Environment Dynamics Synthesis

Reintegrating AI Project Final Paper

Jefferson Bernard Cruz Fernandez Thomas Kim
Aaron Kirtland Shane Parr

May 10, 2023

Abstract

This paper presents a novel approach for building reinforcement learning (RL) agents that use program synthesis to learn environment dynamics, a process that is often data-inefficient and slow when relying on repetition. We utilize a domain-specific language (DSL), RLang, along with program synthesis tools, DreamCoder and Stitch, to fill in missing environment dynamics. This method is demonstrated on various Markov environments, with results indicating that program synthesis can handle simple cases efficiently, but struggles with larger or more complex environments unless given almost complete input. We conclude with potential improvements and future research directions, such as integrating a curriculum of training sets for the learning agent and modelling the reward function. This study contributes to AI research by integrating program synthesis, model-learning for RL, and DSLs, with potential for faster generalization from few samples, thereby improving data efficiency.

1 Introduction and Related Work

In this work, we combined a domain specific language (DSL), RLang, with program synthesis for the purpose of building a reinforcement learning (RL) agent that uses program synthesis to learn environment dynamics. Currently, RL agents rely on repetition to learn state-action-state transition functions. These methods tend to be data-inefficient and lead to slow exploration. Using program synthesis, RL agents may be able to complete their transition functions using a more generalized and data-efficient method.

This project combines three distinct AI components: DSLs, program synthesis, and RL.

1.1 DSL: RLang

By enabling commands at a higher level than general purpose languages, DSLs provide more direct, expressive, and efficient means to solve domain-specific problems. In this work, we use RLang [4] to communicate domain knowledge to an RL agent. RLang is very useful for describing RL environments, as it can specify information about every element of a Markov Decision Process (MDP), such as features of the state space, transitions, and actions.

Tools similar to RLang include BabyAI and Video Game Description Language (VGDL). BabyAI [2] provides a DSL for each of its environments, but the DSL is primarily used for goal specification rather than modeling transition dynamics. VGDL [5] can conveniently describe game-like MDPs, namely those that are structured on a grid with certain dynamics like in video games, but is limited in its expressive power compared to RLang.

1.2 Program Synthesis: DreamCoder and Stitch

For program synthesis, we primarily used DreamCoder [3]. DreamCoder is a powerful method for learning new tasks through several phases of hierarchical abstraction. The phases we focus on are the Wake and "Sleep: Abstraction" phases. In the Wake phase, DreamCoder enumerates programs created from primitive and invented programs in a library L that may fit the observed input and output data. In the Sleep: Abstraction phase, the algorithm compresses observed programs found during Wake to create a new library L'. In doing so, DreamCoder is able to learn complex tasks such as manipulating lists and drawing in Logo. However, it is extremely compute and memory hungry, relying on many repetitions to achieve the desired output.

Stitch [1] is a recent program aimed to improve the Sleep: Abstraction phase. It implements a corpus-based top-down synthesis and runs in both Python and Rust, and is faster than the part of DreamCoder it replaces by at least two orders of magnitude.

1.3 Reinforcement Learning

Third, we combine a DSL for MDPs with program synthesis for learning in an RL setting. While many works have used program synthesis to learn policies [7], comparatively fewer works have applied it to learning environment dynamics. In fact, we found just one paper that does this applied to learning VGDL [6]. Unfortunately, because VGDL is limited in the scope of MDPs that it describes (see above), this work does not explore the possibility of learning environment dynamics sufficiently robustly. RLang is built to handle a larger scope of MDPs (see above) and additionally has simpler syntax than VGDL, which we suspect may help program synthesis learn more easily. Lastly, Schneider's work was too slow to be feasible for realistic programs.

2 Environment Dynamics Synthesis

The following steps outline the algorithm we developed in order to fill in missing environment dynamics using RLang and DreamCoder.

