL program that is consistent with the user's demonstrations.

While this programmatic approach has been shown to offer several advantages over black-box behavior cloning in terms of data-efficiency, generalizability and interpretability [Holtz et al. 2021; Verma et al. 2018], existing work in this space suffers from three key shortcomings: First, prior techniques can only synthesize simple decision-list like programs that choose an action to be taken in a given state; hence, they can only synthesize *Markovian* robot policies. Second, these techniques have only been applied to restricted domains with limited object and interaction types, such as robot soccer playing where the entities of interest are known a-priori and comprise a small set.

In this paper, we propose a new programmatic LfD approach that can be used to teach robots how to perform *long-horizon tasks* [Gao et al. 2023], such as complex household chores. For example, given a single demonstration of the user going to some subset of rooms in the house and collecting some of the towels from those rooms and taking them to a specific other room in the house, the robot should be able to synthesize a programmatic policy to *visit all bathrooms, check for towels on the floor, and bring any such towels to the laundry room.*[1]

The class of programmatic LfD tasks considered in this paper impose several challenges that are not addressed in prior work. First, long-horizon tasks necessitate a richer DSL that is capable of expressing repetitive actions and reasoning about properties of objects as well as relationships between them. Second, since the target tasks involve complex environments with a large number

[1]Our approach does not use natural language as inputs or representation, but we use English descriptions to aid exposition.

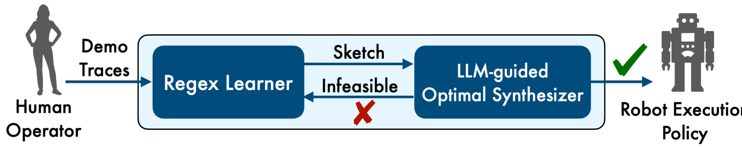


Fig. 1. Overview of PROLEX

of objects, directly applying syntax-guided synthesis [Alur et al. 2015a] is not computationally tractable in this setting. Third, since the demonstrations are *traces* of a target program, there may be a large number of possible programmatic structures, including nested loops that may be consistent with the demonstrations. We propose to overcome these challenges using a novel program synthesis algorithm that leverages two key insights:

(1) ***Traces as strings; Programs as regexes:*** To perform fast and effective generalization from user demonstrations, our synthesis algorithm views these demonstrations as *strings* over a finite alphabet and abstracts programs in our DSL as *regular expressions* over the same alphabet. This insight allows reducing the problem of learning the target program's control flow structure to the task of learning regular expressions from positive string examples. Hence, our approach can use off-the-shelf regex learning techniques to effectively learn a useful *program sketch*. Our approach further leverages this duality to aid sketch completion and to prune significant parts of the search space that the synthesizer would otherwise need to explore.

(2) ***Reasoning about semantic object relations using LLMs:*** In many robotics tasks, there are a very large number of object types in the environment and an even larger number of relations that can hold between them. Thus, even if we have a detailed sketch of the target program, the number of possible completions of that sketch can be prohibitively large. Our approach tackles this problem by leveraging a Large Language Model (LLM) to query for semantic relations between entities and actions and uses such information to prioritize likely programs.

Figure 1 presents a schematic illustration of our proposed approach based on the above insights. Given a demonstration performed by the user, our approach first infers a suitable *program sketch* that captures the high-level control flow structure of the program, including loops and conditionals. To generate such a program sketch, our method first treats the user demonstrations as positive string examples, then learns a regex that can match all of these strings, and finally inserts any necessary variable bindings and *perception* operations to ensure that the resulting sketch is realizable. In the second sketch completion phase, our method guides top-down enumerative search by using a large language model to reason about semantic relationships between entities that may occur in the program. Specifically, the algorithm iteratively refines the sketch until it finds a complete program that is consistent with the demonstrations. In each refinement step, the LLM is used to produce a probability distribution over refinements, and the most likely candidates are prioritized when performing future refinement steps. Additionally, each refinement step utilizes a *trace compatibility checker* to ensure that the proposed refinement will not lead to a dead end. This trace compatibility check also utilizes the duality between programs and regexes and constructs a regular expression abstraction of the program under a given environment.

We have implemented the proposed LfD technique in a tool called PROLEX[2] and evaluate it on a benchmark set containing 120 long-horizon robotics tasks involving household activities. Given a 2 minute time limit, our approach can complete 80% of the synthesis tasks, and can handle

[2]**P**rogramming **RO**bots with **L**anguage models and regular **EX**pressions

tasks that require multiple loops with several conditionals as well as environments with up to *thousands* of objects and dozens of object types. Furthermore, for 81% learning tasks that PROLEX is able to complete within the 2 minute time limit, PROLEX learns a program that matches the ground truth from just a *single* demonstration. To put these results in context, we compare our approach against two relevant baselines, including a state-of-the-art SyGuS solver and a neural program synthesizer, and experimentally demonstrate the advantages of our approach over other alternatives. Furthermore, we report the results of a series of ablation studies and show that our proposed ideas contribute to successful synthesis.

In summary, this paper makes the following contributions:

- We introduce a generic DSL to capture long-horizon robotic tasks and define the programmatic LfD problem based on this DSL.
- We propose a novel synthesis technique that leverages two key insights to tackle the unique challenges of this domain. In particular, we propose abstracting programs as regular expressions to (a) learn program sketches from demonstrations, and (b) identify dead-ends in the search space. We also propose guiding synthesis using an LLM to reason about semantic relationships between objects and actions in a rich environment.
- We implement these ideas in a tool called PROLEX and evaluate its efficacy in the context of 120 benchmarks involving 40 unique household chores in three environments. PROLEX can complete the synthesis task within 2 minutes for 80% of the benchmarks, and, for 81% of the completed tasks, PROLEX is able to learn the ground truth program from just a *single* demonstration.

2 MOTIVATING EXAMPLE

In this section, we present a motivating example to illustrate our programmatic LfD approach. Imagine a hotel worker who wants to instruct a robot to collect dirty sheets from guest rooms and place them in a laundry bin in the room. The goal of LfD is to teach this task through demonstrations rather than explicitly programming the robot. We formalize user demonstrations in a form that is amenable to be captured using smart hand-held devices, similar to existing end-user robots like the iRobot Roomba [iRobot 2023] and Amazon Astro [Lee et al. 2023].

User Demonstration. For the above task, suppose that the hotel worker performs a demonstration consisting of the 12 actions shown in the left side of Figure 2a. The demonstration takes place in two rooms, r_1 and r_2; Figure 3 shows the state of these two rooms before the demonstration takes place. Each room contains a large number of objects, including a bed, a laundry bin, and a few sheets on the bed. In particular, there is a clean sheet (s_2) and a dirty sheet (s_1) on the bed in r_1, and there are two dirty sheets (s_3 and s_4) on the bed in r_2. Note that the complete representation of the rooms includes many more object types and properties, which are omitted from the figure due to space constraints. The first five steps of the demonstration sequence shown in Figure 2a (left) correspond to the actions performed in r_1, and the remaining seven steps indicate the actions performed in r_2. Specifically, $goto(l)$ indicates going to the location l, and $act(a, \overline{o})$ indicates performing a specific action a on objects $\overline{o}$. Hence, in our example demonstration, the user first visits room r_1, where they open bin bn_1, grab and place sheet s_1 in that bin, and finally close the bin. Next, they go to the second room, r_2, and repeat a similar sequence of actions with the bin bn_2, and sheets s_3 and s_4.

Desired Output. Since our goal is to perform *programmatic* LfD, we wish to learn a programmatic *robot execution policy* from the provided demonstration. Figure 2d shows the desired policy that generalizes from the user's only demonstration. Intuitively, this policy encodes that the robot should go to each room (line 2), identify a bin and all beds present in that room, collect all *dirty* sheets from the top of each bed (line 8), and place them in the bin (line 9). The program also contains

```
1: goto(r1)
2: act(open, bn1)
3: act(grab, s1)
4: act(put-in, s1, bn1)
5: act(close, bn1)
6: goto(r2)
7: act(open, bn2)
8: act(grab, s3)
9: act(put-in, s3, bn2)
10: act(grab, s4)
11: act(put-in, s4, bn2)
12: act(close, bn2)
```

$\xrightarrow{\alpha}$

$$\mathcal{G}_{room}\ \mathcal{A}_{open,bin}\ \mathcal{A}_{grab,sheet}\ \mathcal{A}_{put,sheet,bin}\ \mathcal{A}_{close,bin}\ \mathcal{G}_{room}\ \mathcal{A}_{open,bin}\ \mathcal{A}_{grab,sheet}\ \mathcal{A}_{put,sheet,bin}\ \mathcal{A}_{grab,sheet}\ \mathcal{A}_{put,sheet,bin}\ \mathcal{A}_{close,bin}$$

(a) Trace of demonstrated actions (left) and the abstracted string using function α (right)

$$(\mathcal{G}_{room}\mathcal{A}_{open,bin}(\mathcal{A}_{grab,sheet}\mathcal{A}_{put,sheet,bin})^*\mathcal{A}_{close,bin})^*$$

$$(\mathcal{G}_{room}(\mathcal{A}_{open,bin})?((\mathcal{A}_{grab,sheet}\mathcal{A}_{put,sheet,bin})?)^*\mathcal{A}_{close,bin})^*$$

$$(\mathcal{G}_{room}\mathcal{A}_{open,bin}(((\mathcal{A}_{grab,sheet}\mathcal{A}_{put,sheet,bin})?)^*)^*\mathcal{A}_{close,bin})^*$$

C1 L3 L2 L1

(b) Regexes learned from the string abstracted from the demonstration. The correct regex (bottom) is highlighted to show looping and conditional structures.

```
L1    1: foreach(v1 ∈ scanLoc(??)){
      2:   goto(?? : room);
P1    3:   let v2 := getFirst(scanObj(??));
      4:   actUnary(open, ?? : bin);
L2    5:   foreach(v3 ∈ scanObj(??)){
L3    6:     foreach(v4 ∈ scanObj(??)){
C1    7:       if(??){
      8:         actUnary(grab, ?? : sheet);
      9:         actBinary(put-in, ?? : sheet, ?? : bin);
     10:       }
     11:     }
     12:   }
     13:   actUnary(close, ?? : bin);
     14: }
```

(c) Inferred Sketch

```
1: foreach(v1 ∈ scanLoc(room)){
2:   goto(v1);
3:   let v2 := getFirst(scanObj(bin));
4:   actUnary(open, v2);
5:   foreach(v3 ∈ scanObj(bed)){
6:     foreach(v4 ∈ scanObj(sheet)){
7:       if(checkProp(dirty, v4) ∧ checkRel(on-top-of, v4, v3)){
8:         actUnary(grab, v4);
9:         actBinary(put-in, v4, v2);
10:      }
11:    }
12:  }
13:  actUnary(close, v2);
14: }
```

(d) Synthesized Program

Fig. 2. Motivating Example

perception primitives that enable the robot to become aware of its environment (lines 1, 3, 5, and 6). In particular, the function scanObj(τ) allows the robot to identify all objects of type τ that are visible from its current location and reason about their properties and relations. Function scanLoc(τ) is similar but returns all locations of type τ. Synthesis of appropriate perception primitives is crucial for effective LfD, since the robot must be able to observe and reason about the state of objects in new and unseen environments.

Loc	State	
r_1	*Objs*	$bed\colon[b_1],\ sheet\colon[s_1,s_2],\ bin\colon[bn_1],\ldots$
	Props	$(s_1, dirty), (bn_1, closed),\ \ldots$
	Rels	$(s_1, on\text{-}top\text{-}of, b_1),\ (s_2, on\text{-}top\text{-}of, b_1),\ \ldots$
r_2	*Objs*	$bed\colon[b_2],\ sheet\colon[s_3,s_4],\ bin\colon[bn_2],\ldots$
	Props	$(s_3, dirty), (s_4, dirty), (bn_2, closed),\ \ldots$
	Rels	$(s_3, on\text{-}top\text{-}of, b_2),\ (s_4, on\text{-}top\text{-}of, b_2),\ \ldots$

Fig. 3. Partial representation of the initial environment.

Synthesis Challenges. In this context, generating the desired policy from the user's demonstration is challenging for several reasons: (1) the desired program has complex control flow, with three nested loops and an if statement with multiple predicates, (2) the desired program requires performing appropriate perception actions that have no correspondence in the given demonstrations, and (3) the desired program requires reasoning about high-level concepts that are also not indicated in the demonstrations, such as being dirty or being on top of some other object. Additionally, observe that the synthesized program needs to refer to object types (e.g., bed) that are not involved in the demonstration. Hence, the synthesizer cannot *only* consider those objects in the demonstration, as the desired program could refer to any of the entities in the environment. This means that the difficulty of the synthesis task is inherently sensitive to the complexity of the environment.

The PROLEX Approach. Our approach tackles the above challenges through two novel ideas: First, given the user demonstration, our approach generates a set of *sketches* of the target program by reducing the sketch inference problem to synthesis of regexes from positive strings. Second, our method employs a novel LLM-guided *sketch completion* algorithm to figure out the missing pieces.

Sketch generation. Here we give a brief overview of our sketch generation approach. Our method first *abstracts* the user's demonstrations as a set of strings. For instance, in our running example, Figure 2a (right) shows the string abstracted from the user's sole demonstration using an abstraction function α, where $\mathcal{G}$ and $\mathcal{A}$ denote *goto* and *act* in the demonstration, and the subscripts indicate the *type* of their arguments. For example, even though the demonstration specifies that the user grabbed specific sheets (namely, s_1, s_3, and s_4), the string abstraction omits such details and represents all three instances of this action using $\mathcal{A}_{grab,sheet}$. Each character in the abstract string is highlighted using a different color to visually aid the reader. The idea of converting the demonstration to a more abstract form is a crucial first step towards generalization.

Next, our approach utilizes existing techniques to synthesize a regex that matches the string encoding of the demonstrations. For our running example, Figure 2b presents three regular expressions that all match the string abstraction of the given demo. Note that there is an obvious parallel between regex operators and the program's control flow: Since Kleene star denotes repetition, it naturally corresponds to a looping construct in our DSL. Similarly, since the optional operator (i.e., $(r)?$) denotes choice, it is naturally translated into an `if` statement in our DSL. Thus, given a candidate regex r for the demonstrations, our approach translates it into a sketch by utilizing syntax-translated translation rules to convert regex operators to the target program's control flow. For instance, the third regular expression shown in Figure 2b is translated to the sketch shown in Figure 2c, where the three nested loops and the conditional block are marked using the same colored dashed lines both in the regex and in the sketch. Additionally, because the program should not manipulate any objects before it perceives them, our sketch generation procedure also inserts any necessary perception primitives to the sketch. For instance, the translated program sketch in Figure 2c contains an inferred let binding (labelled `P1`), since the object of type *bin* at line 4 cannot be acted upon before it is first perceived by the robot.

