RLang: A Declarative Language for Describing Partial World Knowledge to Reinforcement Learning Agents

Rafael Rodriguez-Sanchez [*1] **Benjamin A. Spiegel** [*1] **Jennifer Wang** [1]
Roma Patel [1] **Stefanie Tellex** [1] **George Konidaris** [1]

Abstract

We introduce RLang, a domain-specific language (DSL) for communicating domain knowledge to an RL agent. Unlike existing RL DSLs that ground to *single* elements of a decision-making formalism (e.g., the reward function or policy), RLang can specify information about every element of a Markov decision process. We define precise syntax and grounding semantics for RLang, and provide a parser that grounds RLang programs to an algorithm-agnostic *partial* world model and policy that can be exploited by an RL agent. We provide a series of example RLang programs demonstrating how different RL methods can exploit the resulting knowledge, encompassing model-free and model-based tabular algorithms, policy gradient and value-based methods, hierarchical approaches, and deep methods.

1. Introduction

Reinforcement learning tasks are often impractically hard to solve *tabula rasa*. Fortunately, even a small amount of prior knowledge about the world—knowledge that is often either seemingly obvious or easy for a human to produce—can dramatically improve learning. For instance, knowing about the action dynamics of a game (e.g., you can jump to avoid falling into pits) or properties of its state (e.g., that a particular state variable indicates a block of lava) can prevent an agent from making poor decisions. In long-horizon tasks, especially those with sparse rewards, such knowledge may even be a prerequisite for finding an acceptable policy.

Languages, both formal and natural, have been used in various ways to add prior knowledge into decision-making (Luketina et al., 2019). Formal languages benefit from unambiguous syntax and semantics, and can therefore be reliably used to represent knowledge. These have proven useful in specifying advice to agents in the form of hints about actions (Maclin & Shavlik, 1996) or policy structure (Andreas et al., 2017; Sun et al., 2020). Communicating using natural language would be more intuitive, but that requires converting natural language sentences into grounded knowledge usable by the agent; most approaches in this area can ground limited subsets of natural language (e.g., only commands). For example, for describing task objectives (Artzi & Zettlemoyer, 2013; Patel et al., 2020), or other individual task components such as rewards (Goyal et al., 2019; Sumers et al., 2021) and policies (Branavan et al., 2010). All of the above approaches provide information about a single component of a chosen decision-making formalism; there exists no unified framework able to express information about *all* the components of a task.

We therefore introduce RLang, a domain-specific language (DSL) that can specify information about every component of a Markov decision process (MDP), including flat and hierarchical policies, state factors, state features, transition functions, and reward functions. RLang is human-readable and compiles into simple data structures that can be accessed by any learning algorithm. We explain how to write statements that inform each MDP component, and demonstrate RLang's versatility through a series of example programs expressing different types of domain knowledge. We also show how to integrate this knowledge into several RL methods to improve learning performance.[1]

2. Background

Domain-Specific Languages Domain-specific languages (DSLs) are formal languages designed to specify information relevant to a target domain. Compared to general-purpose programming languages like Python (Van Rossum & Drake Jr, 1995) and C (Kernigham & Ritchie, 1973), DSLs typically contain a smaller set of narrower semantics

[*]Equal contribution [1]Department of Computer Science, Brown University, Providence, RI, USA. Correspondence to: Benjamin Spiegel <bspiegel@cs.brown.edu>.

Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

[1]RLang source code, documentation, and examples are available at `rlang.ai`.

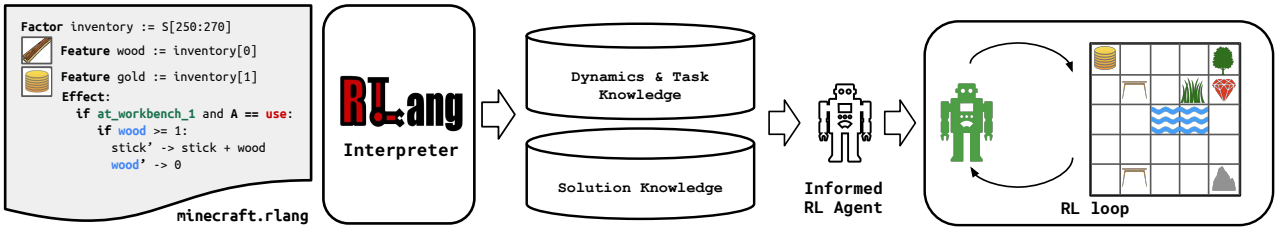


Figure 1: RLang provides users with a precise language to provide domain knowledge to RL agents. RLang programs are parsed by the interpreter to create a partial model of the MDP (`Dynamics and Task Knowledge`) and its solution (`Solution Knowledge`). Finally, the knowledge objects inform an RL agent that can leverage the grounded information during its learning loop.

better suited to a specific application. That is, DSLs sacrifice computational expressivity for ease-of-use within a particular domain. Commonly-used DSLs include the Standard Query Language (SQL) used for querying relational databases and the Planning Domain Definition Language (PDDL; Ghallab et al. 1998) for defining planning tasks.

Decision-Making Formalisms Reinforcement learning tasks are typically modeled as Markov Decision Processes (MDPs; Puterman, 1990), defined by a tuple $(\mathcal{S}, \mathcal{A}, R, T, \gamma)$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a transition probability distribution, $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a reward function, and $\gamma \in (0, 1]$ is a discount factor. A solution to an MDP is a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ maximizing expected discounted return $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, where r_t is the reward obtained at time step t. The value function $V^\pi : \mathcal{S} \to \mathbb{R}$ for a policy π captures the expected return an agent would receive from executing π starting in a state s. The action-value function $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ of a policy is the expected return from executing an action a in a state s and following policy π thereafter.

Hierarchical Decision-Making Solving MDPs with high-dimensional state and action spaces can be difficult, especially in domains where long sequences of actions are required to achieve a goal. In these environments, hierarchical reinforcement learning (Barto & Mahadevan, 2003) may be more applicable, as temporally-extended actions can reduce the complexity of the space of solution policies. The *options framework* (Sutton et al., 1999) formalizes this notion by modeling high-level actions as *options*: closed-loop policies defined by a tuple (I, π, β), where $I \subseteq \mathcal{S}$ is a set of states in which the option can be executed, π is an option policy, and $\beta : \mathcal{S} \to [0, 1]$ describes the probability that the option will terminate upon reaching a given state. If O is the set of options the agent can execute, then the MDP tuple is extended to $(\mathcal{S}, \mathcal{A} \cup O, R, T, \gamma)$ in the hierarchical setting.

3. RLang: Describing Partial World Knowledge about Tasks

If RL is to become widely used in practice, we must reduce the infeasible amount of trial-and-error required to learn to solve a task from scratch. One promising approach is to avoid *tabula rasa* learning by including the sort of background knowledge that humans typically bring to a new task. Such background knowledge is often easy to obtain—in many cases, it is simply obvious to anyone: *try not to fall off cliffs!*—and need not be perfect or complete to be useful.

Unfortunately, however, there is no standardized approach to communicating such background knowledge to an RL agent. In most cases, the same person who implements the learning algorithm also hand-codes the background knowledge, typically in an ad-hoc fashion in the same general-purpose programming language in which the algorithm is implemented. This has two primary drawbacks. First, prior knowledge is often task-specific, and the lack of a means to express it hinders the development of learning algorithms that can exploit varying types and degrees of background knowledge. Second, it is not accessible to end-users or other consumers of RL agents, who do not write the algorithms themselves and cannot be expected to master the relevant programming languages and mathematical details, but who might nevertheless wish to accelerate learning.

The alternative is to design a standardized, human-interpretable DSL for expressing prior knowledge about reinforcement learning tasks. Such a DSL should have two important properties not present in existing RL DSLs (Maclin & Shavlik, 1996; Denil et al., 2017; Sun et al., 2020). First, it should be agnostic of the learning algorithm used. Separating the question of how to express prior knowledge from how that knowledge is exploited by a learning algorithm introduces a standardized interface that can be used to inform a wide variety of RL agents, even ones based on algorithms not yet developed. Second, it should be *complete*: able to express all the information that could possibly be informative about a particular task. We therefore propose RLang, a new DSL designed to fulfill these criteria.

RLang can name state features (using `Features` and `Propositions`), specify goal states (using `Goals`), define abstract actions (using `Options`), describe policies and hierarchical policy structure (using `Policies`), restrict the action space (using `ActionRestrictions`), provide partial world models (using `Effects`), and shape reward (also using Effects). Our Python package parses RLang into an algorithm-agnostic data structure (see Section 3.2) that can be integrated into nearly any RL algorithm.

3.1. RLang Elements

An RLang program consists of a set of declarations, where each one grounds to one or more components of an $(\mathcal{S}, \mathcal{A}, O, R, T, \pi)$ tuple. More specifically, every RLang Element grounds to a function with a domain in $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ and a co-domain in $\mathcal{S}$, $\mathcal{A}$, $\mathbb{R}^n$ where $n \in \mathbb{N}$, or $\{\top, \bot\}$. We describe the main RLang element types in the rest of this section and summarize them in Table 1. A full formal semantics and grammar are in Appendix A.

