Proceedings of the Nineteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2023)

Learning of Generalizable and Interpretable Knowledge in Grid-Based Reinforcement Learning Environments

Manuel Eberhardinger[1,2], **Johannes Maucher**[1], **Setareh Maghsudi**[2]
[1]Hochschule der Medien Stuttgart, Germany
[2]University of Tübingen, Germany
eberhardinger@hdm-stuttgart.de, maucher@hdm-stuttgart.de, setareh.maghsudi@uni-tuebingen.de

Abstract

Understanding the interactions of agents trained with deep reinforcement learning is crucial for deploying agents in games or the real world. In the former, unreasonable actions confuse players. In the latter, that effect is even more significant, as unexpected behavior cause accidents with potentially grave and long-lasting consequences for the involved individuals. In this work, we propose using program synthesis to imitate reinforcement learning policies after seeing a trajectory of the action sequence. Programs have the advantage that they are inherently interpretable and verifiable for correctness. We adapt the state-of-the-art program synthesis system DreamCoder for learning concepts in grid-based environments, specifically, a navigation task and two miniature versions of Atari games, Space Invaders and Asterix. By inspecting the generated libraries, we can make inferences about the concepts the black-box agent has learned and better understand the agent's behavior. We achieve the same by visualizing the agent's decision-making process for the imitated sequences. We evaluate our approach with different types of program synthesizers based on a search-only method, a neural-guided search, and a language model fine-tuned on code.

Introduction

Humans can easily explain other agents' behavior, living or artificial, using a single demonstration. However, generating explanations post-hoc in (deep) reinforcement learning (RL) after observing an agent's interactions in an environment remains underexplored. Moreover, it is challenging to produce an informative explanation to understand the agent's reasoning for selecting a specific action for a particular state.

Nevertheless, understanding the behavior of artificial agents is crucial for deploying agents in the real world or games. In games, one wants to ensure the agent behaves similarly and avoid confusing the players with unreasonable actions. In real-world scenarios, this is even more important because, for example, in self-driving cars unpredictable actions can cause accidents and lead to serious harm for those involved. Therefore, RL is not yet applicable in real-world scenarios since the behavior of agents trained with RL is not always predictable and, thus, cannot be verified for all edge cases (Dulac-Arnold et al. 2021).

In this work, we propose using program synthesis to imitate RL policies after seeing a trajectory of the action sequence. Program synthesis is the task of finding a program for a given specification, such as a natural language description or input-output examples (Gulwani, Polozov, and Singh 2017). By distilling neural network policies into programmatic policies, we are able to verify the program for correctness and use traditional formal verification tools (Lathouwers and Zaytsev 2022) to analyze the behavior and edge cases of synthesized programs. Another benefit of distilling policies into programs is that software developers can adapt the policy to their own needs, which makes it easy to further improve the programmatic policies or adapt them to other scenarios (Trivedi et al. 2021).

Ideally, we desire to extract programs that can explain decisions and solve the environment; Nevertheless, in this work, we start by dividing complete trajectories into subtrajectories to be able to find programs at all. Therefore, we intend to lay the foundation for a complete policy extraction algorithm.

To accomplish this, we adopt a state-of-the-art program synthesis system DreamCoder (Ellis et al. 2021) for learning concepts in grid-based RL environments and demonstrate that it can extract a library of functions. We collect trajectories of the policy, i.e., state-action pairs, from an oracle trained with RL and use the collected data for DreamCoder to imitate these state-action sequences with programs. We use these programs to extract core concepts from the environment, represented as functions. To enable the system to learn a library, we introduce a domain-agnostic curriculum based on the length of the state-action sequences to imitate. By inspecting the generated library, we can make inferences about the concepts the agent has learned and better understand the agent's behavior. We achieve the same by visualizing the agent's decision-making process for the imitated sequences. We evaluate our approach with three different program synthesizers: a search-only approach, a neural-guided search, and a language model fine-tuned on code.

Our main contributions are as follows:

- Introducing a framework for learning reusable and interpretable knowledge that can reason about agent behavior in grid-based reinforcement learning environments
- An evaluation of the method on a navigation task through a maze and on two simplified Atari game environments,

Asterix and Space Invaders

- A comparison of different program synthesis algorithms, including enumerative search, neural-guided enumerative search, and a fine-tuned language model with and without library learning
- An analysis of extracted functions of the generated libraries
- We open-source the code to enable further research[1].

Related Work

Program Synthesis and Library Learning Program Synthesis has a long history in the artificial intelligence research community (Waldinger and Lee 1969; Manna and Waldinger 1975). In recent years, many researchers have combined deep learning with program synthesis to make program search more feasible by reducing or guiding the search space (Balog et al. 2017; Nye et al. 2019; Ritchie et al. 2016; Chaudhuri et al. 2021). In contrast to the heuristic-based search algorithms, one can also use language models to synthesize programs from text prompts (Li et al. 2022; Austin et al. 2021; Wang et al. 2021; Poesia et al. 2022; Scholak, Schucher, and Bahdanau 2021). Another promising method is learning a library of functions from previously solved problems. These functions are then reusable in an updated domain-specific language to solve more challenging problems (Hewitt, Le, and Tenenbaum 2020; Ellis et al. 2021, 2018; Cao et al. 2023; Bowers et al. 2023).

Explainable Reinforcement Learning There exists a variety of methods in the explainable reinforcement learning (XRL) domain. In a recent comprehensive survey (Qing et al. 2022), the authors divide XRL into four explainable categories: model, reward, state and task. Programmatic policies, where a policy is represented by a program, are part of the model-based explanations (Verma et al. 2019, 2018; Anderson et al. 2020; Trivedi et al. 2021; Qiu and Zhu 2022). Other works in this category synthesize finite state machines to represent policies (Inala et al. 2020) or use models based on decision trees (Silver et al. 2020; Bastani, Pu, and Solar-Lezama 2018). Our method belongs to the same category since we explain sub-trajectories of policies, and our main goal in the future is to extract a program that can represent the full policy.

Background

In this section we give a brief introduction of the different research topics and concepts, that we combine to learn structured and reusable knowledge in grid-based reinforcement learning environments.

