Towards Explainable Decision Making with Neural Program Synthesis and Library Learning

Manuel Eberhardinger[1,2,*], Johannes Maucher[1] and Setareh Maghsudi[2]

[1]Hochschule der Medien Stuttgart, Germany
[2]University of Tuebingen, Germany

Abstract

Generating explanations for the behavior of artificial or living agents is still an underexplored problem. We propose a new framework to make agent behavior explainable based on program synthesis. Programs have the advantage that they are inherently interpretable and can be verified for correctness. In detail, we use neural program synthesis with a language model in combination with library learning. In addition, we introduce a domain-agnostic curriculum that is necessary for the system to learn at all. We compare our methods with traditional and state-of-the-art program synthesis systems and justify the necessity of the curriculum and library learning module in ablation studies, followed by an analysis of the extracted libraries.

Keywords

Neural Program Synthesis, Library Learning, Imitation Learning, Explainable Reinforcement Learning

1. Introduction

Humans can easily explain other agents' behavior, living or artificial, after observing a single demonstration. However, generating explanations post-hoc in reinforcement learning (RL) after seeing an agent interact in an environment is still an underexplored problem. Moreover, it is unclear how to produce an informative explanation that helps to understand the agent's reasoning for selecting a specific action for a particular state.

In this work, we propose using program synthesis to create post-hoc explanations for the agent's behavior after seeing a trajectory of the action sequence. We argue that programs are inherently interpretable and therefore are a good choice for generating explanations of the agents' decision making process. This is possible by constructing a program call graph, which can be traversed to follow the "reasoning process" of the agent. Furthermore, we propose a new neural program synthesis framework for RL. This new framework combines code generation with a language model and library learning guided by a curriculum. Programs are generated by imitating state-action pairs, which serve as input-output examples for the inductive program synthesis task. Library learning extracts functions from programs synthesized from previously solved tasks. The use of library learning enables a more detailed analysis of the extracted

NeSy 2023, 17th International Workshop on Neural-Symbolic Learning and Reasoning, Certosa di Pontignano, Siena, Italy

*Corresponding author.
eberhardinger@hdm-stuttgart.de (M. Eberhardinger); maucher@hdm-stuttgart.de (J. Maucher); setareh.maghsudi@uni-tuebingen.de (S. Maghsudi)

functions and thus a better understanding of the agents' behavior. The curriculum is domain-agnostic, and works in theory on all environments, as the curriculum is based on the action sequence lengths to imitate. This curriculum is necessary for the system to build up the functions in the library, because without a curriculum the system is not able to solve tasks at all.

To the best of our knowledge, this is the first work that proposes to combine neural program synthesis with library learning for reasoning about the decision making process in reinforcement learning environments. Our contributions include:

- Introducing a framework for learning reusable and interpretable knowledge that can reason about agent behavior in reinforcement learning environments
- A case study on data collected from a partial observable maze environment (see the environment on the top left in Figure 2)
- 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

2. Related Work

Program Synthesis and Library Learning Program synthesis has a long history in the artificial intelligence research community [1, 2]. 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 [3, 4, 5, 6]. In contrast to the heuristic-based search algorithms, one can also use language models to synthesize programs from text prompts [7, 8, 9, 10, 11]. 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 [12, 13, 14, 15, 16].

Explainable Reinforcement Learning There exists a variety of methods in the explainable reinforcement learning (XRL) domain. In a recent comprehensive survey [17], 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 [18, 19, 20, 21, 22]. Other works in this category synthesize finite state machines to represent policies [23] or use models based on decision trees [24, 25]. 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.

3. Background

Our work is based on several research topics, which we combine to learn structured and reusable knowledge in grid-based reinforcement learning environments. In this section, we provide a brief introduction of those topics.

Neural Program Synthesis One of the core component in this work is the neural program synthesizer. We use CodeT5, a finetuned T5 model[26] on multiple programming languages and code related tasks. In [26], Raffel et al. 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 domain-specific language 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. That is similar to software engineers using open source libraries to improve their programming efficiency, since someone else implemented the needed concepts in a specific domain. We use the DreamCoder [13] 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 [13].

ACCEL ACCEL is a training framework for RL agents that improves their generalization capabilities using Unsupervised Environment Design (UED) [27, 28]. UED generates a distribution of valid environments at the frontier of the agents capabilities by providing a curriculum for the agent. Dennis et al. uses an additional agent as the environment designer to increase the difficulty of the environment after each episode [27]. ACCEL improves this method by using an evolutionary approach to adapt the current environments [28]. In contrast to UED our curriculum depends not on another agent or method.

4. Problem Formulation

Our goal is to make the agents' decision making process interpretable by imitating trajectories collected from black-box neural network policies with programmatic policies. Ultimately, our main goal is to extract programs that can explain decisions and solve the environment. Therefore, we intend to lay the foundation for a complete policy extraction algorithm in this paper.

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 [29]. The main components 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 or empty cells. The control flows include if-else statements and Boolean operators to formulate more complex conditions. The full DSL is included in Appendix B.

