Towards Learning-Augmented Languages

Xinyuan Sun
University of California, Davis
U.S.A
sxysun@ucdavis.edu

ABSTRACT

Reinforcement learning (RL) has seen tremendous success at solving a variety of problems ranging from industrial automation to games. This paper describes how existing programming languages can be augmented with new features so as to allow developers to exploit the power of modern RL algorithms and implementations.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; *Formal language definitions*; • **Computing methodologies** → Artificial intelligence;

KEYWORDS

Domain-specific languages, Reinforcement learning, Program synthesis

ACM Reference Format:

Xinyuan Sun. 2018. Towards Learning-Augmented Languages. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3236024.3275432

1 INTRODUCTION

Programmers often hard-code heuristics in programs that determine which action should be performed to achieve a particular goal. Such heuristics have a significant impact on the performance of the code. For instance, an implementation of a SAT solver involves various heuristics that determine the restart policy of the search algorithm [4]. This heuristic controlling when to perform a restart not only changes across different input formulas to the SAT solver, but can also change during the execution of the SAT solver on a specific input. Contemporary programming languages do not provide features that explicitly support specifying such adaptive heuristic decision points in programs.

Reinforcement learning (RL) [11] is a machine-learning paradigm where an agent learns a policy function that maximizes some reward in an environment. Recent progress in deep reinforcement learning (DRL) enable agents to learn complex policies for a variety of tasks [7, 9].

This paper describes the first steps towards designing and implementing a *learning-augmented language* (LAL) that bridges the gap between the programmer's need for specifying adaptive decisions in programs and the available agent-based DRL frameworks. In particular, a LAL supports two primary functions: **`choice`**, which enables the programmer to specify where a decision needs to be made, and **`reward`**, which specifies the metric that needs to maximized. The implementation of the language uses DRL to synthesize RL agents that implement the various `choice` and `reward` functions. As a simple illustrative example, consider the following program:

Listing 1: reachTen

```
cnt = 0
n = 0
while (cnt < 10 and n < 20):
   r = random(-5, 2)
   if (choice(numChoices=2) == 0):
      cnt += r
   reward(amount=-1)
   n += 1
if (cnt >= 10):
  reward(amount=100)  // Success!
else:
  reward(amount=-100)  // Failure.
```

The goal of **reachTen** is to ensure that `cnt` reaches 10 within 20 tries as quickly as possible starting from the initial value of 0. At each iteration of the loop, the program has to decide whether it should add `r` to `cnt`. The function `choice` can return two values, 0 and 1, indicated by setting the argument `numChoices` to 2. The implementation of the LAL should synthesize a `choice` function that, in effect, returns `0` only if `r` is positive.

The above simple example illustrates the following challenges in the implementation of the primitives `choice` and `reward`:
Reward attribution: Calls to `choice` and `reward` can be scattered across the code. Thus, we need to associate each `reward` with the appropriate `choice`, that is, those that impacted the reward.
Environment Synthesis: The implementation of `choice` has to be a function of the current state of the program. This state (or environment) should be synthesized and should incorporate relevant program variables and expressions.
Safety: To guide the learned `choice` function to make safe decisions, we incorporate `assert` functions into our LAL.

2 DESIGN OF THE LANGUAGE

Our learning-augmented language is built using the existing agent-based model used in reinforcement learning. An *RL agent* supports the following two functions: (i) `agent.query(state) : action`, which asks the agent to provide the best action given the state of the environment, and (ii) `agent.update(state, action, reward)`, which updates the agent by providing information about the reward associated with choice the specified action in the given state.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11.
https://doi.org/10.1145/3236024.3275432

As a wrapper around this agent-based model, we introduce two primitives for our first learning-augmented language LAL_1: (a) `choice(agentId: Int, numChoices: Int, env: Set): Int`, which queries agent `agentId` for the best action (represented by an integer from 0 to `numChoices-1`) for the environment `env`, and (b) `reward(agentId: List[Int], amount: List[Int])`, which associates the specified reward amount to the previous `choice` made by the agent `agentId`. There could be multiple calls to `reward` between subsequent calls to `choice` for the same agent. In this case, the reward amounts are added up. Implementing LAL_1 using an the agent-based model for RL described above is straight forward.

Synthesizing the agent IDs and the right environment is a burden for the programmer. Thus, we propose the following two related primitives that do not require the programmer to specify the agent IDs and the environment: (1) `choice(numChoices: Int): Int`, and (2) `reward(amount: Int)`. We call this language LAL_2. In the rest of the section, we describe how to transpile LAL_2 to LAL_1.

2.1 Reward Attribution

Each call to a `choice` function is associated with a unique agent ID. Reward attribution links each `reward` function with an agent Id, and, hence, with a `choice` function.