Program and Domain-specific Language This work considers programs defined in a typed domain-specific language (DSL) which is based on the Lisp programming language (McCarthy 1960). The primitives, i.e. provided functions and constants, of the DSL are control flows, the actions the agent can use, and also modules to perceive the agent's environment. Since we work with grid environments, the agent's perception consists of modules to determine certain positions on the grid and compare them with the available objects in the environment such as walls, empty cells or game-specific objects. The control flows include if-else statements and Boolean operators to formulate more complex conditions. The DSL is a probabilistic grammar with a uniform distribution over the primitives, i.e., each primitive is assigned the same probability of being used. Listing 1 shows an example program that gets an object on the map and compares it to a wall object. The map is the input parameter x. If there is a wall at the specified position, the left action is chosen, otherwise the forward action. The full DSL is included in Appendix .

Listing 1: An example program generated from the DSL.

```
λ(x) (
  (if (eq-obj? wall-obj (get x 1 0))
      left-action forward-action)
)
```

Program Synthesis One of the core component in this work is the program synthesizer. Our work is based on DreamCoder, a state-of-the-art program synthesis system that combines program synthesis with library learning. DreamCoder provides an implementation of a search algorithm that enumerates programs in decreasing order of their probability of being generated based on a given probabilistic grammar. The programs are checked if they fit a given specification until the top k most likely solutions are found or a timeout is reached (Ellis et al. 2021).

To improve search time, a neural network is trained to predict a distribution over the programs defined by the DSL, thus, adapting the uniform distribution to the one that fits the training data of the neural network. This means that the network predicts the probability of the primitives in the DSL, which results in programs being found faster because they are checked earlier by the enumerative search algorithm (Ellis et al. 2021).

In order to perform a study between different types of program synthesizers, we replace DreamCoder's program synthesis component with a neural program synthesizer. For this, we use CodeT5 (Wang et al. 2021), a finetuned T5 model (Raffel et al. 2020) on multiple programming languages and code related tasks. Raffel et al. (2020) introduced the T5 model with an encoder-decoder architecture and unified different natural language processing (NLP) tasks into a single one by converting them into a text-to-text format. That allows the authors to treat every problem in one way, i.e., using text as input and producing text as output. In this work, CodeT5 is further trained on Lisp programs to synthesize programs in the provided DSL by converting the agent's observation into a text prompt and synthesizing programs in text format as output.

Library Learning The goal of learning libraries is to build a library of specialized concepts in a particular domain that allow programs to be expressed in a concise way. This is similar to software engineers using open source libraries to improve their programming efficiency, since someone else

[1]https://github.com/ManuelEberhardinger/ec-rl

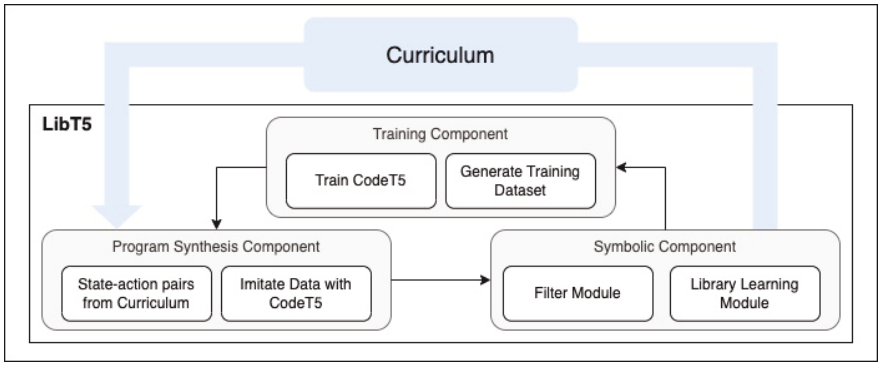


Figure 1: The architecture for creating explanations for a given reinforcement learning environment. The framework can be decomposed into three components that are executed iteratively and are guided by a curriculum. First, we train the CodeT5 model, then we synthesize programs for the provided data from the curriculum. Finally, the synthesized programs are evaluated by the symbolic component for correctness and analyzed to extract a library of functions. We describe the method in detail in Section .

already implemented the needed concepts in a given domain. We use the DreamCoder library learning module to extract functions from solved tasks by analyzing synthesized programs. These programs are refactored to minimize their description length while growing the library. Instead of only extracting syntactic structures from programs, DreamCoder refactors programs to find recurring semantic patterns in the programs (Ellis et al. 2021). In this work, we use library learning to build a library of functions for grid-based RL environments that allow us to make inferences about the knowledge acquired by the black box agent during training.

Imitation Learning To simplify the problem, we limit ourselves to imitation learning, instead of directly finding programs from rewards (Hussein et al. 2017). Although directly finding programs from rewards is an interesting challenge for future research, our main objective is to make the agent's behavior interpretable. We define the problem we want to solve as an imitation of sub-trajectories of state-action pairs collected from a previously trained agent.

Method

Figure 1 shows a high-level overview how we adapted DreamCoder with a neural program synthesizer based on the CodeT5 model (Wang et al. 2021), therefore we call this approach LibT5. The framework consists of three components and a curriculum. In addition, we need an oracle for collecting the data to be imitated. The collected data is provided to the curriculum in the beginning of the training process. For evaluating DreamCoder on the problem of imitating sub-trajectories of state-action sequences, we exchange the LibT5 component with DreamCoder and integrate it into the curriculum. The main differences of LibT5 compared to DreamCoder are the program synthesizer and that DreamCoder first performs an enumerative search before training the neural network and then includes programs found from solved tasks in the training data set. LibT5 is trained only on random programs.

As the oracle is dependent on the domain and not part of the method, we describe the used oracles in Section .

LibT5

LibT5 consists of three components that are executed iteratively. First, we train the CodeT5 model, then we synthesize programs for the provided test data. Finally, the synthesized programs are evaluated by the symbolic component for correctness and analyzed to extract a library of functions.

Training Component The first part is the training process of the CodeT5 model (Wang et al. 2021) with randomly generated programs from the current DSL. The randomly generated programs are executed in a given environment for t steps to collect input-output examples, i.e., sequences of state-action pairs to be imitated, as a training dataset. t is chosen randomly for each program, so we do not overfit on a specific sequence length.

In our setup, we generate 50000 random programs. They are then executed in a randomly selected environment of the provided ones for each domain to collect data to imitate (see Figure 2 for example environments). We execute each program with a random sequence length t between t_{min} and t_{max}. The programs do not have a specific target or reward since they are sampled from the DSL. Our goal in creating a training data set is to exhibit the behavior of programs in a specific RL domain, i.e., how the agent is controlled by given programs in a domain. Random program generation is limited to a maximum depth d_{max} of the abstract syntax tree. We train the model for five epochs in each iteration.

CodeT5 is used without any modifications, as we generate text prompts from the state-action pairs which the model maps to the random programs. The agent's observation, a 2D array of integers, is converted into a string representation, where each integer represents an object in the environment, such as a wall or an enemy. Then the action is appended after the observation. We explain the text prompt generation in Appendix .

