Autonomous Robots (2023) 47:999–1012
https://doi.org/10.1007/s10514-023-10135-3

PROGPROMPT: program generation for situated robot task planning using large language models

Ishika Singh[1] · Valts Blukis[2] · Arsalan Mousavian[2] · Ankit Goyal[2] · Danfei Xu[2] · Jonathan Tremblay[2] · Dieter Fox[2,3] · Jesse Thomason[1] · Animesh Garg[2,4]

Received: 1 May 2023 / Accepted: 3 August 2023 / Published online: 28 August 2023
© The Author(s) 2023

Abstract

Task planning can require defining myriad domain knowledge about the world in which a robot needs to act. To ameliorate that effort, large language models (LLMs) can be used to score potential next actions during task planning, and even generate action sequences directly, given an instruction in natural language with no additional domain information. However, such methods either require enumerating all possible next steps for scoring, or generate free-form text that may contain actions not possible on a given robot in its current context. We present a programmatic LLM prompt structure that enables plan generation functional across situated environments, robot capabilities, and tasks. Our key insight is to prompt the LLM with program-like specifications of the available actions and objects in an environment, as well as with example `programs` that can be executed. We make concrete recommendations about prompt structure and generation constraints through ablation experiments, demonstrate state of the art success rates in VirtualHome household tasks, and deploy our method on a physical robot arm for tabletop tasks. Website and code at progprompt.github.io

Keywords Robot task planning · LLM code generation · Planning domain generalization · Symbolic planning

1 Introduction

Everyday household tasks require *both* commonsense understanding of the world and situated knowledge about the current environment. To create a task plan for "Make dinner," an agent needs common sense: *object affordances*, such as that the stove and microwave can be used for heating; *logical sequences of actions*, such as an oven must be preheated before food is added; and *task relevance of objects and actions*, such as heating and food are actions related to "dinner" in the first place. However, this reasoning is infeasible without *state feedback*. The agent needs to know what food is available *in the current environment*, such as whether the freezer contains fish or the fridge contains chicken.

Autoregressive large language models (LLMs) trained on large corpora to *generate* text sequences conditioned on input *prompts* have remarkable multi-task generalization. This ability has recently been leveraged to generate plau-

✉ Ishika Singh
ishikasi@usc.edu

Valts Blukis
vblukis@nvidia.com

Arsalan Mousavian
amousavian@nvidia.com

Ankit Goyal
angoyal@nvidia.com

Danfei Xu
danfeix@nvidia.com

Jonathan Tremblay
jtremblay@nvidia.com

Dieter Fox
dieterf@nvidia.com

Jesse Thomason
jessetho@usc.edu

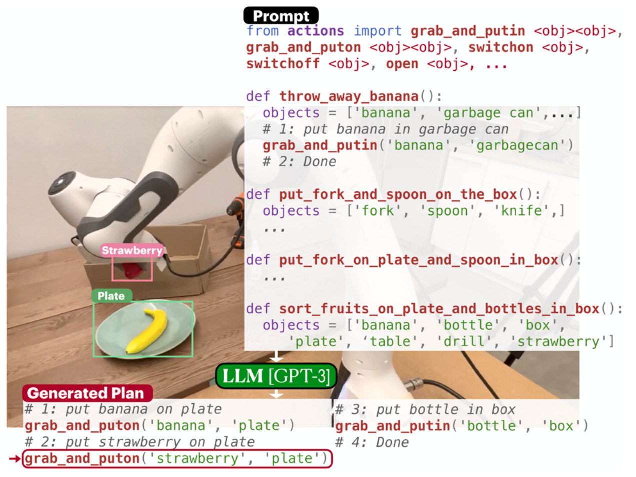
Animesh Garg
animeshg@nvidia.com

1 Computer Science, University of Southern California, Los Angeles, CA 90089, USA

2 Seattle Robotics Lab, NVIDIA, Seattle, WA 98105, USA

3 Computer Science and Engineering, University of Washington, Seattle, WA 98195, USA

4 School of Interactive Computing, Georgia Institute of Technology, Atlanta, GA 30308, USA



```
from actions import grab_and_putin <obj><obj>,
grab_and_puton <obj><obj>, switchon <obj>,
switchoff <obj>, open <obj>, ...

def throw_away_banana():
  objects = ['banana', 'garbage can',...]
  # 1: put banana in garbage can
  grab_and_putin('banana', 'garbagecan')
  # 2: Done

def put_fork_and_spoon_on_the_box():
  objects = ['fork', 'spoon', 'knife',]
  ...

def put_fork_on_plate_and_spoon_in_box():
  ...

def sort_fruits_on_plate_and_bottles_in_box():
  objects = ['banana', 'bottle', 'box',
     'plate', ‘table', 'drill', 'strawberry']
```

```
# 1: put banana on plate
grab_and_puton('banana', 'plate')
# 2: put strawberry on plate
grab_and_puton('strawberry', 'plate')
# 3: put bottle in box
grab_and_putin('bottle', 'box')
# 4: Done
```

Fig. 1 ProgPrompt leverages LLMs' strengths in both world knowledge and programming language understanding to generate situated task plans that can be directly executed

sible action plans in context of robotic task planning (Ahn et al., 2022; Huang et al., 2022b, a; Zeng et al., 2022) by either scoring next steps or generating new steps directly. In scoring mode, the LLM evaluates an enumeration of actions and their arguments from the space of what's possible. For instance, given a goal to "Make dinner" with first action being "open the fridge", the LLM could score a list of possible actions: "pick up the chicken", "pick up the soda", "close the fridge", . . . , "turn on the lightswitch." In text-generation mode, the LLM can produce the next few words, which then need to be mapped to actions and world objects available to the agent. For example, if the LLM produced "reach in and pick up the jar of pickles," that string would have to neatly map to an executable action like "pick up jar." A key component missing in LLM-based task planning is state feedback from the environment. The fridge in the house might not contain chicken, soda, or pickles, but a high-level instruction "Make dinner" doesn't give us that world state information. **Our work introduces situated-awareness in LLM-based robot task planning.**

We introduce ProgPrompt, a prompting scheme that goes beyond conditioning LLMs in natural language. ProgPrompt utilizes *programming language* structures, leveraging the fact that LLMs are trained on vast web corpora that includes many programming tutorials and code documentation (Fig. 1). ProgPrompt provides an LLM a Pythonic program header with an `import` statement for available actions and their expected parameters, a `list` of environment objects, and function definitions like `make_dinner` whose bodies are sequences of actions operating on objects. We incorporate situated state feedback from the environment by asserting preconditions of our plan, such as being close to the fridge before attempting to open it, and responding to failed assertions with recovery actions. What's more, we show that including natural language *comments* in ProgPrompt programs to explain the goal of the upcoming action improves task success of generated plan programs.

2 Background and related work

2.1 Task planning

For high-level planning, most works in robotics use search in a pre-defined domain (Fikes and Nilsson, 1971; Jiang et al., 2018; Garrett et al., 2020). Unconditional search can be hard to scale in environments with many feasible actions and objects (Puig et al., 2018; Shridhar et al., 2020) due to large branching factors. Heuristics are often used to guide the search (Baier et al., 2007; Hoffmann, 2001; Helmert, 2006; Bryce and Kambhampati, 2007). Recent works have explored learning-based task & motion planning, using methods such as representation learning, hierarchical learning, language as planning space, learning compositional skills and more (Akakzia et al., 2021; Eysenbach et al., 2019; Jiang et al., 2019; Kurutach et al., 2018; Mirchandani et al., 2021; Nair and Finn, 2020; Shah et al., 2022; Sharma et al., 2022; Silver et al., 2022; Srinivas et al., 2018; Xu et al., 2018, 2019; Zhu et al., 2020). Our method *sidesteps* search to directly generate a plan that includes conditional reasoning and error-correction.