State Factors In Factored MDPs (Boutilier et al., 2000), the state space is a collection of conditionally independent variables: $\mathcal{S} = \mathcal{X}_1 \times .. \times \mathcal{X}_n$. Some algorithms might find it useful to reference these variables individually. For example, consider a 2-D version of Minecraft, as represented in Figure 2, where an agent has to collect ingredients to craft new tools and objects. In this environment the state is the concatenation of a position vector, a flattened map representation, and an inventory vector: $s = (\text{pos}, \text{map}, \text{inventory})$. `Factors` can be used to reference these state variables:

```
Factor position := S[0:2]
Factor map := S[2:250]
Factor inventory := S[250:270]
```

S is a reserved keyword referring to the current state. **A** and **S'** are also keywords referring to the current action and the next state, respectively. Factors can be sliced and indexed:

```
Factor iron := inventory[0]
Factor wood := inventory[1]
```

State Features RLang can also be used to define more complex functions of state. For instance, if the agent's goal is to build axes, we can define a `Feature` capturing the number of axes that could be built at the current state:

```
Feature number_of_axes := wood + iron
```

Propositions `Propositions` in RLang, which are functions of the form $\mathcal{S} \rightarrow \{\top, \bot\}$, identify states that share relevant characteristics:

```
Constant workbench_locations := [[1,
    0], [1, 3]]
Proposition at_workbench := position
    in workbench_locations
Proposition have_bridge_material :=
    iron >= 1 and wood >= 1
```

Goals `Goals` specify goal states via a proposition. For example, **Goal** `get_gold := gold >= 1` encodes that the agent must collect at least one gold unit.

Objects and Classes RLang supports the usage of Object-Oriented MDPs (Diuk et al., 2008) via `Objects` and `Classes`, with support for sub-classes and various object attribute types (including integers, floats, and booleans). Objects that are in the state space can be accessed, and new objects and classes can be easily instantiated:

```
Class Arm:
  length: int
Object robot_arm := Arm(S.forearm.
    length + S.end_affector.length)
```

Markov Features Markov functions like the action-value or transition function take the form $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$. We extend the co-domain of this function class to $\mathbb{R}^n$ and introduce `Markov Features`, which allow users to compute features on an (s, a, s') experience tuple. The following Markov Feature represents a change in inventory elements.

```
Markov Feature inventory_change :=
    inventory' - inventory
```

The prime (`'`) operator references the value of an RLang name when evaluated on the next state.

Policies Policy functions can also be specified in RLang using conditional expressions:

```
Policy main:
  if iron >= 2:
    if at_workbench:
      Execute Use # This is an action
    else:
      Execute go_to_workbench # This
    is a policy
  else:
    Execute collect_iron
```

The `Execute` keyword executes an action or calls another policy. The above policy instructs the agent to craft iron tools at a workbench by first collecting iron and then navigating to the workbench. Policies can also be probabilistic:

```
Policy random_move:
  Execute up with P(0.25)
  or Execute down with P(0.25)
  or Execute left with P(0.25)
  or Execute right with P(0.25)
```

Users can specify multiple `policy` functions in an RLang program and designate a primary policy by naming it `main`.

Options Abstract actions are specified using `Options`, which include initiation and termination propositions:

```
Option build_bridge:
  init have_bridge_material and
    at_workbench
    Execute craft_bridge
  until bridge in inventory
```

Table 1: `RLang` declarations for corresponding MDP elements. The first column shows a component of the MDP, the second shows an `RLang` expression that can inform it, while the last column contains a description of the expression.

MDP Component	RLang Declaration	Natural Language Interpretation
State Factor $\phi : \mathcal{S} \to \mathbb{R}^n$	`Factor inventory := S[250:260]`	Your inventory is a small factor of the state space.
State Feature $\phi : \mathcal{S} \to \mathbb{R}^n$	`Feature inventory_value :=` `5 * gold + 2 * iron`	The value of your inventory is 5 for each gold you have plus 2 for each iron.
Proposition $\sigma : \mathcal{S} \to \{\top, \bot\}$	`Proposition at_workbench :=` `position in workbench_locations`	You are at a workbench if your position is one of the workbench locations.

Objects and Class Definitions
$(\mathcal{C} = \{C_1, ..., C_c\},$
$\mathcal{O} = \{o_1, ..., o_n\})$
	`Class Block:`
	`  id:  int`
	`Object dirt := Block(1)`
		Create a Block class and a new dirt Block object.

Markov Feature $f : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}^n$	`MarkovFeature inv_change :=` `inventory' - inventory`	The change in your inventory in a time step

Policy
$\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$
	`Policy build_bridge:`
	`  if at_workbench:`
	`  Execute use`
		If you are at a workbench, craft using it.

Option
$(\sigma_I, \pi, \sigma_\beta)$
	`Option build_axe:`
	`  init (wood >= 1 and iron >= 1)`
	`  Execute build_axe_policy`
	`  until (axe >= 1)`
		When you have at least one wood and one iron, you can build axes until you have at least one.

Reward and Transition Function
$(R_e : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R},$
$T_e : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1])$
	`Effect resource_consumption:`
	`  if wood >= 1 and A == use:`
	`  stick' -> stick + wood`
	`  wood' -> 0`
	`  Reward wood`
		Crafting will convert your wood into sticks. You will also be rewarded 1 for every wood you have.

Action Restrictions Restrictions to the set of possible actions an agent can take in a given circumstance can be specified using `ActionRestrictions`:

```
ActionRestriction dont_get_burned:
  if (position + [0, 1]) in
    lava_locations:
    Restrict up
```

Effects `Effects` provide an interface for specifying partial information about the transition and reward functions. In a factored MDP, RLang can specify factored transition functions (i.e., transition functions for individual factors):

```
Effect movement_effect:
  if x_position >= 1 and A == left:
    x_position' -> x_position - 1
  Reward -0.1
```

The above Effect captures the predicted consequence of moving left on the `x_position` factor, stating that the x position of the agent in the next state will be 1 less than in the current state. This Effect also specifies a -0.1 step penalty regardless of the current state or action. In simpler MDPs, predictions can be made about the whole state vector:

```
Effect tic_tac_toe:
  if three_in_a_row:
    S' -> empty_board # Board is reset
```

Effects can reference previously defined effects similarly:

```
Effect main:
  -> movement_effect
  -> crafting_effect
```

A `main` Effect designates the primary environment dynamics, and grounds to a partial factored world model $(\overline{\mathcal{T}}, \overline{\mathcal{R}})$. As with policies, Effects can be probabilistic using `with`.

Finally, it is important to note that RLang, as we have seen across these examples, does not require the specification of `Effects` and `Policies` to be complete. Therefore, a user is not required to provide extensive and complex programs to fully specify the MDP—although this is a possibility with RLang—to accelerate learning. RL agents must learn to solve the task by filling in the missing pieces.

3.2. Accessing Parsed RLang Knowledge

Using our Python package, users can parse RLang programs into the following queryable `knowledge` objects, which can be integrated directly into a learning algorithm:

- The **Dynamics and Task Knowledge** object contains a queryable model of the environment and the task (i.e., transition dynamics $\overline{T}$ and reward function $\overline{R}$) that are derived from the **`Effect`** `main` declaration and the collection of defined goals;
- The **Solution Knowledge** object that contains information about the collection of newly defined options O and the `main` policy $\overline{\pi}$.

These knowledge objects are implemented as partial functions; an `unknown` flag is returned when querying for a element of the domain where the RLang has no knowledge.

3.3. Selecting a Suitable RLang-informed Agent

Once an RLang program is compiled into partial MDP components, the parser must select an informed agent that leverages the given information. We propose the following simple mechanism to select among a suite of RLang-informed agents: Let A be a set of RL agents capable of taking RLang advice. Let c_i^α be a binary variable that determines whether the agent $\alpha \in \mathrm{A}$ is capable of exploiting a type of advice (e.g., transition dynamics, rewards, etc.). Therefore, for each agent, we have a characterizing vector $C^\alpha = [c_1^\alpha, c_2^\alpha, ..., c_n^\alpha]$. Similarly, the RLang parser generates an analogous binary vector P that represents the RLang information types expressed in the RLang program. Finally, we can select among the agents that exploit as much information as possible by selecting an agent such that $P \approx C^\alpha$.

3.4. Specifying Complex Groundings with a Vocabulary

RLang comes built-in with a set of simple arithmetic, Boolean, and set operations that can be used in RLang object declarations. However, users may wish to include more complex grounding functions in their RLang programs. For instance, when dealing with problems with high-dimensional observation spaces (e.g., pixel frames), the user may wish to use an object classifier as a means to output a propositional value. Users can therefore import and reference RLang objects defined using our Python library. By specifying a **vocabulary file** (in JSON) and a corresponding **grounding file** (a Python file containing RLang objects), users can construct their own RLang objects and reference them directly in RLang programs. This allows users to provide complex expert groundings or, more generally, learned groundings that hold the necessary semantic information to derive *new* grounded knowledge easily with RLang programs.

4. Demonstrations

We now provide a few RLang use-cases, focusing on examples that show how different types of prior information can be concisely expressed, and effectively exploited, for varying degrees of environment complexity and different families of RL methods.