Sketch completion. For each sketch generated in the first step, our method tries to find a completion that matches the user's demonstration. In practice, several of these sketches are infeasible. For example, consider the first regex presented above in Figure 2b that matches the string abstraction of the demonstration. This regex does not include the optional operator ? used in the correct regex. Intuitively, sketches generated from this regex will be infeasible because, without an if statement in the sketch, the resulting programs would end up grabbing *both* sheets in room r_1, rather than only the single dirty sheet. Hence, the second phase of our technique considers multiple sketches and tries to find a completion of *any* sketch that is consistent with the given demonstrations.

Our sketch completion algorithm is based on top-down enumerative search but (a) utilizes a novel trace compatibility checking procedure to quickly detect dead-ends and (b) leverages an LLM to guide exploration. In particular, starting from a sketch, the synthesis algorithm maintains a worklist of partial programs containing holes to be filled. When dequeuing a partial program from the worklist, we consider its probability according to the LLM, so more promising partial programs are prioritized compared to less likely ones. Going back to our example, consider a partial program where we test the color of a bedsheet without putting it in the bin. Because the concepts "laundry bin" and "cleanliness" are much more related than than "color" and "laundry bin", our technique prioritizes a program that includes the conditional `checkProp`(*dirty*, ...) over one that is based on color. As mentioned earlier, this strategy not only allows *faster* synthesis but also makes it more likely that the generated program will match the user's demonstrations.

τ_l	$\in$	Location Type*	:=	$\{bedroom,\ kitchen,\ basement,\ mailroom,\ \dots\}$
τ_o	$\in$	Object Type*	:=	$\{plate,\ drawer,\ dishwasher,\ sink,\ \dots\}$
p	$\in$	Property*	:=	$\{empty,\ broken,\ green,\ dry,\ \dots\}$
r	$\in$	Relation*	:=	$\{inside\text{-}of,\ next\text{-}to,\ on\text{-}top\text{-}of,\ belongs\text{-}to,\ \dots\}$
a	$\in$	Action*	:=	$\{grab,\ open,\ pour\text{-}into,\ scrub\text{-}with,\ put\text{-}in,\ \dots\}$
ρ	$\in$	Item List	:=	$\texttt{scanObj}(\tau_o) \mid \texttt{scanLoc}(\tau_l)$
ϕ	$\in$	Conditional	:=	$\texttt{checkProp}(p, v) \mid \texttt{checkRel}(r, v_1, v_2) \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi$
π	$\in$	Program	:=	$\texttt{actUnary}(a, v) \mid \texttt{actBinary}(a, v_1, v_2) \mid \texttt{goto}(v) \mid \texttt{if}(\phi)\{\pi\} \mid \texttt{skip} \mid$ $\texttt{foreach}(v \in \rho)\{\pi\} \mid \texttt{let}\ v := \texttt{getNth}(\rho, n) \mid \pi; \pi$

Fig. 4. DSL syntax where v denotes variables and n is a natural number. Rules marked with * domain-specific.

In addition to utilizing an LLM, our sketch completion method also leverages the demonstration to reason about *compatibility* between the partial programs and the demo trace. As an example, consider the variant of the sketch from Figure 2c but without the conditional at line 7, and suppose that the algorithm has already instantiated the question marks at lines 1 and 6 with *room* and *sheet* respectively. Because none of the actions in the sketch modify object locations and because there are two sheets in each room, any completion of this partial program would end up performing the *grab* action *at least four times* but there are only three *grab* actions in the trace. Hence, this partial program is a dead end, and our approach can detect its infeasibility by constructing a regex abstraction of the partial program (for this specific environment) and checking whether it accepts the demo trace.

3 ROBOT EXECUTION POLICIES

In this section, we introduce a domain-specific language (DSL) for programming robots and provide a formal definition of the robot learning from demonstrations (LfD) problem.

3.1 Syntax

The syntax of the our DSL is presented in Figure 4. A robot program (π) contains functions to perform various operations on a single object (`actUnary`) or a pair of objects (`actBinary`). The robot can move between locations using the `goto` function. The robot becomes aware of its location by scanning the environment using `scanLoc`, and it becomes aware of objects in its current location using the `scanObj` function. The result of running a scan operation is an ordered list of location or object instances (denoted by ρ) of the specified type τ. For example, `scanObj`(*plate*) yields all plates at the current location of the robots. Specific elements in the scan result can be bound to variables using a restricted `let` binding of the form `let` $v :=$ `getNth`(ρ, n). This expression introduces a new variable v and assigns the n'th element of list ρ to v. As standard, the DSL also contains typical conditional and looping control structures. Conditional expressions check properties of objects (`checkProp`), relationships between them (`checkRel`), as well as their boolean compositions.

Note that the DSL presented in Figure 4 is parametrized over a set of domain-specific terminals, indicated by an asterisk. For example, location types τ_l are not fixed and can vary based on the target application domain. For example, for robot execution policies targeting household chores, locations might be kitchen, living room, basement etc. Similarly, object types, properties, relations, and actions are also domain-specific and can be customized for a given family of tasks.

3.2 Operational Semantics

In this section, we present the operational semantics of our robot DSL using the small-step reduction relation $\Rightarrow$ shown in Figure 5. This relation formalizes how the robot interacts with its environment while executing the program. Specifically, the relation $\Rightarrow$ is defined between tuples of the form $(\pi, \mathcal{E}, \sigma, t)$, where π is a program, $\mathcal{E}$ is the robot's execution environment, σ is a valuation (mapping

$$\textsc{(sequence)}\quad \frac{\pi, \mathcal{E}, \sigma, t \Rightarrow \pi', \mathcal{E}', \sigma', t'}{\pi; \pi'', \mathcal{E}, \sigma, t \Rightarrow \pi'; \pi'', \mathcal{E}', \sigma', t'} \qquad \textsc{(skip)}\quad \frac{}{\texttt{skip}; \pi, \mathcal{E}, \sigma, t \Rightarrow \pi, \mathcal{E}, \sigma, t}$$

$$\textsc{(if-t)}\quad \frac{\phi \Downarrow_{\mathcal{E},\sigma} \top}{\texttt{if}(\phi)\{\pi\}, \mathcal{E}, \sigma, t \Rightarrow \pi, \mathcal{E}, \sigma, t} \qquad \textsc{(if-f)}\quad \frac{\phi \Downarrow_{\mathcal{E},\sigma} \bot}{\texttt{if}(\phi)\{\pi\}, \mathcal{E}, \sigma, t \Rightarrow \texttt{skip}, \mathcal{E}, \sigma, t}$$

$$\textsc{(act-unary)}\quad \frac{\begin{array}{c}\pi = \texttt{actUnary}(a, v) \\ o = \sigma(v) \qquad \mathcal{E} \xrightarrow{a,o} \mathcal{E}' \\ t' = act(a, o)\end{array}}{\pi, \mathcal{E}, \sigma, t \Rightarrow \texttt{skip}, \mathcal{E}', \sigma, t; t'}$$

$$\textsc{(act-binary)}\quad \frac{\begin{array}{c}\pi = \texttt{actBinary}(a, v_1, v_2) \\ o_1 = \sigma(v_1) \qquad o_2 = \sigma(v_2) \\ \mathcal{E} \xrightarrow{a,o_1,o_2} \mathcal{E}' \qquad t' = act(a, o_1, o_2)\end{array}}{\pi, \mathcal{E}, \sigma, t \Rightarrow \texttt{skip}, \mathcal{E}', \sigma, t; t'}$$

$$\textsc{(goto)}\quad \frac{\begin{array}{c}\pi = \texttt{goto}(v) \\ l = \sigma(v) \qquad \mathcal{E}' = \mathcal{E}[\ell \mapsto l] \\ t' = goto(l)\end{array}}{\pi, \mathcal{E}, \sigma, t \Rightarrow \texttt{skip}, \mathcal{E}', \sigma, t; t'}$$

$$\textsc{(let-obj)}\quad \frac{\begin{array}{c}\pi = \texttt{let } v := \texttt{getNth}(\texttt{scanObj}(\tau_o), n) \\ \sigma' = \sigma[v \mapsto \mathcal{E}.\texttt{objs}(\ell, \tau_o)[n]]\end{array}}{\pi, \mathcal{E}, \sigma, t \Rightarrow \texttt{skip}, \mathcal{E}, \sigma', t}$$

$$\textsc{(foreach-obj)}\quad \frac{\begin{array}{c}n = |\mathcal{E}.\texttt{objs}(\ell, \tau_o)| \qquad \forall_{0 \le i < n}.\ \sigma_i = \sigma[v \mapsto \mathcal{E}.\texttt{objs}(\ell, \tau_o)[i]] \\ \mathcal{E}_0 = \mathcal{E} \qquad t_0 = t \qquad \forall_{0 \le i < n}.\ \pi, \mathcal{E}_i, \sigma_i, t_i \Rightarrow \texttt{skip}, \mathcal{E}_{i+1}, \_, t_{i+1}\end{array}}{\texttt{foreach}(v \in \texttt{scanObj}(\tau_o))\{\pi\}, \mathcal{E}, \sigma, t \Rightarrow \texttt{skip}, \mathcal{E}_n, \sigma, t_n}$$

$$\textsc{(let-loc)}\quad \frac{\begin{array}{c}\pi = \texttt{let } v := \texttt{getNth}(\texttt{scanLoc}(\tau_l), n) \\ \sigma' = \sigma[v \mapsto \mathcal{E}.\texttt{locs}(\tau_l)[n]]\end{array}}{\pi, \mathcal{E}, \sigma, t \Rightarrow \texttt{skip}, \mathcal{E}, \sigma', t}$$

$$\textsc{(foreach-loc)}\quad \frac{\begin{array}{c}n = |\mathcal{E}.\texttt{locs}(\tau_l)| \qquad \forall_{0 \le i < n}.\ \sigma_i = \sigma[v \mapsto \mathcal{E}.\texttt{locs}(\tau_l)[i]] \\ \mathcal{E}_0 = \mathcal{E} \qquad t_0 = t \qquad \forall_{0 \le i < n}.\ \pi, \mathcal{E}_i, \sigma_i, t_i \Rightarrow \texttt{skip}, \mathcal{E}_{i+1}, \_, t_{i+1}\end{array}}{\texttt{foreach}(v \in \texttt{scanLoc}(\tau_l))\{\pi\}, \mathcal{E}, \sigma, t \Rightarrow \texttt{skip}, \mathcal{E}_n, \sigma, t_n}$$

Fig. 5. Operational semantics. Relation $\Downarrow$ is defined in Figure 6, and $\rightarrow$ is given in Appendix A.

variables to their values), and t is a program trace. In more detail, the environment $\mathcal{E}$ is a quadruple $(\mathcal{L}, \mathcal{O}, \ell, \mathcal{I})$ where $\mathcal{L}$ is a set of typed location identifiers; $\mathcal{O}$ is a mapping from (typed) object identifiers to their corresponding location; ℓ is the current location of the robot; and $\mathcal{I}$ is an interpretation for all the relation symbols. That is, for a relation p, $\mathcal{I}(p)$ yields the set of tuples of objects for which p evaluates to true. Given object type τ_o and a location l, we write $\mathcal{E}.\texttt{objs}(l, \tau_o)$ to denote the list of all objects that are at location l and that have type τ_o. Similarly, given a location type τ_l, the list of locations of this type is denoted by $\mathcal{E}.\texttt{locs}(\tau_l)$. Note that running a robot execution policy can modify the environment – for example, the location of the robot or some properties of an object can change after executing π. Finally, a trace is a sequence of actions performed by the robot. Robot actions are denoted using $act(a, \bar{o})$ and $goto(l)$, where a is an action that was performed on objects $\bar{o}$, and l is a location that the robot visited.

With the above notations in place, we now explain the operational semantics from Figure 5 in more detail. The first rule, labeled (SEQUENCE), defines how the robot takes a step by executing the first statement in the program. The next rule (SKIP) defines the semantics of executing a `skip` statement, which has no effect on the execution state. The rules (IF-T) and (IF-F) describe the flow of the program when a conditional statement $\texttt{if}(\phi)\{\pi\}$ is executed. First, the Boolean expression ϕ is evaluated to $\bot$ or $\top$, depending on the result, the program either skips or executes π. Boolean expressions are evaluated using the relation $\Downarrow$ defined in Figure 6. This relation is parameterized by the execution environment $\mathcal{E}$ and valuation σ.

The (ACT-UNARY) and (ACT-BINARY) rules specify the outcomes of executing a unary and a binary action, respectively, using the auxiliary relation $\rightarrow \subseteq \mathcal{E} \times \mathcal{E}$. Given an environment $\mathcal{E}$, an action a and affected object instance(s), the relation $\rightarrow$ formalizes how the environment $\mathcal{E}$ is modified based on the semantics of action a. Since our DSL is parameterized over the set of actions, we do not discuss the $\rightarrow$ relation in detail in the main body of the paper and refer the interested reader to Appendix A for a representative subset of actions used in our evaluation.

$$\text{(CHECK\_PROP\_T)}\ \frac{\mathbf{o} \in \mathcal{I}(p)}{\texttt{checkProp}(p, v) \Downarrow_{\mathcal{E},\sigma} \top} \qquad \text{(CHECK\_PROP\_F)}\ \frac{\sigma(v) \notin \mathcal{I}(p)}{\texttt{checkProp}(p, v) \Downarrow_{\mathcal{E},\sigma} \bot} \qquad \text{(CHECK\_REL\_T)}\ \frac{(\sigma(v_1), \sigma(v_2)) \in \mathcal{I}(r)}{\texttt{checkRel}(r, v_1, v_2) \Downarrow_{\mathcal{E},\sigma} \top} \qquad \text{(CHECK\_REL\_F)}\ \frac{(\sigma(v_1), \sigma(v_2)) \notin \mathcal{I}(r)}{\texttt{checkRel}(r, v_1, v_2) \Downarrow_{\mathcal{E},\sigma} \bot}$$

$$\text{(NEGATION)}\ \frac{\phi \Downarrow_{\mathcal{E},\sigma} b}{\neg\phi \Downarrow_{\mathcal{E},\sigma} \neg b} \qquad \text{(CONJUNCTION)}\ \frac{\phi_1 \Downarrow_{\mathcal{E},\sigma} b_1 \quad \phi_2 \Downarrow_{\mathcal{E},\sigma} b_2}{\phi_1 \land \phi_2 \Downarrow_{\mathcal{E},\sigma} b_1 \land b_2} \qquad \text{(DISJUNCTION)}\ \frac{\phi_1 \Downarrow_{\mathcal{E},\sigma} b_1 \quad \phi_2 \Downarrow_{\mathcal{E},\sigma} b_2}{\phi_1 \lor \phi_2 \Downarrow_{\mathcal{E},\sigma} b_1 \lor b_2}$$

Fig. 6. Boolean expression evaluation.

