RLang: A Declarative Language for Expressing Prior Knowledge for Reinforcement Learning

Rafael Rodriguez-Sanchez
Department of Computer Science
Brown University
Providence, RI
rrs@brown.edu

Benjamin Spiegel
Department of Computer Science
Brown University
Providence, RI
bspiegel@cs.brown.edu

Jennifer Wang
Department of Computer Science
Brown University
Providence, RI
jennifer_wang2@brown.edu

Roma Patel
Department of Computer Science
Brown University
Providence, RI
romapatel@brown.edu

Stefanie Tellex
Department of Computer Science
Brown University
Providence, RI
stefie10@cs.brown.edu

George Konidaris
Department of Computer Science
Brown University
Providence, RI
gdk@cs.brown.edu

Abstract

Communicating useful background knowledge to reinforcement learning (RL) agents is an important and effective method for accelerating learning. Oftentimes, a concise piece of information might considerably improve the agent's learning performance. For instance, *do not fall in lava pits!*. However, there is no standardized and expressive enough medium to provide such type of information. Therefore, we introduce RLang, a domain-specific language (DSL) for communicating domain knowledge to an RL agent. Unlike other existing DSLs proposed by the RL community that ground to *single* elements of a decision-making formalism (e.g., the reward function or policy function), RLang can specify information about every element of a Markov decision process. We define precise syntax and grounding semantics for RLang such that RLang programs ground to algorithm-agnostic *partial* world model and policy that can be exploited by an RL agent. Finally, we provide some example RLang programs to introduce the language expressions, and provide a simple example that show how RL methods can effectively exploit the resulting knowledge.

Keywords: Reinforcement Learning, Language

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 and Shavlik [1996] or policy structure Andreas et al. [2017]. Communicating such knowledge using natural language would be more intuitive, though this approach would require converting natural language sentences into grounded knowledge usable by the agent; most of the approaches in this area restrict the possible groundings by translating natural language into expressions of a restricted grammar. For example, for describing task objectives Artzi and Zettlemoyer [2013], Patel et al. [2020], or other individual components of decision-making systems 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. Finally, we show a simple example of how RLang can help improve performance of an RL agent.

2 Notation

Reinforcement learning tasks are typically modeled as Markov Decision Processes (MDPs; Puterman, 1990), which are 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, $R : \mathcal{S} \times \times \to R$ is a reward function, and $\gamma \in (0, 1]$ is a discount factor. A solution to an MDP is a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ that maximizes the expected discounted return $[\sum_{t=0}^{\infty} \gamma^t r_t]$, where r_t is the reward obtained at time step t. The value function $V^\pi : \mathcal{S} \to R$ for a policy π captures the expected return an agent would receive from executing π starting in a state s. The action-value function $Q^\pi : \mathcal{S} \times \mathcal{A} \to R$ of a policy is the expected return from executing an action a in a state s and following policy π thereafter. Options [Sutton et al., 1999] formalizes temporally-extended actions: closed-loop policies defined by a tuple (I, π, β), where $I \subseteq$ is a set of states in which the option can be executed, π is an option policy, and $\beta :\to [0, 1]$ describes the probability that the option will terminate upon reaching a given state.

3 RLang: Expressing Prior Knowledge about Reinforcement Learning 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 in order to be useful.

Unfortunately, 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 the same general-purpose programming language in which the algorithm is implemented, typically in an ad-hoc fashion. This has two primary drawbacks. First, prior knowledge is often task-specific, and the lack of a medium to express it hinders the development of general-purpose learning algorithms that can exploit varying types and degrees of background knowledge. Second, this approach is not accessible to end-users or other consumers of RL agents, who do not write the algorithms themselves and cannot necessarily be expected to master the relevant programming languages and mathematical details, but who might nevertheless wish to accelerate learning.

We propose RLang, a DSL designed to be a standardized medium to provide background knowledge to RL agents in an algorithm-agnostic manner. RLang programs can be parsed using our Python package into an algorithm-agnostic data structure that can be integrated into nearly any reinforcement learning algorithm. In this section, we describe the main RLang element types and expressions that compose RLang programs.