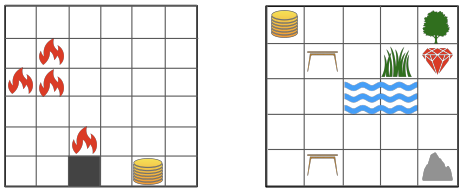


Figure 2: On the left, the Lava-Gap environment. On the right, 2D Minecraft based on Andreas et al. (2017).

Hierarchical Policy Structure: 2D Minecraft We first consider a 2D version of Minecraft based on Andreas et al. (2017), consisting of a gridworld (see Figure 2) that contains workbenches where the agent can craft new objects, and raw materials like wood, stone and gold. To build an item, the agent must have the required ingredients and be in the correct workbench. The agent has the action `use` to interact with elements, and actions to move in the cardinal directions.

To show how providing the sub-policy structure of the task improves performance, we provide the agent with initiation and termination conditions for a few options (to collect wood, go to the three different workshops and to build the required elements). The following program concisely defines 3 options fully and 4 options with uninformed policies.

```
Option go_to_workshop_0:
  init(any):
    Execute go_to_workshop_0_learnable_policy
  until(at_workshop_0)
Option go_to_workshop_1:
  init(any):
    Execute go_to_workshop_1_learnable_policy
  until(at_workshop_1)
Option go_to_workshop_2:
  init(any):
    Execute go_to_workshop_2_learnable_policy
  until(at_workshop_2)
Option get_wood:
  init(there_is_wood):
    Execute get_wood_learnable_policy
  until delta_wood >= 1
Option build_plank:
  init(wood >= 1 and at_workshop_1):
    Execute use
  until (delta_plank >= 1)
Option build_stick:
  init (wood >= 1 and at_workshop_1)
    Execute use
```

```
    until (delta_stick >= 1)
  Option build_ladder:
    init (stick >= 1 and plank >= 1)
      Execute use
    until (delta_ladder >= 1)
```

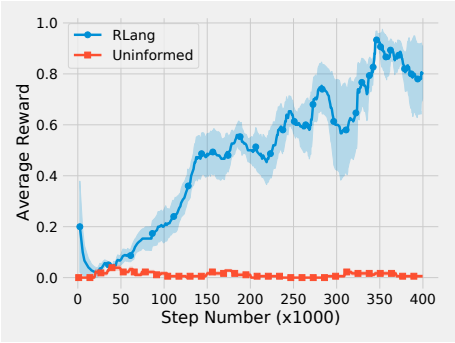
To exploit this information, the agent must learn both the policy over options to maximize reward, and the option policies that achieve each option's termination condition. For both the high-level and low-level agents, we use DDQN (Van Hasselt et al., 2016) (details are in Appendix B.4).

Figure 3a show the average return of RLang-informed hierarchical DDQN vs. the uninformed (flat) performance of a DDQN agent. A concise program partially describing a hierarchical solution was sufficient to successfully learn to solve the task, in stark contrast with the uninformed agent.

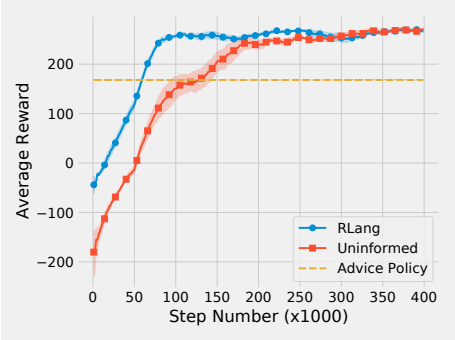
Policy Prior: Lunar Lander Next, we consider programs that provide prior policy knowledge. Such policy information need not be optimal or complete, but it can still improve learning performance. We first consider Lunar Lander (Brockman et al., 2016), which requires learning an optimal control policy to gently land a ship on the moon. The environment has a dense reward encoding both the goal and cost constraints, a continuous state space, and four discrete actions that either do nothing, fire the main engine, or fire the left or right thruster. We provide the agent with an initial policy using the following RLang program:

```
  Policy land:
    if (left_leg_in_contact == 1.0) or
     (right_leg_in_contact == 1.0):
      if (velocity_y/2 * -1.0) > 0.05:
        Execute main_engine
      else:
        Execute do_nothing
    elif remaining_hover >
    remaining_angle and
    remaining_hover > -1 *
    remaining_angle and
    remaining_hover > 0.05:
      Execute main_engine
    elif remaining_angle < -0.05:
      Execute right_thruster
    elif remaining_angle > 0.05:
      Execute left_thruster
    else:
      Execute do_nothing
```

We implemented an RLang-informed agent using PPO (Schulman et al., 2017), a policy gradient method, as our base method. We probabilistically mixed the RLang-defined advice policy with a learnable policy network using mixing parameter $\beta \in [0, 1]$, following Fernández & Veloso (2006), which is annealed during learning process. In this way, the RLang and the learned policies shared control stochastically. Figure 3b shows the average return curves resulting from an uninformed PPO agent (Schulman et al., 2017) and the RLang-informed version. The informed agent exploits the given policy and learns to improve it further, resulting in a clear performance improvement.



(a) Craftworld + hierarchical information



(b) Lunar Lander + policy prior

Figure 3: RLang-informed algorithms given prior **policy structure**. (a) an RLang-informed DDQN agent vs. uninformed DDQN on 2-D Minecraft. (b) an RLang-informed PPO agent exploits policy advice vs. learning from scratch.

We also considered two classic control problems: CartPole and Mountain Car. For CartPole, we obtain analogous results using REINFORCE (Williams, 1992) as the base method (see Appendix B.5). In Mountain Car, a hard exploration problem in RL, a very concise RLang policy results in near-optimal performance; the simple program below gets a -119 average return over 100 episodes, where the task is considered solved with a -110 average return.

```
  Policy gain_momentum:
  if velocity < 0:
    Execute go_left
  else:
    Execute go_right
```

Dynamics and Rewards: Lava-Gap We now show how to provide transition and reward information using RLang in the Lava-Gap environment (Figure 2), a gridworld where an agent must navigate to a goal position. The agent can move in the cardinal directions but each action has a probability of failure of $1/3$. Moreover, moving into walls causes the agent to stay in the same position and falling into a lava pit

results in a high negative reward. The agent would typically need to fall into lava pits many times to learn to avoid it. With RLang, however, we can easily inform agents about the dynamics and the high cost of lava pits.

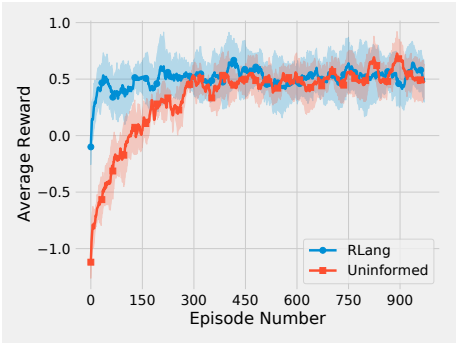


Figure 4: RLang-informed Q-Learning agents given partial **rewards** and **dynamics** information in Lava-Gap

The program below starts by defining an effect `moving_effect` that predicts the effect of an action in most cases—i.e., when walls are not in the way. Next, the effect `dynamics` extends this by describing walls. Finally, the reward function is provided through the effect `reward`.

```
Effect moving_effect:
  if A == up:
    x' -> x + 1
    y' -> y
  elif A == down:
    x' -> x - 1
    y' -> y
  elif A == left:
    x' -> x
    y' -> y - 1
  elif A == right:
    x' -> x
    y' -> y + 1
Effect dynamics:
  if at_wall:
    S' -> S
  else:
    -> moving_effect
Effect reward:
  if in_lava:
    Reward -1
  elif at_goal:
    Reward 1.
  else:
    Reward 0.
Effect main:
  -> dynamics
  -> reward
```

Classic tabular Q-Learning is suitable here. We designed a Q-Learning agent that exploits the transition dynamics and reward information. The agent first estimates an initial Q-table using Value Iteration based on the partial transition and reward models—when information is `unknown` for a transition tuple (s, a, s') the Q-value defaults to 0. See Appendix B.1 for more details.

Average return curves are shown for an RLang-informed Q-Learning agent (Watkins & Dayan, 1992) in Figure 4. The curves show that the informed agent clearly leverages the information to gain high return early in the training process. We obtain analogous results informing an RMax agent (Brafman & Tennenholtz, 2002), a model-based method. These results are in Appendix B.1.

Object-Oriented Dynamics and Rewards: Taxi

We now provide dynamics and reward information for an object-oriented environment using RLang. We use the object-oriented version of the Taxi environment in Diuk et al. (2008), a gridworld where the agent must pick up a passenger and drop them off at one of four destinations. DOORmax (Diuk et al., 2008) is suitable for Object-Oriented MDPs. We implement an RLang-enabled DOORmax algorithm which allows for initialization of transition dynamics and rewards. We show these results in Figure 5.

```
Effect no_movement_effect:
  if S.taxi.touch_n and A == move_n:
    S' -> S
  if S.taxi.touch_s and A == move_s:
    S' -> S
  if S.taxi.touch_e and A == move_e:
    S' -> S
  if S.taxi.touch_w and A == move_w:
    S' -> S

Effect main:
  if S.taxi.on_passenger and A ==
    pick_up:
    S'.passenger.in_taxi -> True
  if S.passenger.in_taxi and A ==
    drop_off:
    S'.passenger.in_taxi -> False
    if S.taxi.on_destination:
      Reward 20
    else:
      Reward -10
  elif A == pick_up or A == drop_off:
    Reward -10
  else:
    Reward -1
    -> no_movement_effect
```

OO-MDPs contain an abstract layer of object-centric information about an underlying MDP. RLang provides compact semantics for referencing these abstractions, which object-aware agents can exploit for improved performance. Some predicates given by the Taxi domain describe the taxi's position in the underlying MDP (e.g., `touch_n(taxi)`), which can be used to specify where the taxi is allowed to move. Using these predicates, the above RLang program succinctly explains that the taxi cannot move through walls, a crucial piece of information that a *tabula rasa* agent might learn for each wall it encounters over many timesteps.