We formulate task planning as the tuple $\langle \mathcal{O}, \mathcal{P}, \mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G}, \mathbf{t} \rangle$. $\mathcal{O}$ is a set of all the objects available in the environment, $\mathcal{P}$ is a set of properties of the objects which also informs object affordances, $\mathcal{A}$ is a set of executable actions that changes depending on the current environment state defined as $\mathbf{s} \in \mathcal{S}$. A state $\mathbf{s}$ is a specific assignment of all object properties, and $\mathcal{S}$ is a set of all possible assignments. $\mathcal{T}$ represents the transition model $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$, $\mathcal{I}$ and $\mathcal{G}$ are the initial and goal states. The agent does not have access to the goal state $\mathbf{g} \in \mathcal{G}$, but only a high-level task description $\mathbf{t}$.

Consider the task $\mathbf{t}$ = "*microwave salmon*". Task relevant objects *microwave*, *salmon* $\in \mathcal{O}$ will have properties modified during action execution. For example, action $\mathbf{a} = \texttt{open}(\textit{microwave})$ will change the state from $\texttt{closed}(\textit{microwave}) \in \mathbf{s}$ to $\neg\texttt{closed}(\textit{microwave}) \in \mathbf{s}'$ if $\mathbf{a}$ is admissible, i.e., $\exists(\mathbf{a}, \mathbf{s}, \mathbf{s}')$ $s.t.$ $\mathbf{a} \in \mathcal{A} \wedge \mathbf{s}, \mathbf{s}' \in \mathcal{S} \wedge \mathcal{T}(\mathbf{s}, \mathbf{a}) = \mathbf{s}'$. In this example a goal state $\mathbf{g} \in \mathcal{G}$ could contain the conditions $\texttt{heated}(\textit{salmon}) \in \mathbf{g}$, $\neg\texttt{closed}(\textit{microwave}) \in \mathbf{g}$ and $\neg\texttt{switchedOn}(\textit{microwave}) \in \mathbf{g}$.

2.2 Planning with LLMs

A Large Language Model (LLM) is a neural network with many parameters—currently hundreds of billions (Brown et al., 2020; Chen et al., 2021)—trained on unsupervised learning objectives such as next-token prediction or maskedlanguage modelling. An autoregressive LLM is trained with a maximum likelihood loss to model the probability of a sequence of tokens $\mathbf{y}$ conditioned on an input sequence $\mathbf{x}$, i.e. $\theta = \arg\max_{\theta} P(\mathbf{y}|\mathbf{x};\theta)$, where θ are model parameters. The trained LLM is then used for prediction $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}\in\mathbb{S}} P(\mathbf{y}|\mathbf{x};\theta)$, where $\mathbb{S}$ is the set of all text sequences. Since search space $\mathbb{S}$ is huge, approximate decoding strategies are used for tractability (Holtzman et al., 2020; Luong et al., 2015; Wiseman et al., 2017).

LLMs are trained on large text corpora, and exhibit multi-task generalization when provided with a relevant prompt input $\mathbf{x}$. Prompting LLMs to generate text useful for robot task planning is a nascent topic (Ahn et al., 2022; Jansen, 2020; Huang et al., 2022a, b; Li et al., 2022; Patel and Pavlick, 2022). Prompt design is challenging given the lack of paired natural language instruction text with executable plans or robot action sequences (Liu et al., 2021). Devising a prompt for task plan prediction can be broken down into a *prompting function* and an *answer search* strategy (Liu et al., 2021). A *prompting function*, $f_{\text{prompt}}(.)$ transforms the input state observation $\mathbf{s}$ into a textual prompt. *Answer search* is the generation step, in which the LLM outputs from the entire LLM vocabulary or scores a predefined set of options.

Closest to our work, Huang et al. (2022a) generates open-domain plans using LLMs. In that work, planning proceeds by: 1) selecting a similar task in the prompt example (f_{prompt}); 2) open-ended task plan generation (answer search); and 3) 1:1 prediction to action matching. The entire plan is generated *open-loop without any environment interaction*, and later tested for executability of matched actions. However, action matching based on generated text doesn't ensure the action is admissible in the current situation. INNER-MONOLOGUE (Huang et al., 2022b) introduces environment feedback and state monitoring, but still found that LLM planners proposed actions involving objects not present in the scene. Our work shows that a *programming language-inspired prompt generator* can inform the LLM of both situated environment state and available robot actions, ensuring output compatibility to robot actions.

The related SAYCAN (Ahn et al., 2022) uses natural language prompting with LLMs to generate a set of feasible planning steps, re-scoring matched admissible actions using a learned value function. SayCan constructs a set of all admissible actions expressed in natural language and scores them using an LLM. This is challenging to do in environments with combinatorial action spaces. Concurrent with our work are Socratic models (Zeng et al., 2022), which also use code-completion to generate robot plans. We go beyond (Zeng et al., 2022) by leveraging additional, familiar features of programming languages in our prompts. We define an f_{prompt} that includes import statements to model robot capabilities, natural language comments to elicit common sense reasoning, and assertions to track execution state. Our answer search is performed by allowing the LLM to generate an entire, executable plan program directly.

2.3 Recent developments following PROGPROMPT

Vemprala et al. (2023) further explores API-based planning with ChatGPT[1] in domains such as aerial robotics, manipulation and visual navigation. They discuss the design principles for constructing interaction APIs, for action and perception, and prompts that can be used to generate code for robotic applications. Huang et al. (2023) builds on SAYCAN (Ahn et al., 2022) and generates planning steps token-by-token while scoring the tokens using both the LLM and the grounded pretrained value function. Cao and Lee (2023) explores generating behavior trees to study hierarchical task planning using LLMs. Skreta et al. (2023) proposes iterative error correction via a syntax verifier that repeatedly prompts the LLM with previous query appended with a list of errors. Mai et al. (2023), similar in approach as Zeng et al. (2022), Huang et al. (2022b), integrates pretrained models for perception, planning, control, memory, and dialogue zero-shot, for active exploration and embodied question answering tasks. Gupta and Kembhavi (2022) extends the LLM code generation and API-based perceptual interaction approach for a variety of vision-langauge tasks. Some recent works Xie et al. (2023a), Capitanelli and Mastrogiovanni (2023) use PDDL as the translation language instead of code, and use the LLM to generate either a PDDL plan or the goal. A classical planner then plans for the PDDL goal or executes the generated plan. This approach ablated the need to generate preconditions using the LLM, however, needs the domain rules to be specified for the planner.

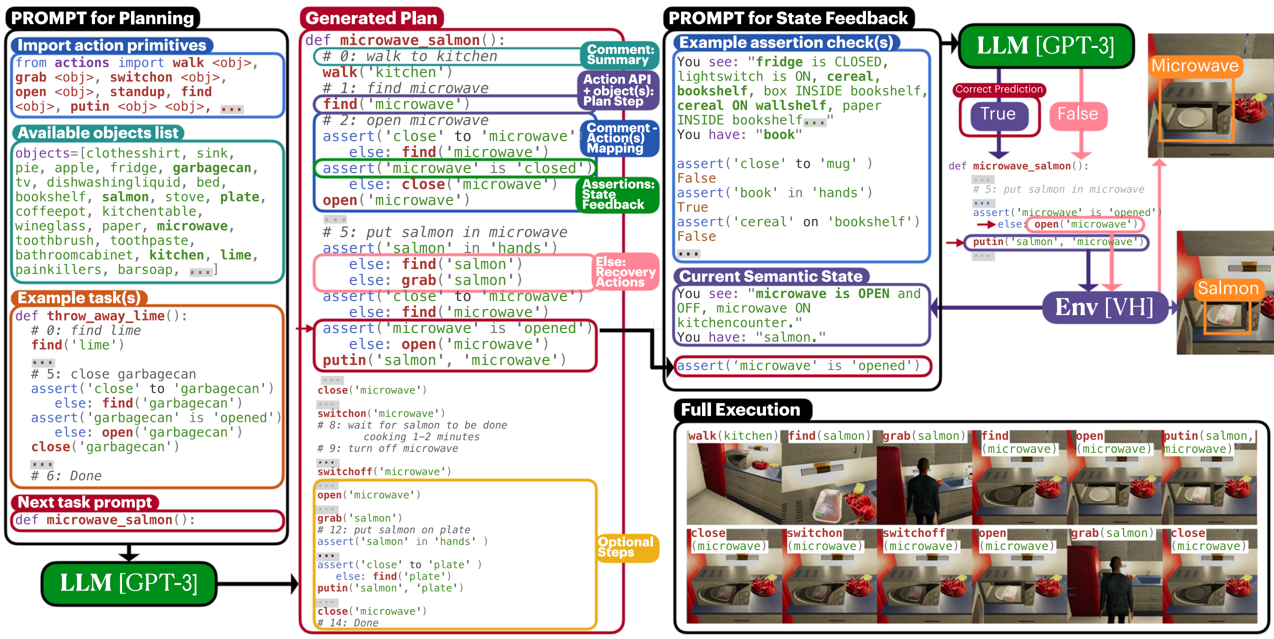
3 Our method: PROGPROMPT

We represent robot plans as *pythonic* programs. Following the paradigm of LLM prompting, we create a prompt structured as pythonic code and use an LLM to complete the code (Fig. 2). We use features available in Python to construct prompts that elicit an LLM to generate situated robot task plans, conditioned on a natural language instruction.