3.1 RLang Elements

State Factors In Factored MDPs, the state space is a collection of conditionally independent variables: $\mathcal{S} = \mathcal{X}_1 \times .. \times \mathcal{X}_n$. For example, consider a 2-D version of Minecraft 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: $s = (pos, map, inventory)$. `Factors` can be used to reference these independent 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 which refer to the current action and the next state, respectively.

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` that captures the number of axes that can be potentially built at the current state: `Feature number_of_axes := wood + iron`.

Propositions `Propositions` in RLang, which are functions of the form $\mathcal{S} \to \{,\}$, 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
```

Markov Features Markov Functions like the action-value function or transition function take the form $\times\times \to R$. We extend the co-domain of this function class to R^n, where $n \in N$, 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 can be used to execute an action or call 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.

Options Temporally-extended actions can be 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
```

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. When using a factored MDP, RLang can also be used to 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. Moreover, a `main` Effect designates the primary environment dynamics, and grounds to a partial factored world model $(\overline{\mathcal{T}}, \overline{\mathcal{R}})$. Similar to policies, Effects can be made probabilistic using `with`.

4 Demonstrations

We consider a 2D version of Minecraft based on Andreas et al. [2017], consisting of a gridworld 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. We show how providing the sub-policy structure of the task improves performance. Specifically, 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), leaving the agent to learn the policy over options. The following program concisely defines 3 options fully and 4 options with uninformative policies. This is an example of a simple RLang program that conveys partial hierarchical structure that effectively help the agent.

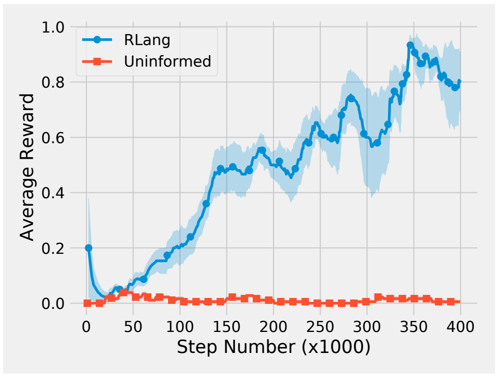

Figure 1: Average return curves for hierarchical agent informed with relevant options.

```
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 the DDQN algorithm Van Hasselt et al. [2016].

Figure 1 show the average return of RLang-informed hierarchical DDQN Van Hasselt et al. [2016] vs. the uninformed (flat) performance of a DDQN agent. The results show that providing a concise program partially describing a hierarchical solution was sufficient to successfully learn to solve the task, in stark contrast with the uninformed DDQN agent.

5 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 and 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 and Shavlik [1996] propose an RL paradigm in which the agent may request advice, as provided through 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 for the agent. RLang expands on all of these DSLs to include information beyond the policy and the reward function, thus allowing a wider array of information to be parsed and interpreted by the agent.

6 Conclusion

RLang is a precise, 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 provide use-cases in which RLang is used to provide diverse type of domain knowledge and structure, clearly demonstrating that simple and intuitive RLang programs describing task knowledge can effectively improve learning performance over *tabula rasa* methods.

References

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

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

SRK Branavan, Luke S Zettlemoyer, and Regina Barzilay. Reading between the lines: Learning to map high-level instructions to commands. Association for Computational Linguistics, 2010.

Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl—the planning domain definition language. 1998.

Prasoon Goyal, Scott Niekum, and Raymond J Mooney. Using natural language for reward shaping in reinforcement learning. *arXiv preprint arXiv:1903.02020*, 2019.

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

Michael L Littman, Ufuk Topcu, Jie Fu, Charles Isbell, Min Wen, and James MacGlashan. Environment-independent task specifications via gltl. *arXiv preprint arXiv:1704.04341*, 2017.

Jelena Luketina, Nantas Nardelli, Gregory Farquhar, Jakob Foerster, Jacob Andreas, Edward Grefenstette, Shimon Whiteson, and Tim Rocktäschel. A survey of reinforcement learning informed by natural language. *arXiv preprint arXiv:1906.03926*, 2019.

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

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

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

Theodore R Sumers, Mark K Ho, Robert D Hawkins, Karthik Narasimhan, and Thomas L Griffiths. Learning rewards from linguistic feedback. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6002–6010, 2021.

Shao-Hua Sun, Te-Lin Wu, and Joseph J. Lim. Program guided agent. In *International Conference on Learning Representations*, 2020.

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

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

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