Table 2: Comparison of DSLs proposed for RL agents and the types of MDP information that can be expressed

RL Language	Policy Hint	Action Structure	Policy Constraints	State Structure	Rewards	Transition Dynamics
ALisp (Andre & Russell, 2002)	✓	✓				
Advice RL (Maclin & Shavlik, 1996)	✓		✓			
Program-guided Agent (Sun et al., 2020)	✓	✓				
Programable Agents (Denil et al., 2017)				✓		
Policy sketches (Andreas et al., 2017)		✓				
GLTL (Littman et al., 2017)					✓	
SPECTRL (Jothimurugan et al., 2019)					✓	
RLang	✓	✓	✓	✓	✓	✓

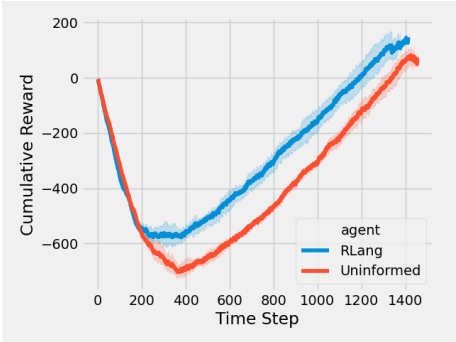


Figure 5: An RLang-enabled DOORmax agent in object-oriented Taxi informed with dynamics information

5. Related Work

There has been a recent surge of interest in language methods for RL agents (Luketina et al., 2019). These fall under methods that use natural language to instruct, or to reward, agents as a form of supervision, and methods that use formal languages to represent goals or other components of an MDP.

Formal Languages in Reinforcement Learning In classical planning it is standard to use the Planning Domain Description Language (PDDL; Ghallab et al. 1998) and its probabilistic extension PPDDL (probabilistic PDDL; Younes & Littman, 2004) to specify the complete dynamics of a factored-state environment. RLang is inspired by these but it is intended for a fundamentally different task: providing *partial* knowledge to a learning agent, where the knowledge might correspond to any component of the underlying MDP. Maclin & Shavlik (1996) propose an RL paradigm in which the agent may request advice via a DSL that uses propositional statements to provide policy hints. Similarly, Sun et al. (2020) propose to learn a policy conditioned on a program from a DSL. Andreas et al. (2017) use a simple grammar to represent policies as a concatenation of primitives (sub-policies) to provide RL agents with knowledge about the hierarchical structure of the tasks. Other languages include linear temporal logic (LTL; Littman et al., 2017; Jothimurugan et al., 2019) which has been used to describe goals for instruction-following agents. These methods ground LTL formulae to reward functions. RLang expands on all of these DSLs to include information beyond the policy and the reward function, thus enabling a wider array of information to be provided to the agent.

Table 2 summarizes existing DSLs for RL and shows their relative expressive power. Besides RLang, no other DSL is sufficiently powerful to express the wide range of information that could be of use to an RL agent.

Natural Language Grounding and Learning Methods Several works have attempted to learn mappings for the semantic meaning of natural language to grounded information for agents. Some approaches learn to ground the natural language input to a single grounding function type (e.g., reward function) directly from data. For instance, RL agents that learn to play Civilization II by grounding linguistic information from manuals as features relevant to estimating the Q-function (Branavan et al., 2012), and grounding textual specifications of goals and dynamics of the game to learn a language-conditioned policy (Zhong et al., 2019). In instruction-following, some approaches learn to map instructions to reward functions (Misra et al., 2017; Bahdanau et al., 2018; Goyal et al., 2020).

However, other approaches translate natural language sentences to an intermediate semantic representation language. In general, these languages are restricted grammars that can be easily mapped to the desired grounding element. For example, some methods translate instructions to sequences of primitive actions (Misra & Artzi, 2017) or to LTL formulae (Williams et al., 2018; Gopalan et al., 2018; Patel et al., 2020). In future work we plan to use RLang as the semantic representation language, since it is capable of expressing a much wider range of information than existing DSLs.

6. Conclusion

RLang is a concise and unambiguous domain-specific language designed to make it easy for a human to provide background knowledge—about any component of a task—to an RL agent. RLang's formal semantics also serve as a unified framework under which to study and compare RL algorithms capable of exploiting background knowledge to improve learning. The examples in this paper show that RLang can be used to provide diverse types of domain knowledge and structure via simple and intuitive RLang programs describing task knowledge that can effectively improve performance over *tabula rasa* methods.

Acknowledgements

We would like to thank our colleagues Sam Lobel, Akhil Bagaria, Saket Tiwari and members in the BigAI group for useful conversations, discussions and feedback during the earlier iterations of this project. We thank Michael Littman and Ellie Pavlick whose insights were useful to kick-start this project. This research was supported by DARPA under award number FA8750-18-2-0117, NSF under award numbers CNS-2038897 and IIS-1955361, and with support from Echo Labs to Tellex. It was supported in part by NSF grant #1955361, NSF CAREER award #1844960 to Konidaris, and NSF GRFP #2022339942 to Spiegel, and was conducted using computational resources and services at the Center for Computation and Visualization, Brown University.

References

Abel, D. simple_rl: Reproducible reinforcement learning in python, 2019.

Andre, D. and Russell, S. J. State abstraction for programmable reinforcement learning agents. In *Eighteenth National Conference on Artificial Intelligence*, pp. 119–125, USA, 2002. American Association for Artificial Intelligence. ISBN 0262511290.

Andreas, J., Klein, D., and Levine, S. Modular multitask reinforcement learning with policy sketches. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 166–175. JMLR. org, 2017.

Artzi, Y. and Zettlemoyer, L. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62, 2013.

Bahdanau, D., Hill, F., Leike, J., Hughes, E., Hosseini, A., Kohli, P., and Grefenstette, E. Learning to understand goal specifications by modelling reward. In *International Conference on Learning Representations*, 2018.

Barto, A. G. and Mahadevan, S. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1):41–77, 2003.

Barto, A. G., Sutton, R. S., and Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077.

Boutilier, C., Dearden, R., and Goldszmidt, M. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000.

Brafman, R. I. and Tennenholtz, M. R-max: A general polynomial-time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3 (Oct):213–231, 2002.

Branavan, S., Zettlemoyer, L., and Barzilay, R. Reading between the lines: Learning to map high-level instructions to commands. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pp. 1268–1277, 2010.

Branavan, S., Silver, D., and Barzilay, R. Learning to win by reading manuals in a monte-carlo framework. *Journal of Artificial Intelligence Research*, 43:661–704, 2012.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

Denil, M., Colmenarejo, S. G., Cabi, S., Saxton, D., and de Freitas, N. Programmable agents. *arXiv preprint arXiv:1706.06383*, 2017.

Dietterich, T. G. The maxq method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pp. 118–126, 1998.

Diuk, C., Cohen, A., and Littman, M. L. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pp. 240–247, 2008.

Fernández, F. and Veloso, M. Probabilistic policy reuse in a reinforcement learning agent. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 720–727, 2006.

Fujita, Y., Nagarajan, P., Kataoka, T., and Ishikawa, T. Chainerrl: A deep reinforcement learning library. *Journal of Machine Learning Research*, 22(77):1–14, 2021.

Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., and Wilkins, D. Pddl—the planning domain definition language, 1998.

Gopalan, N., Arumugam, D., Wong, L., and Tellex, S. Sequence-to-sequence language grounding of non-markovian task specifications. In *Robotics: Science and Systems*, 2018.

Goyal, P., Niekum, S., and Mooney, R. Using natural language for reward shaping in reinforcement learning. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 2019.

Goyal, P., Niekum, S., and Mooney, R. Pixl2r: Guiding reinforcement learning using natural language by mapping pixels to rewards. In *Conference on Robot Learning*, 2020.

Jothimurugan, K., Alur, R., and Bastani, O. A composable specification language for reinforcement learning tasks. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Kernigham, B. W. and Ritchie, D. M. *The C Programming Language*. Prentice Hall of India, 1973.

Littman, M. L., Topcu, U., Fu, J., Isbell, C., Wen, M., and MacGlashan, J. Environment-independent task specifications via gltl. *arXiv preprint arXiv:1704.04341*, 2017.

Luketina, J., Nardelli, N., Farquhar, G., Foerster, J., Andreas, J., Grefenstette, E., Whiteson, S., and Rocktäschel, T. A survey of reinforcement learning informed by natural language. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, August 10-16 2019, Macao, China.*, volume 57, pp. 6309–6317. AAAI Press (Association for the Advancement of Artificial Intelligence), 2019.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. Rectifier nonlinearities improve neural network acoustic models, 2013.