Program Synthesis Component The second component converts the data provided from the curriculum for the cur-

Parameters	Navigation Task (5x5)	MinAtar Games (10x10)
t_{min}	5	3
t_{max}	60	20
d_{max}	6	20
P	100	500

Table 1: The parameters for the different domains. MinAtar games have a grid size of 10x10, while the navigation task has a partial observation of size 5x5.

rent sequence length into the text prompt for the model, and then the model synthesizes P programs for the state-action sequences to be imitated. These programs are then passed to the symbolic part of the framework.

Symbolic Component In this component, the filter module evaluates P programs for syntactic and functional correctness in the provided DSL that imitates the state-action sequence.

The library learning module uses the correct programs to generate a library by extracting functions from the found programs and adding them to the current DSL. It extracts functions only if a part in a program occurs multiple times in other synthesized programs on the oracle data. That way, the extracted functions are beneficial for the DSL since they have been synthesized several times for different state-action sequences.

Curriculum

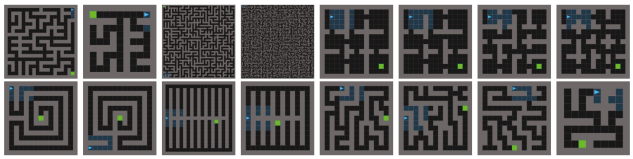
The curriculum is based on the action sequence length and, therefore domain-agnostic. We start with an initial sequence length of three, and at each iteration, after the symbolic component has completed the library learning phase, we check whether the sequence length should be increased. We always sample new random programs from the DSL and run them in the environment as the library is updated each iteration to represent more diverse programs. We increment the action sequence length if at least 10% of the oracle's data is imitated and stop the training process if the action sequence length has not been incremented twice in a row.

This curriculum strategy is based on the assumption that longer sequence lengths are more complex than shorter ones. Programs that need to imitate three actions do not need to represent as much information as programs that imitate five actions; Thus, the program length is shorter. Shorter programs are easier to synthesize compared to long ones because of the smaller search space. Ellis et al. (2021) showed that building up a library of complex functions, enables DreamCoder to synthesize programs for more difficult tasks.

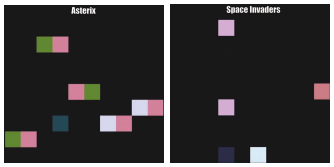
Table 1 shows the different parameters for the evaluated domains, as they depend on the complexity and observation space of the environment.

Experiments

In this section we evaluate the different program synthesis systems on the problem of imitating sub-trajectories on two different domains. We first introduce both domains. Then, we evaluate the conducted experiments, followed by an introduction of the method for generating explanations from



(a) Gridworld: The different environments used for generating the training data. The evaluation is always performed on the medium sized perfect maze environment which is placed on the top left.



(b) MinAtar: The two different miniaturized versions of the Atari 2600 games, Asterix and Space Invaders. In all environments the agent is displayed in dark blue. The other colors are game-specific objects, such as gold coins, enemies and bullets in Asterix or Space Invaders (from (Young and Tian 2019)).

Figure 2: The environments for the Gridworld and the MinAtar domain used for the training data generation.

programs. Finally, we perform a thorough analysis of the extracted functions in the library.

Domains

Gridworld The first domain we evaluate on is a navigation task from the grid-world domain (Chevalier-Boisvert, Willems, and Pal 2018). We trained the agent with the default hyperparameters from Parker-Holder et al. (2022) and then collected state-action pairs from the medium-sized perfect grid environment displayed on the top left in Figure 2.

MinAtar Young and Tian (2019) introduced a miniature version for five games of the Arcade Learning Environment (Bellemare et al. 2013). The games are simplified to enable more efficient experimentation. MinAtar converted the game into a symbolic 10x10 grid representation with the same game dynamics. Each environment provides a one-hot encoded 10x10xn observation, where n channels correspond to specific game objects such as cannon, enemy or bullet objects in Space Invaders. For our experiments we converted the state into a single 10x10 grid. We evaluate our method on Asterix and Space Invaders. We trained both agents with the default parameters provided from Young and Tian (2019).

To generate diverse training data that starts not always from similar game states, we let the oracle play the episode for a random time before executing the program. This ensures that the training data set is capturing different aspects of the policy.

Evaluation

We evaluate the problem of imitating sub-trajectories for RL environments with four different program synthesizers:

- Search: Program synthesis with an enumerative search algorithm. We use the implementation from Ellis et al.

(2021) and the same DSL as DreamCoder to show the benefits of the neural-guided search.

- DreamCoder: A neural-guided search algorithm with a library learning module (Ellis et al. 2021).
- CodeT5: A language model fine-tuned on Lisp programs on our data (Wang et al. 2021).
- LibT5: The CodeT5 model combined with DreamCoder's library learning module.

For the final evaluation, we use data collected from the same agent but on different runs to ensure that we do not evaluate and train on the same data. The performance is measured by

$$Accuracy = \frac{1}{N} \sum_{\tau \in D} f(\mathrm{P}, \tau),$$

$$f(\mathrm{P}, \tau) = \begin{cases} 1, & \text{if } \sum_{\rho \in \mathrm{P}} g(\rho, \tau) > 0 \\ 0, & \text{otherwise} \end{cases}$$

$$g(\rho, \tau) = \underbrace{\mathbb{1}\{\ \mathrm{EXEC}(\rho, s) == a,\ \forall (s, a) \in \tau\ \}}_{\text{is 0 after the first } (\rho,s) \text{ where } \mathrm{EXEC}(\rho,s)\ != a}$$

where N is the size of the dataset D to imitate, τ is a sub-trajectory from D that consists of state-action pairs (s, a), and P is the set of all synthesized programs from a given method. $f(\mathrm{P}, \tau)$ checks if there exists any program ρ out of all synthesized programs P that is correct. $g(\rho, \tau)$ evaluates if a given program ρ can imitate the full rollout τ and returns 1 if this is the case and otherwise 0. $\mathrm{EXEC}(\rho, s)$ executes the program on a given state s and returns an action a. The identity function $\mathbb{1}$ maps Boolean values to 0 and 1.

Since we have formulated the problem as an imitation of sub-trajectories, we cannot use a more appropriate metric for evaluation. In RL, there are often many different trajectories to achieve a goal in an environment, but in our case we need to evaluate our framework using a more rigorous metric until a complete policy can be distilled into a program with our method.