The GOTO rule rule defines the effect of executing a $\texttt{goto}(v)$ statement, where the environment is updated to reflect the robot's new location, and a new trace element $goto(l)$ is generated and appended to the existing trace, where l is the location stored in variable v. Next, the rules LET-OBJ and LET-LOC, define the semantics of the $\texttt{let}\ v := \texttt{getNth}(\rho, n)$ statement, which assigns to variable v the n^{th} element of list ρ obtained via either `scanObj` or `scanLoc`. Specifically, $\texttt{scanObj}(\tau_o)$ yields all objects of type τ_o that are present at the robot's current location, and $\texttt{scanLoc}(\tau_l)$ yields all locations of type τ_l. The rules (FOREACH-OBJ) and (FOREACH-LOC) describe the semantics of loops of the form $\texttt{foreach}(v \in \rho)\{\pi\}$, where ρ is the result of a scan operation. As expected, these rules iteratively bind v to each of the elements in ρ and execute the loop body π under this new valuation.

Finally, we use the $\Rightarrow$ relation to define the semantics of executing a policy π on environment $\mathcal{E}$. Given $\mathcal{E}$ and robot execution policy π, we write $\pi(\mathcal{E}) = t$ iff $\pi, \mathcal{E}, \text{Nil}, \text{Nil} \Rightarrow \texttt{skip}, \_, \_, t$ where Nil denotes an empty list/mapping.

3.3 Problem Statement

In this section, we formally define the LfD problem that we address in this paper. Informally, given a set of demonstrations $\mathcal{D}$ our LfD problem is to find a robot execution policy π^* (in the DSL of Figure 4) such that π is consistent with $\mathcal{D}$. To make the notion of consistency more precise, we represent a demonstration δ as a pair $(\mathcal{E}, t)$ where $\mathcal{E}$ is the initial environment and t is a trace of the user's demonstration in this environment.

Definition 3.1. **(Consistency with demonstration)** We say that a robot execution policy π^* is consistent with a demonstration $\delta = (\mathcal{E}, t)$, denoted $\pi \models \delta$, iff $\pi(\mathcal{E}) = t$.

We also extend this notion of consistency to a set of demonstrations $\mathcal{D}$, and we write $\pi \models \mathcal{D}$ iff $\pi \models \delta$ for every demonstration δ in $\mathcal{D}$. We can now formalize our problem statement as follows:

Definition 3.2. **(Programmatic LfD)** Given a set of demonstrations $\mathcal{D}$, the *programmatic LfD* problem is to find a robot execution policy π^* such that $\pi^* \models \mathcal{D}$.

4 SYNTHESIS ALGORITHM

In this section, we present our synthesis technique for solving the programmatic LfD problem defined in the previous section. We start by giving an overview of the top-level algorithm and then describe each of its key components in more detail.

4.1 Top-Level Algorithm

Our top-level learning procedure is presented in Algorithm 1. This algorithm takes as input a set of demonstrations $\mathcal{D}$ and returns a policy π such that for all $\delta \in \mathcal{D}$, we have $\pi \models \delta$. If there is no programmatic policy that is consistent with all demonstrations, the algorithm returns $\bot$.

The synthesis procedure starts by constructing an *abstraction* of each demonstration $\delta \in \mathcal{D}$ as a string over the alphabet $\Sigma = \{\mathcal{G}_\tau, \mathcal{A}_{a,\tau}, \mathcal{A}_{a,\tau,\tau'}\}$ where τ, τ' indicate location and object types (e.g.,

Algorithm 1: Top-level Synthesis Algorithm

Input: *A set of demonstrations* $\mathcal{D}$*, a statistical completion model* θ
Output: *A policy consistent with the demonstrations or* $\perp$ *if none exists*

```
1:  Synthesize(𝒟, θ)
2:    A := {α(t) | (_, t) ∈ 𝒟}   # get the abstraction of given traces using function α
3:    while (true) {
4:      r := GetNextRegex(A)   # get a regular expression that matches the abstractions of all demos
5:      if(r = ⊥) break
6:      while (true) {
7:        s := GetNextSketch(r)   # get the next lazily generated sketch from r
8:        if(s = ⊥) break
9:        π := CompleteSketch_θ(s, 𝒟)   # search for a consistent completion of s and return the result
10:       if(π ≠ ⊥) return π
11:     }
12:   }
13:   return ⊥
```

$$
\begin{array}{llll}
\rho_s & \in & \text{Item List} & := \ \texttt{scanObj}(??_{\tau_o}) \mid \texttt{scanLoc}(??_{\tau_l}) \\
s & \in & \text{Sketch} & := \ \texttt{actUnary}(a, ??_v : \tau_o) \mid \texttt{actBinary}(a, ??_v : \tau_o, ??_v : \tau_o') \mid \texttt{goto}(??_v) \mid \texttt{if}(??_\phi)\{s\} \mid \\
 & & & \quad \ \texttt{foreach}(v \in \rho_s)\{s\} \mid \texttt{let } v := \texttt{getNth}(\rho_s, ??_n) \mid s; s
\end{array}
$$

Fig. 7. Syntax of Program Sketches. Domain specific definitions (i.e., τ_o, τ_l, a) are identical to Figure 4.

plate, *kitchen*) and a denotes a specific type of action (e.g., *grab*). This abstraction is performed at line 2 of Algorithm 1 using the function α, defined as follows:

$$\alpha(goto(l)) := \mathcal{G}_{\tau_l} \quad \alpha(act(a, o) := \mathcal{A}_{a,\tau_o} \quad \alpha(act(a, o, o')) := \mathcal{A}_{a,\tau_o,\tau_o'}$$

where τ_l, τ_o denote the type of location l and object o respectively. In other words, when abstracting a trace as a string, the algorithm replaces specific object instances with their corresponding types. Intuitively, this abstraction captures the commonality between different actions in the trace, allowing generalization from a specific sequence of actions to a more general program structure.

Next, given the string abstraction A of the demonstrations $\mathcal{D}$, the loop in lines 3–12 alternates between the following key steps:

- **Regex synthesis:** The GetNextRegex procedure at line 4 finds a regular expression r matching all strings in A. Intuitively, this regex captures the main control flow structure of the target program and can be used to generate a set of program sketches.
- **Sketch generation:** The inner loop in lines 6–10 translates a given regex to a *set* of program sketches. As shown in Figure 7, a sketch has almost the same syntax as programs in our DSL except that the arguments of most constructs are unknown, as indicated by question marks. In particular, note that (1) the types of objects and locations being scanned are unknown, (2) predicates of `if` statements are yet to be determined, and (3) the specific objects and locations being acted on are also unknown (although their *types* are known).
- **Sketch completion:** Given a candidate program sketch s, line 9 of the algorithm invokes CompleteSketch to find a completion π of s that is consistent with the demonstrations. If CompleteSketch does not return failure ($\perp$), the synthesized policy is guaranteed to satisfy all demonstrations; hence, Synthesize returns π as a solution at line 10.

In the remainder of this section, we describe sketch generation and sketch completion in more detail. Because learning regexes from a set of positive string examples is a well-understood problem,

$$\frac{}{\mathcal{G}_{\tau_l} \triangleright \texttt{goto}(??_v : \tau_l)} \qquad \frac{}{\mathcal{A}_{a,\tau_o} \triangleright \texttt{actUnary}(a, ??_v : \tau_o)} \qquad \frac{}{\mathcal{A}_{a,\tau_o,\tau'_o} \triangleright \texttt{actBinary}(a, ??_v : \tau_o, ??_v : \tau'_o)}$$

$$\frac{r \triangleright s}{(r)^* \triangleright \texttt{foreach}(v \in \texttt{scanLoc}(??_{\tau_l}))\,\{s\}} \qquad \frac{r \triangleright s}{(r)^* \triangleright \texttt{foreach}(v \in \texttt{scanObj}(??_{\tau_o}))\,\{s\}} \qquad \frac{r_1 \triangleright s_1 \quad r_2 \triangleright s_2}{r_1 r_2 \triangleright s_1; s_2}$$

$$\frac{r \triangleright s}{(r)? \triangleright \texttt{if}(??_\phi)\,\{s\}} \qquad \frac{r \triangleright s}{r \triangleright \texttt{let}\ v := \texttt{getNth}(\texttt{scanObj}(??_{\tau_o}), ??_n); s} \qquad \frac{r \triangleright s}{r \triangleright \texttt{let}\ v := \texttt{getNth}(\texttt{scanLoc}(??_{\tau_l}), ??_n); s}$$

Fig. 8. Regex to sketch inference rules. Double-lined rules are used to make the sketch perception-complete.

we do not describe it in this paper, as our implementation uses an off-the-shelf tool customized via some post-processing (described later in Section 5).

4.2 Sketch Inference

Given a regex r over the alphabet $\Sigma = \{\mathcal{G}_{\tau_l}, \mathcal{A}_{a,\tau_o}, \mathcal{A}_{a,\tau_o,\tau'_o}\}$, the goal of sketch inference is to (lazily) generate a set of program sketches. The inputs to the sketch inference procedure are regular expression of the following form:

$$r \quad := \quad \mathcal{A}_{a,\tau_o} \mid \mathcal{A}_{a,\tau_o,\tau'_o} \mid \mathcal{G}_{\tau_l} \mid rr \mid (r)^* \mid (r)?$$

Given such a regex, sketch inference consists of two steps:

(1) **Syntax-directed translation:** In the first step, sketch inference converts the given regex to control flow operations using syntax-directed translation. Intuitively, string concatenation is translated into to sequential composition; Kleene star corresponds to loops; and, optional regexes translate into conditionals.
(2) **Perception inference:** While the sketches generated in step (1) are syntactically valid, they may lack essential perception operations (i.e., `scanObj` and `scanLoc`). Hence, in the second step, our sketch inference procedure inserts these perception operations such that the resulting sketch is *perception-complete*, meaning that it contains at least the minimum number of required scan operation. However, since the target program may require additional `scan` operations, the second step of sketch inference yields a set of sketches that only differ with respect to the placement of these perception operations.

Figure 8 presents our syntax-directed translation rules for converting regular expressions to a syntactically valid sketch using judgments of the form $r \triangleright s$, meaning that regex r is translated to sketch s. As expected, characters $\mathcal{G}_\tau, \mathcal{A}_{a,\tau}, \mathcal{A}_{a,\tau,\tau'}$ are translated to `goto`, `actUnary`, and `actBinary` constructs respectively. The Kleene star operator is translated into a looping construct, but may iterate either over locations or objects. Finally, regex concatanation is translated into sequential composition, and $(r)?$ is translated into a conditional with an unknown predicate.

Recall that our DSL also allows `let` bindings that assign a new variable to the result of a perception operation. Since program traces (and, hence, the inferred regexes) do not contain these perception operations, the last two rules in Figure 8 (double-lined) allow inserting `let` bindings at arbitrary positions in the sketch. In particular, if r can be translated into a sketch s, then the last two rules of Figure 8 state that r can also be translated into a sketch of the form $l; s$ where l is a new `let` binding which assigns a fresh variable v to an entity that is obtained by scanning objects or locations.

In general, observe that a regex can give rise to a large number of program sketches, as we do not a priori know where to insert `let` bindings. To tackle this problem, our lazy sketch inference procedure first translates a regex into a sketch *without* using the last two rules in Figure 8 for inserting `let` bindings. In a second step, it infers where perception operations are needed and inserts let bindings according to the results of this analysis.

Algorithm 2: Sketch Completion Algorithm

Input: *A set of demonstrations $\mathcal{D}$, a sketch s*
Output: *A completion of s consistent with $\mathcal{D}$ or $\perp$ if none exists*

CompleteSketch$_\theta$$(s, \mathcal{D})$ {
  $W := [(s, 1.0)]$ # the given sketch s is the only partial program initially
  while $(W \neq \emptyset)$ {
    $(\partial, p) := W$.dequeue()   # get the partial program with the highest probability
    if (IsComplete(∂)) {   # check if there is any holes left to be filled
      if $(\partial \models \mathcal{D})$ **return** ∂ # return the completed program if it is consistent with the demos
      else continue
    if $\neg$Compatible$(\partial, \mathcal{D})$ **continue**   # check compatibility of the partial program and demos
    $h :=$ GetNextHole(∂)
    foreach $(c \in$ Fill$(\partial, h, \mathcal{D}))$ {
      $\partial' := \partial[h \mapsto c]; p' := \theta(c \mid \partial, h)$ # fill the chosen hole and assign a probability to the result
      W.enqueue$((\partial', p'))$ # add the partial program to the worklist
    }}}

This second step of our sketch inference procedure is formalized using the notion of *perception completeness*. Intuitively, a sketch is *perception complete* if the program perceives (using `scan` operations) all objects that it manipulates, before it manipulates them. If a sketch is *not* perception complete, it can never be realized into a valid program; hence, it is wasteful to consider such sketches. We formalize the notion of perception completeness using the following definition:

Definition 4.1. **(Perception Completeness)** Let $\prec$ denote a standard partial order between program points.[3] A sketch s is said to be *perception complete* if the following conditions are satisfied:

(1) for all $s_1 :=$ `actUnary`$(a, ?? : \tau_o)$ in s, there exists a $s_2 :=$ `scanObj`(τ_o) such that $s_2 \prec s_1$.
(2) for all $s_1 :=$ `actBinary`$(a, ?? : \tau_o, ?? : \tau'_o)$ in s, there exist $s_2 :=$ `scanObj`(τ_o) and $s_3 :=$ `scanObj`(τ'_o), such that $s_2 \prec s_1$ and $s_3 \prec s_1$.
(3) for all $s_1 :=$ `goto`$(?? : \tau_l)$ in s, there exists a statement $s_2 :=$ `scanLoc`(τ_l) in s such that $s_2 \prec s_1$.