3.1 Representing robot plans as pythonic functions

Plan functions consist of API calls to action primitives, comments to summarize actions, and assertions for tracking execution (Fig. 3). Primitive actions use objects as arguments. For example, the "*put salmon in the microwave*" task includes API calls like `find`(*salmon*).

[1] https://openai.com/blog/chatgpt/



```
from actions import walk <obj>,
grab <obj>, switchon <obj>,
open <obj>, standup, find
<obj>, putin <obj> <obj>, ...
```

```
objects=[clothesshirt, sink,
pie, apple, fridge, garbagecan,
tv, dishwashingliquid, bed,
bookshelf, salmon, stove, plate,
coffeepot, kitchentable,
wineglass, paper, microwave,
toothbrush, toothpaste,
bathroomcabinet, kitchen, lime,
painkillers, barsoap, ...]
```

```
def throw_away_lime():
  # 0: find lime
  find('lime')
  ...
  # 5: close garbagecan
  assert('close' to 'garbagecan')
     else: find('garbagecan')
  assert('garbagecan' is 'opened')
     else: open('garbagecan')
  close('garbagecan')
  ...
  # 6: Done
```

```
def microwave_salmon():
```

```
def microwave_salmon():
  # 0: walk to kitchen
  walk('kitchen')
  # 1: find microwave
  find('microwave')
  # 2: open microwave
  assert('close' to 'microwave')
     else: find('microwave')
  assert('microwave' is 'closed')
     else: close('microwave')
  open('microwave')
  ...
  # 5: put salmon in microwave
  assert('salmon' in 'hands')
     else: find('salmon')
     else: grab('salmon')
  assert('close' to 'microwave')
     else: find('microwave')
  assert('microwave' is 'opened')
     else: open('microwave')
  putin('salmon', 'microwave')
  ...
  close('microwave')
  ...
  switchon('microwave')
  # 8: wait for salmon to be done
        cooking 1-2 minutes
  # 9: turn off microwave
  ...
  switchoff('microwave')
  ...
  open('microwave')
  ...
  grab('salmon')
  # 12: put salmon on plate
  assert('salmon' in 'hands' )
  ...
  assert('close' to 'plate' )
     else: find('plate')
  putin('salmon', 'plate')
  ...
  close('microwave')
  # 14: Done
```

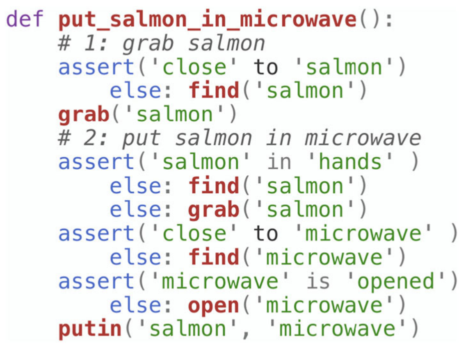
```
You see: "fridge is CLOSED,
lightswitch is ON, cereal,
bookshelf, box INSIDE bookshelf,
cereal ON wallshelf, paper
INSIDE bookshelf..."
You have: "book"

assert('close' to 'mug' )
False
assert('book' in 'hands')
True
assert('cereal' on 'bookshelf')
False
...
```

```
You see: "microwave is OPEN and
OFF, microwave ON
kitchencounter."
You have: "salmon."
```

```
assert('microwave' is 'opened')
```

```
def microwave_salmon():
  ...
  # 5: put salmon in microwave
  ...
  assert('microwave' is 'opened')
  else: open('microwave')
  putin('salmon', 'microwave')
  ...
```

Fig. 2 Our ProgPrompts include `import` statement, object list, and example tasks (*PROMPT for Planning*). The *Generated Plan* is for `microwave salmon`. We highlight prompt comments, actions as imported function calls with objects as arguments, and assertions with recovery steps. *PROMPT for State Feedback* represents example assertion checks. We further illustrate the execution of the program via a scenario where an assertion succeeds or fails, and how the generated plan corrects the error before executing the next step. *Full Execution* is shown in bottom-right. '...' used for brevity



```
def put_salmon_in_microwave():
    # 1: grab salmon
    assert('close' to 'salmon')
        else: find('salmon')
    grab('salmon')
    # 2: put salmon in microwave
    assert('salmon' in 'hands' )
        else: find('salmon')
        else: grab('salmon')
    assert('close' to 'microwave' )
        else: find('microwave')
    assert('microwave' is 'opened')
        else: open('microwave')
    putin('salmon', 'microwave')
```

Fig. 3 Pythonic ProgPrompt plan for "*put salmon in the microwave*"

We utilize comments in the code to provide natural language summaries for subsequent sequences of actions. Comments help break down the high-level task into logical sub-tasks. For example, in Fig. 3, the "*put salmon in microwave*" task is broken down into sub-tasks using comments "# grab salmon" and "# put salmon in microwave". This partitioning could help the LLM to express its knowledge about tasks and sub-tasks in natural language and aid planning. Comments also inform the LLM about immediate goals, reducing the possibility of incoherent, divergent, or repetitive outputs. Prior work Wei et al. (2022) has also shown the efficacy of similar intermediate summaries called 'chain of thought' for improving performance of LLMs on a range of arithmetic, commonsense, and symbolic reasoning tasks. We empirically verify the utility of comments (Table 1; column Comments).

Assertions provide an environment feedback mechanism that encourages preconditions to be met, and allow error recovery possibility when they are not. For example, in Fig. 3, before the `grab`(*salmon*) action, the plan asserts the agent is `close` to *salmon*. If not, the agent first executes `find`(*salmon*). In Table 1, we show that such assert statements (column Feedback) benefit plan generation, and improve success rates.

3.2 Constructing programming language prompts

We provide information about the environment and primitive actions to the LLM through prompt construction. As done in few-shot LLM prompting, we also provide the LLM with examples of sample tasks and plans. Figure 2 illustrates our prompt function f_{prompt} which takes in all the information (observations, action primitives, examples) and produces a Pythonic prompt for the LLM to complete. The LLM then predicts the `<next_task>`(.) as an executable function (`microwave_salmon` in Fig. 2).

In the task `microwave_salmon`, a reasonable first step that an LLM could generate is `take_out`(*salmon*, *grocery* *bag*). However, the agent responsible for executing the plan might not have a primitive action to `take_out`. To inform the LLM about the agent's action primitives, we provide them as Pythonic `import` statements. These encourage the LLM to restrict its output to only functions that are available in the current context. To change agents, ProgPrompt just needs a new list of imported functions representing agent actions. A *grocery bag* object might also not exist in the environment. We provide the available objects in the environment as a list of strings. Since our prompting scheme explicitly lists out the set of functions and objects available to the model, the generated plans typically contain actions an agent can take and objects available in the environment.

ProgPrompt also includes a few example tasks—fully executable program plans. Each example task demonstrates how to complete a given task using available actions and objects in the given environment. These examples demonstrate the relationship between task name, given as the function handle, and actions to take, as well as the restrictions on actions and objects to involve.

3.3 Task plan generation and execution

The given task is fully inferred by the LLM based on the ProgPrompt prompt. Generated plans are executed on a virtual agent or a physical robot system using an interpreter that executes each action command against the environment. Assertion checking is done in a closed-loop manner during execution, providing current environment state feedback.