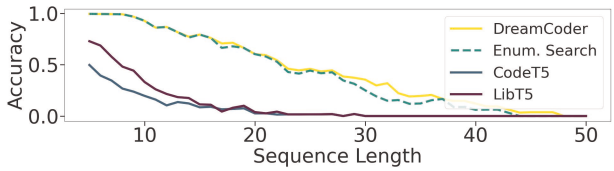
Fairness of evaluation Considering fundamental differences, a fair comparison of the used algorithms can be challenging. We describe the used hardware resources and the main distinctions of the experimental setup in more detail in Appendix .

Results Figure 3 shows the final evaluation of newly collected data in the same environment used to extract functions from found programs on the solved test tasks. Depending on the domain, different program synthesis methods are well suited. Figure 3a shows the evaluation for the maze environment with the smallest observation space of 5x5. The search-based methods can solve sub-trajectories almost twice as long as the neural-based models. For Asterix all methods show similar performance (Fig. 3b). For Space Invaders, LibT5 performs worse compared to the other methods (Fig. 3c).

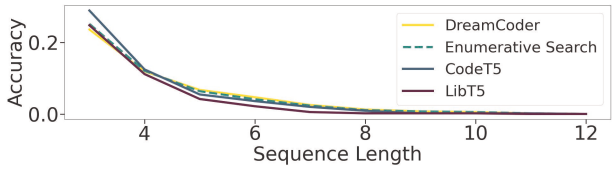
Figure 3a shows that library learning can be useful for neural program synthesizers, but also detrimental depending on the environment. For both MinAtar environments, LibT5 is not as good as CodeT5. This suggests that the more diverse DSL can lead to the problem of "catastrophic forgetting" (Kirkpatrick et al. 2017), and previously solved programs become unsolvable. In addition, longer action sequences are no longer solved as well. Our hypothesis is that the library is not beneficial, although more functions have been extracted from LibT5 compared to DreamCoder (see Table 2). Inspecting programs synthesized with LibT5 for Space Invaders shows that they are too complicated compared to programs synthesized with CodeT5. The reason is that LibT5 uses the extracted functions of the library, even if the task is easier to solve with the initial primitives.

From both observations, we conclude that neural program synthesizers may be useful for larger observation spaces. Catastrophic forgetting could be mitigated by adjusting the probabilities of the functions in the probabilistic grammar according to their usefulness. By lowering the probability of the more complex functions, the grammar will produce simpler programs. In addition, we need to improve the generation of training data by collecting different runs for each program or trying different representations for encoding the state-action pairs. Therefore, further research is imperative to better integrate the library learning module into the framework with neural program synthesizers.

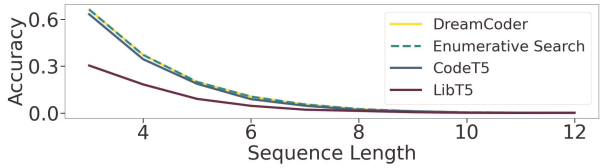
It is also evident from Figure 3 that a system without a curriculum cannot imitate complete action sequences, as it can currently imitate up to sequence lengths of 50. In comparison, complete trajectories are up to 1000 steps long depending on the environment.



(a) The perfect maze environment.



(b) The Asterix environment.



(c) The Space Invaders environment.

Figure 3: The evaluation of the different methods on the three environments. We evaluated the percentage of the correct imitated sub-trajectories for an increasing sequence length until no more programs were found.

Inspecting the Program Library

In this section, we analyze the libraries extracted from the evaluated methods. Appendix includes the full libraries. Table 2 shows the number of extracted functions for the different program synthesis methods. LibT5 extracts for all environments the most functions.

Figure 5 shows the discovered library of DreamCoder for the maze environment with a deep hierarchical structure. `f10` is using seven previous discovered functions. Figure 6 shows the library of LibT5. In contrast to DreamCoder, more functions were extracted, but more often semantic similar functions are found and the deep hierarchical structure is missing. The use of a language model for program search in combination with library learning raises a new problem similar to one previously addressed in Inductive Logic Programming. Cropper (2020) analyzed what the perfect library size is and how to forget unnecessary programs in the library. This is also necessary in our case, as we assume that LibT5 synthesizes many programs that are semantically the same but differ syntactically. Therefore, the library learning module extracts many similar functions and adds them to the library. A similar problem is also observable in the AlphaCode system, which clusters synthesized programs before selecting solutions for submission to programming competitions (Li et al. 2022). From this we conclude, that a larger library is not always beneficial for the program synthesizer.

Visualization of the Decision Making Process

Since programs are inherently interpretable, we developed a method to visualize the agent's decision-making process by highlighting those grid positions responsible for choosing a particular action. Since one position is not always sufficient to select the correct action, we create step-by-step explanations of the "reasoning process" by traversing the program call graph (Ryder 1979) and logging all function calls and their parameters.

Listing 2 shows a program synthesized by DreamCoder. The data was collected from the maze environment for a sub-trajectory of 24 state-action pairs. We show the full implementation of the program, where all implementations of the used functions are shown. The first λ denotes the start of the program and receives two input parameters, the map and the direction of the agent. All subsequent λ represent discovered functions from the library, followed by the input parameters. In Listing 3, we show the program when we use the function `f5` from the library. This shows the effectiveness of using library learning in combination with program synthesis.

Figure 4 visualizes three examples of the reasoning process by highlighting the responsible grid cells in yellow. The agent's position is in blue, which is the same in all visualizations because the partial observation of the agent is aligned to the same position by the environment. The direction above the images indicates in which direction the agent is looking on the full map, since the surrounding area is only partially visible. The walls are gray, and the path through the maze is black. The program first checks if the direction is `direction-3` and returns an `goal-object` or `empty-object`. Since for all three visualizations the direction is one or zero, the `empty-obj` is always returned. Then the `empty-obj` is compared with the cell on position (x=3,y=0). The result is an input parameter for `f4` and `f3`. The coordinates (3,0) are also input parameters of `f5`. Then `f4` is called which gets the object on position (1, 2). This object is then compared with an empty object in `f3` and it is checked whether the other input parameter of `f5` is also true. Depending on this result, the object at position (0,1) or a `wall-obj` is returned. The returned object is then compared to the position (1,2) and finally the agent decides whether to use the left or the forward action.