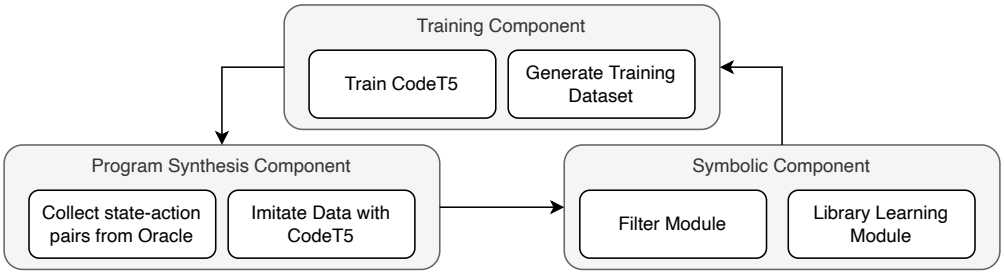

Figure 1: An overview of the overall architecture for creating explanations for a given reinforcement learning environment. The framework can be decomposed into three components that are executed iteratively and are guided by a curriculum. We describe the method in detail in Section 5.

Imitation Learning To simplify the problem, we limit ourselves to imitation learning [30], instead of directly finding programs from rewards. Although directly finding programs from rewards is an interesting challenge for future research, our main objective is to make the agent's behavior interpretable. To achieve this, we want to build a library of functions that allow us to draw conclusions about the knowledge the black-box agent acquired during the training process. We define the problem we want to solve as an imitation of sub-trajectories of state-action pairs collected from a previously trained agent.

5. Method

Figure 1 shows a high-level overview of our approach. The framework consists of three components and a curriculum. In addition, we need an oracle for collecting the data to be imitated.

Training Component The first part is the training process of the CodeT5 model [9] 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. We then execute them in a randomly selected environment from Figure 2 to collect data for imitating. We execute each program with a random sequence length t between 5 and 50. The programs do not have a specific target or reward since they are sampled from the DSL. Our goal in creating a training dataset 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 of six 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 partial 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 the goal position. Then the action is appended after the observation. We explain the text prompt generation in Appendix C.

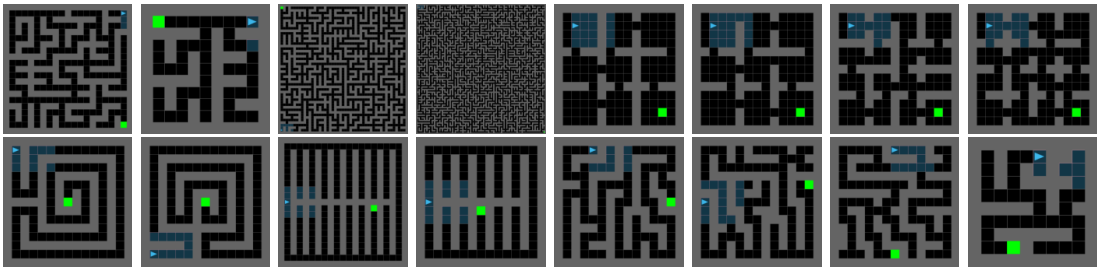

Figure 2: Different environments used for generating the training data. The evaluation is always performed on the medium sized perfect maze environment which is placed on the top left.

There are still a lot of improvements possible for the generation of the training data. Currently, we only create one rollout of the program on a single environment. We also do not check whether programs are semantically interesting, i.e., if clauses where the Boolean condition is a tautology, leading to programs that consistently evaluate the same branch in the if clause.

Program Synthesis Component The second component converts the data collected from an oracle into the desired representation of the model for the current action sequence length, and then the model synthesizes P programs for the state-action sequences to be imitated. In our setup, 100 programs are synthesized for each text prompt, which 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 beenficial for the DSL since they have been synthesized several times for different state-action sequences.

Curriculum The curriculum is only based on the action sequence length and, therefore domain-agnostic. We start with an initial sequence length of five and increment it at the end of each iteration after the library learning module completed the extraction phase. We repeat all three steps until no more improvement is observable and no more functions are extracted from solved tasks. 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. In future work, we will design a more complicated curriculum that also takes into account the number of functions extracted in each iteration or the percentage of imitated data from the oracle.

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. After building up a library of more complex functions, the model can also synthesize programs for longer sequence lengths. In the experiments, we validated that our method cannot build up a library without using a curriculum (see Section E).

For a better understanding of the whole process, Algorithm 1 in the Appendix summarizes all of the steps.

The Oracle We use the grid-world domain [31] for evaluation and restrict the problem space to navigation tasks. We trained the ACCEL agent with the default hyperparameters from [28]. We then collected state-action pairs from the medium-sized perfect grid environment displayed on the top left in Figure 2.

We decided to use the ACCEL agent as an oracle, as it shows strong zero-shot capabilities to a wide-ranging domain of tasks [28] and therefore can be used to collect data for different problems without retraining an agent from scratch. In this work, we only conduct one case study on the perfect maze environment. We plan to evaluate our method on multiple tasks and domains in the future, to validate that it is domain-agnostic.