Maclin, R. and Shavlik, J. W. Creating advice-taking reinforcement learners. *Machine Learning*, 22(1):251–281, 1996.

Misra, D. and Artzi, Y. Reinforcement learning for mapping instructions to actions with reward learning, 2017.

Misra, D., Langford, J., and Artzi, Y. Mapping instructions and visual observations to actions with reinforcement learning. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 1004–1015, 2017.

Nota, C. The autonomous learning library. `https://github.com/cpnota/autonomous-learning-library`, 2020.

Patel, R., Pavlick, E., and Tellex, S. Grounding language to non-markovian tasks with no supervision of task specifications. In *Proceedings of Robotics: Science and Systems*, June 2020.

Puterman, M. L. Markov decision processes. *Handbooks in Operations Research and Management Science*, 2:331–434, 1990.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Sumers, T. R., Ho, M. K., Hawkins, R. D., Narasimhan, K., and Griffiths, T. L. Learning rewards from linguistic feedback. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(7):6002–6010, May 2021. doi: 10.1609/aaai.v35i7.16749. URL `https://ojs.aaai.org/index.php/AAAI/article/view/16749`.

Sun, S.-H., Wu, T.-L., and Lim, J. J. Program guided agent. In *International Conference on Learning Representations*, 2020.

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

Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Van Rossum, G. and Drake Jr, F. L. *Python Reference Manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

Watkins, C. J. and Dayan, P. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Williams, E. C., Gopalan, N., Rhee, M., and Tellex, S. Learning to parse natural language to grounded reward functions with weak supervision. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1–7. IEEE, 2018.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Younes, H. L. and Littman, M. L. Ppddl 1.0: The language for the probabilistic part of ipc-4. In *Proc. International Planning Competition*, 2004.

Zhong, V., Rocktäschel, T., and Grefenstette, E. Rtfm: Generalising to new environment dynamics via reading. In *International Conference on Learning Representations*, 2019.

A. RLang: Grammar and Semantics

In this section, we define the semantics of RLang expressions.

A.1. Grammar

```
⟨program⟩ ::= import ⟨declarations⟩
⟨declaration⟩ ::= ⟨constant⟩
                | ⟨action⟩
                | ⟨factor⟩
                | ⟨proposition⟩
                | ⟨goal⟩
                | ⟨feature⟩
                | ⟨markov_feature⟩
                | ⟨object⟩
                | ⟨class_definition⟩
                | ⟨option⟩
                | ⟨policy⟩
                | ⟨effect⟩
                | ⟨action_restriction⟩
⟨constant⟩ ::= Constant ⟨identifier⟩ :=
               ⟨arithmetic_expression⟩
⟨action⟩ ::= Action ⟨identifier⟩ :=
             ⟨arithmetic_expression⟩
⟨factor⟩ ::= Factor ⟨identifier⟩ :=
             ⟨special_variable⟩
⟨proposition⟩ ::= Proposition ⟨identifier⟩ :=
                  ⟨boolean_expression⟩
⟨goal⟩ ::= Goal ⟨identifier⟩ :=
           ⟨boolean_expression⟩
⟨feature⟩ ::= Feature ⟨identifier⟩ :=
              ⟨arithmetic_expression⟩
⟨markov_feature⟩ ::= MarkovFeature ⟨identifier⟩ :=
                     ⟨arithmetic_expression⟩
⟨object⟩ ::= Object ⟨identifier⟩ :=
             ⟨object_instantiation⟩
⟨policy⟩ ::= Policy ⟨identifier⟩ :
             ⟨policy_statement⟩
⟨policy_statement⟩ ::= ⟨execute_statement⟩
                     | ⟨conditional_policy_statement⟩
                     | ⟨probabilistic_policy_statement⟩
⟨execute_statement⟩ ::= Execute
                        ⟨arithmetic_expression⟩
⟨option⟩ ::= Option ⟨identifier⟩:
             ⟨option_init⟩
             ⟨policy_statement⟩
             ⟨option_until⟩
⟨option_init⟩ ::= init ⟨boolean_exp⟩
⟨option_until⟩ ::= until ⟨boolean_exp⟩
⟨class_definition⟩ ::= Class ⟨identifier⟩:
                       ⟨attribute_definitions⟩
⟨effect⟩ ::= Effect ⟨identifier⟩:
             ⟨effect_statements⟩
⟨effect_statement⟩ ::= ⟨reward⟩
                     | ⟨prediction⟩
                     | ⟨effect_reference⟩
                     | ⟨conditional_effect⟩
                     | ⟨probabilistic_effect⟩
⟨reward⟩ ::= Reward
             ⟨arithmetic_expression⟩
⟨prediction⟩ ::= ⟨identifier⟩' ->
                 ⟨arithmetic_expression⟩
⟨effect_reference⟩ ::= -> ⟨identifier⟩
⟨action_restriction⟩ ::= ActionRestriction
                         ⟨identifier⟩:
                         ⟨restrict_statements⟩
```

A.2. Semantics: Basic Syntactic Elements

RLang allows to express information that grounds to functions defined over the State-Action space of an MDP. Moreover, these functions have to be Markov, in the MDP sense, allowing to define functions with domain X that can be $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ and, its simplifications, $\mathcal{S}$, $\mathcal{A}$, $\mathcal{S} \times \mathcal{A}$, $\mathcal{S} \times \mathcal{S}$. The range of these functions can include real vectors $\mathbb{R}^d$ (with $d \in \mathbb{N}$), Booleans $\{\top, \bot\}$ and sets. The following are the basic expressions that are used to build the MDP specific elements of RLang.

- **Real Expressions** ⟨arithmetic_expression⟩ are functions of the form $f : X \to \mathbb{R}^d$ for some dimension d. Syntactically, RLang allows element-wise arithmetic operations $(+, -, *, /)$, numeric literals, and references to previously defined real functions to define new functions. These functions are Markov and defined over the State-Action Space of the MDP, i.e. $X \in \{\mathcal{S}, \mathcal{A}, \mathcal{S} \times \mathcal{A}, \mathcal{S} \times \mathcal{S}, \mathcal{S} \times \mathcal{A} \times \mathcal{S}\}$.
- **Constant expressions** ⟨constant⟩ allows to bind a

name (⟨identifier⟩) to literal value or a list of literal values.

- **Boolean Expressions** ⟨boolean_expression⟩ Analogous to Real Expressions, these are functions of the form $f : X \to \{\top, \bot\}$ with domain $X \in \{\mathcal{S}, \mathcal{A}, \mathcal{S} \times \mathcal{A}, \mathcal{S} \times \mathcal{S}, \mathcal{S} \times \mathcal{A} \times \mathcal{S}\}$.

  In order to define new Boolean expressions, RLang allow for logical operators `(and, or, not)` and order relations of the real numbers (<, <= , >, >=, =, !=).

- **Relations and Partial Functions** A relation of domains X and Y are a subset $R \subseteq X \times Y$. Partial functions are specifications of functions of the form $f : X \to Y$. A partial function is, then, a relation such that

$$PF := \{(x, y) \in X \times Y \text{ and } \forall (x, y), (x', y')\ x = x' \implies y = y'\}.$$

  Therefore, it is partial because not every element of the domain is defined by the function. We consider that undefined domain elements map to the special element `unknown`.

A.3. Semantics: RLang Expressions

Core RLang Type Definitions are necessary in order to derive new information from the base vocabulary.

- **State Space Definitions** allows to define important features and set of states for the agent.
  1. **State Features** ⟨feature⟩ These ground to functions of the form $\phi : \mathcal{S} \to \mathbb{R}^d$. Hence, given the base vocabulary and using real expressions, we can derive new features;
  2. **State Factors** ⟨factor⟩ In the particular case the state space is $\mathcal{S} \subseteq \mathbb{R}^n$ ($n \in \mathbb{N}$) and factored, we have specific state features that correspond to the State Factors whose definition is given by a list of integers that correspond to the positions in the state vectors that are part of the factor. A factorization of the state correspond to a set of disjoint factors whose union is the full state vector;
  3. **Propositions** ⟨proposition⟩ ground to Boolean functions with domain in $\mathcal{S}$ that represents a set of states $S_\sigma := \{s \in \mathcal{S} \text{ and } s \models \sigma\}$;

- **Action Definition** ⟨action⟩ names a particular action. Consider that the action space $\mathcal{A} \subseteq \mathbb{R}^d$ with $d \in \mathbb{N}$. Then, an action definition grounds to a point in $\mathcal{A}$.

- **Option Definition** ⟨option⟩ allows to define an temporally-abstracted action based on the options framework (Sutton et al., 1999). Therefore, the statement directly maps to the triple (I, π_o, β), where I is the initiation proposition defined by the syntactical element ⟨option_init⟩, β is the termination proposition defined by ⟨option_until⟩ and π_o is the policy function defined by a policy statement ⟨policy_statement⟩

- **Class and Object Definition** ⟨class⟩ and ⟨object⟩ allow for the definition of new classes and object instances. Classes can have any number of attributes of any of the types `str`, `int`, `float`, `bool`, `object` and object instances can have attributes that are functions of the (s, a, s') tuple.

- **Markov Feature Definition** ⟨markov_feature⟩ allow for the definition of real-valued features of a transition tuple (s, a, s') of the MDP. These expressions ground to real functions of the form $f : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}^n$ with $n \in \mathbb{N}$.