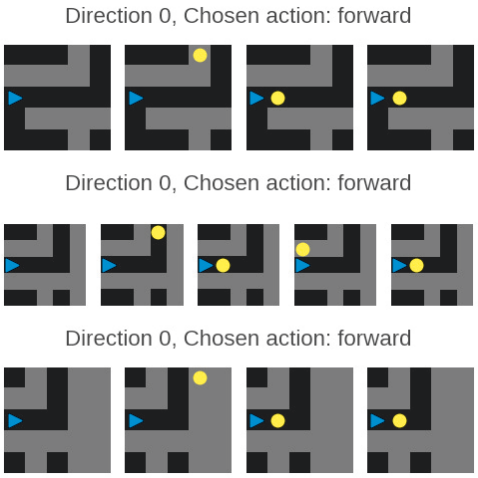


Figure 4: The decision-making process, when executing the program on the state-action sequence. We show explanations for three of 24 states of a given sub-trajectory. The grid positions that are checked in the program are yellow. The agent's position is marked blue and faces to the right. Grey and black indicate walls and empty cells, respectively. The forward action moves the agent one grid cell to the right. The left action only turns the agent 90° in the left direction but does not move it. The cell on position (2,1) is checked multiple times, as at first it is checked in `f4` and then later in `f1`. We give a detailed explanation of the program in Section .

Environment	DreamCoder	LibT5
Maze	17	27
Asterix	4	8
Space Invaders	15	23

Table 2: Number of extracted functions for different program synthesis methods.

Currently, the explanations of the policy can be incorrect, as we do not have a complete policy extraction algorithm and only imitate sub-trajectories collected from an oracle. Without imitating the complete trajectories, the created explanations can be wrong; As such, the programs found for longer action sequences are more reliable as they explain more of the policy.

Listing 2: The program for a given sub-trajectory synthesized by DreamCoder.

```
(λ (x y) (
  (λ (z u v w) ( // start f5
    (λ (a b) ( // start f4
      (λ (c d e) ( // start f3
        (λ (f g) ( // start f2
          (λ (h i) ( // start f1
            (λ (j k) (if (eq-obj? k j) left-action forward-action)) // f0
            h (get i 2 1))) // end f1
          (if g (get-game-obj (get f 1 0)) wall-obj))) // end f2
        c (and (eq-obj? e empty-obj) d) c)) // end f3
    a b (get a 2 1))) // end f4
  z (eq-obj? (get z u w) v))) // end f5
x 0 (if (eq-direction? y direction-3) goal-obj empty-obj) 3))
```

Listing 3: The same program as in Listing 2 when `f5` is called from the library.

```
(λ (x y) (f5 x 0 (if (eq-direction? y
    direction-3) goal-obj empty-obj) 3))
```

Discussion & Limitations

In our experiments, we have demonstrated that DreamCoder and LibT5 are able to learn a library of functions for a navigation task and two game environments with a discrete state and action space. By traversing the program call graph of synthesized programs, we created visual explanations for the agent's decision-making process. We concluded our experiments with an analysis of the generated libraries for the given domains and discussed the implications.

While learning a library of concepts showed promising results for grid-based environments with a small observation space, we need to further improve our framework for medium-sized and large observation spaces. We have shown that it is possible for the MinAtar environments to learn a library and imitate short sequences using the CodeT5 model, but the library was not used effectively and the synthesized programs were too complicated compared to the data to be imitated. Therefore, we need to further investigate how the library learning module can benefit neural program synthesizers without compromising its ability to imitate shorter state-action sequences. If this is possible, this opens up other interesting domains, such as AlphaGo (Silver et al. 2016), where humans struggle to comprehend the reasoning process of strategies discovered by artificial agents through self-play.

Additionally, for these environments, it is straightforward to define the functional primitives for the agent's perceptions and actions. However, that becomes challenging for continuous- state and action spaces, or when an image represents the state. For images, we could use an object detection model which parses the images before generating text prompts for the program synthesizer, similar to (Yi et al. 2018), where an image is parsed into a structural representation that is then used in a program. For continuous representations, further research is imperative to verify the effectiveness of this method.

Conclusion and Future Work

In this paper, we adapted the DreamCoder system to learn structured and reusable knowledge in grid-based reinforcement learning environments that allows reasoning about the behavior of black-box agents. We further evaluated the use of a neural network as a program synthesizer and discussed the positive and negative aspects of both methods. The main disadvantage of the proposed framework is its dependence on an oracle for collecting trajectories, whereas it does not depend on much background knowledge except for the initial primitives in the DSL.

This work opens many possibilities for future work. The main focus is a policy extraction algorithm that can imitate the entire state-action sequences and not only parts of them. Additionally, we want to evaluate our method on continuous or image-based domains to validate that it is domain-agnostic.

Domain-specific Language

Table 3 shows the initial domain-specific language, which contains only the primitives necessary to get different cells on the grid, the control flow structures and Boolean operators. Since we use a typed DSL, we show the types for each function or value. If our primitive is a value, only one type appears in the type column. For functions, multiple types are combined with an arrow $\rightarrow$. The last type represents the return value of the function. The types before it are the types of the input parameters. The type `func` represents a function because if-clauses returns a new function to execute depending on the condition since partial programs are also functions in Lisp.

To generate random programs, we can specify the types of program to be generated. In our case, we always want programs of type `map` $\rightarrow$ `direction` $\rightarrow$ `action` or `map` $\rightarrow$ `action`, so a random program is always defined from two input parameters of type `map` and `direction` or one input parameter `map` and returns a value of type `action`. We restrict the input parameter types of the `eq-obj?` function to `mapObject` and `object` so that sampling programs from the grammar always results in the comparison of at least one object from the map.