Our sketch inference algorithm leverages this notion of perception completeness to lazily enumerate program sketches as follows: First, it translates a given regex into a set of sketches using the inference rules shown in Figure 8 but *without* using the last two (double-lined) rules. It then infers a minimal set of applications of the last two rules needed to make the sketch perception complete and then augments the resulting sketches with the inferred `let` bindings. Finally, because additional `let` bindings may be needed, it lazily inserts more `let` bindings (up to a bound) if the current sketch does not produce a valid completion.

4.3 Sketch Completion

We now turn our attention to the sketch completion procedure, shown in Algorithm 2, for finding a sketch instantiation that satisfies the given demonstrations. Given a sketch s and demonstrations $\mathcal{D}$, CompleteSketch either returns $\perp$ to indicate failure a policy π that is consistent with all demonstrations. Note that the sketch completion procedure is parameterized over a statistical model θ for assigning probabilities to possible sketch completions.

CompleteSketch is a standard top-down enumerative search procedure that iteratively expands partial programs until a solution is found.[4] However, our sketch completion procedure has two

[3]We refer the interested reader to the chapter 6 of "Programming Language Pragmatics" by Michael L. Scott [2000] for a detailed discussion on program orders and their usage in control flow analysis.
[4]Partial programs belongs to the grammar from Figure 4 augmented with productions $M \rightarrow ??$ for each non-terminal M.

Algorithm 3: Checking compatibility between partial programs and demonstrations

Input: *A partial program ∂, a set of demonstrations $\mathcal{D}$*
Output: *True if the given partial program is compatible with $\mathcal{D}$ and otherwise false*

1: Compatible$(\partial, \mathcal{D})$
2:   **foreach** $((\mathcal{E}, t) \in \mathcal{D})$
3:     $\partial^* :=$ PartialEval$(\partial, \mathcal{E})$ # Partially evaluate the given partial program on the demonstration environment
4:     $r :=$ ProgToRegex$(\partial^*, \alpha(\mathcal{E}))$ # Find the regex that over-approximates behaviors of ∂^*
5:     **if**$(\alpha(t) \notin r)$ **return** *false* # check if the trace is not accepted by the over-approximating regex
6:   **return** *true*

novel aspects: First, it assigns probabilities to partial programs using a large language model, and second, it uses a novel *compatibility checking* procedure for pruning the search space.

In more detail, the sketch completion procedure initializes the worklist to a singleton containing the input sketch s, with corresponding probability 1.0 (line 2). It then enters a loop (lines 3–14) where each iteration processes the highest probability partial program ∂ in the worklist. If the dequqed partial program ∂ is complete, meaning that it has no holes (line 5), the algorithm checks whether all demonstrations are satisfied (line 6). If so, ∂ is returned as a solution; otherwise, it is discarded. If ∂ is incomplete, the algorithm performs a *compatibility* check at line 8 to ensure that the partial program is compatible with provided demonstrations. This procedure is based on regular expressions and is explained in more detail later. Next, if ∂ is a compatible with the demonstrations, the algorithm considers one of the holes h in ∂ and *all* well-typed grammar productions that can be used to fill that hole. In particular, given a hole $?_N$, the procedure Fill yields a set of expressions $c_1, \ldots, c_n$ such that (1) $N \rightarrow c_i$ is a production in the grammar, and (2) replacing h with c_i can result in a well-typed program. Hence, for each such expression c_i, we obtain a new partial program ∂' at line 11 by replacing hole h in ∂ with expressions c_i.[5] However, since some completions are much more likely than others, our algorithm assigns probabilities to completions using a large language model. Hence, when dequeuing partial programs from the worklist, the algorithm prioritizes programs that are assigned a higher probability.

4.3.1 Compatibility Checking using Regular Expressions. A key component of our sketch completion procedure is a novel pruning technique, based on *regex abstractions*, for detecting partial programs that are incompatible with the provided demonstrations. As shown in Algorithm 2, this compatibility checking procedure takes as input a partial program ∂ and a set of demonstrations $\mathcal{D}$ and returns false if it can prove that there is no completion π of ∂ such that $\pi \models \mathcal{D}$.

Compatibility checking is implemented in Algorithm 3. At a high level, this algorithm iterates over all demonstrations (lines 2–5) and returns false if it can prove that ∂ is incompatible with some demonstration $(\mathcal{E}, t)$ in $\mathcal{D}$. To check compatibility with $(\mathcal{E}, t)$, the algorithm first partially evaluates ∂ on the initial environment $\mathcal{E}$ to obtain a simplified program ∂^*, as done in existing work [Feng et al. 2017]. The novel part of our technique lies in constructing a regex abstraction of the partial program ∂ under a given environment $\mathcal{E}$. Specifically, our compatibility checking procedure constructs a regex r that *over-approximates* the possible behaviors of ∂ under initial environment $\mathcal{E}$. In particular, the regex r is constructed at line 4 in such a way that if $\alpha(t)$ is not accepted by r, then no completion of ∂ can be compatible with $(\mathcal{E}, t)$.

Hence, the crux of the compatibility checking algorithm is the ProgToRegex procedure (formalized as inference rules shown in Figures 9 and 10) for generating a regex that over-approximates the behavior of ∂ under environment $\mathcal{E}$. These rules utilize the notion of an *abstract environment* which is a triple $\hat{\mathcal{E}} :=$ (CurLoc, Locs, Objs) where (1) CurLoc is a set containing all possible locations that

[5] When doing the replacement $\partial[h \mapsto c]$, all non-terminals N occurring in c are replaced with a hole $?_N$.

the robot *could* currently be at; (2) Locs is a mapping from location types to the set of locations of that type; (3) Objs is a mapping from each location to the set of objects of each type at that location (or $\top$ if unknown). Because statements in our DSL can modify the environment, this notion of abstract environment is used to conservatively capture (the possibly unknown) side effects of partial programs on the environment. The inference rules shown in Figures 9 and 10 formalize the ProgToRegex procedure using two types of judgments:

(1) **Scan rules** (shown in Figure 9) are of the form $\hat{\mathcal{E}} \vdash \texttt{scan}(...) : \Theta$, indicating that the *cardinality* of the set returned by `scan` must be some element of Θ. For example, if $\Theta = \{1, 4\}$, this means that the number of objects/locations returned by scan is either 1 or 4. On the other hand, if Θ includes the special $\star$ element, then the number of elements is unknown.
(2) **Partial program rules** (shown in Figure 10) are of the form $\hat{\mathcal{E}} \vdash \pi : \hat{\mathcal{E}}', r$, meaning that, under initial abstract environment $\hat{\mathcal{E}}$, (a) the behavior of π is over-approximated by regex r, and (b) $\hat{\mathcal{E}}'$ is a new environment that captures all possible environment states after executing π on $\hat{\mathcal{E}}$.

Before we explain these rules in detail, we first describe the high level idea, which is analogous to how we construct sketches from regexes (but slightly more complex). The basic idea is to encode (a) atomic actions using characters drawn from the (parametrized) alphabet $\{\mathcal{G}_\tau, \mathcal{A}_{a,\tau}, \mathcal{A}_{a,\tau,\tau'}\}$, (b) if statements using optional regexes, and (c) loops using regexes of the form r^n where n corresponds to the number of times the loop will execute (or as r^* if the number of loop iterations is completely unknown). With this intuition in mind, we now explain the rules shown in Figures 9 and 10.

Scan rules. There are two sets of rules for scan operations, (LOC-KNOWN) and (LOC-UNKNOWN) for scanning locations, and (OBJ-KNOWN) and (OBJ-UNKNOWN) for scanning objects. For a `scanLoc` operation, if its argument is a known location type τ_l, we simply look up the number of locations of that type from the given abstract environment. If it is unknown, we take the union over all possible location types. The rules for `scanObj` are similar: The (OBJ-KNOWN) rule handles the case where the argument is a known object type τ_o. In this case, we consider all the locations that the robot could be currently at and take the union of the number of objects of type τ_o for all of those locations. In the OBJ-UNKNOWN rule, we additionally take the union over all possible object types, since the argument of the `scanObj` operation is unknown.

Atomic actions. The rule labeled (ATOMIC) in Figure 10 deals with `goto`, `actUnary`, and `actBinary` statements and serves two roles. First, it abstracts the performed action as a letter in our regex alphabet using the abstraction function α. Second, it produces a new abstract environment $\hat{\mathcal{E}}'$ by considering all possible effects of the action on the input environment via the UpdateAbsEnv function. Since the UpdateAbsEnv function is domain-specific and depends on the types of actions of interest, we do not describe in detail but provide a few representative examples in the Appendix A.

Sequence. Sequential composition is abstracted using regex concatanation and its final effect on the environment is captured by threading the environment through the two premises of the rule.

Conditionals. As expected, if statements are abstracted using the optional operator ($(r)?$). Furthermore, since we do not know whether the predicate evaluates to true or not[6], we take the *join* of the two abstract environments. Intuitively, the join of abstract environments $\hat{\mathcal{E}}$ and $\hat{\mathcal{E}}'$, denoted by $\hat{\mathcal{E}} \sqcup \hat{\mathcal{E}}'$, is the smallest environment that over-approximates both $\hat{\mathcal{E}}$ and $\hat{\mathcal{E}}'$. An abstract environment $\hat{\mathcal{E}}$ is said to over-approximate an abstract environment $\hat{\mathcal{E}}'$, denoted by $\hat{\mathcal{E}}' \sqsubseteq \hat{\mathcal{E}}$, if and only if:

$$\forall_{l \in \hat{\mathcal{E}}.\text{Locs}} \forall_{\tau.\text{ObjTypes}(\hat{\mathcal{E}}')}.\ \hat{\mathcal{E}}'.\text{Objs}(l, \tau) \leq \hat{\mathcal{E}}.\text{Objs}(l, \tau)$$

[6] Recall that we apply partial evaluation before performing this step. Hence, if the predicate of a conditional can be fully evaluated, it will be simplified away and rewritten as conditional-free code.

$$\frac{}{\hat{\mathcal{E}} \vdash \texttt{scanLoc}(\tau_l) : \{|\hat{\mathcal{E}}.\mathsf{Locs}(\tau_l)|\}}\ (\textsc{loc-known}) \qquad \frac{\mathsf{LocTypes}(\hat{\mathcal{E}}) = \{\tau_1, \ldots, \tau_n\}}{\hat{\mathcal{E}} \vdash \texttt{scanLoc}(??) : \bigcup_{1 \le i \le n} \{|\hat{\mathcal{E}}.\mathsf{Locs}(\tau_i)|\}}\ (\textsc{loc-unknown})$$

$$\frac{\hat{\mathcal{E}}.\mathsf{CurLocs} = \{l_1, \ldots, l_n\}}{\hat{\mathcal{E}} \vdash \texttt{scanObj}(\tau_o) : \bigcup_{1 \le i \le n} \{|\hat{\mathcal{E}}.\mathsf{Objs}(l_i, \tau_o)|\}}\ (\textsc{obj-known}) \qquad \frac{\hat{\mathcal{E}}.\mathsf{CurLocs} = \{l_1, \ldots, l_n\} \quad \mathsf{ObjTypes}(\hat{\mathcal{E}}) = \{\tau_0, \ldots, \tau_k\}}{\hat{\mathcal{E}} \vdash \texttt{scanObj}(??) : \bigcup_{1 \le i \le n} \bigcup_{1 \le j \le k} \{|\hat{\mathcal{E}}.\mathsf{Objs}(l_i, \tau_j)|\}}\ (\textsc{obj-unknown})$$

Fig. 9. Over-approximation of robot perception used in ProgToRegex function.

$$\frac{\mathsf{AtomicAction}(s) \quad \hat{\mathcal{E}}' = \mathsf{UpdateAbsEnv}(\hat{\mathcal{E}}, s)}{\hat{\mathcal{E}} \vdash s : \hat{\mathcal{E}}', \alpha(s)}\ (\textsc{atomic}) \qquad \frac{\hat{\mathcal{E}} \vdash s_1 : \hat{\mathcal{E}}_1, r_1 \quad \hat{\mathcal{E}}_1 \vdash s_2 : \hat{\mathcal{E}}_2, r_2}{\hat{\mathcal{E}} \vdash s_1; s_2 : \hat{\mathcal{E}}_2, r_1 r_2}\ (\textsc{seq})$$

$$\frac{\hat{\mathcal{E}} \vdash s : \hat{\mathcal{E}}', r}{\hat{\mathcal{E}} \vdash \texttt{if}(\_)\ \{s\} : \hat{\mathcal{E}} \sqcup \hat{\mathcal{E}}', (r)?}\ (\textsc{if}) \qquad \frac{}{\hat{\mathcal{E}} \vdash \texttt{let}\ v := e : \hat{\mathcal{E}}, \epsilon}\ (\textsc{let}) \qquad \frac{\hat{\mathcal{E}} \vdash \rho_s : \{n_1, \ldots, n_k\} \quad \hat{\mathcal{E}}' \vdash s : \hat{\mathcal{E}}', r \quad \hat{\mathcal{E}} \sqsubseteq \hat{\mathcal{E}}'}{\hat{\mathcal{E}} \vdash \texttt{foreach}(v \in \rho_s)\{s\} : \hat{\mathcal{E}}', r^{n_1} | r^{n_2} | \ldots | r^{n_k}}\ (\textsc{loop})$$

Fig. 10. Over-approximation of partial programs used in ProgToRegex function.

Hence, we can now formally define the join operator on abstract environments as follows:

$$\hat{\mathcal{E}} \sqcup \hat{\mathcal{E}}' = \hat{\mathcal{E}}'' \iff (\hat{\mathcal{E}} \sqsubseteq \hat{\mathcal{E}}'') \wedge (\hat{\mathcal{E}}' \sqsubseteq \hat{\mathcal{E}}'') \wedge (\forall_{\hat{\mathcal{E}}'''}(\hat{\mathcal{E}} \sqsubseteq \hat{\mathcal{E}}''' \wedge \hat{\mathcal{E}}' \sqsubseteq \hat{\mathcal{E}}''' \Rightarrow \hat{\mathcal{E}}'' \sqsubseteq \hat{\mathcal{E}}'''))$$

The last conjunct in the right-hand-side of the implication states that $\hat{\mathcal{E}}'''$ (the result of joining $\hat{\mathcal{E}}'$ and $\hat{\mathcal{E}}''$) is the minimal environment (with respect to $\sqsubseteq$) that satisfies the first two conjuncts.