We say that a `reward` shares the same agent ID as a `choice` function if the execution of the `reward` function depends on the execution of the `choice` function. Specifically, we associate a `reward` with all the `choice` along the backward static slice [14] of the `reward`. If there are multiple `choice` functions in the slice, then they all get associated with the same `reward`. Applying reward attribution to **reachTen** in Listing 1 gives us:

Listing 2: reachTen with Reward Attribution

```
cnt = 0
n = 0
while (cnt < 10 and n < 20):
   r = random(-5, 2)
   if (choice(agentId=0, numChoices=2) == 0):
      cnt += r
   reward(agentId=0, amount=-1)
   n += 1
if (cnt >= 10):
   reward(agentId=0, amount=100)  // Success!
else:
   reward(agentId=0, amount=-100)  // Failure.
```

2.2 Environment Synthesis

Environment synthesis determines the program variables and expressions that should be passed to the `choice` function. A given `choice` function might have multiple `reward` functions associated with it. The environment for a given `choice` function consists of the variables and expressions that occur in the backward static slice of any of the associated `reward` functions that are also in scope. Furthermore, if a variable `x` is in the environment, then expressions of the form `x>0` and `x==0` are also added. Performing environment synthesis on Listing 2 gives us:

```
cnt = 0
n = 0
while (cnt < 10 and n < 20):
   r = random(-5, 2)
   if (choice(agentId=0, numChoices=2, env={cnt<10,
                cnt, cnt==0, cnt>0, r, r==0, r>0,
                n<20, n, n==0, n>0}) == 0):
     cnt += r
   reward(agentId=0, amount=-1)
   n += 1
if (cnt >= 10):
   reward(agentId=0, amount=100)  // Success!
else:
   reward(agentId=0, amount=-100)  // Failure.
```

2.3 Safety

A programmer can use an `assert` to ensure that the synthesized `choice` satisfies a safety property. Ensuring safety in RL is not a solved problem [6]. By giving a large negative reward when the assert is violated, the LAL can guide the underlying RL agent away from learning policies that would be unsafe. For example, an assert specifying that `cnt` should not equal 8, `assert (cnt != 8)`, would be translated into `if (cnt == 8): reward(−9999)`. This translated code would then be subject to the reward attribution and environment synthesis algorithms described above.

3 IMPLEMENTATION AND RESULTS

We have implemented LAPy, which is an instantiation of LAL implemented as a deep embedded domain specific language in Python. LAPy uses TensorForce [8], an RL framework supporting a wide range of learning algorithms, including Deep-Q networks.

A preliminary evaluation based on implementing standard RL tasks shows that LAPy is expressive, and results in learning rates comparable to hand-tuned RL implementations.

4 RELATED WORK

There have been a few attempts at integrating reinforcement learning with a programming language [3, 10]. However, we believe the specific interface described in this paper along with the use of Python might lead to greater adoption of LAL. Past work has investigated adaptation via compilation-time algorithmic choices [1, 2, 5].

An approach orthogonal to ours converts a trained RL model into a deterministic, interpretable program expression [13]. The goal of LAL is to provide a flexible and robust interface to RL. Solver-aided languages [12], which perform program synthesis based on purely symbolic reasoning, are complementary to LAL.

5 CONTRIBUTION

The concept of learning-augmented language (LAL) is a robust and flexible abstraction of specifying adaptive behavior in a programming language. LAL frees the programmer from manually specifying heuristics, and avoids bugs caused by the programmer missing out on heuristic relationships as the code evolves. LAL supports flexible choice-reward pairing as well as multi-agent reward attribution. Finally, LAL demonstrates how static analysis can be used for automatic reward attribution and environment synthesis, resulting in an approach for integrating safe reinforcement learning into programming languages.

REFERENCES

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. *PetaBricks: a language and compiler for algorithmic choice*. Vol. 44. ACM.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*. IEEE, 303–315.

[3] Tim Bauer, Martin Erwig, Alan Fern, and Jervis Pinto. 2012. Faster program adaptation through reward attribution inference. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 103–111.

[4] Armin Biere. 2008. Adaptive restart strategies for conflict driven SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 28–33.

[5] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 379–390.

[6] Javier Garcıa and Fernando Fernández. 2015. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research* 16, 1 (2015), 1437–1480.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[8] Michael Schaarschmidt, Alexander Kuhnle, and Kai Fricke. 2017. TensorForce: A TensorFlow library for applied reinforcement learning. Web page. https://github.com/reinforceio/tensorforce

[9] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[10] Christopher Simpkins, Sooraj Bhat, Charles Isbell Jr, and Michael Mateas. 2008. Towards adaptive programming: integrating reinforcement learning into a programming language. *ACM Sigplan Notices* 43, 10 (2008), 603–614.

[11] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.

[12] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 135–152.

[13] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. 2018. Programmatically Interpretable Reinforcement Learning. *arXiv preprint arXiv:1804.02477* (2018).

[14] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.