Core MDP Expressions are related to specifications of the main functions of the MDP:

Transition Dynamics and Rewards represented syntactically by ⟨effect_statement⟩. Such statements ground to tuples (R, T) of reward functions and next-state probabilities. More precisely, grounding functions are $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ and $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$. The following are the semantics of possible effect expressions.

- **Reward** ⟨reward⟩ statement allows to specify a reward value using real expressions. A reward statement grounds to a function $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ defined by scalar arithmetic expressions.

- **Next State Prediction** ⟨prediction⟩ ground to functions $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ that gives a probability of transitioning to the next state s' after executing action a at state s. The following are possible groundings:
  1. **Null Effect**: `S'` -> `S`. This grounds to
  $$T(s', a, s) = \begin{cases} 1 & \text{if } s' = s \\ 0 & \text{otherwise} \end{cases};$$
  2. **Singleton Prediction**: `S'` -> ⟨constant⟩. Let the constant ground to a valid state vector $\hat{s} \in \mathcal{S}$. Thus, the expression grounds to
  $$T(s', a, s) = \begin{cases} 1 & \text{if } s' = \hat{s} \\ 0 & \text{otherwise} \end{cases};$$
  3. **Real Prediction**: `S'` -> ⟨arithmetic_expression⟩. Let the

⟨arithmetic_expression⟩ ground to a real function of the form $e : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Then, the prediction expression grounds to

$$T(s', a, s) = \begin{cases} 1 & \text{if } s' = e(s, a) \\ 0 & \text{otherwise} \end{cases};$$

4. **Factor Prediction** `factor_name ->` ⟨arithmetic_expression⟩. Let `factor_name` ground to the factor $\phi : \mathcal{S} \to \mathbb{R}^d$ where $d \in \mathbb{N}$ is the dimension of the factor. Let the ⟨arithmetic_expression⟩ ground to $e_\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$. Then, a factored prediction grounds to the function

$$T_\phi(\phi(s'), a, s) = \begin{cases} 1 & \text{if } \phi(s') = e(s, a) \\ 0 & \text{otherwise} \end{cases};$$

A collection of factor predictions for a set of disjoint factors $\{\phi_i\}_i$ that partition the state vector can ground to a full transition function

$$T(s', a, s) = \prod_i T_{\phi_i}(\phi_i(s'), a, s)$$

5. **Probabilistic Effect Statements** ⟨probabilistic_effect_statement⟩ allows to explicitly indicate probabilities for a collection of predictions. Consider that the probabilistic statement is a collection of tuples $\{(T_i, p_i)\}_i$ where T_i is the grounding function of the prediction and p_i a probability, subject to the correctness of the probabilities $\sum_i p_i \leq 1$ and for all $p_i \geq 0$. Thus, it grounds to

$$T(s', a, s) = \sum_i p_i T_i(s', a, s).$$

If $\sum_i p_i < 1$, then the remaining probability is construed to be assigned to `unknown`.

In the case of Factor predictions for a given factor, the grounding function T_ϕ is defined analogously.

6. **Conditional Effect Statements** ⟨conditional_effect_statement⟩ The conditional context allows to define subsets of the domain $D \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ through a ⟨boolean_expression⟩ that defines when a particular `Execute` statement is valid. Hence, a ⟨conditional_policy_statement⟩ grounds to partial functions $R = \{(D_i, R_i)\}_i$ where each tuple is the result of a branch from the parsing of `if-elif-else` blocks. Analogously, for the grounding of dynamics information of the statement $T = \{(D_i, T_i)\}_i$, where the D_i are disjointed subsets of the domain defined by the Boolean expressions, the order of the conditional branches and the T_i and R_i are defined by reward and prediction statements defined above.

7. **Effect References** ⟨effect_reference⟩ allows to refer to previously defined effect statements to compose new ones. Each referred effect is tuple of (R_i, T_i) of groundings for rewards and transition dynamics.

A collection of effect references ground to:

Rewards $R(s, a, s') = \sum_{i \in I(s,a,s')} R_i(s, a, s')$ where $I(s, a, s')$ is the set of referred effects that have information for rewards in the tuple (s, a, s'). Hence, rewards are composed additively.

Transition Dynamics Let $S'_i(s, a) = \{s' \in \mathcal{S} : T_i(s', a, s) > 0\}$ be the set of next states from state-action pair (s, a) about which T_i provide knowledge. Thus, a set of effect references are well-defined if $\bigcap_i S'_i(s, a) = \emptyset$ and $\sum_{s' \in \bigcup_i S'_i(s,a)} T(s', a, s) \leq 1$ for all (s, a). Hence, the grounding function is

$$T(s', a, s) = \sum_i T_i(s', a, s).$$

Policies ⟨policy⟩ ground to policy functions $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$. The simplest expression to specify a policy is an **execute statement** ⟨execute_statement⟩. The name after the `Execute` keyword represents either an ⟨action⟩ $a' \in \mathcal{A}$ and, hence, the statement grounds to

$$\pi(s, a) = \begin{cases} 1 & \text{if } a = a' \\ 0 & \text{otherwise} \end{cases},$$

or it can refer to a previously defined ⟨policy⟩ that grounds to $\hat{\pi}$ and, then, the statement grounds to $\pi = \hat{\pi}$. Therefore, the ⟨execute_statement⟩ functions analogously to a return statement in function definitions in imperative programming languages: when an action name is found, it maps the querying state s to the first action referenced by an execute statement.

Probabilistic policy expressions ⟨probabilistic_policy_statement⟩: Probability statements allow to extend the execute statement with explicit probability values. Therefore, a probabilistic policy statement grounds to a collection of execute statements-probability pairs $\{(\pi_i, p_i)\}_i$. In this way, probabilistic policy statements ground to

$$\pi(s, a) = \sum_i p_i \pi_i(s, a)$$

If $\sum_i p_i < 1$, then the remaining probability is construed to be assigned to `unknown`.

Conditional Policy Expressions ⟨conditional_policy_statement⟩: The conditional context allows to define subsets of the domain $S' \subseteq \mathcal{S}$ through a ⟨proposition⟩ that defines when a particular execute statement is valid. Hence, a ⟨conditional_policy_statement⟩ grounds to a partial function $\pi' : \{(S_i, \pi_i)\}_i$ where each pair (S'_i, π_i) the result of a branch from the parsing of `if-elif-else` blocks. The S'_i are disjointed and they are the result of the ⟨proposition⟩, the order of the statements and the returning semantics of the execute statements. The π_i are defined by execute statements or probabilistic policy statements.

Action Restrictions ⟨action_restriction⟩ are defined analogously to conditional policy statements. They reduce the possible set of actions to consider in a given situation. They ground to functions of the form $A : \mathcal{S} \rightarrow \mathcal{A}$ that defines then subset of prohibited actions to take in state s—i.e., $A(s) \subset \mathcal{A}$.

Goals ⟨goal⟩ ground to set of states that are considered goal states for the MDP, i.e. terminating and highly rewarding. RLang represents goals through propositions.

B. Experimental Details and Additional Results

In this section, we extend the discussion on the implementation details of RLang demonstrations in Section 4. We provide details about the implementation of the RLang-informed variations of the RL algorithms, descriptions of the environments and the hyperparameters used.

For each of the experiments below we report average return curves over 5 different random seeds and report 95% confidence intervals. Moreover, we use a running average with window size of 50.

B.1. Lava-Gap

Lava-Gap is a 6×6 grid-world with coordinates $x, y \in \{1, 6\}$. There is a wall in position $(3, 1)$ and 4 lava pits in locations $(3, 2), (1, 4), (2, 4), (2, 5)$. The goal position is $(5, 1)$. The agent has 4 discrete actions that allows it to move in one of the cardinal directions by 1 step. Each action has a probability of failure of $1/3$ that would move the agent to a random neighboring position (in the cardinal directions). The state s_t at time t is represented by the x and y coordinates of the position at time t. The agent receives a reward of -1 for falling in a lava pit and the episode terminates. Similarly, the agent receives a reward of 1 for reaching the goal and the episode terminates. In any other case, the reward is 0. At the start of every episode, the agent begins executing at position $(1, 1)$. We use a discount factor of $\gamma = 0.95$. We use `simple_rl`'s implementation (Abel, 2019) of this gridworld and of the RMax and Q-Learning algorithms.

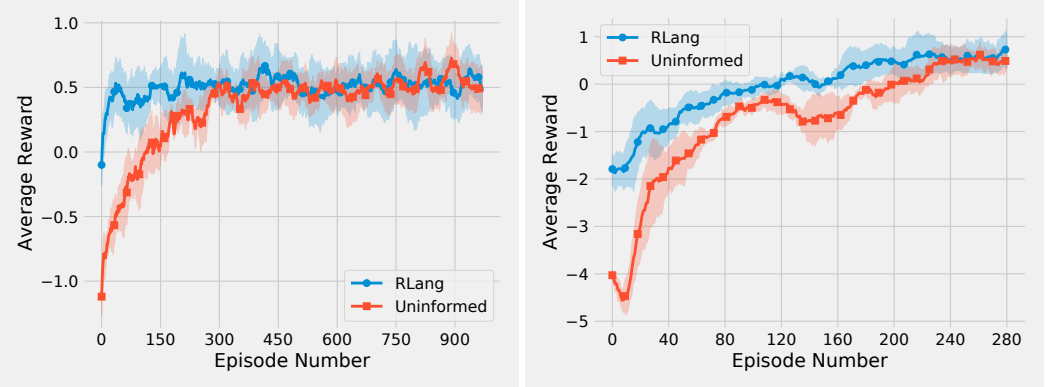
RLang-informed Q-Learning In the RLang-informed example for Lava-gap, we leverage the transition and reward information from the RLang program to initialize the Q-table for Q-Learning and R-Max. We compute the Q-table by executing value iteration considering in the state pairs where transition information is available. In Algorithm 1, we show how the Q-table of a Q-Learning agent is initialized using the information provided in a RLang program given as input as the RLangKnowledge objects. The UpdateValue function computes the new value using the standard TD-error.