Loops. The most interesting aspect of the ProgToRegex procedure is the treatment of loops. First, since the environment may be modified in the loop body, this rule first computes an *inductive abstract environment*, $\hat{\mathcal{E}}'$, for the loop. In particular, the premise $\hat{\mathcal{E}} \sqsubseteq \hat{\mathcal{E}}'$ ensures that $\hat{\mathcal{E}}'$ over-approximates the *initial* environment, thereby establishing our base case. Second, the premise $\hat{\mathcal{E}}' \vdash \pi : \hat{\mathcal{E}}', r$ ensures that $\hat{\mathcal{E}}'$ is preserved in the loop body. Furthermore, because $\hat{\mathcal{E}}'$ is an over-approximation of the environment that the loop body π operates in, the regular expression r also over-approximares the behavior of π. Finally, to over-approximate the behavior of the entire loop, we determine the possible number of loop executions using the rules from Figure 9 under the initial environment $\hat{\mathcal{E}}$. If ρ_s can yield n different objects, then the behavior of the loop is captured as r^n. However, since we may not be able to compute the exact number of objects returned by a scan operation, we consider all possible cardinalities $n_1, \ldots, n_k$ of the resulting set. Hence, the behavior of the loop is captured by the disjunction of the regexes $r^{n_1}, \ldots, r^{n_k}$.

4.3.2 Using Large Language Model for Sketch Completion. We conclude this section with a discussion of how to use a large language model to guide sketch completion. Given a partial program ∂ with a hole h to fill, our approach encodes the context of the hole as a natural language prompt with unknown *masks* [Devlin et al. 2019]. In particular, the LLM is instructed to infer a probability distribution over completions for each mask from a given set of target completions (which are chosen by the Fill procedure in Algorithm 2 to ensure that the resulting program will be well-typed).

To illustrate how our sketch completion utilizes LLMs, Figure 11a (top) shows a partial program containing six unfilled holes, denoted as $??_i$. Figure 11b (top) shows the prompt used for completing hole $??_1$, which essentially corresponds to a natural description of the program. To generate such a prompt, our approach translates control-flow constructs to natural language in a syntax-directed way and replaces some of the holes ($??_1$ and $??_2$ in this example) with masks. Because the remaining holes $??_3$–$??_6$ will be replaced with synthetic variable names like v_1, v_2, they are not meaningful to the LLM; so our approach simply uses the types of these holes rather than masks when translating the partial program to natural language. As we can see from the LLM output in Figure 11b, the

```
foreach(v1 ∈ scanLoc(room)){
  goto(v1 : room);
  let v2 := getNth(scanObj(bin), 0);
  actUnary(open, v2 : bin);
  foreach(v3 ∈ scanObj(??1)){
   foreach(v4 ∈ scanObj(sheet)){
     if(??2){
       actUnary(grab, ??3 : sheet);
       actBinary(put-in, ??4 : sheet, ??5 : bin);
   }}}
  actUnary(close, ??6 : bin);
}
```

```
  ...
  foreach(v3 ∈ scanObj(bed)){
   foreach(v4 ∈ scanObj(sheet)){
     if(checkProp(??2a, v4) ∧ checkRel(??2b, v4, v3)){
  ...
```

(a) Partial programs ∂ (top) and ∂' (bottom)

Prompt for $??_1$**:** For each room; Go to room; Look for bins; Open a bin; For each $[M]_1$ do; For each sheet do; If $[M]_2$, grab sheet and put sheet in bin; Close bin.

LLM Output for $[M]_1$**:** {Bed: 0.8, Chair: 0.1, Mug: 0.05, ...}

Prompt for $??_{2a}$**:** For each room; Go to room; Look for bins; Open a bin; For each bed do; For each sheet do; If sheet is $[M]_{2a}$ and sheet is $[M]_{2b}$ bed, grab sheet and put sheet in bin; Close bin.

LLM Output for $[M]_{2a}$**:** {Dirty: 0.75, Folded: 0.2, White: 0.02, ...}

Prompt for $??_{2b}$**:** For each room; Go to room; Look for bins; Open a bin; For each bed do; For each sheet do; If sheet is dirty and sheet is $[M]_{2b}$ bed, grab sheet and put sheet in bin. Close bin.

LLM Output for $[M]_{2b}$**:** {On-top-of: 0.9, Under: 0.05, ...}

(b) LLM prompts for $??_1$, $??_{2a}$ and $??_{2b}$

Fig. 11. Partial programs during synthesis and the generated LLM prompts to choose the next completion

highest likelihood completion of $??_1$ is deemed to be *bed* by the model, so the sketch completion algorithm prioritizes this completion over other alternatives such as *chair* or *mug*.

As another example, consider the process of filling hole $??_2$, and suppose that the algorithm has already refined $??_2$ to the conjunct `checkProp`$(??_{2a}, v_4) \wedge$ `checkRel`$(??_{2b}, v_4, v_3)$ as shown in the bottom part of Figure 11a. When generating the prompt for $??_{2a}$, both holes $??_{2a}$ and $??_{2b}$ are filled with masks, and the LLM outputs *dirty* as the most likely completion for $??_{2a}$. When querying the remaining hole ($??_{2b}$), $??_{2a}$ has already been filled with *dirty*, so the prompt only contains a single mask, and the LLM outputs *on-top-of* as the most likely completion. As illustrated by these examples, the LLM-guided search strategy allows the sketch completion engine to quickly home in on the right concepts (such as *bed*, *dirty*, and *on-top-of* in this example) and therefore allows the search procedure to focus on the most promising sketch completions.

5 IMPLEMENTATION

We implemented the proposed approach in a tool called PROLEX written in Python[7]. In this section, we discuss salient aspects of PROLEX that are not covered in the technical sections.

Regex Learner. Our implementation leverages an open-source library, called XML-Schema-Learner [Nordmann 2014], to learn regular expressions from positive samples. Given a set of XML files, this library can learn the shared schema of the files as a regular expression. PROLEX encodes the given demonstrations as XML files, uses this software to learn a regex, and performs some post-processing to the learnt regex so that they better generalize in our application domain. Specifically, our post-processing step applies a small set of rewrite rules to expand the learned regular expression into a *set* of regexes that also match the input demonstrations. For instance, since our target regexes only involve the optional operator rather than arbitary disjunction, one of the rewrite rules is $(x|y)^* \longrightarrow (xy?)^*$, which replaces a disjunction with an optional operator. The full list of our rewrite rules is presented in Appendix A.

Large Language Model. Our sketch completion module utilizes the BERT large language model [Devlin et al. 2019], with pre-trained weights obtained from the HuggingFace library [Wolf et al. 2020].

[7] We intend to make all of our benchmarks, implementations and experiments publicly accessible.

Table 1. A selection of tasks from PROLEX-DS. We provide the task description in English to give the reader a general idea; however, the English descriptions do not capture all details of the tasks.

Task Description	Ground Truth		
	loop	cond.	percp.
Clean the plates and cups on the table and push in all the chairs	3	1	3
Grab the lamps near the door and place them near the bed	1	1	3
Grab all non-empty boxes near the kitchen's door and put them on the table	1	2	3
Grab and wipe all objects inside the baskets and put them back inside	3	1	4

Since our algorithm generates *masked language modeling* (MLM) queries, we chose to use the `bert-base-uncased` model. This model is primarily fine-tuned for tasks that make use of the whole sentence, potentially with masked words, to make decisions. It is a lightweight model, and in our experiments, it consistently returns responses in less than 70 *ms* on average.

Parallel Sketch Completion. The inherent independence of program sketches naturally lends itself to parallelization of the search process. To take advantage of this, PROLEX spawns a new process to execute sketch completion (Algorithm 2) for each generated sketch.

6 EVALUATION

In this section, we present a comprehensive empirical study of PROLEX, showcasing its efficacy in synthesizing complex programs for long-horizon robotic tasks. Our approach is evaluated against alternative methodologies, and we perform an ablation study to validate our design decisions in PROLEX. In particular, our experiments are designed to answer the following research questions:

(RQ1) How effective is PROLEX at learning policies from human demonstrations?
(RQ2) What is the relative significance of each of the key components in our synthesis algorithm?
(RQ3) How does PROLEX compare against relevant baselines in terms of learning policies that match the user's intention?

6.1 Benchmarks and Experimental Set-up

Tasks. To assess the effectiveness of PROLEX, we developed a benchmark-set, henceforth called PROLEX-DS, comprised of 40 programmatic LfD problems involving long-horizon tasks in a household environment. PROLEX-DS is inspired by household tasks defined in the Behavior Project [Li et al. 2023; Srivastava et al. 2022], which is an interactive simulation platform for a virtual embodied AI agent. The Behavior benchmark includes user demonstrations of the tasks, but there is no currently existing algorithm that is capable of utilizing the demonstrations for LfD in this challenging setting.[8] In recognition of the challenges of learning tasks from raw sensory data and controlling low-level motions of a robot, the Behavior benchmark introduces a custom symbolic abstraction of environment states and action primitives. PROLEX-DS leverages a similar symbolic abstraction consistent with the grammar introduced in Section 3, and includes 25 representative tasks from Behavior, and an additional 15 similar tasks (collected through a user survey), for a total of 40 tasks. Table 1 presents a selection of tasks from PROLEX-DS and provides statistics about their corresponding ground-truth programs.[9]

[8] There exist reinforcement learning (RL) baselines (discussed in Section 7) that are shown to have very poor performance in the Behavior benchmark set, despite having access to a formally specified task goal and extensive environment interactions during training. In our problem formulation, there is no formal specification of the task, hence we omit the RL baselines.
[9] The full list of tasks in PROLEX-DS is provided in Appendix A.

Env	# Obj. Types	# Obj. Instances	# Props.
Easy	40	140	609
Medium	60	295	2455
Hard	80	1109	13944

(a) Environments

Obj. Type	Properties
Mug	{empty, broken, warm, green, large, ...}
Bed	{king, firm, bunk, clean, messy, wooden, ...}
Sheet	{clean, white, beige, soft, large, ...}
...	...

(b) Object types

Fig. 12. Statistics about the environments used in PROLEX-DS benchmarks.

Environments. The tasks in PROLEX-DS are defined in three different household environments, as summarized in Figure 12. Because the difficulty of the synthesis task depends on the number of object types and objects in the environment, we classify the three environments as Easy, Medium, and Hard based on these numbers. As we can see from Figure 12a, there are up to thousands of objects in these environments. Because these objects correspond to constants in our DSL, the number of objects directly affects the difficulty of the corresponding synthesis task. Finally, we note that object types in these environments can involve many different relations and properties, a subset of which is shown in Figure 12b. This representation of objects aligns with the standard instance-level semantic knowledge graphs for everyday human environments [Li et al. 2023; Liu et al. 2021] which can be constructed using various robot perception technologies.

Full benchmark set. Overall, PROLEX-DS contains a total of 120 benchmarks, with 40 unique tasks and 3 different environments. For each of the 40 tasks, we manually write the ground truth program in our DSL and obtain a demonstration for each of the 120 benchmarks by running the ground truth program in the corresponding environment.

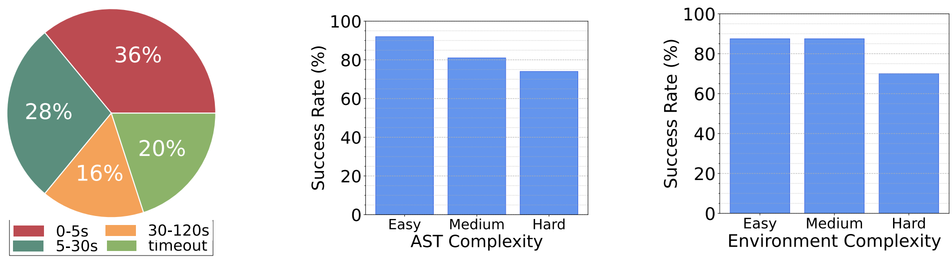
Experimental set-up. Our experiments were conducted on a server with 64 available Intel Xeon Gold 5218 CPUs @2.30GHz, 264GB of available memory, and running Ubuntu 22.04.2.

6.2 Main Results for Prolex

We now report on our experience with evaluating PROLEX on the collected benchmarks. In this evaluation, we run PROLEX with a time-limit of 120 seconds per task. We consider a task to be *completed* if PROLEX is able to learn a robot execution policy consistent with the provided demonstration within the time limit, and *solved* if the learned program also matches the user's intent. We determine if a task is solved by comparing it against the ground truth program (written manually) and checking if the learned program is semantically equivalent. Since the manually written policy is intended to work in all environments, tasks that are classified as "solved" have ultimate generalization power.

Running time. Figure 13a summarizes the main results as a pie chart where each slice shows the percentage of tasks that can be completed within a given time interval. As we can see from this figure, PROLEX is able to complete 36% of the tasks within the first 5 seconds, 28% of the tasks within 6-30 seconds, and 16% of the tasks within 31-120 seconds. Overall, PROLEX is able to complete 80% of the tasks within the time limit of 120 seconds.

Figure 13 also provides a more detailed look at these statistics by showing the percentage of tasks completed with respect to the complexity of the task and the environment. We categorized tasks into three levels of complexity, based on the AST size of their ground truth program. As expected, the learning problem becomes harder as the complexity of target policy or the environment increases. However, even for the more complex programs, PROLEX is still able to complete the learning task for 74% of the benchmarks, and similarly, in the "Hard" environment, PROLEX is able to successfully complete 70% of the tasks within the 2 minute time limit.



(a) Distribution of synthesis time (b) success vs program size (c) success vs environment

Fig. 13. PROLEX's Synthesis Results

Env.	Generalizable
Easy	71%
Medium	82%
Hard	90%

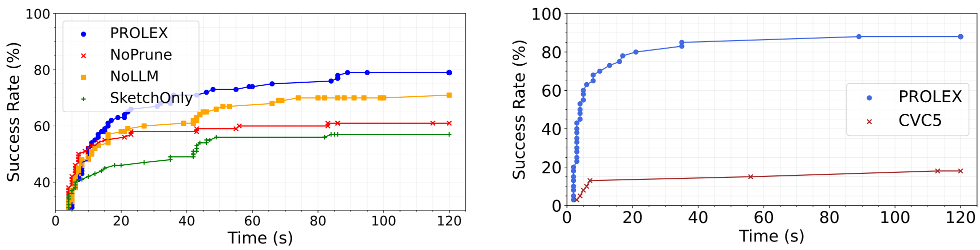
Fig. 14. % of completed tasks that match the ground-truth.