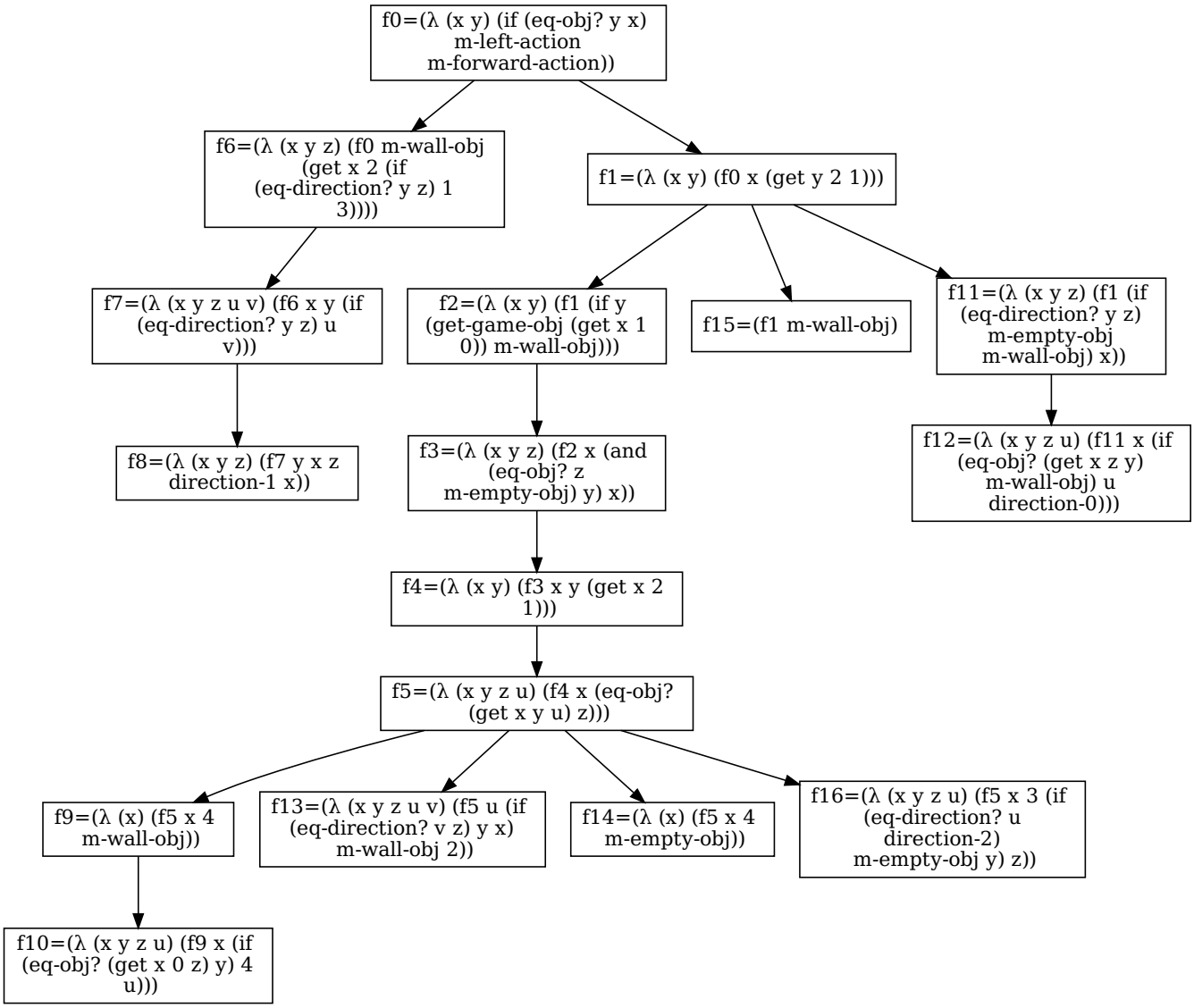


Figure 5: Maze: The extracted functions from programs found by using DreamCoder. The function `f10` is using seven previous discovered functions.

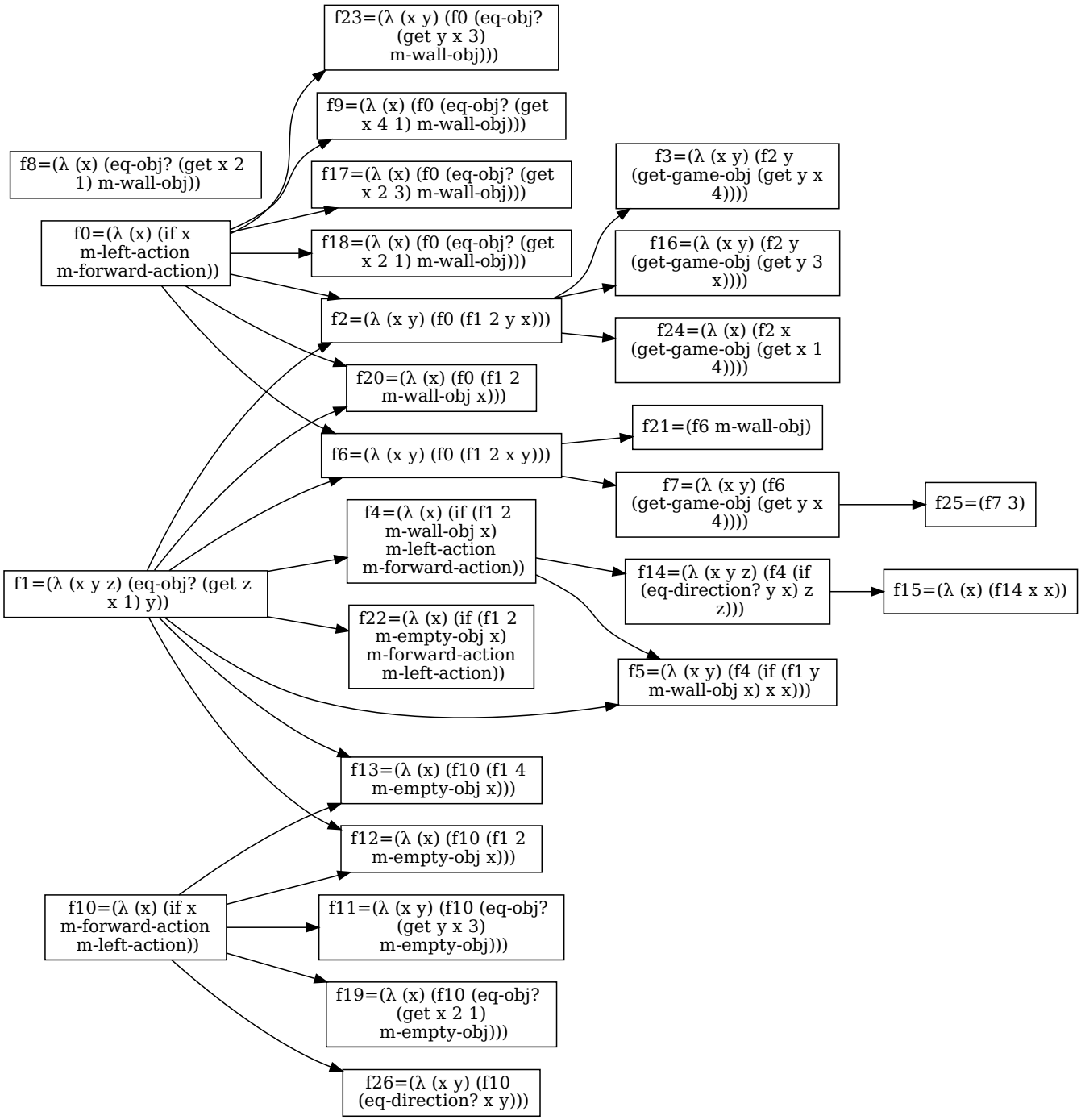


Figure 6: Maze: The extracted functions from programs found by using LibT5.