Algorithm 1 Q-Table Initialization

```
function InitQTable(RLangKnowledge, QAgent)
  for (s,a) ∈ S × A do
    if RLangKnowledge.Reward(s,a) is known then
      QAgent.Q(s,a) ← RLangKnowledge.Reward(s,a)
    end if
  end for
  for N iterations do
    for (s,a,s') ∈ S × A × S do
      if RLangKnowledge.Transition(s,a) is known
      then
        T(s, a, s') ←
        RLangKnowledge.Transition(s, a)
        R(s, a, s') ← RLangKnowledge.Reward(s,a)
        QAgent.Q(s,a) ←
        UpdateValue(s, a, s', T(s, a, s'), QAgent)
      end if
    end for
  end for
end function
```

RLang-informed RMax RMax (Brafman & Tennenholtz, 2002) is a model-based RL algorithm. Hence, we directly use the RLang-provided information to initialize the model of the world (i.e., Transition and Reward tables) in addition to initializing the Q-table. We set the count in the respective tables to be a hyper-parameter K, less that the threshold used by RMax to considered a transition known.

Hyperparameters For uninformed Q-Learning we have the exploration $\epsilon = 0.1$ and the step size $\alpha = 0.05$ and for the RLang-informed Q-Learning we use $\epsilon = 0.01$ and the same α. For RMax and RLang-informed RMax, we use a threshold of 30 samples to consider the transition learned and set $K = 1$.



(a) RLang-informed Q-Learning

(b) RLang-informed RMax

Figure 6: Average return curves for Lava-Gap environment. We provide the agent with an RLang program that contains information about the reward function and the transition dynamics.

RMax Results In Figure 6b, we show the average return curves for an RMax agent informed with the program below, in which we see consitent results with the results obtained for an RLang-informed Q-Learning agent.

B.2. Taxi (Flat)

We use `simple_rl`'s implementation of the Taxi environment (Dietterich, 1998) with 2 passengers in a 5×5 grid. The state vector has the position of the agent and a binary variable that is 0 when the taxi does not carry a passenger and 1 otherwise. Moreover, it has the current position of the passengers, the destination of the passenger and a binary variable that indicates if the passenger is in the taxi. The agent has 4 movement actions and a special action for picking up a passenger that is at the same position than the agent and another for dropping off the passenger currently in the taxi. The reward function is 1 when all passenger are in destination and 0 otherwise. The discount factor $\gamma = 0.95$.

RLang-informed hierarchical Q-Learning In this experiment, we use a hierarchical RL agent based on the options framework. In this particular case, we use Q-Learning to learn both the policy over options and the intra-option policies. We consider that an RLang-defined option is learnable if the policy function is not provided, i.e. only initiation and termination conditions are specified. We use such termination condition as a goal represented by a pseudo-reward function that is 1 when the termination condition is achieved and 0 otherwise that the inner agent uses to learn the intra-option policy. We initialize the intra-option learning agents of those learnable options defined in the input RLang program with the procedure in Algorithm 2.

Hyperparameters For our Q-Learning baseline, we use an exploration $\epsilon = 0.1$ and a step size of $\alpha = 0.1$. For our hierarchical Q-Learning agents, we have a Q-Learning agent for each subpolicy to be learnt with $\epsilon = 0.1$ and $\alpha = 0.1$ and a Q-Learning agent to learn the policy over options with $\epsilon = 0.01$ and $\alpha = 0.5$. We implemented our hierarchical Q-Learning agent as Sutton et al..

Algorithm 2 Hierarchical Agent Initialization

function InitializeOptions(RLangKnowledge)
  for $o \in$ RLangKnowledge.Options **do**
    if o is `learnable` **then**
      o.agent $\leftarrow$ `InitializeAgent()`
    end if
  end for
end function

Results In Figure 7a, we show the average return curves for Taxi. The RLang program, shown below, defines the options given to the agent—a simple variation of this program is provided to the agent that needs to learn the intra-option policies. The plot shows the most-informed agent, i.e. intra-option policies are provided and it only needs to learn the policy over options, is represented in blue, the RLang-informed agent that only knows the initiation and termination conditions of the options (in red) and an uninformed Q-Learning agent. We observe that both of the informed agents are able to exploit the knowledge to gain a steeper learning curve with respect to the uninformed agent.

```
Option pick_up_passenger0:
  init(not(passenger_in_taxi) and
  not(passenger_0_in_dest))
    Execute pick_up_passenger0
  until passenger_0_intaxi

Option dropoff_passenger0:
  init(passenger_0_intaxi)
    Execute dropoff_passenger0
```

```
    until passenger_0_in_dest and not(
    passenger_0_intaxi)

  Option pick_up_passenger1:
    init(not(passenger_in_taxi) and
    not(passenger_1_in_dest))
      Execute pick_up_passenger1
    until passenger_1_intaxi

  Option dropoff_passenger1:
    init(passenger_1_intaxi)
      Execute dropoff_passenger1
    until passenger_1_in_dest and not(
    passenger_1_intaxi)
```

B.3. Taxi (Object Oriented)

For object-oriented Taxi we used `efficient_rl`'s implementation of the environment presented in Diuk et al. (2008) with 1 passenger in a 5×5 grid. The agent has access to the same information as the flat Taxi environment with the addition of a set of seven predicates whose truth values are available at each state:

```
touch_north(taxi, wall)
touch_south(taxi, wall)
touch_east(taxi, wall)
touch_west(taxi, wall)
on(taxi, passenger)
on(taxi, destination)
in_taxi(passenger)
```

The agent's action space is the same as the flat Taxi environment: 4 movement actions and a special action for picking up a passenger. The reward function is different than flat Taxi, however, and matches one given in Diuk et al. (2008): a reward of 20 for dropping off a passenger at a depo, a reward of -10 for dropping off a passenger anywhere else or attempting a pickup when the taxi is not on a passenger, and a reward of -1 otherwise. The discount factor is likewise $\gamma = 0.95$.

RLang-informed DOORmax In this experiment we use the DOORmax algorithm implemented by `efficient_rl` that was originally presented in Diuk et al. (2008) as a baseline and create an RLang-enabled DOORmax agent. Similar to the RLang-enabled Q-Leaning agent, the RLang-enabled DOORmax agent starts by initializing its internal representation of the transition and reward function with the partial dynamics given by an RLang program.

Hyperparameters Both DOORmax and RLang-DOORmax use the same hyperparameters. We set $r_{max} = 20$, $\gamma = 0.95$, $\delta = 0.01$, and $K = 5$, the maximum number of different effects possible for any action.

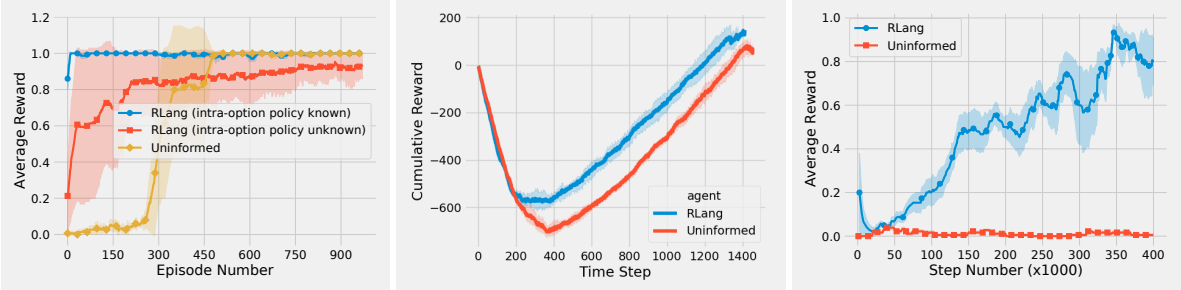
Results In 7b, we show the cumulative reward curves for DOORmax agents in object-oriented Taxi. The RLang program, shown below, describes the full reward function and partial transition dynamics, specifically what happens when the agent tries to drive into walls and what happens when a passenger is picked up and dropped off. The plot shows a significant speedup for the RLang-enabled agent, even when partial transition dynamics are described.

```
Effect movement_effect:
  if S.taxi.touch_n and A == move_n:
    S' -> S
  if S.taxi.touch_s and A == move_s:
    S' -> S
  if S.taxi.touch_e and A == move_e:
    S' -> S
  if S.taxi.touch_w and A == move_w:
    S' -> S

Effect main:
  if S.taxi.on_passenger and A ==
    pick_up:
    S'.passenger.in_taxi -> True
  if S.passenger.in_taxi and A ==
    drop_off:
    S'.passenger.in_taxi -> False
    if S.taxi.on_destination:
      Reward 20
    else:
      Reward -10
  elif A == pick_up or A == drop_off:
    Reward -10
  else:
    Reward -1
```

B.4. 2D Minecraft

2D Minecraft is a crafting environment based on Andreas et al. implemented as a 10×10 grid. The state vector include a map of the environment, an inventory vector and the change on inventory with respect to the previous time step. The map is represented by $10 \times 10 \times 22$ tensor that represent with a one-hot vector the element at position (x, y). The agent has 4 actions to move in the cardinal directions by one position and a special action `use` to interact with the element *in front*, i.e., given the current position and orientation of the agent, the position with which the agent interacts in the one the agent is facing. If such element is a primitive, the agent adds it to its inventory; if the element is a workbench and it has any of the primitive elements to build an object in the workbench, then those objects are built and added to the inventory (any primitive element used is removed from the inventory); if the agent is in front of water and has a bridge, it can `use` it and cross the water. If the agent has to interact with stone, it needs an axe in inventory. This is a goal-oriented environment; when the agent has the goal object in inventory, it receives a reward of 1 and the episode terminates. When an episode starts, the agent is randomly placed in any free cell of the grid. We use a discount factor $\gamma = 0.99$.