Algorithm 1 Environment Dynamics Synthesis

```
1: procedure EnvironmentDynamicsSynthesis(RLang Partial Dynamics Specification)
2:     while RLang model incomplete do
3:         The following two steps can occur in parallel:
4:         Convert RLang to $\lambda$ calculus
5:         Train an agent with the partial RLang model and gather transitions from episodes
6:         Run DreamCoder with $\lambda$ calculus and transition data inputs to produce a filled in transition model.
7:         Convert this transition model back to a partial RLang model.
8:     end while
9: end procedure
```

Our method is visualized in the diagram in Fig. 1.

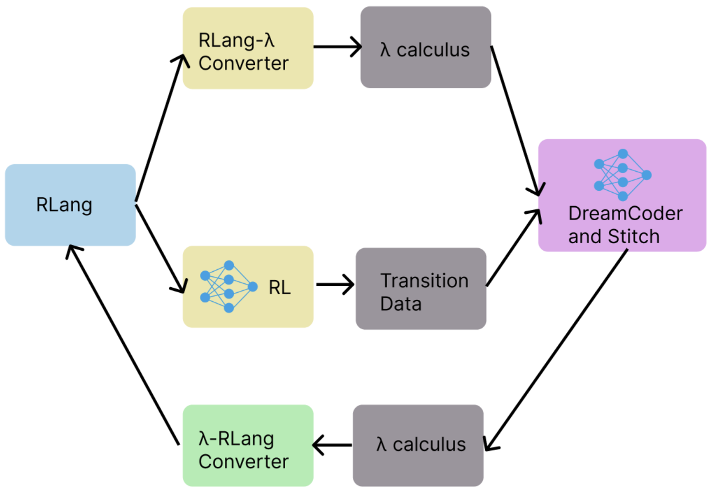

Figure 1: Our method can be visualized as a loop of RLang and lambda calculus, with RL and DreamCoder acting as intermediary components. RLang is fed into two programs in parallel, which produces lambda calculus and transition data with DreamCoder uses to produce a complete lambda calculus program. The lambda calculus is then converted back into RLang for execution.

Next, we will explain each step of our implementation in more detail.

2.1 RLang to λ calculus

The allowable syntax for Stitch is well-defined and we must convert at least the transition function aspects of the RLang syntax to it [1]. This is clearly possible because transitions will form a computable function between $\mathcal{S} \times \mathcal{A} \times S \to [0, 1]$, and lambda calculus can be used to represent any computable function.

We developed a script that takes an RLang file as input and outputs the Effects described in the file (the transition and reward functions) in a lambda calculus format. This allows us to extract transition information from RLang files and input them to Stitch and DreamCoder.

An example translation is shown below: **Example 1:** An RLang file representing a simple Gridworld environment with four actions and their effects described. The λ calculus translation of it using the script we developed is shown below.

```
Constant dim := 3

Factor position := S[0, 1]
Factor ox := position[0]
Factor oy := position[1]

Effect action_effect:
    if a == 0:
        ox -> ox
        oy -> if oy == dim-1:
                oy -> oy
              else:
                  oy -> oy + 1
    elif a == 1:
        ox -> ox
        oy -> if oy == 0:
                oy -> oy
              else:
                  oy -> oy - 1
    elif a == 2:
        ox -> if ox == 0:
                ox -> ox
              else:
                  ox -> ox - 1
        oy -> oy
    elif a == 3:
        ox -> if ox == dim-1:
                ox -> ox
              else:
                  ox -> ox + 1
        oy -> oy
```

λ **Calculus translation**:

```
(if (eq? $0 0) (cons $3 (cons (if (eq? $1 $2) $1  (+ $1 1)) empty))
    (if (eq? $0 1) (cons $3 (cons (if (eq? $1 0) $1  (- $1 1)) empty))
        (if (eq? $0 2) (cons (if (eq? $3 0) $3  (- $3 1)) (cons $1 empty))
            (if (eq? $0 3) (cons (if (eq? $3 $2) $3  (+ $3 1)) (cons $1 empty ())))))
```

2.2 Running agent with partial RLang model

In order to collect transition data for learning a model, we used the Simple RL library. We ran QLearning for 5000 timesteps, collected the (s, a, s') tuples for each observed transition, randomized the order, and passed it to DreamCoder as a program synthesis task.