Generalizability. As mentioned earlier, completing a task is not the same as "solving" it, since the learnt policy may not match user intent despite being consistent with the demonstrations. We manually inspected all programs synthesized by PROLEX for all tasks and environments, and found that it is equivalent to the ground truth program in 81% of the completed cases. Interestingly, as shown in Figure 14 we found that PROLEX's generalization power improves as the complexity of the environment increases. While not immediately obvious, this observation actually makes sense: the more complex the environment, the more objects there are with different properties, so it becomes harder to find multiple programs that "touch" exactly the same objects as the demonstration.

Failure analysis. As discussed above, there are two reasons why PROLEX may fail to solve a task: (1) it fails to find a policy consistent with the demonstrations within the time limit, or (2) the learnt policy does not adequately capture the user intent. We have manually inspected both classes of failure cases and report on our findings.

The main cause of PROLEX's timeouts is the synthesizer's incorrect assumptions about what to scan for. Many of our environments contain a large number of object types, all of which can be arguments of `scan` operations. Our approach tries to overcome this issue by using an LLM to guide search, but in some cases, the LLM proposes the wrong object type to scan for. This causes the synthesizer to go down a rabbit hole, particularly in cases when the proposed object type has many properties associated with it. We believe more advanced LLMs that can reason about finer-grained properties between the environment and the context of the task can potentially mitigate this issue.

We also inspected the cases where PROLEX finds *a* robot execution policy consistent with the demonstrations, but the learnt policy does not generalize to different environments (i.e., it *completes* the task but fails to *solve* it). There are two main reasons for this, both due to the inadequacy of the demonstrations with respect to the desired task. Specifically, if there is only *one* instance of a particular object type in the environment, the synthesizer may not return a program with a `foreach` loop over that object type, even though the ground truth program contains such a loop. Likewise, if *all* instances of a manipulated object type in the scene satisfy (or dissatisfy) a property, the synthesizer cannot learn the conditional block for such manipulations, as all object instances of that type are acted upon. As noted earlier, this generalizability issue becomes less of a problem in more complex environments containing many instances of an object type.



(a) Ablation results over all environments.

(b) PROLEX vs. CVC5 in **Easy** environments. CVC5 is provided the **ground truth sketch** but PROLEX is not

Fig. 15. Experimental Results

RQ1 Summary. Given a 2 minute time limit, PROLEX is able to find a policy consistent with the demonstations for 80% of the benchmarks. Furthermore 81% of the synthesized programs correspond to the ground truth, meaning that they can generalize perfectly in any unseen environment.

6.3 Ablation Studies

As mentioned throughout the paper, there are three key components underlying our learning approach, namely (1) sketch generation using regexes, (2) LLM-guided sketch completion, and (3) trace compatibility checking using regexes. To better understand the relative importance of each component, we present the results of an ablation study where we disable each component or combinations of components. Specifically, for our ablation study, we consider the following variants of PROLEX:

- **PROLEX-NoSketch:** This is a variant of PROLEX that does not generate program sketches using regex learning.
- **PROLEX-NoLLM:** This is a variant of PROLEX that does not utilize LLMs for sketch completion.
- **PROLEX-NoPrune:** This is a variant of PROLEX that does not utilize the compatibility checking procedure (Algorithm 2) for pruning the search space during sketch completion.
- **PROLEX-SketchOnly:** This is a variant of PROLEX that uses regexes to produce a sketch but neither utilizes LLM nor compatibility checking during sketch completion.

Figure 15a shows the results of our ablation study in the form of a Cumulative Distribution Function (CDF). The x-axis represents the cumulative running time, while the y-axis shows the percentage of benchmarks solved in all environments. The results indicate a significant gap between the number of the tasks solved by PROLEX and its variants defined above. In particular, PROLEX is able to solve 9% more tasks than **NoLLM**, 18% more tasks than **NoPrune**, and 22% more tasks than **SketchOnly** variants, within the 120-second time limit. However, the results for the **NoSketch** variant are particularly poor, with *none* of tasks solved. This is not surprising, due to the astronomical number of programs in our DSL.

RQ2 Summary. All of the key components of our proposed synthesis algorithm contribute to the practicality of our learning approach. The most important component is regex-based sketch generation, without which none of the tasks can be solved. Regex-based pruning helps solve an additional 18% of the tasks, and LLM guidance increases success rate by another 9%.

6.4 Comparison with Alternative Approaches

In this section, we report on our experience comparing PROLEX against alternative approaches. While there is no existing off-the-shelf LfD approach that targets our problem domain (see Section 6.1), we compare PROLEX against the following two baselines:

- **CVC5:** We formulate our learning problem as an instance of syntax-guided synthesis and use one of the leading SyGuS solvers (namely, CVC5 [Barbosa et al. 2022]) as a programmatic policy synthesizer.
- **GPT-Synth:** We use an LLM as a *neural program synthesizer* in our domain. To this end, we consider a baseline called "GPT-Synth" that synthesizes programs in our DSL from demonstrations.

Case Study with CVC5. Our programmatic LfD task can be reduced to an instance of the syntax-guided synthesis (SyGuS) problem [Alur et al. 2015a], which is the standard formulation for synthesis problems. To compare PROLEX against state-of-the-art SyGuS solvers, we encoded the tasks in PROLEX-DS as instances of SyGuS and leveraged an off-the-shelf solver, namely, CVC5 [Barbosa et al. 2022], which is the winner of the most recent SyGuS competition.

To perform this comparison, we defined our DSL using the syntactic constraints in SyGuS, and we incorporated semantic constraints based on the *initial and final environment states* in the demonstrations. Note that SyGuS solvers are unable to perform synthesis from demonstrations, as demonstrations correspond to intermediate program states, which are not expressible in the SyGuS formulation. Hence, when comparing against CVC5, we only use the initial and final environments and consider a task to be completed if the solver returns a policy that produces the desired environment. Furthermore, we only compare the performance of PROLEX and CVC5 on the "easy" environment as encoding the environments in SyGuS requires significant manual labor.

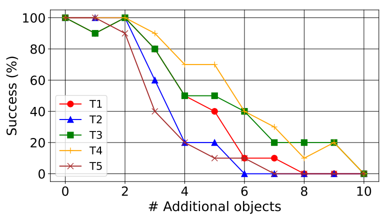
When we tried to use CVC5 to perform synthesis from scratch (i.e., without a sketch, it was not able to complete any task within a reasonable time limit. Hence, to be able to perform any comparison, we manually provided CVC5 with the *ground truth* sketch for the specific task. The results of this comparison are presented in Figure 15b. As in Figure 15a, this figure plots the cumulative distribution of the percentage of programs solved against the solver time. The results demonstrate that CVC5 struggles to solve the majority of the given tasks within the time limit, even when provided with the ground truth sketch. In particular, CVC5 only solved 18% of the tasks in the given time limit, compared to 88% solved by PROLEX. This outcome is not surprising since the space of programs in our DSL is vast, so any unbiased search, including the one implemented in CVC5, is unlikely to scale well.

Case Study with GPT-Synth. For this experiment, we use the GPT 3.5 LLM[10] to generate programmatic policies from demonstrations. This study aims to evaluate the effectiveness of LLMs as an end-to-end program synthesizer, henceforth referred to as GPT-Synth.

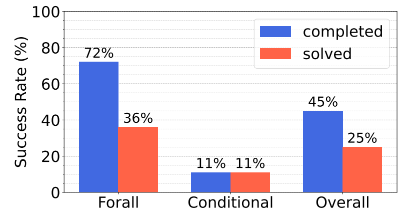
The prompts used in this study were developed using the "*template*" pattern as described by White et al. [2023]. This pattern has been shown to be highly effective in situations where the generated output must conform to a specific format that may not be part of the LLM's training data. In particular, for each benchmark task, we created a prompt that includes a description of our DSL, and a set of example demonstrations and the environment states, along with the correct programs for the desired task. For a novel task, we provide the language model with the prompt, the demonstrations and environment states for the novel task, and ask it to generate a corresponding program.

To use GPT 3.5 as an end-to-end synthesizer, we adopt the following methodology, as done in prior work [Chen et al. 2021b]. If the first program returned by the LLM is consistent with the

[10]We specifically use the `text-davinci-003` model, which is the most capable publicly available LLM from OpenAI finetuned for completion, including natural language and code.



(a) GPT-Synth with environment



(b) GPT-Synth without environment

Fig. 16. LLM Experiments

demonstration, GPT-Synth returns that program as the solution. Otherwise, it asks the model to produce another program, for up to 10 iterations, after which it declares failure. Unfortunately, we found that GPT-Synth is unable to solve any of the benchmarks in PROLEX-DS when we provide both the demonstrations and the environment. This behavior seems to be caused by the large number of entities and relations in the environment – prior work has reported similar results in other domains (e.g., planning [Mahowald et al. 2023]) where LLMs were used to perform tasks in non-trivial environments.

To gain more intuition about how the language model scales with the environment size, we report on our experience with using GPT-Synth on five representative tasks involving toy environments. We construct these toy environments by incorporating *only* the objects (and their properties) required for that task plus some additional objects and properties. Figure 16a shows how the success rate of GPT-Synth scales with respect to environment size. Here, the x-axis shows the number of additional objects (and their properties) in the environment and the y-axis shows the success rate. As we can see from Figure 16a , GPT-Synth works well if it is given *only* the relevant objects (which is not a realistic usage scenario), but, as environment size increases, its success rate drops dramatically. In fact, when the environment contains only 10 additional objects — a tiny fraction of the entities present in our "Easy" environment — the success rate of , GPT-Synth already drops to zero.

The reader may wonder if the environment is actually necessary for GPT-Synth to learn the correct program. To answer this question, we perform an additional experiment where we provide GPT-Synth with only the demonstration, but not the environment. The results of this evaluation are presented in Figure 16b, where we classify tasks into two categories as "Forall" and "Conditional". The former class of tasks does not involve branching, whereas the latter does. Here, "Completed" shows the percentage of tasks for which GPT-Synth finds a program consistent with the demonstration, and "Solved" shows the percentage of tasks for which GPT-Synth returns a program that *also* matches the ground truth. As we can see, GPT-Synth returns a program consistent with the demonstration for 45% of all tasks, but it is only able to identify the ground truth program in 25% of the cases. Furthermore, as one might expect, GPT-Synth is much more effective at the much simpler "Forall" category of tasks that involve acting on *all* instances of a particular type. In contrast, the "Conditional" category is much more challenging without having access to the environment, and the success rate of GPT-Synth is only 11% for this category. Intuitively, without knowing which objects have what properties, GPT-Synth has little chance of knowing that there should be a conditional and what its corresponding predicate should be. We came across a few cases where GPT-Synth is able to "hallucinate" the right predicates after several rounds of interaction; but, in general, guessing the user's intent without knowing the environment, is at best a matter of sheer luck.

RQ3 Summary. PROLEX performs significantly better than the two CVC5 and GPT-Synth baselines. Even when given the ground truth sketches, CVC5 is only able to return a program consistent with the final environment in 18% of the cases. On the other hand, the GPT-based synthesizer cannot solve any tasks when provided with both the demonstration *and* the full environment, but it is able to solve 25% of the tasks when it is given *only* the demonstration.

7 RELATED WORK

Robot Learning from Demonstrations. Our approach builds upon a substantial body of literature on the use of Learning from Demonstration (LfD) techniques to learn robot execution policies [Argall et al. 2008, 2009; Sosa-Ceron et al. 2022]. This literature can be broadly categorized into two approaches: (1) learning neural models to represent robot behaviors [Ho and Ermon 2016; Kober et al. 2013; Ly and Akhloufi 2021; Rusu et al. 2017; Sünderhauf et al. 2018; Taylor and Stone 2009; Xiao et al. 2021; Ziebart et al. 2008], and (2) synthesizing programmatic representations of execution policies [French et al. 2019; Holtz et al. 2021, 2020a; Niekum et al. 2015]. The most well-established techniques for learning neural models from demonstrations include behavior cloning within the framework of imitation learning [Ho and Ermon 2016; Ly and Akhloufi 2021], and (deep) reinforcement learning (RL) methods [Kober et al. 2013; Sünderhauf et al. 2018; Ziebart et al. 2008]. Empirical studies have demonstrated the efficacy of these neural policies in perception tasks and their ability to perform well in unknown or ill-defined environments. However, such neural models lack robust interpretability and generalization capabilities — as a testament to this, there exist no neural LfD algorithms to date capable of leveraging the user demonstrations in the Behavior benchmarks [Srivastava et al. 2022]. The field of transfer learning [Pan and Yang 2010; Rusu et al. 2017; Taylor and Stone 2009] aims to resolve generalization problems to some degree and also enhance data efficiency. A related setting to our work is applying reinforcement learning (RL) by specifying the task via formal specification of the goal conditions; in fact, the Behavior benchmark set [Srivastava et al. 2022] reports the results of such an approach. However, even in the simplest 12 activities, the RL algorithms are unable to complete the tasks even when initiated close to the goal states. Further, even when the actions are abstracted into symbols in Behavior-1K [Li et al. 2023], RL approaches demonstrate very poor performance. These results mainly highlight the complexity of the tasks that we tackle in this paper.

Recently, there has also been growing interest in developing techniques to enhance the transparency and reliability of RL systems through formal explanations [Glanois et al. 2021; Krajna et al. 2022; Li et al. 2019; Topin and Veloso 2019]. These techniques aim to explain different aspects of the learned models, such as inputs and transitions, by finding interpretable representations of neural policies, such as Abstracted Policy Graphs [Topin and Veloso 2019] or structures in a high-level DSL [Verma et al. 2018]. More recently, there has also been interest in utilizing program synthesis methods [Holtz et al. 2020b; Xin et al. 2023] to learn robot execution policies from demonstrations as an alternative to neural model learning [Holtz et al. 2021, 2020a]. These approaches provide improved interpretability, generalizability [Holtz et al. 2018], and data efficiency. PROLEX falls into the same class of techniques as these approaches but broadens their applicability in several ways: First, it can learn non-Markovian policies to handle long-horizon tasks; second, it can synthesize programs with complex control flow, such as loops with nested conditionals and loops; and, third, it can handle environments with a large number of objects and properties.