4 Experiments

We evaluate our method with experiments in a virtual household environment and on a physical robot manipulator.

4.1 Simulation experiments

We evaluate our method in the Virtual Home (VH) Environment (Puig et al., 2018), a deterministic simulation platform for typical household activities. A VH state $\mathbf{s}$ is a set of objects $\mathcal{O}$ and properties $\mathcal{P}$. $\mathcal{P}$ encodes information like `in`(*salmon*, *microwave*) and `agent_close_to`(*salmon*). The action space is $\mathcal{A}$ = `{grab, putin, putback, walk, find, open, close, switchon, switchoff, sit, standup}`.

We experiment with 3 VH environments. Each environment contains 115 unique object instances (Fig. 2), including class-level duplicates. Each object has properties corresponding to its action affordances. Some objects also have a semantic state like `heated`, `washed`, or `used`. For example, an object in the *Food* category can become `heated` whenever `in`(*object*, *microwave*) $\wedge$ `switched_on`(*microwave*).

We create a dataset of 70 household tasks. Tasks are posed with high-level instructions like "*microwave salmon*". We collect a ground-truth sequence of actions that completes the task from an initial state, and record the final state $\mathbf{g}$ that defines a set of symbolic goal conditions, $\mathbf{g} \in \mathcal{P}$.

When executing generated programs, we incorporate environment state feedback in response to assertions. VH provides observations in the form of state graph with object properties and relations. To check assertions in this environment, we extract information about the relevant object from the state graph and prompt the LLM to return whether the assertion holds or not given the state graph and assertion as a text prompt (Fig. 2 *Prompt for State Feedback*). We choose this design over a rule-based checking since it's more general.

4.2 Real-robot experiments

We use a Franka-Emika Panda robot with a parallel-jaw gripper. We assume access to a pick-and-place policy. The policy takes as input two pointclouds of a target object and a target container, and performs a pick-and-place operation to place the object on or inside the container. We use the system of Danielczuk et al. (2021) to implement the policy, and use MPPI for motion generation, SceneCollisionNet (Danielczuk et al., 2021) to avoid collisions, and generate grasp poses with Contact-GraspNet (Sundermeyer et al., 2021).

We specify a single `import` statement for the action `grab_and_putin(obj1, obj2)` for ProgPrompt. We use ViLD (Gu et al., 2022), an open-vocabulary object detection model, to identify and segment objects in the scene and construct the available object list for the prompt. Unlike in the virtual environment, where object list was a global variable in common for all tasks, here the object list is a local variable for each plan function, which allows greater flexibility to adapt to new objects. The LLM outputs a plan containing function calls of form `grab_and_putin(obj1, obj2)`. Here, objects `obj1` and `obj2` are text strings that we map to pointclouds using ViLD segmentation masks and the depth image. Due to real world uncertainty, we do not implement assertion-based closed loop options on the tabletop plans.

4.3 Evaluation metrics

We use three metrics to evaluate system performance: success rate (*SR*), executability (*Exec*), and goal conditions recall (*GCR*). The task-relevant goal-conditions are the set of goal-conditions that changed between the initial and final state in the demonstration. *SR* is the fraction of executions that achieved all task-relevant goal-conditions. *Exec* is the fraction of actions in the plan that are executable in the environment, even if they are not relevant for the task. *GCR* is measured using the set difference between ground truth final state conditions $\mathbf{g}$ and the final state achieved $\mathbf{g}'$ with the

Table 1 Evaluation of generated programs on Virtual Home

#	— Prompt Format and Parameters —						
	Format	COMMENTS	FEEDBACK	LLM Backbone	*SR*	*Exec*	*GCR*
*	PROGPROMPT	✓	✓	GPT4	0.37 ± 0.06	0.87 ± 0.01	0.64 ± 0.02
*	PROGPROMPT	✓	✓	DAVINCI- 003	**0.47** ± 0.15	0.85 ± 0.02	**0.74**± 0.07
1	PROGPROMPT	✓	✓	CODEX	0.40 ± 0.11	**0.90** ± 0.05	0.72 ± 0.09
2	PROGPROMPT	✓	✓	DAVINCI	0.22 ± 0.04	0.60 ± 0.04	0.46 ± 0.04
3	PROGPROMPT	✓	✓	GPT3	*0.34* ± 0.08	*0.84* ± 0.01	*0.65* ± 0.05
4	PROGPROMPT	✓	✗	GPT3	0.28 ± 0.04	0.82 ± 0.01	0.56 ± 0.02
5	PROGPROMPT	✗	✓	GPT3	0.30 ± 0.00	0.65 ± 0.01	0.58 ± 0.02
6	PROGPROMPT	✗	✗	GPT3	0.18 ± 0.04	0.68 ± 0.01	0.42 ± 0.02
7	LANGPROMPT	–	–	GPT3	0.00 ± 0.00	0.36 ± 0.00	0.42 ± 0.02
8	Baseline from HUANG ET AL.			GPT3	0.00 ± 0.00	0.45 ± 0.03	0.21 ± 0.03

PROGPROMPT uses 3 fixed example programs, except the DAVINCI backbone which can fit only 2 in the available API. Huang et al. (2022a) use 1 dynamically selected example, as described in their paper. LANGPROMPT uses 3 natural language text examples. Best performing model with a GPT3 backbone is shown in *italic* (used for our ablation studies); best performing model overall shown in **bold**. PROGPROMPT significantly outperforms the baseline Huang et al. (2022a) and LANGPROMPT. We also showcase how each PROGPROMPT feature adds to the performance of the method

Table 2 PROGPROMPT performance on the VH test-time tasks and their ground truth actions sequence lengths $|A|$

Task desc	$\|A\|$	*SR*	*Exec*	*GCR*
watch tv	3	0.20 ± 0.40	0.42 ± 0.13	0.63 ± 0.28
turn off light	3	0.40 ± 0.49	1.00 ± 0.00	0.65 ± 0.30
brush teeth	8	0.80 ± 0.40	0.74 ± 0.09	0.87 ± 0.26
throw away apple	8	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
make toast	8	0.00 ± 0.00	1.00 ± 0.00	0.54 ± 0.33
eat chips on the sofa	5	0.00 ± 0.00	0.40 ± 0.00	0.53 ± 0.09
put salmon in the fridge	8	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
wash the plate	18	0.00 ± 0.00	0.97 ± 0.04	0.48 ± 0.11
bring coffeepot and cupcake to the coffee table	8	0.00 ± 0.00	1.00 ± 0.00	0.52 ± 0.14
microwave salmon	11	0.00 ± 0.00	0.76 ± 0.13	0.24 ± 0.09
Avg: $0 \leq \|A\| \leq 5$		0.20 ± 0.40	0.61 ± 0.29	0.60 ± 0.25
Avg: $6 \leq \|A\| \leq 10$		0.60 ± 0.50	0.95 ± 0.11	0.79 ± 0.29
Avg: $11 \leq \|A\| \leq 18$		0.00 ± 0.00	0.87 ± 0.14	0.36 ± 0.16

generated plan, divided by the number of task-specific goal-conditions; *SR*= 1 only if *GCR*= 1.

5 Results

We show that PROGPROMPT is an effective method for prompting LLMs to generate task plans for both virtual and physical agents.

5.1 Virtual experiment results

Table 1 summarizes the performance of our task plan generation and execution system in the *seen* environment of VirtualHome. We utilize a GPT3 as a language model backbone to receive PROGPROMPT prompts and generate plans. Each result is averaged over 5 runs in a single VH environment across 10 tasks. The variability in performance across runs arises from sampling LLM output. We include 3 Pythonic task plan examples per prompt after evaluating performance on VH for between 1 prompt and 7 prompts and finding that 2 or more prompts result in roughly equal performance for GPT3. The plan examples are fixed to be: "*put the wine glass in the kitchen cabinet*", "*throw away the lime*", and "*wash mug*".