Primitive Function/Values	Description	Type	Note
left, right, forward, down, fire, no-op	possible actions	action	
0, 1, ..., 9	integer values	int	maze: 0,..,5
2D array	2D grid observation of the agent	map	
direction-0, direction-1 direction-2, direction-3	represents either north, east, south and west	direction	only for maze
depends on the environment	possible objects on the map	object	
depends on the environment	a object on a map with (x, y)-coordinates	mapObject	
if	standard if-clause	bool → func → func → func	
eq-direction?	checks if two directions are equal	direction → direction → bool	only for maze
eq-obj?	checks if two objects are equal	mapObject → object → bool	
get	get a object on the map for two coordinates	map → int → int → mapObject	
get-game-obj	get the object type of a mapObject	mapObject → object	
not	negates a Boolean value	bool → bool	
and	conjunction of two Boolean values	bool → bool → bool	
or	disjunction of two Boolean values	bool → bool → bool	
get-x	get the X coordinate of a mapObject	mapObject → tx	
get-y	get the Y coordinate of a mapObject	mapObject → ty	
eq-x?	checks if two X coordinates are the same	tx → tx → bool	
eq-y?	checks if two Y coordinates are the same	ty → ty → bool	
gt-x?	checks if the first X is greater than the second	tx → tx → bool	
gt-y?	checks if the first Y is greater than the second	ty → ty → bool	

Table 3: The used domain-specific language at the beginning. The type column shows one type for values and several types separated by an arrow for functions. The type after the last arrow is the return type of the function. The types before it are the types of the input parameters.

Text Prompts

Text prompts are generated by converting the agent's observation into a string representation and then concatenating the string representation with the action. This is repeated for all state-action pairs until each pair in the sequence is represented as a string. Then all strings are combined into a single text prompt.

Figure 7 shows an example of a state-action sequence of length five. On the right of the first line is the 2D array, followed by the string representation and the selected action. On the left is the corresponding maze represented by the 2D array. The final text prompt is then generated by iteratively concatenating all the string representations and actions for the entire sequence. The final text prompt for the state-action sequence is:

```
22222222221222212222122220 left
↪  12222222221222222222222223 left
11121121221211122222222222 left
↪  22222222221111122122111211 forward
22222222221111121222112111 forward
```

The 1 represents empty grid cells and the 2 represents wall objects on the map.

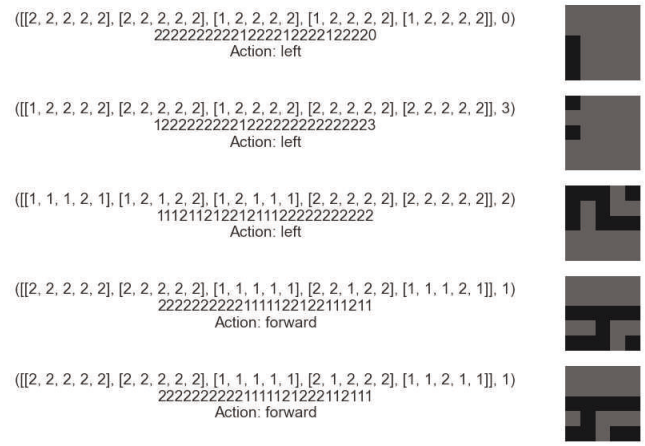


Figure 7: Text prompts are created by converting the 2D array into a string were all values are concatenated without spaces.

Experimental Setup & Hardware Resources

Table 4 shows the hyperparameters for the different methods. We use the same hyperparameters for each iteration. We adopted the hyperparameters from the DreamCoder system to our problem and tried out a few timeouts; since our system is executed iteratively, the runtimes of the experiments are very long, so an extensive hyperparameter search was not possible. For the neural-guided search, we train an encoder-decoder neural network. The encoder for the state obser-

Hyperparameters	Enum. Search	DreamCoder	CodeT5	LibT5
Search Timeout	720 seconds	720 seconds	-	-
Training time	-	5000 steps	5 epochs	5 epochs
Training Programs	-	5000	50000	50000
Arity	3	3	-	3

Table 4: The hyperparameters for the different methods.

vations of the maze was adapted from Parker-Holder et al. (2022). For the MinAtar environments we adapted the encoder from Young and Tian (2019). The decoder, which predicts which functional primitives are used in the program to be synthesized, is generated automatically from the DreamCoder system (Ellis et al. 2021). We train the neural-guided search model for 5000 update steps. Ellis et al. (2021) explain that DreamCoder does not require a large data set because it is based on a search method that is already defined by the DSL. The neural network only improves the search. CodeT5, on the other hand, learns everything from data and therefore requires many more programs. The library learning module is restricted to an arity of three, which means that extracted functions can have up to three input parameters.

Extracted Libraries

In this section, we briefly discuss the full libraries created by DreamCoder and LibT5 for the two other evaluated environments. The libraries can be accessed online[2]. For the Asterix environment both methods could not extract that many functions, we think that this shows the difficulty of the game compared to the maze environment. For Space Invaders, LibT5's library shows a deeper hierarchical structure compared to DreamCoder's extracted functions, but the evaluation has shown that a larger library is not always useful for the program synthesizer, especially for neural program synthesis.

Acknowledgements

The work of S. M. was supported by the German Research Foundation under Grant MA 7111/7-1.

References

Anderson, G.; Verma, A.; Dillig, I.; and Chaudhuri, S. 2020. Neurosymbolic Reinforcement Learning with Formally Verified Exploration. In *Advances in Neural Information Processing Systems*, volume 33, 6172–6183. Curran Associates, Inc.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; and Sutton, C. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732.

Balog, M.; Gaunt, A. L.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2017. DeepCoder: Learning to Write Programs. In *International Conference on Learning Representations*.

Bastani, O.; Pu, Y.; and Solar-Lezama, A. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47: 253–279.

Bowers, M.; Olausson, T. X.; Wong, L.; Grand, G.; Tenenbaum, J. B.; Ellis, K.; and Solar-Lezama, A. 2023. Top-Down Synthesis for Library Learning. *Proceedings of the ACM on Programming Languages*, 7(POPL): 1182–1213.