Program Synthesis from Demonstrations. This paper is related to a long history of research on program synthesis, with the ultimate objective of automatically generating a program that meets a specified requirement [Alur et al. 2015a,b; Bornholt et al. 2016; Chen et al. 2021a; Feng et al. 2018a;

Gulwani 2011; Gulwani et al. 2017; Jha et al. 2010; Kalyan et al. 2018; Solar-Lezama 2008; Wang et al. 2020, 2018b]. Different synthesizers adopt different types of specifications, such as input-output examples [Feser et al. 2015], demonstrations [Chasins et al. 2018], logical constraints [Miltner et al. 2022], refinement types [Polikarpova et al. 2016], or a reference implementation [Wang et al. 2019a]. Among these, our method is mostly related to the synthesizers that enable programming by demonstration (PbD) [Chasins et al. 2018; Lau et al. 2003]. Existing PbD techniques generalize programs either from sequences of user actions [Chasins et al. 2018; Dong et al. 2022] or sequences of program states [Lau et al. 2003]. Our approach is similar to the former, specifically similar to WebRobot [Dong et al. 2022], which can synthesize challenging programs with multiple nested loops from sequences of user actions. However, WebRobot utilizes a term-rewriting engine to complete sketches, whereas we incorporate a regex-based program abstraction to prune the search space, and an LLM to guide the search. Moreover, WebRobot is fine-tuned for web process automation tasks and is not equipped to handle challenges specific to the domain of robotics, such as perception. Finally, unlike our approach, WebRobot cannot synthesize programs with conditional blocks, which are essential for successfully performing long-horizon tasks.

Enhancing Synthesis using ML Models. Machine learning has proven to be highly effective for improving time and accuracy of synthesis [Cambronero et al. 2023; Kalyan et al. 2018; Pailoor et al. 2021; Rahmani et al. 2021; Verbruggen et al. 2021; Ye et al. 2020]. For example, neural generators trained on partial programs (i.e., sketches) have been shown to accurately predict the full body of a method from just a few API calls or data types [Chen et al. 2020; Murali et al. 2017; Nye et al. 2019]. In addition, LLMs have been utilized to guide program search [Jain et al. 2022]. For example, the GPT-3 language model has been applied to mine program components and their distributions for multi-modal program synthesis tasks [Rahmani et al. 2021]. Our work is similar in approach and leverages an LLM to improve program synthesis. However, to the best of knowledge, PROLEX is the first approach to leverage the LLM's prior knowledge of the semantic relations between real-world entities and actions to guide the search towards reasonable completions.

A related field of research, *neurosymbolic programming*, seeks to combine advances in end-to-end machine learning techniques with program synthesis by leveraging compositional programming abstractions as a means of reusing learned modules across various tasks [Bowers et al. 2023; Chaudhuri et al. 2021; Chen et al. 2021a; Huang et al. 2020; Inala et al. 2020; Mao et al. 2019; Sun et al. 2022; Verma et al. 2019; Witt et al. 2023; Zhan et al. 2021]. Because our current approach is based on a symbolic environment representation, it does not require a neurosymbolic DSL.

Program Sketching. Program sketches have been introduced as a syntactic framework to guide the generation of candidate programs during a search process. This approach was initially presented in [Solar-Lezama 2008] and has since been widely used [Bornholt et al. 2016; Dong et al. 2022; Solar-Lezama 2008; Solar-Lezama 2009; Solar-Lezama et al. 2006; Wang et al. 2019b; Yaghmazadeh et al. 2017]. While some approaches utilize program sketches that are crafted by the user, others automatically generate a sketch based on natural language [Chen et al. 2020; Yaghmazadeh et al. 2017] or reference implementation [Wang et al. 2019b]. Our method also decomposes the synthesis task into two separate sketch generation and sketch completion step but utilizes regex learning to find a sketch that is likely to be a consistent generalization of the user demonstrations.

Pruning Techniques for Synthesis. Many prior techniques enhance program synthesis by using some form of static analysis and logical reasoning to prune parts of the search space [Feng et al. 2018b; Lee et al. 2016; Tiwari et al. 2015; Vechev et al. 2010; Wang et al. 2018a]. For instance, some synthesis techniques use SMT-based pruning [Feng et al. 2018b, 2017; Polikarpova et al. 2016], whereas others use abstract interpretation or domain-specific static analysis to perform pruning more efficiently [Chen et al. 2020; Lee et al. 2016; Wang et al. 2018a]. Our technique also utilizes program abstractions to prune partial programs, but the main difference is that it (a) abstracts programs using regular expressions, and (b) uses the abstraction to check compatibility between user demonstrations and partial programs. A similar notion of *trace compatibility* has been proposed for synthesis-based transpilation [Mariano et al. 2022]; however, that work differs from our pruning technique in several ways: First, NGST2's notion of trace compatibility is defined between traces of two different programs, whereas ours is defined between a program and user demonstration. Second, their technique for checking compatibility between traces is very different from ours and relies on a collecting semantics [Cousot and Cousot 1977] of the programming language.

8 CONCLUSION AND FUTURE WORK

We proposed a new programmatic LfD approach, based on program synthesis, for learning robot execution policies for long-horizon tasks in complex environments. Our approach is based on two key insights: First, it utilizes regular expressions for inferring useful sketches and for pruning the space of all possible completions of the sketch. Second, because the synthesized program can refer to objects that are in the environment but *not* in the demonstration, our approach utilizes an LLM to predict which objects and properties are most likely to occur in the target program.

We have evaluated our implementation, PROLEX, on 120 benchmarks and show that PROLEX is able to synthesize complex policies with several (nested) loops and conditionals and that it scales to large environments containing thousands of objects and dozens of distinct objects types. Overall, given a 120 second time limit, PROLEX is able to find a program consistent with the demonstrations for 80% of the benchmarks. Furthermore, for 81% of the completed tasks, PROLEX can learn the ground truth program from a single demonstration. To put these numbers in context, we also compare PROLEX against two baselines, including a state-of-the-art SyGuS solver and a neural LLM-based synthesizer, and show that PROLEX significantly outperforms both of them.

In future work, we are interested in deploying this technique on real robots in physical environments. To this end, we plan to integrate a semantic-aware perception frontend like Kimera [Rosinol et al. 2021] to extract the symbolic state of the world as a semantic scene graph, as such a representation would be directly compatible with the PROLEX DSL. We are interested in building a web-based graphical interface to our robots to gather user demonstrations for deployments, building on existing robot deployment management systems like RoboFleet [Sikand et al. 2021].

REFERENCES

Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015a. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. 1–25.

Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. 2015b. Synthesis Through Unification. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 163–179.

Brenna D. Argall, Brett Browning, and Manuela Veloso. 2008. Learning robot motion control with demonstration and advice-operators. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 399–404. https://doi.org/10.1109/IROS.2008.4651020

Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. 2009. A survey of robot learning from demonstration. *Robotics and Autonomous Systems* 57, 5 (2009), 469–483. https://doi.org/10.1016/j.robot.2008.10.024

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science)*, Dana Fisman and Grigore Rosu (Eds.), Vol. 13243. Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing synthesis with metasketches. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 775–788.

Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. 2023. Top-Down Synthesis for Library Learning. *Proc. ACM Program. Lang.* 7, POPL, Article 41 (jan 2023), 32 pages. https://doi.org/10.1145/3571234

José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL, Article 33 (jan 2023), 30 pages. https://doi.org/10.1145/3571226

Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. Association for Computing Machinery, New York, NY, USA, 963–975. https://doi.org/10.1145/3242587.3242661

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, and Yisong Yue. 2021. Neurosymbolic Programming. *Found. Trends Program. Lang.* 7 (2021), 158–243.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021b. Evaluating Large Language Models Trained on Code. arXiv:cs.LG/2107.03374

Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021a. Web Question Answering with Neurosymbolic Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3453483.3454047

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 487–502.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. 238–252.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

Rui Dong, Zhicheng Huang, Ian Iong Lam, Yan Chen, and Xinyu Wang. 2022. WebRobot: Web Robotic Process Automation Using Interactive Programming-by-Demonstration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 152–167. https://doi.org/10.1145/3519939.3523711

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018a. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018b. Program synthesis using conflict-driven learning. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 420–435.

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proc. of PLDI*. 422–436.

John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239.

Kevin French, Shiyu Wu, Tianyang Pan, Zheming Zhou, and Odest Chadwicke Jenkins. 2019. Learning Behavior Trees From Demonstration. In *2019 International Conference on Robotics and Automation (ICRA)*. 7791–7797. https://doi.org/10.1109/ICRA.2019.8794104

Qiaozi Gao, Govind Thattai, Xiaofeng Gao, Suhaila Shakiah, Shreyas Pansare, Vasu Sharma, Gaurav Sukhatme, Hangjie Shi, Bofei Yang, Desheng Zheng, et al. 2023. Alexa Arena: A User-Centric Interactive Platform for Embodied AI. *arXiv preprint arXiv:2303.01586* (2023).

Claire Glanois, Paul Weng, Matthieu Zimmer, Dong Li, Tianpei Yang, Jianye Hao, and Wulong Liu. 2021. A Survey on Interpretable Reinforcement Learning. https://doi.org/10.48550/ARXIV.2112.13112

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proc. of POPL*. 317–330.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2, 1–119.

Jonathan Ho and Stefano Ermon. 2016. Generative Adversarial Imitation Learning. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2016/file/cc7e2b878868cbae992d1fb743995d8f-Paper.pdf

Jarrett Holtz, Simon Andrews, Arjun Guha, and Joydeep Biswas. 2021. Iterative Program Synthesis for Adaptable Social Navigation. In *Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference on*. 6256–6261. https://doi.org/10.1109/IROS51168.2021.9636540

Jarrett Holtz, Arjun Guha, and Joydeep Biswas. 2018. Interactive Robot Transition Repair With SMT. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 4905–4911. https://doi.org/10.24963/ijcai.2018/681

Jarrett Holtz, Arjun Guha, and Joydeep Biswas. 2020a. Robot Action Selection Learning via Layered Dimension Informed Program Synthesis. In *Conference on Robot Learning*. 1471–1480. https://joydeepb.com/Publications/corl2020_ldips.pdf

Jarrett Holtz, Arjun Guha, and Joydeep Biswas. 2020b. Robot Action Selection Learning via Layered Dimension Informed Program Synthesis. In *Conference on Robot Learning*.

Jiani Huang, Calvin Smith, Osbert Bastani, Rishabh Singh, Aws Albarghouthi, and Mayur Naik. 2020. Generating Programmatic Referring Expressions via Program Synthesis. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Hal Daumé III and Aarti Singh (Eds.), Vol. 119. PMLR, 4495–4506. https://proceedings.mlr.press/v119/huang20h.html

Jeevana Priya Inala, Yichen Yang, James Paulos, Yewen Pu, Osbert Bastani, Vijay Kumar, Martin Rinard, and Armando Solar-Lezama. 2020. Neurosymbolic Transformers for Multi-Agent Communication. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1141, 12 pages.

iRobot. 2023. iRobot Home App. https://www.irobot.com/en_US/irobot-home-app.html Accessed on March 30, 2023.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models Meet Program Synthesis. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3510003.3510203

Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 215–224.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. https://doi.org/10.48550/ARXIV.1804.01186

Jens Kober, J. Andrew Bagnell, and Jan Peters. 2013. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research* 32, 11 (2013), 1238–1274. https://doi.org/10.1177/0278364913495721 arXiv:https://doi.org/10.1177/0278364913495721

Agneza Krajna, Mario Brcic, Tomislav Lipic, and Juraj Doncevic. 2022. Explainability in reinforcement learning: perspective and position. https://doi.org/10.48550/ARXIV.2203.11547

Tessa A. Lau, Steven A. Wolfman, Pedro M. Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53 (2003), 111–156.

Jin Joo Lee, Amin Atrash, Dylan Glas, and Hanxiao Fu. 2023. Developing autonomous behaviors for a consumer robot to hang out near people in the home. In *AAAI 2023 Spring Symposium Series*. https://www.amazon.science/publications/developing-autonomous-behaviors-for-a-consumer-robot-to-hang-out-near-people-in-the-home

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2016)*. Association for Computing Machinery, New York, NY, USA, 70–80. https://doi.org/10.1145/2993236.2993244

Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Michael Lingelbach, Jiankai Sun, Mona Anvari, Minjune Hwang, Manasi Sharma, Arman Aydin, Dhruva Bansal, Samuel Hunter, Kyu-Young Kim, Alan Lou, Caleb R Matthews, Ivan Villa-Renteria, Jerry Huayang Tang, Claire Tang, Fei Xia, Silvio Savarese, Hyowon Gweon, Karen Liu, Jiajun Wu, and Li Fei-Fei. 2023. BEHAVIOR-1K: A Benchmark for Embodied AI with 1,000 Everyday Activities and Realistic Simulation. In *Proceedings of The 6th Conference on Robot Learning (Proceedings of Machine Learning Research)*, Karen Liu, Dana Kulic, and Jeff Ichnowski (Eds.), Vol. 205. PMLR, 80–93. https://proceedings.mlr.press/v205/li23a.html

Xiao Li, Zachary Serlin, Guang Yang, and Calin Belta. 2019. A formal methods approach to interpretable reinforcement learning for robotic planning. *Science Robotics* 4, 37 (2019), eaay6276. https://doi.org/10.1126/scirobotics.aay6276 arXiv:https://www.science.org/doi/pdf/10.1126/scirobotics.aay6276

Weiyu Liu, Dhruva Bansal, Angel Andres Daruna, and S. Chernova. 2021. Learning Instance-Level N-Ary Semantic Knowledge At Scale For Robots Operating in Everyday Environments. *Robotics: Science and Systems XVII* (2021).

Abdoulaye O. Ly and Moulay Akhloufi. 2021. Learning to Drive by Imitation: An Overview of Deep Behavior Cloning Methods. *IEEE Transactions on Intelligent Vehicles* 6, 2 (2021), 195–209. https://doi.org/10.1109/TIV.2020.3002505

Kyle Mahowald, Anna A Ivanova, Idan A Blank, Nancy Kanwisher, Joshua B Tenenbaum, and Evelina Fedorenko. 2023. Dissociating language and thought in large language models: a cognitive perspective. *arXiv preprint arXiv:2301.06627* (2023).

Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B. Tenenbaum, and Jiajun Wu. 2019. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJgMlhRctm

Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. 2022. Automated Transpilation of Imperative to Functional Code Using Neural-Guided Program Synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 71 (apr 2022), 27 pages. https://doi.org/10.1145/3527315

Michael Lee Michael L. Scott. 2000. *Programming language pragmatics*. Morgan Kaufmann.