We also include results on the recent GPT4 backbone. Unlike the GPT3 language model, GPT4 is a chat-bot model trained with reinforcement learning with human feedback (RLHF) to act as a helpful digital assistant OpenAI (2023). GPT4 takes as input a system prompt followed by one or more user prompts. Instead of simply auto-completing the code in the prompt, GPT4 interprets user prompts as questions and generates answers as an assistant. To make GPT4 autocomplete our prompt, we used the following system prompt: *You are a helpful assistant.*. The user prompt is the same f_{prompt} as shown in Fig. 2.

We can draw several conclusions from Table 1. First, ProgPrompt (rows 3–6) outperforms prior work (Huang et al., 2022a) (row 8) by a substantial margin on all metrics using the same large language model backbone. Second, we observe that the Codex (Chen et al., 2021) and Davinci models (Brown et al., 2020)—themselves GPT3 variants—show mixed success at the task. In particular, Davinci, the original GPT3 version, does not match base GPT3 performance (row 2 versus row 3), possibly because its prompt length constraints limit it to 2 task examples versus the 3 available to other rows. Additionally, Codex *exceeds* GPT3 performance on every metric (row 1 versus row 3), likely because Codex is explicitly trained on programming language data. However, Codex has limited access in terms of number of queries per minute, so we continue to use GPT3 as our main LLM backbone in the following ablation experiments. Our recommendation to the community is to utilize a program-like prompt for LLM-based task planning and execution, for which base GPT3 works well, and we note that an LLM fine-tuned further on programming language data, such as Codex, can do even better. We additionally report results on Davinci- 003 and GPT4 (row *), which is the latest GPT3 variant and the latest GPT variant in the series respectively at the time of this submission. Davinci- 003 has a better *SR* and *GCR*, indicating it might have an improved commonsense understanding, but lower *Exec* compared to Codex. The newest model, GPT- 4 does not seem to be better than latest GPT3 variant, on our tasks. Most of our results use the Davinci- 002 variant (that we refer to as GPT3 in this paper), which was the latest model available when this study was conducted.

We explore several ablations of ProgPrompt. First, we find that Feedback mechanisms in the example programs, namely the assertions and recovery actions, improve performance (rows 3 versus 4 and 5 versus 6) across metrics, the sole exception being that *Exec* improves a bit without Feedback when there are no Comments in the prompt example code. Second, we observe that removing Comments from the prompt code substantially reduces performance on all metrics (rows 3 versus 5 and 4 versus 6), highlighting the usefulness of the natural language guidance within the programming language structure.

We also evaluate LangPrompt, an alternative to ProgPrompt that builds prompts from natural language text description of objects available and example task plans (row 7). LangPrompt is similar to the prompts built by Huang et al. (2022a). The outputs of LangPrompt are generated action sequences, rather than our proposed, program-like structures. Thus, we finetune GPT2 to learn a policy $P(\mathbf{a}_t|\mathbf{s}_t, \text{GPT3 step}, \mathbf{a}_{1:t-1})$ to map those generated sequences to executable actions in the simulation environment. We use the 35 tasks in the training set, and annotate the text steps and the corresponding action sequence to get 400 data points for training and validation of this policy. We find that while this method achieves reasonable partial success through *GCR*, it does not match (Huang et al., 2022a) for program executability *Exec* and does not generate any fully successful task executions.

Table 3 ProgPrompt results on Virtual Home in additional scenes. We evaluate on 10 tasks each in two additional VH scenes beyond scene Env- 0 where other reported results take place

VH scene	*SR*	*Exec*	*GCR*
Env- 0	0.34 ± 0.08	0.84 ± 0.01	0.65 ± 0.05
Env- 1	0.56 ± 0.08	0.85 ± 0.02	0.81 ± 0.07
Env- 2	0.56 ± 0.05	0.85 ± 0.03	0.72 ± 0.09
Average	0.48 ± 0.13	0.85 ± 0.02	0.73 ± 0.10

Task-by-Task Performance ProgPrompt performance for each task in the test set is shown in Table 2. We observe that tasks that are similar to prompt examples, such as *throw away apple* versus *wash the plate* have higher *GCR* since the ground truth prompt examples hint about good stopping points. Even with high *Exec*, some task *GCR* are low, because some tasks have multiple appropriate goal states, but we only evaluate against a single "true" goal. For example, after microwaving and plating salmon, the agent may put the salmon on a table or a countertop.

Other Environments We evaluate ProgPrompt in two additional VH environments (Table 3). For each, we append a new object list representing the new environment after the example tasks in the prompt, followed by the task to be completed in the new scene. The action primitives and other ProgPrompt settings remain unchanged. We evaluate on 10 tasks with 5 runs each. For new tasks like *wash the cutlery in dishwasher*, ProgPrompt is able to infer that *cutlery* refers to *spoons* and *forks* in the new scenes, despite that *cutlery* always refers to *knives* in example prompts.

5.2 Qualitative analysis and limitations

We manually inspect generated programs and their execution traces from ProgPrompt and characterize common failure modes. Many failures stem from the decision to make ProgPrompt *agnostic* to the deployed environment and its peculiarities, which may be resolved through explicitly communicating, for example, object affordances of the target environment as part of the ProgPrompt prompt.

- *Environment artifacts*: the VH agent cannot find or interact with objects nearby when `sitting`, and some

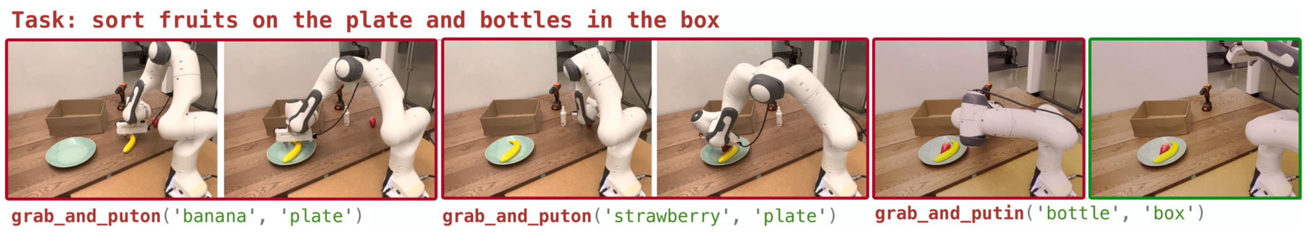

Task: sort fruits on the plate and bottles in the box
grab_and_puton('banana', 'plate')
grab_and_puton('strawberry', 'plate')
grab_and_putin('bottle', 'box')

Fig. 4 Robot plan execution rollout example on the sorting task showing relevant objects *banana*, *strawberry*, *bottle*, *plate* and *box*, and a distractor object *drill*. The LLM recognizes that *banana* and *strawberry* are *fruits*, and generates plan steps to place them on the *plate*, while placing the *bottle* in the *box*. The LLM ignores the distractor object *drill*. See Fig. 1 for the prompt structure used

common sense actions for objects, such as `open` *tvstand*'s cabinets, are not available in VH.

- *Environment complexities*: when an object is not accessible, the generated assertions might not be enough. For example, if the agent finds an object in a *cabinet*, it may not plan to `open` the *cabinet* to `grab` the object.
- *Action success feedback* is not provided to the agent, which may lead to failure of the subsequent actions. Assertion recovery modules in the plan can help, but aren't generated to cover all possibilities.
- *Incomplete generation:* Some plans are cut short by LLM API caps. One possibility is to query the LLM again with the prompt and partially generated plan.

In addition to these failure modes, our strict final state checking means if the agent completes the task *and some*, we may infer failure, because the environment goal state will not match our precomputed ground truth final goal state. For example, after making *coffee*, the agent may take the *coffeepot* to another *table*. Similarly, some task descriptions are ambiguous and have multiple plausible correct programs. For example, "*make dinner*" can have multiple possible solutions. ProgPrompt generates plans that cooks *salmon* using the *fryingpan* and *stove*, and sometimes the agent adds *bellpepper* or *lime*, or sometimes with a side of *fruit*, or served in a *plate* with *cutlery*. When run in a different VH environment, the agent cooks *chicken* instead. ProgPrompt is able to generate plans for such complex tasks as well while using the objects available in the scene and not explicitly mentioned in the task. However, automated evaluation of such tasks requires enumerating all valid and invalid possibilities or introducing human verification.