(a) Taxi (Flat) (b) Taxi (Object-Oriented) (c) 2D Minecraft

Figure 7: Cumulative reward curves for object-oriented Taxi (middle) when the reward function and partial transition dynamics is provided using RLang. Average return curves for flat Taxi (left) and Craftworld (right) when information about the hierarchical structure of the problem is provided using RLang. We provide the agents with a program that specifies the initiation and termination conditions of the options required to solve the problem and the RLang-informed agent learns the intra-option policies and policy over options. In the particular case of flat Taxi, we also include the learning curve when we also provide the intra-option policies.

RLang-informed hierarchical DDQN Agent To solve this environment, analogously to the Taxi environment, we use a hierarchical agent based on options. We use DDQN (Van Hasselt et al., 2016) as the algorithm to learn both the policy over options and intra-option policies. Algorithm 2 is used to initialized the intra-option policies. To implement DDQN and its hierarchical variation based on options, we based it on the Autonomous Learning Library (Nota, 2020).

Neural Network architecture and parameters We use a CNN with 4 ReLU-activated layers with filter banks of size $32, 32, 32, 64$, a kernel size of 3 and stride of 2 (padding was used to keep the dimension 10×10). The inventory and inventory change were processed with a ReLU MLP with hidden layer of size 32 and output 32. These two vectors are concatenated and passed through a linear layer of size 256 (for the flat DDQN) and 64 for the agents in the hierarchical DDQN implementation. Finally, this output vector is passed through a ReLU MLP with a hidden layer of size 64 to get the value predictions for each action.

Hyperparameters For DDQN, we use a linear schedule for $\epsilon-$greedy exploration with start with $\epsilon = 1$ and $\epsilon = 0$ and final exploration step 10000. We use a learning rate 0.001 and mini-batch of size 64. We use a Prioritized replay buffer of size 10000. The target network update frequency is 100 steps. We set a time of 1000 steps.

For hierarchical DDQN:

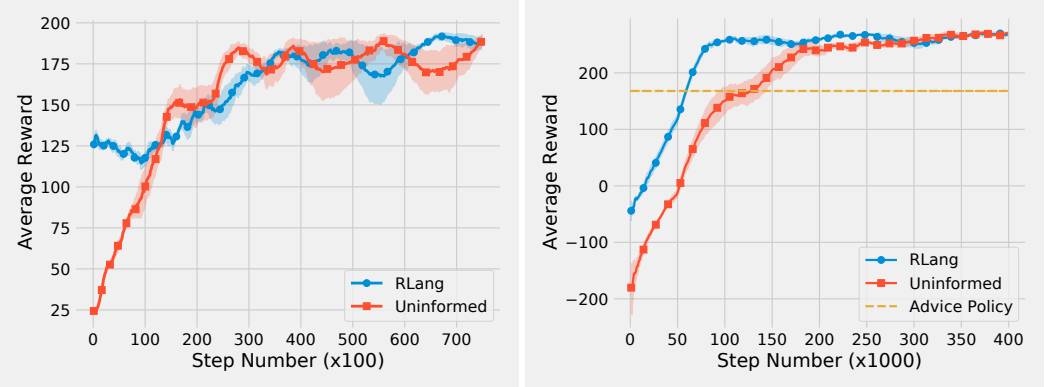
- Outer Agent (policy over options): we use a linear schedule for $\epsilon-$greedy exploration with start with $\epsilon = 1$ and $\epsilon = 0$ and final exploration step 60000. We use a learning rate 10^{-5} and mini-batch of size 64. We use a Prioritized replay buffer of size 10000. The target network update frequency is 100 steps. This outer agent had a timeout of 1000 steps;
- Inner Agents (intra-option policies): we use a linear schedule for $\epsilon-$greedy exploration with start with $\epsilon = 1$ and $\epsilon = 0.001$ and final exploration step 30000. We use a learning rate 10^{-4} and mini-batch of size 128. The target network update frequency is 100 steps. We use a Prioritized replay buffer of size 10000. Each subpolicy had a timeout of 100 steps.

B.5. Classic Control

Environments In this section, we consider Cartpole and Lunar Lander, two classic environments for RL research that have continuous state-space and discrete action space. For both environments, we use OpenAI Gym's implementations (Brockman et al., 2016), i.e. CartPole-v0 and LunarLander-v2.

Cartpole (Barto et al., 1983) consists of an underactuated pole attached to a cart. At the beginning, the pole starts in a vertical position (0 degrees) and the agent has to learn a policy to keep it within 15 degrees. The state consists of the position and the velocity of the cart, and the angle and angular velocity of the pole. The action space is to apply momentum to move the cart to the right or to the left. An episode ends when the pole absolute angle is greater than 15 degrees, the position of the cart is greater than 2.4 units or after 200 time steps. The agent gets a reward of 1 every time step. This task is considered solved is the average return is at least 195 over 100 episodes.

Lunar Lander simulates the task of landing a ship in a landing pad on the moon. The state consists of the ship's position and velocity, angle and angular velocity, and Boolean flags that indicate if the ship's leg is in contact with the ground.



(a) CartPole using REINFORCE

(b) Lunar Lander using PPO

Figure 8: Average return curves for classic control tasks with continuous state spaces. We use RLang to provide the agent with an initial policy (non-optimal) that the agent can leverage to improve learning performance. We compare with the uninformed counterparts.

The actions are to fire the main engine, the right orientation engine, the left orientation engine and doing nothing. The agent gets a reward of -100 if the ship crashed and $+100$ if it lands correctly. It receives $+10$ for each leg that touches the ground and -0.3 if the main engine is fired. The task is solved with a return of at least 200.

RLang-informed Policy Gradient Agent To solve these environments, we provided non-optimal policies through RLang programs and use policy gradients methods to leverage this knowledge while learning. Algorithm 3 is derived from (Fernández & Veloso, 2006), in which, we probabilistically share control between the learning policy π_θ and the RLang-provided policy $\hat{\pi}$. At each time step, we choose which policy to follow by drawing a sample from a Bernoulli distribution with parameter β. We use such probabilistic mixing to collect trajectories and then optimize, θ, using policy gradient methods. We use REINFORCE (Williams, 1992) for Cartpole and PPO (Schulman et al., 2017) for Lunar Lander. The mixing parameter β is annealed exponentially using a decay rate α.

Algorithm 3 Hierarchical Agent Initialization

1: **function** PolicyMixing(RLangKnowledge,decay_rate)
2: Trajectories $\leftarrow$ `Rollout`(Env, $\pi_\theta, \hat{\pi}, \beta$)
3: $\theta \leftarrow$`PolicyGradient`(Trajectories, π_θ)
4: $\beta \leftarrow \beta *$ decay_rate
5: **end function**

Neural Network architecture and parameters For Cartpole and REINFORCE, we use a policy network using an MLP with hidden size 64 and Leaky ReLU activations (Maas et al., 2013) with parameter 0.2

In the case of Lunar Lander and PPO, we use a policy network based on a MLP with hidden size 64 and Leaky ReLU activations (Maas et al., 2013) with parameter 0.2. As a value network, we use an MLP with hidden size 64 and Tanh activations.

Hyperparameters For Cartpole, we use PFRL's REINFORCE implementation (Fujita et al., 2021). We use an initial mixing parameter $\beta = 0.7$ and a decay rate $\alpha = 0.99$. For REINFORCE, we use a learning rate of 0.001 and a batch size of 5. In Lunar Lander, we use PFRL's PPO implementation and use a mixing parameter $\beta = 0.5$ and a decay rate $\alpha = 0.9999$. For PPO, we used a learning rate of 0.0002.

Cartpole Results For Cartpole, we provide the RLang program below with a very simple prior policy. In Figure 8a, we show the average return curves for RLang-informed REINFORCE and its uninformed performance, which show a jump-start performance gain for the agent then improves through experience.

```
Policy balance_pole:
  if pole_angular_velocity > 0:
    Execute move_right
  else:
    Execute move_left
```