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (jan 2022), 29 pages. https://doi.org/10.1145/3498682

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2017. Neural Sketch Learning for Conditional Program Generation. https://doi.org/10.48550/ARXIV.1703.05698

Scott Niekum, Sarah Osentoski, George Konidaris, Sachin Chitta, Bhaskara Marthi, and Andrew G. Barto. 2015. Learning grounded finite-state representations from unstructured demonstrations. *The International Journal of Robotics Research* 34, 2 (2015), 131–157. https://doi.org/10.1177/0278364914554471 arXiv:https://doi.org/10.1177/0278364914554471

Kore Nordmann. 2014. XML-Schema-learner. https://github.com/kore/XML-Schema-learner.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to Infer Program Sketches. https://doi.org/10.48550/ARXIV.1902.06349

Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2021. Synthesizing Data Structure Refinements from Integrity Constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3453483.3454063

Sinno Jialin Pan and Qiang Yang. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* 22 (2010), 1345–1359.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proc. of PLDI*. 522–538.

Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-Modal Program Inference: A Marriage of Pre-Trained Language Models and Component-Based Synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 158 (oct 2021), 29 pages. https://doi.org/10.1145/3485535

Antoni Rosinol, Andrew Violette, Marcus Abate, Nathan Hughes, Yun Chang, Jingnan Shi, Arjun Gupta, and Luca Carlone. 2021. Kimera: From SLAM to spatial perception with 3D dynamic scene graphs. *The International Journal of Robotics Research* 40, 12-14 (2021), 1510–1546.

Andrei A. Rusu, Matej Vecerík, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. 2017. Sim-to-Real Robot Learning from Pixels with Progressive Nets. In *1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings (Proceedings of Machine Learning Research)*, Vol. 78. PMLR, 262–270. http://proceedings.mlr.press/v78/rusu17a.html

Kavan Singh Sikand, Logan Zartman, Sadegh Rabiee, and Joydeep Biswas. 2021. Robofleet: Open source communication and management for fleets of autonomous robots. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 406–412.

Armando Solar-Lezama. 2008. *Program synthesis by sketching*. Citeseer.

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Proc. of APLAS*. 4–13.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proc. of ASPLOS*. 404–415.

Arturo Daniel Sosa-Ceron, Hugo Gustavo Gonzalez-Hernandez, and Jorge Antonio Reyes-Avendaño. 2022. Learning from Demonstrations in Human and Robot Collaborative Scenarios: A Survey. *Robotics* 11, 6 (2022). https://doi.org/10.3390/robotics11060126

Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Martín-Martín, Fei Xia, Kent Elliott Vainio, Zheng Lian, Cem Gokmen, Shyamal Buch, Karen Liu, Silvio Savarese, Hyowon Gweon, Jiajun Wu, and Li Fei-Fei. 2022. BEHAVIOR: Benchmark for Everyday Household Activities in Virtual, Interactive, and Ecological Environments. In *Proceedings of the 5th Conference on Robot Learning (Proceedings of Machine Learning Research)*, Aleksandra Faust, David Hsu, and Gerhard Neumann (Eds.), Vol. 164. PMLR, 477–490. https://proceedings.mlr.press/v164/srivastava22a.html

Jennifer J. Sun, Megan Tjandrasuwita, Atharva Sehgal, Armando Solar-Lezama, Swarat Chaudhuri, Yisong Yue, and Omar Costilla-Reyes. 2022. Neurosymbolic Programming for Science. https://doi.org/10.48550/ARXIV.2210.05050

Niko Sünderhauf, Oliver Brock, Walter Scheirer, Raia Hadsell, Dieter Fox, Jürgen Leitner, Ben Upcroft, Pieter Abbeel, Wolfram Burgard, Michael Milford, and Peter Corke. 2018. The limits and potentials of deep learning for robotics. *The International Journal of Robotics Research* 37, 4-5 (2018), 405–420. https://doi.org/10.1177/0278364918770733 arXiv:https://doi.org/10.1177/0278364918770733

Matthew E. Taylor and Peter Stone. 2009. Transfer Learning for Reinforcement Learning Domains: A Survey. *J. Mach. Learn. Res.* 10 (dec 2009), 1633–1685.

Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. 2015. Program Synthesis Using Dual Interpretation. In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 482–497.

Nicholay Topin and Manuela Veloso. 2019. Generation of Policy-Level Explanations for Reinforcement Learning. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence (AAAI'19/IAAI'19/EAAI'19).* AAAI Press, Article 310, 8 pages. https://doi.org/10.1609/aaai.v33i01.33012514

Martin Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-Guided Synthesis of Synchronization. *SIGPLAN Not.* 45, 1 (jan 2010), 327–338. https://doi.org/10.1145/1707801.1706338

Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic Programming by Example with Pre-Trained Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 100 (oct 2021), 25 pages. https://doi.org/10.1145/3485477

Abhinav Verma, Hoang M. Le, Yisong Yue, and Swarat Chaudhuri. 2019. *Imitation-Projected Programmatic Reinforcement Learning.* Curran Associates Inc., Red Hook, NY, USA.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. 2018. Programmatically Interpretable Reinforcement Learning. (2018). https://doi.org/10.48550/ARXIV.1804.02477

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018a. Program synthesis using abstraction refinement. *PACMPL* 2, POPL (2018), 63:1–63:30.

Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019a. Synthesizing Database Programs for Schema Refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019).* ACM, New York, NY, USA, 286–300. https://doi.org/10.1145/3314221.3314588

Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019b. Synthesizing database programs for schema refactoring. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI.*

Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data migration using datalog program synthesis. *VLDB* (2020).

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018b. Relational Program Synthesis. *PACMPL* 2, OOPSLA (2018), 155:1–155:27.

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. arXiv:cs.SE/2302.11382

Jonas Witt, Stef Rasing, Sebastijan Dumančić, Tias Guns, and Claus-Christian Carbon. 2023. A Divide-Align-Conquer Strategy for Program Synthesis. https://doi.org/10.48550/ARXIV.2301.03094

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations.* Association for Computational Linguistics, Online, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

Xuesu Xiao, Joydeep Biswas, and Peter Stone. 2021. Learning Inverse Kinodynamics for Accurate High-Speed Off-Road Navigation on Unstructured Terrain. https://doi.org/10.48550/ARXIV.2102.12667

Jimmy Xin, Linus Zheng, Jiayi Wei, Kia Rahmani, Jarrett Holtz, Isil Dillig, and Joydeep Biswas. 2023. PLUNDER: Probabilistic Program Synthesis for Learning from Unlabeled and Noisy Demonstrations. https://doi.org/10.48550/ARXIV.2303.01440

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *PACMPL* 1, OOPSLA (2017), 63:1–63:26.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2020. Optimal Neural Program Synthesis from Multimodal Specifications. *arXiv preprint arXiv:2010.01678* (2020).

Eric Zhan, Jennifer J. Sun, Ann Kennedy, Yisong Yue, and Swarat Chaudhuri. 2021. Unsupervised Learning of Neurosymbolic Encoders. https://doi.org/10.48550/ARXIV.2107.13132

Brian D. Ziebart, Andrew Maas, J. Andrew Bagnell, and Anind K. Dey. 2008. Maximum Entropy Inverse Reinforcement Learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3 (AAAI'08).* AAAI Press, 1433–1438.

A SUPPLEMENTARY MATERIAL

Following is a list of rewrite rules used in PROLEX's implementation. x and y are meta-variables for any regex:

- $(x|y)^* \rightarrow (xy?)^*$
- $(x|y)^* \rightarrow (x?y)^*$
- $(x|y)^* \rightarrow ((xy)?)^*$
- $x^* \rightarrow (x^*)^*$

$$\frac{v \in \text{Domain}(\Gamma)}{\Gamma, \mathcal{E} \vdash v \rightarrow \Gamma(v)} \qquad \frac{v \notin \text{Domain}(\Gamma)}{\Gamma, \mathcal{E} \vdash v \rightarrow v} \qquad \frac{\ell \in \text{Domain}(\Gamma) \quad \mathcal{E}.\texttt{objs}(\Gamma(\ell), \tau_o) = O}{\Gamma, \mathcal{E} \vdash \texttt{scanObj}(\tau_o) \rightarrow O}$$

$$\frac{\ell \notin \text{Domain}(\Gamma)}{\Gamma, \mathcal{E} \vdash \texttt{scanObj}(\tau_o) \rightarrow \texttt{scanObj}(\tau_o)} \qquad \frac{\begin{array}{c}\Gamma, \mathcal{E} \vdash \rho \rightarrow [o_1, o_2, \ldots, o_n] \\ \forall_{1 \le i \le n}.\ \pi_i = \pi[v \mapsto o_i] \qquad \Gamma, \mathcal{E} \vdash \pi_1; \pi_2; \ldots \pi_n \rightarrow \pi'\end{array}}{\Gamma, \mathcal{E} \vdash \texttt{foreach}(v \in \rho)\{\pi\} \rightarrow \pi'}$$

$$\frac{\Gamma, \mathcal{E} \vdash \rho \rightarrow \rho' \qquad \Gamma, \mathcal{E} \vdash \pi \rightarrow \pi'}{\Gamma, \mathcal{E} \vdash \texttt{foreach}(v \in \rho)\{\pi\} \rightarrow \texttt{foreach}(v \in \rho')\{\pi'\}}$$

Fig. 17. Partial Evaluation Rules. Γ is a partial store mapping a subset of variables to concrete values. $\Gamma \vdash P \rightarrow P'$ denotes partial evaluation result under Γ

Table 2. PROLEX-DS benchmark set of long-horizon robotic tasks

Task	Description	# loops	# conds	# vars	Source
B1	Sort books by colour into corresponding drawers	1	2	3	Behavior
B2	Grab fruit from the fridge and place them on the counter	1	0	3	Behavior
B3	Box all books up for storage	1	0	2	Behavior
B4	Bring wood into the kitchen and place it near the fridge	2	0	2	Behavior
B5	Brush lint off of all the clothing and place them in the drawer	1	0	4	Behavior
B6	Close all the open doors in the house	2	1	1	Behavior
B7	Grab all bottles and matchboxes and put them in the living room garbage	3	0	3	Behavior
B8	Clean the plates and cups on the table and push in all the chairs	3	3	3	Behavior
B9	Store the recently bought groceries in their proper locations	2	0	4	Behavior
B10	Grab the lamps near the door and place them near the bed	1	1	3	Behavior
B11	go to the kitchen, grab the brush, clean the stove	1	0	3	Behavior
B12	go to the kitchen put the dirty Plates in the dishwasher	1	1	3	Behavior
B13	go to the kitchen and clean all Plates	1	0	2	Behavior
B14	go to the kitchen, put all empty mugs into the sink	1	1	3	Behavior
B15	go to the kitchen, clean all fruits	1	0	2	Behavior
B16	go to the kitchen, grab a cleaning tool, clean all windows	1	0	3	Behavior
B17	go to the bedroom, put all clothes into the drawer	1	0	3	Behavior
B18	go to the kitchen, grab all plates and put them inside the sink	1	0	3	Behavior
B19	go to the livingroom, pickup all trash, take it to the kitchen	1	0	2	Behavior
B20	Go to the kitchen, clean the floor	0	0	3	Behavior
B21	go to the kitchen, open the fridge grab spoiled fruits, put them in the sink	1	1	4	Behavior
B22	go to the bedroom, grab all lights that are off, put them on bed	1	1	3	Behavior
B23	go to all rooms, close all doors	1	0	2	Behavior
B24	go to all rooms, close all windows	1	0	2	Behavior
B25	go to each room, grab all bottles, go to the kitchen, empty all bottles	2	0	3	Behavior
S0	Grab all the dry clothes from the laundry room and put them in a basket	1	1	2	Survey
S1	Grab the towel in the kitchen and clean all the chairs in all the rooms	2	0	2	Survey
S2	Grab all the clothes in the living room, put them in the basket and bring the basket to the bedroom	1	0	2	Survey
S3	Grab all the matchboxes in the living room and put them in the drawer	1	0	2	Survey
S4	Grab all plates and mugs from the dishwasher. Put the plates in the sink, put the mugs in the drawer	2	2	4	Survey
S5	Turn off all lamps in all rooms	2	0	1	Survey
S6	Pickup the basket in the living room and grab all the pillows on all beds	2	1	3	Survey
S7	Grab all red books and put them on the living room table	2	1	2	Survey
S8	Grab all the boxes near the living room door and put them on the table	1	1	3	Survey
S9	Grab the basket in the living room and put it next to the bed in the bedroom. Then grab all the clothes in the bedroom and put them in the basket	1	0	3	Survey
S10	Grab all vegetables and fruits from inside the fridge and place them in the garbage bin	2	1	4	Survey
S11	Open all windows in all rooms, then sweep all the floors, then close all the windows	6	0	3	Survey
S12	Grab an empty box from the living room, go to all the bedroom and grab all the books on the bed and put them in the box. Then take the box back to the living room	2	2	3	Survey
S13	Open the kitchen window if the stove is on	0	1	2	Survey
S14	Go to each bedroom and grab a pillow from the cupboard and put it on the bed	1	0	3	Survey

$$\textsc{(open)}\quad \frac{type(o) \in \{frdige, drawer, box, \dots\} \qquad \mathcal{I}' = \mathcal{E}.\mathcal{I}[(openned, o) \mapsto \top]}{\mathcal{E} \xrightarrow{open,\, o} \mathcal{E}[\mathcal{I} \mapsto \mathcal{I}']}$$

$$\textsc{(put-in)}\quad \frac{\mathcal{E}.\mathcal{I}(open, o_2) = \top \qquad \mathcal{I}' = \mathcal{E}.\mathcal{I}[(empty, o_2) \mapsto \bot][(inside\text{-}of, o_1, o_2) \mapsto \top]}{\mathcal{E} \xrightarrow{put-in,\, o_1, o_2} \mathcal{E}[\mathcal{I} \mapsto \mathcal{I}']}$$

$$\textsc{(grab)}\quad \frac{\begin{array}{c} o = type(o) \qquad \texttt{objs}' = \mathcal{E}.\mathcal{O}.remove((\mathcal{E}.\ell, o), o).append(loc_r, o) \\ \mathcal{I}' = \mathcal{E}.\mathcal{I}[(o, \_, on\text{-}top\text{-}of) \mapsto \bot][(o, \_, inside\text{-}of) \mapsto \bot][\dots] \end{array}}{\mathcal{E} \xrightarrow{grab,\, o} \mathcal{E}[\mathcal{I} \mapsto \mathcal{I}']}$$

Fig. 18. Auxiliary relation $\rightarrow$ for updating environments following robot actions. These rules are specific to each domain and model the dynamics of the robot's environment. A subset of rules implemented for PROLEX, modeling a typical household environment, are shown above. Remember that an environment is defined as $\mathcal{E} := (\mathcal{L}, \mathcal{O}, \ell, \mathcal{I})$. A special location loc_r is used for objects that are being carried by the robot.