2.3 DreamCoder

Specifying a DreamCoder domain essential requires two parts, specifying primitives for programs, and specifying tasks.

Each program in a DreamCoder program library is composed of primitives. For the experiments mentioned in this paper, these include "0" , "1", "2", "3", "4" , "5", "6", "7", "8", "9", "-1", "increment", "decrement", "eq?", "if", "emptylist", "cons", "index", "add", and "mod". DreamCoder primitives must be defined in both OCaml and Python, and we chose these primitives because they had preexisting OCaml definitions in the code base.

To specify the task list, we did two things. First, we wrote Python programs to generate input-output pairings that DreamCoder tries to learn. Second, we also specify tasks by reading in input-output transition data from RLang episodes. We input an observed (state, action) pair as a (x-coordinate, y-coordinate, action) tuple of integers, and output an (x-coordinate, y-coordinate) tuple of integers. We then copy the output of DreamCoder into the script for the next stage.

2.4 λ calculus to RLang

We have written a second script to convert λ calculus back to RLang. This is fairly straightforward and simply the reverse of the previous RLang to λ calculus script we mentioned. This allows us to ingest the λ calculus output from DreamCoder/Stitch and translate it back to RLang, therefore completing the loop illustrated in Figure 1.

Example 1: A λ calculus expression describing a simple Gridworld transition function with 4 actions. The output of the λ calculus to RLang script on this input is shown below.

```
(if (eq? $0 0)
    (cons $3 (cons (if (eq? $1 $2) $1  (+ $1 1)) empty))
    (if (eq? $0 1)
    (cons $3 (cons (if (eq? $1 0) $1  (- $1 1)) empty))
    (if (eq? $0 2)
    (cons (if (eq? $3 0) $3  (- $3 1)) (cons $1 empty))
    (if (eq? $0 3) (cons (if (eq? $3 $2) $3  (+ $3 1)) (cons $1 empty)) ())))))
```

RLang translation

```
if param0 == 0:
    [param3, if param1 == $2):
            param1
         else:
    param1 + 1]
else:
    if param0 == 1:
        [param3, if param1 == 0:
                    param1
                 else:
                    param1 - 1]
    else:
        if param0 == 2:
            [if param3 == 0:
                param3
            else:
                param3 - 1, param1]
        else:
            if param0 == 3:
                [if param3 == param2):
                    param3
                else:
                    param3 + 1, param1]
```

2.5 Experimental design

Our primary experiment uses an $n \times n$ grid Block World. However, in total we ran DreamCoder on the following MDP-like environments:

- Hallway with single action that moves agent to the right and wraps around.
- Hallway with single action that moves agent to the left and wraps around.
- Agent must produce $x + y$ given inputs x and y.
- $d \times d$ gridworld that wraps around (on a torus)
- $d \times d$ gridworld that does not wrap around (standard)

We observed that DreamCoder can very quickly synthesize the dynamics for the first three cases, while the latter two cases are considerably harder for it. Additionally, we find that DreamCoder has very similar results whether the observed state transitions were generated by a random program directly

Components	**Project status**	
	Attempted	Completed
RLang-λ Converter	✓	✓
RL running on RLang domain	✓	✓
DreamCoder running on simple inputs	✓	✓
DreamCoder running on RL Data	✓	✓
λ-RLang Converter	✓	✓

Table 1: Overall project status (updated)

by sampling random states and actions or by trajectories from the Simple RL agent.

We achieve good results on spaces with large programs (our gridworld transition program has around 40-50 primitives) by including in the library a nearly complete transition function with one or two holes, or missing primitives. Doing so, we were able to get DreamCoder to synthesize large $d \times d$ gridworlds, while without this hint, DreamCoder could not finish. This suggests that the problem is related to program depth and explosion of possibilities, versus not being able to successfully represent our target functions using the primitives that we implemented at all. This is also evidenced by the fact that DreamCoder was often able to complete environments in which d was relatively small, i.e. able to be expressed with few primitives like the numbers 1 through 10, but with $d \geq 11$, more primitives are required, and it sometimes failed to find solutions.