Cao, D.; Kunkel, R.; Nandi, C.; Willsey, M.; Tatlock, Z.; and Polikarpova, N. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-Unification. *Proceedings of the ACM on Programming Languages*, 7(POPL): 396–424.

Chaudhuri, S.; Ellis, K.; Polozov, O.; Singh, R.; Solar-Lezama, A.; and Yue, Y. 2021. Neurosymbolic Programming. *Foundations and Trends® in Programming Languages*, 7(3): 158–243. Publisher: Now Publishers, Inc.

Chevalier-Boisvert, M.; Willems, L.; and Pal, S. 2018. Minimalistic Gridworld Environment for Gymnasium. https://github.com/Farama-Foundation/Minigrid. Accessed: 2023-07-25.

Cropper, A. 2020. Forgetting to Learn Logic Programs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04): 3676–3683. Number: 04.

Dulac-Arnold, G.; Levine, N.; Mankowitz, D. J.; Li, J.; Paduraru, C.; Gowal, S.; and Hester, T. 2021. Challenges of Real-World Reinforcement Learning: Definitions, Benchmarks and Analysis. *Machine Learning*, 110(9): 2419–2468.

Ellis, K.; Morales, L.; Sablé-Meyer, M.; Solar-Lezama, A.; and Tenenbaum, J. 2018. Learning Libraries of Subroutines for Neurally– Guided Bayesian Program Induction. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Ellis, K.; Wong, C.; Nye, M.; Sablé-Meyer, M.; Morales, L.; Hewitt, L.; Cary, L.; Solar-Lezama, A.; and Tenenbaum, J. B. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, 835–850. Association for Computing Machinery.

Gulwani, S.; Polozov, A.; and Singh, R. 2017. *Program Synthesis*, volume 4. NOW. Publication Title: Foundations and Trends in Programming Languages.

Hewitt, L.; Le, T. A.; and Tenenbaum, J. 2020. Learning to learn generative programs with Memoised Wake-Sleep. In *Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence (UAI)*, 1278–1287. PMLR.

Hussein, A.; Gaber, M. M.; Elyan, E.; and Jayne, C. 2017. Imitation Learning: A Survey of Learning Methods. *ACM Comput. Surv.*, 50(2).

Inala, J. P.; Bastani, O.; Tavares, Z.; and Solar-Lezama, A. 2020. Synthesizing Programmatic Policies that Inductively Generalize. In *International Conference on Learning Representations*.

[2]https://github.com/ManuelEberhardinger/ec-rl/extracted-libs/

Kirkpatrick, J.; Pascanu, R.; Rabinowitz, N.; Veness, J.; Desjardins, G.; Rusu, A. A.; Milan, K.; Quan, J.; Ramalho, T.; Grabska-Barwinska, A.; Hassabis, D.; Clopath, C.; Kumaran, D.; and Hadsell, R. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13).

Lathouwers, S.; and Zaytsev, V. 2022. Modelling Program Verification Tools for Software Engineers. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, 98–108. Association for Computing Machinery.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A. D.; Hubert, T.; Choy, P.; d'Autume, C. d. M.; Babuschkin, I.; Chen, X.; Huang, P.-S.; Welbl, J.; Gowal, S.; Cherepanov, A.; Molloy, J.; Mankowitz, D. J.; Robson, E. S.; Kohli, P.; Freitas, N. d.; Kavukcuoglu, K.; and Vinyals, O. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624): 1092–1097.

Manna, Z.; and Waldinger, R. 1975. Knowledge and Reasoning in Program Synthesis. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'75, 288–295. Morgan Kaufmann Publishers Inc.

McCarthy, J. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM*, 3(4): 184–195.

Nye, M.; Hewitt, L.; Tenenbaum, J.; and Solar-Lezama, A. 2019. Learning to Infer Program Sketches. In *Proceedings of the 36th International Conference on Machine Learning*, 4861–4870. PMLR.

Parker-Holder, J.; Jiang, M.; Dennis, M.; Samvelyan, M.; Foerster, J.; Grefenstette, E.; and Rocktäschel, T. 2022. Evolving Curricula with Regret-Based Environment Design. In *Proceedings of the 39th International Conference on Machine Learning*, 17473–17498. PMLR.

Poesia, G.; Polozov, A.; Le, V.; Tiwari, A.; Soares, G.; Meek, C.; and Gulwani, S. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *International Conference on Learning Representations*.

Qing, Y.; Liu, S.; Song, J.; Wang, H.; and Song, M. 2022. A Survey on Explainable Reinforcement Learning: Concepts, Algorithms, Challenges. arXiv:2211.06665.

Qiu, W.; and Zhu, H. 2022. Programmatic Reinforcement Learning without Oracles. In *International Conference on Learning Representations*.

Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; and Liu, P. J. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140): 1–67.

Ritchie, D.; Thomas, A.; Hanrahan, P.; and Goodman, N. D. 2016. Neurally-guided procedural models: amortized inference for procedural graphics programs using neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, 622–630. Curran Associates Inc.

Ryder, B. 1979. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, SE-5(3): 216–226.

Scholak, T.; Schucher, N.; and Bahdanau, D. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587): 484–489.

Silver, T.; Allen, K. R.; Lew, A. K.; Pack Kaelbling, L.; and Tenenbaum, J. 2020. Few-Shot Bayesian Imitation Learning with Logical Program Policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34.

Trivedi, D.; Zhang, J.; Sun, S.-H.; and Lim, J. J. 2021. Learning to Synthesize Programs as Interpretable and Generalizable Policies. In *Advances in Neural Information Processing Systems*, volume 34.

Verma, A.; Le, H.; Yue, Y.; and Chaudhuri, S. 2019. Imitation-Projected Programmatic Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Verma, A.; Murali, V.; Singh, R.; Kohli, P.; and Chaudhuri, S. 2018. Programmatically Interpretable Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning*, 5052–5061. PMLR.

Waldinger, R. J.; and Lee, R. C. T. 1969. PROW: A Step Toward Automatic Program Writing. In *International Joint Conference on Artificial Intelligence*.

Wang, Y.; Wang, W.; Joty, S.; and Hoi, S. C. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Yi, K.; Wu, J.; Gan, C.; Torralba, A.; Kohli, P.; and Tenenbaum, J. B. 2018. Neural-Symbolic VQA: Disentangling Reasoning from Vision and Language Understanding. In *Neural Information Processing Systems*.

Young, K.; and Tian, T. 2019. MinAtar: An Atari-Inspired Testbed for Thorough and Reproducible Reinforcement Learning Experiments. arXiv:1903.03176.