Furthermore, we note that while the reasoning capabilities of current state LLMs are impressive, our proposed method does not make any claims of providing guarantees. However, the evaluations reported in Table 1 offer insights into the capabilities of different LLMs within our task settings. While our method effectively prevents the LLM from generating unavailable actions or objects, it is worth acknowledging that depending on the LLM's generation quality and reasoning capabilities, there is still a possibility of hallucination.

Table 4 Results on the physical robot by task type

Task description	**Distractors**	*SR*	Plan *SR*	*GCR*
put the banana in the bowl	0	1	1	1/1
	4	1	1	1/1
put the pear on the plate	0	1	1	1/1
	4	1	1	1/1
put the banana on the plate	0	1	1	2/2
and the pear in the bowl	3	1	1	2/2
sort the fruits on the plate	0	0	1	2/3
and the bottles in the box	1	1	1	3/3
	2	0	0	2/3

5.3 Physical robot results

The physical robot results are shown in Table 4. We evaluate on 4 tasks of increasing difficulty listed in Table 4. For each task we perform two experiments: one in a scene that only contains the necessary objects, and with one to four distractor objects added.

All results shown use ProgPrompt with comments, but not feedback. Our physical robot setup did not allow reliably tracking system state and checking assertions, and is prone to random failures due to things like grasps slipping. The real world introduces randomness that complicates a quantitative comparison between systems. Therefore, we intend the physical results to serve as a qualitative demonstration of the ease with which our prompting approach allows constraining and grounding LLM-generated plans to a physical robot system. We report an additional metric *Plan SR*, which refers to whether the plan would have *likely succeeded*, provided successful pick-and-place execution without gripper failures.

Across tasks, with and without distractor objects, the system almost always succeeds, failing only on the *sort* task. The run without distractors failed due to a random gripper failure. The run with 2 distractors failed because the model mistakenly considered a *soup can* to be a *bottle*. The executability for the generated plans was always *Exec*=1. An execution rollout example is illustrated in Fig. 4.

After this study was conducted, we re-attempted plan generation of the failed plan with GPT- 4, using the same system prompt as in Sect. 5.1. GPT- 4 was able to successfully predict the correct plan and not confuse the soup can for a bottle.

6 Conclusions and future work

We present an LLM prompting scheme for robot task planning that brings together the two strengths of LLMs: commonsense reasoning and code understanding. We construct prompts that include situated understanding of the world and robot capabilities, enabling LLMs to directly generate executable plans as programs. Our experiments show that ProgPrompt programming language features improve task performance across a range of metrics. Our method is intuitive and flexible, and generalizes widely to new scenes, agents and tasks, including a real-robot deployment.

As a community, we are only scratching the surface of task planning as robot plan generation and completion. We hope to study broader use of programming language features, including real-valued numbers to represent measurements, nested dictionaries to represent scene graphs, and more complex control flow. Several works from the NLP community show that LLMs can do arithmetic and understand numbers, yet their capabilities for complex robot behavior generation are still relatively under-explored.

7 FAQs and discussion

Question 1 *How does this approach compare with end-to-end robot learning models, and what are the current limitations?*

ProgPrompt is a hierarchical solution to task planning where the abstract task descriptions leverage LLM's reasoning and maps the task plan to the grounded environment labels. On the other hand, in end-to-end approaches, generally the model implicitly learns reasoning, planning, and grounding, while mapping the abstract task description to the action space directly.

Pros:

- LLMs can do long-horizon planning from an abstract task description.
- Decoupling the LLM planner from the environment makes generalization to new tasks and environments feasible.
- ProgPrompt enables LLMs to intelligently combine the robot capabilities with the environment and their own reasoning ability to generate an executable and valid task plan.
- The precondition checking helps recover from some failure modes that can happen if actions are generated in the wrong order or are missed by the base plan.

Cons:

- Requires action space discretization, formalization of environments and objects.
- Plan generation is open-loop, with commonsense precondition checking-based environment interaction.
- Plan generation doesn't consider low-level continuous aspects of the environment state, and only reasons with the semantic state for planning as well as precondition checking.
- The amount of information exchange between language models and other modules such as the robot's perceptual or proprioceptive state encoders is limited, since API-based access to these recent LLMs only allows textual queries. However, this is still promising as it indicates the need for a multimodal encoder that can work with input such as vision, touch, force, temperature, etc.

Question 2 *How does it compare with the concurrent work: Code-as-Policies (CaP) (Liang et al., 2023)?*

- We believe that the general approach is quite similar to ours. CaP defines Hints and Examples which may correspond to Imports/Object lists and Task Plan examples in ProgPrompt .
- CaP uses actions as API calls with certain parameters for the calls such as robot arm pose, velocity, etc. We use actions as API calls with objects as parameters.
- CaP uses APIs to obtain environment information as well, like object pose or segmentation, for the purpose of plan generation. However, ProgPrompt extracts environment information via precondition checking on current environment state, to ensure plan executability. ProgPrompt also generates the prompt conditioned on information from perception models.

Question 3 *During "PROMPT for State Feedback", it seems that the prompt already includes all information about the environment state. Is it necessary to prompt the LLM again for the assertion (compared to a simple rule-based algorithm)?*

- The environment state input to the model is not the full state for brevity. Thus, checking pre-conditions with the full state separately helps, as shown in Table 1.
- The environment state could change during execution.

- Using LLM as opposed to a rule-based algorithm is a design choice made to keep the approach more general, instead of using a hand-coded rule-based algorithm. The assertion checking may also be replaced with a visual state conditioned module, when a semantic state is not available, such as in the real-world scenario. However, we leave these aspects to be addressed in future research.

Question 4 *Is it possible that the generated code might lead the robot to be stuck in an infinite loop?*

LLM code generation could lead to loops by predicting the same actions repeatedly as a generation artifact. LLMs used to suffer from such degeneration, but with latest LLMs (i.e. GPT-3) we have not encountered it at all.

Question 5 *Why are real-robot experiments simpler than virtual experiments?*

The real-robot experiments were done as a demonstration of the approach on a real-robot, while studying the method in depth in a virtual simulator, for the sake of simplicity and efficiency.

Question 6 *What's the difference between various GPT3 model versions used in this project?*

We name GPT3, which is the latest available version of GPT3 model on OpenAI at the time the paper was written: TEXT- DAVINCI- 002. We name DAVINCI as the original version of GPT3 released: TEXT- DAVINCI.[2]

Question 7 *Why not a planning language like PDDL (or other planning languages) be used to construct* PROGPROMPT*? Any advantages of using a pythonic structure?*

- GPT-3 has been trained on data from the internet. There is a lot of python code on the internet, while PDDL is a language of much more narrow interest. Thus, we expect the LLM to better understand python syntax.
- Python is a general purpose language, so it has more features than PDDL. Furthermore, we want to avoid specifying the full planning domain, instead relying on the knowledge learned by the LLM to make common-sense inferences. A recent work Xie et al. (2023b) uses LLMs to generate PDDL goals, however, it requires full domain specification for a given environment.
- Python is an accessible language that a larger community is familiar with.

Question 8 *How to handle multiple instances of the same object type in the scene?*

PROGPROMPT doesn't tackle the issue, however, Xie et al. (2023b) shows that multiple instances of the same objects can be handled by using labels with object IDs such as "book_1, book_2".

Question 9 *Why doesn't the paper compare the performance of the proposed method to InnerMonologue, SAYCAN, or Socratic models?*

At the time of writing, the dataset or model from the above papers were not public. However, we do compare with a proxy approach, similar in underlying idea to the above approaches, in the VirtualHome environment. LANGPLAN in our baselines, uses GPT3 to get textual plan steps, which are then executed using a GPT-2 based trained policy.

Question 10 *So the next step in this direction of research is to create highly structured inputs and outputs that could be compiled, since eventually we want something that compiles on robotic machines?*

The disconnect and information bottleneck between LLM planning module and skill execution module might make it less concrete on "how much" and "what" information should be passed through the LLM during planning. That said, we think that this would be an interesting direction to pursue and test the limits of LLM's highly structured input understanding and generation.