3 Conclusion

3.1 Future Work

From this project, we have found that program synthesis is generally very difficult, and writing program synthesis libraries is quite a struggle. While we initially naively expected DreamCoder to be able to solve a 2×2 gridworld with little issue, we found that it was unsuccessful at learning the correct transition function until we gave it nearly the complete input. This is because synthesizing the correct solution requires synthesizing a long 40-primitive description-length program, as mentioned above. We anticipate that further experimentation with the DreamCoder hyperparameters may assist it, but the largest improvement would come from having a curriculum of training sets to learn to program. We predict that DreamCoder would do much better if we prepared simple environments requiring it to learn primitives such as "create a list from the two inputs" rather than having to reimplement "cons input1 (cons input2 emptylist)". However, this would take an extensive period of time to run (the original DreamCoder experiments took months), and it would take some effort to create a sufficiently robust set of training tasks. More concretely, the original DreamCoder list task sets had as many as 100 distinct tasks; we predict that the agent in this situation may be able to make further progress in learning the environment dynamics if they were, similarly, exposed to data from many environments simultaneously. This is the future work we would like to explore next by writing a set of many basic environments in RLang.

Second, in this work, we focused only on modelling the transition function, while in the future we would like to model the reward function as well. This is a straightforward change that increases the complexity of the DreamCoder computations, but does not change much fundamentally. Third, we would like to remove the manual copy-pasting between loop steps to have the RLang world model seamlessly update after DreamCoder updates. Fourth, we would like to use Stitch to make the compression phase of DreamCoder faster. Like many parts of DreamCoder, the PyStitch code that enables this cooperation is a prototype version not written with the intention of outside use, and thus editing it requires substantial effort. However, the payoff is a compression library faster by orders of magnitude. Lastly, we eagerly await advances in program synthesis to enable DreamCoder-like algorithms for probabilistic environments.

3.2 Code Availability

Our RLang and λ-calculus converter scripts are available now at `https://github.com/jbernar3/synthesize-env-dynamics`. Our code for the remaining parts involving Simple RL and DreamCoder will be available in the next few days.

3.3 Final Thoughts

In conclusion, we integrated several distinct areas of AI research: program synthesis, model-learning for model-based RL, and domain specific languages. Program synthesis provides the potential for fast generalization from few samples during the expensive data collection phase of reinforcement learning. Reaching high performance in a time and data efficient manner is critical for achieving real-world applicability. Once a learned program represents the environment in simulation, model-free RL has a reliable source and size of samples it can use to solve the problem. Finally, RLang provides a flexible and rigorous language with which to connect the components, being one of the few DSLs which supports partial transition dynamics.

References

[1] Matthew Bowers et al. “Top-down synthesis for library learning”. In: *Proceedings of the ACM on Programming Languages* 7.POPL (2023), pp. 1182–1213.

[2] Maxime Chevalier-Boisvert et al. “Babyai: A platform to study twhe sample efficiency of grounded language learning”. In: *arXiv preprint arXiv:1810.08272* (2018).

[3] Kevin Ellis et al. “Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning”. In: *arXiv preprint arXiv:2006.08381* (2020).

[4] Rafael Rodriguez-Sanchez et al. “RLang: A Declarative Language for Expression Prior Knowledge for Reinforcement Learning”. In: *arXiv preprint arXiv:2208.06448* (2022).

[5] Tom Schaul. “A video game description language for model-based or interactive learning”. In: *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. IEEE. 2013, pp. 1–8.

[6] Martin Franz Schneider. “Program synthesis approaches to improving generalization in reinforcement learning”. PhD thesis. Massachusetts Institute of Technology, 2020.

[7] Yichen Yang et al. “Program synthesis guided reinforcement learning for partially observed environments”. In: *Advances in neural information processing systems* 34 (2021), pp. 29669–29683.