Question 11 *How does it compare to a classical planner?*

- Classical planners require concrete goal condition specification. An LLM planner reasons out a feasible goal state from a high level task description, such as "microwave salmon". From a user's perspective, it is desirable to not have to specify a concrete semantic goal state of the environment and just be able to give an instruction to act on.
- The search space would also be huge without common sense priors that an LLM planner leverages as opposed to a classical planner. Moreover, we also bypass the need to specify the domain knowledge needed for the search to roll out.
- Moreover, the domain specification and search space will grow non-linearly with the complexity of the environment.

Question 12 *Is it possible to decouple high-level language planning from low-level perceptual planning?*

It may be feasible to an extent, however we believe that a clean decoupling might not be "all we need". For instance, imagine an agent being stuck at an action that needs to be resolved at semantic level of reasoning, and probably very hard for the visual module to figure out. For instance, while placing a dish on an oven tray, the robot may need to pull the dish rack out of the oven to be successful in the task.

[2] More info on GPT3 models variations and naming can be found here: https://platform.openai.com/docs/models/overview

Question 13 *What are the kinds of failures that can happen with* ProgPrompt*-like 2 stage decoupled pipeline?*

A few broad failure categories could be:

- Generation of a semantically wrong action.
- Robot might fail to execute the action at perception/action /skill level.
- Robot needs to recover from a failure by taking a different high-level action, i.e., a precondition needs to be satisfied. The challenge is to identify that precondition from the current state of the environment and the agent.

Question 14 *What are the assumptions made about the actions used for* ProgPrompt*?*

We assume a set of available action APIs that are implemented on the robot, without assuming the implementation method (e.g. motion planning or reinforcement learning). ProgPrompt abstracts over and complements other research on developing flexible robot skills. This assumption is similar to those made in classical TAMP planners, where the planning space is restricted by the available robot skills.

Question 15 *Can the* ProgPrompt *planner handle more expressive situations when "the embodied agent has to grasp an object in a specific way in order to complete an aspect of the task"?*

This is possible, provided the deployed robot is capable of handling the requested action. For example, one can specify 'how' along with 'what' parameters for an action as function arguments, which may be discrete semantic grounded labels affecting the low-level skill execution, e.g. to select between different modes of grasping intended for different task purposes. However, it is an open question as to what the right level of abstraction is between high-level task specification and continuous control space actions, and the answer might depend on the application domain.

Author Contributions IS lead the research, conducted experiments, and drafted the manuscript; VB provided feedback, conducted experiments, drafted and reviewed the manuscript; JT and AG provided feedback, drafted and reviewed the manuscript; AM, AG, DX, JT, and DF provided feedback and reviewed the manuscript.

Funding Open access funding provided by SCELC, Statewide California Electronic Library Consortium. This project was conducted at and funded by NVIDIA.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., & Yan, M. (2022). Do as i can, not as i say: Grounding language in robotic affordances. arXiv.

Akakzia, A., Colas, C., Oudeyer, P. Y., Chetouani, M., & Sigaud, O. (2021). Grounding language to autonomously-acquired skills via goal generation. In *International conference on learning representations*.

Baier, J. A., Bacchus, F., & McIlraith, S. A. (2007). A heuristic search approach to planning with temporally extended preferences. In *Proceedings of the 20th international joint conference on artifical intelligence* (pp. 1808–1815). Morgan Kaufmann Publishers Inc.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., & Amodei, D. (2020). Language models are few-shot learners. arXiv.

Bryce, D., & Kambhampati, S. (2007). A tutorial on planning graph based reachability heuristics. *AI Magazine, 28*(1), 47.

Cao, Y., & Lee, C. (2023). Robot behavior-tree-based task generation with large language models. arXiv preprint arXiv:2302.12927

Capitanelli, A., & Mastrogiovanni, F. (2023). A framework to generate neurosymbolic pddl-compliant planners. arXiv preprint arXiv:2303.00438

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv.

Danielczuk, M., Mousavian, A., Eppner, C.,& Fox, D. (2021). Object rearrangement using learned implicit collision functions. In *IEEE international conference on robotics and automation (ICRA)*.

Eysenbach, B., Salakhutdinov, R. R., & Levine, S. (2019). Search on the replay buffer: Bridging planning and reinforcement learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems* (vol. 32). Curran Associates, Inc.

Fikes, R. E., & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2nd international joint conference on artificial intelligence* (pp. 608–620). Morgan Kaufmann Publishers Inc.

Garrett, C. R., Lozano-Pérez, T., & Kaelbling, L. P. (2020). Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning. *Proceedings of the International Conference on Automated Planning and Scheduling, 30*(1), 440–448.

Gu, X., Lin, T. Y., Kuo, W., & Cui, Y. (2022). Open-vocabulary object detection via vision and language knowledge distillation. In *International conference on learning representations*.

Gupta, T., & Kembhavi, A. (2022). Visual programming: Compositional visual reasoning without training. arXiv preprint arXiv:2211.11559

Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research, 26*(1), 191–246.

Hoffmann, J. (2001). Ff: The fast-forward planning system. *AI Magazine, 22*(3), 57.

Holtzman, A., Buys, J., Du, L., Forbes, M., & Choi, Y. (2020). The curious case of neural text degeneration. In *International conference on learning representations*.

Huang, W., Abbeel, P., Pathak, D., & Mordatch, I. (2022). Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. arXiv preprint arXiv:2201.07207

Huang, W., Xia, F., Shah, D., Driess, D., Zeng, A., Lu, Y., others (2023). Grounded decoding: Guiding text generation with grounded models for robot control. arXiv preprint arXiv:2303.00855

Huang, W., Xia, F., Xiao, T., Chan, H., Liang, J., Florence, P., & Ichter, B. (2022). Inner monologue: Embodied reasoning through planning with language models. arxiv preprint arxiv:2207.05608.

Jansen, P. (2020). Visually-grounded planning without vision: Language models infer detailed plans from high-level instructions. In *Findings of the association for computational linguistics: Emnlp 2020* (pp. 4412–4417). Online: Association for Computational Linguistics.

Jiang, Y., Gu, S. S., Murphy, K. P., & Finn, C. (2019). Language as an abstraction for hierarchical deep reinforcement learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems.* (vol. 32). Curran Associates, Inc.

Jiang, Y., Zhang, S., Khandelwal, P., & Stone, P. (2018). Task planning in robotics: An empirical comparison of pddl-based and asp-based systems. arXiv.

Kurutach, T., Tamar, A., Yang, G., Russell, S. J., & Abbeel, P. (2018). Learning plannable representations with causal infogan. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in neural information processing systems* (vol. 31). Curran Associates, Inc.

Li, S., Puig, X., Paxton, C., Du, Y., Wang, C., Fan, L., & Zhu, Y. (2022). Pre-trained language models for interactive decision-making. arXiv.

Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., & Zeng, A. (2023). Code as policies: Language model programs for embodied control.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2021). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. arXiv.

Luong, T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 conference on empirical methods in natural language processing* (pp. 1412–1421). Association for Computational Linguistics.

Mai, J., Chen, J., Li, B., Qian, G., Elhoseiny, M., & Ghanem, B. (2023). Llm as a robotic brain: Unifying egocentric memory and control. arXiv preprint arXiv:2304.09349

Mirchandani, S., Karamcheti, S., & Sadigh, D. (2021). Ella: Exploration through learned language abstraction. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, J. W. Vaughan (Eds.), *Advances in neural information processing systems* (vol. 34, pp. 29529–29540). Curran Associates, Inc.

Nair, S., & Finn, C. (2020). Hierarchical foresight: Self-supervised learning of long-horizon tasks via visual subgoal generation. In *International conference on learning representations*.

OpenAI (2023). Gpt-4 technical report. arXiv.

Patel, R., & Pavlick, E. (2022). Mapping language models to grounded conceptual spaces. In *International conference on learning representations*.

Puig, X., Ra, K., Boben, M., Li, J., Wang, T., Fidler, S., & Torralba, A. (2018). Virtualhome: Simulating household activities via programs. In *2018 IEEE/cvf conference on computer vision and pattern recognition* (pp. 8494–8502).

Shah, D., Toshev, A. T., Levine, S., & brian ichter. (2022). Value function spaces: Skill-centric state abstractions for long-horizon reasoning. In *International conference on learning representations*.

Sharma, P., Torralba, A., & Andreas, J. (2022). Skill induction and planning with latent language. In *Proceedings of the 60th annual meeting of the association for computational linguistics (volume 1: Long papers)* (pp. 1713–1726). Association for Computational Linguistics.

Shridhar, M., Thomason, J., Gordon, D., Bisk, Y., Han, W., Mottaghi, R., & Fox, D. (2020). ALFRED: A Benchmark for Interpreting Grounded Instructions for Everyday Tasks. In *The IEEE conference on computer vision and pattern recognition (cvpr)*.

Silver, T., Chitnis, R., Kumar, N., McClinton, W., Lozano-Perez, T., Kaelbling, L. P., & Tenenbaum, J. (2022). Inventing relational state and action abstractions for effective and efficient bilevel planning. In *The multi-disciplinary conference on reinforcement learning and decision making (rldm)*.

Skreta, M., Yoshikawa, N., Arellano-Rubach, S., Ji, Z., Kristensen, L. B., Darvish, K., & Garg, A. (2023). Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting. arXiv preprint arXiv:2303.14100

Srinivas, A., Jabri, A., Abbeel, P., Levine, S., & Finn, C. (2018). Universal planning networks: Learning generalizable representations for visuomotor control. In J. Dy, & A. Krause (Eds.), *Proceedings of the 35th international conference on machine learning* (vol. 80, pp. 4732–4741). PMLR.

Sundermeyer, M., Mousavian, A., Triebel, R., & Fox, D. (2021). Contact-graspnet: Efficient 6-dof grasp generation in cluttered scenes. In *2021 IEEE international conference on robotics and automation (icra)* (pp. 13438–13444).

Vemprala, S., Bonatti, R., Bucker, A., & Kapoor, A. (2023). Chatgpt for robotics: Design principles and model abilities. 2023

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., & Zhou, D. (2022). Chain of thought prompting elicits reasoning in large language models. arXiv.

Wiseman, S., Shieber, S., & Rush, A. (2017). Challenges in data-to-document generation. In *Proceedings of the 2017 conference on empirical methods in natural language processing* (pp. 2253–2263). Association for Computational Linguistics.

Xie, Y., Yu, C., Zhu, T., Bai, J., Gong, Z., & Soh, H. (2023a). Translating natural language to planning goals with large-language models. arXiv preprint arXiv:2302.05128

Xie, Y., Yu, C., Zhu, T., Bai, J., Gong, Z., & Soh, H. (2023b). Translating natural language to planning goals with large-language models.

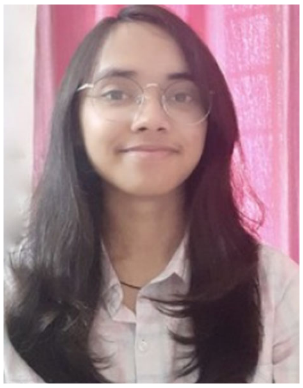
Xu, D., Martín-Martín, R., Huang, D. A., Zhu, Y., Savarese, S., & Fei-Fei, L. F. (2019). Regression planning networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems* (vol. 32). Curran Associates, Inc.

Xu, D., Nair, S., Zhu, Y., Gao, J., Garg, A., Fei-Fei, L., & Savarese, S. (2018). Neural task programming: Learning to generalize across hierarchical tasks. In *2018 IEEE international conference on robotics and automation (icra)* (pp. 3795–3802).

Zeng, A., Attarian, M., Ichter, B., Choromanski, K., Wong, A., Welker, S., & Florence, P. (2022). Socratic models: Composing zero-shot multimodal reasoning with language. arXiv

Zhu, Y., Tremblay, J., Birchfield, S., & Zhu, Y. (2020). Hierarchical planning for long-horizon manipulation with geometric and symbolic scene graphs. arXiv.

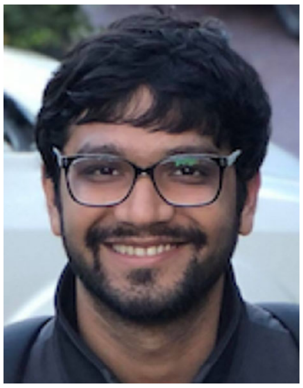
Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



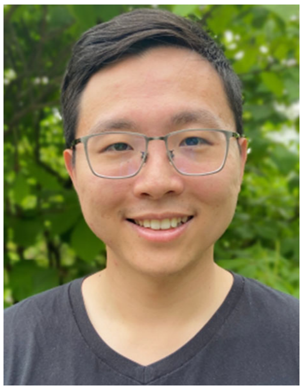
Ishika Singh is a 3rd year PhD student advised by Professor Jesse Thomason in the Computer Science department at the University of Southern California. Her research focuses on problems in language-conditioned robot learning such as vision-language navigation, manipulation and task planning. Previously, she was an undergrad at IIT Kanpur.

Valts Blukis is a research scientist at NVIDIA. His research goal is creating scalable and generalizable machine learning algorithms and models that enable robots to interact with people through natural language while observing the unstructured world through first-person sensor observations. He received his PhD from Cornell University and Cornell Tech.

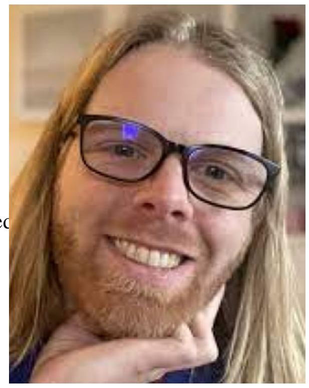
Arsalan Mousavian s a senior research scientist at NVIDIA Seattle Robotics Lab. He is interested in using computer vision and 3D vision for robotics tasks such as object manipulation. Prior to NVIDIA, he finished his PhD in the Computer Science department at George Mason University.



Ankit Goyal is a Research Scientist in Robotics at NVIDIA. He did his Ph.D. in Computer Science at Princeton University. I completed Masters from University of Michigan and Bachelors from IIT Kanpur. He is interested in understanding various aspects of intelligence, especially reasoning and common sense. In particular, he wants to develop computation models for various reasoning skills that humans possess.



Danfei Xu is an Assistant Professor at the School of Interactive Computing at Georgia Tech and a (part-time) Research Scientist at NVIDIA AI. His current research focuses on visuomotor skill learning, long-horizon manipulation planning, and data-driven approaches to human-robot collaboration. He received his Ph.D. in CS from Stanford University.



Jonathan Tremblay is a research scientist at NVIDIA. His research interests are in computer vision, synthetic data, and reinforcement learning for robotics applications. At NVIDIA, Jonathan has focused on using synthetic data to train object detectors, object pose estimation, few shot learning, etc. Jonathan's goal is to create robust and accessible computer vision systems for roboticists to use on their system. Prior to joining NVIDIA, Jonathan received Ph.D. in computer science from McGill University.

Dieter Fox is Senior Director of Robotics Research at Nvidia. His research is in robotics, with strong connections to artificial intelligence, computer vision, and machine learning. He is currently on partial leave from the University of Washington, where he is a Professor in the Paul G. Allen School of Computer Science & Engineering. At UW, he also heads the UW Robotics and State Estimation Lab. From 2009 to 2011, he was Director of the Intel Research Labs Seattle. Dieter obtained his Ph.D. from the University of Bonn, Germany.

Jesse Thomason is an Assistant Professor at USC leading the Grounding Language in Multimodal Observations, Actions, and Robots (GLAMOR) lab. GLAMOR brings together natural language processing and robotics (RoboNLP). Jesse joined USC in 2021 and received his PhD from the University of Texas at Austin in 2018.

Animesh Garg is an Stephen Fleming Early Career Professor in Computer Science at Georgia Tech. Previously, he was an Assistant Professor of Computer Science at University of Toronto and a Faculty Member at the Vector Institute. He is also a Sr. Research Scientist at Nvidia. He earned his Ph.D. in Operations Research from UC Berkeley and postdoc at Stanford. His group focuses on multimodal object-centric and spatiotemporal event representations, self-supervised pre-training for reinforcement learning & control, principle of efficient dexterous skill learning.