6. Experiments

We perform different ablation studies to justify the need for a language model as a program synthesizer and a curriculum. In Section 6.2 the method for generating explanations from programs is introduced. Finally, we perform a thorough analysis of the extracted functions in the library.

6.1. Evaluation

We compare our approach to different program synthesis methods and justify the need for a language model as a program synthesizer and a library learning module by the following baselines and ablation studies:

- Search: Program synthesis with a top-down enumerative search algorithm. We use the implementation from [13].
- DreamCoder: A neural-guided search algorithm with a library learning module [13].
- CodeT5: A language model fine-tuned on Lisp programs on our data [9].
- LibT5: Our method which combines a CodeT5 model with library learning.

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. Additionally, we justify the curriculum by comparing our method to a setup without using a curriculum. The performance is measured by

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

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

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

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 are all synthesized programs from a given method. $f(\text{P}, \tau)$ 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. $\text{EXEC}(\rho, s)$ 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.

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 D.

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. Our presented method can imitate most and the longest action sequences from the test data. CodeT5 without a library learning module struggles to imitate longer action sequences but still outperforms enumerative search and the DreamCoder system by a large margin. That also shows the importance of the library learning module in addition to a language model for synthesizing programs.

The performance of DreamCoder and the enumerative search algorithm was similar, although we experimented with different types of encoders and hyperparameters for the neural-guided search. That was also observed by Banburski et al., who applied DreamCoder on the Abstract and Reasoning Corpus [32].

When inspecting the found programs, it is also noticeable that both search methods do not find programs that represent the agent's decision process. The search methods always focus their attention on the $(0, 0)$ grid position, in contrast, the language models often look at the position in front of the agent. That is also reflected in the library of search-based methods, as no extracted function uses grid positions other than $(0, 0)$. This indicates that only programs that have checked the $(0, 0)$ position were found. The libraries are shown in Figure 9 and Figure 10 in the Appendix. These functions are f2 and f0, and, f0 and f3, respectively.

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 30. In comparison, complete trajectories are up to 100 steps long for this environment. We included the full ablation study in Appendix E.

6.2. 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 [33] and logging all function calls and their parameters.

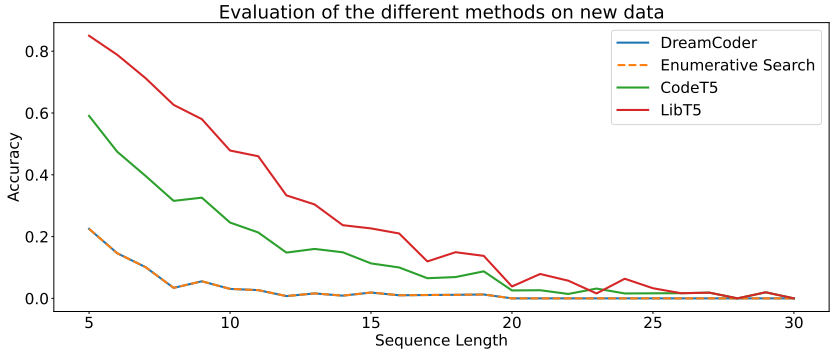

Figure 3: The evaluation of the different methods after no new functions were added to the library. The evaluation data was collected on new rollouts of the ACCEL agent on the perfect maze environment.

```
(lambda (lambda (#(lambda
  (lambda (#(lambda (lambda
    (if (not (eq-obj? $0 $1))
        forward-action left-action)))
    (#(lambda (get $0 2 1)) $0)
    (get $0 3 $1)))) 2 $1)))
```

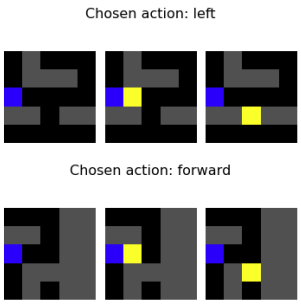

Figure 4: Left: The program for a given sub-trajectory. Right: The decision-making process, when executing the program on the state-action sequence. We show explanations for the first two states of the sub-trajectory. The grid positions that are checked in the program are yellow. The agent's position 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 in the left direction but does not move it.

Figure 4-left shows a synthesized program for a given sub-trajectory. Numbers with a leading dollar sign are variables which are input parameters of a function. Each lambda in a synthesized program corresponds to an input function parameter. So two consecutive lambdas represent two input parameters. The number sign with brackets #(...) denotes extracted functions discovered in a previous iteration. On the right side, we demonstrate two 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 direction. The walls are gray, and the path through the maze is black. The program finds two objects on the map and compares them. In the first row of the pictures, two empty cells are compared, and therefore the agent chooses the left action. The left action only turns the agent 90° in the left direction, without moving it forward. In the second row, the agent compares a wall object with an empty cell and decides to go forward, i.e., one step to the right. In Appendix F, we show more found programs and their visual explanations of the decision-making process.

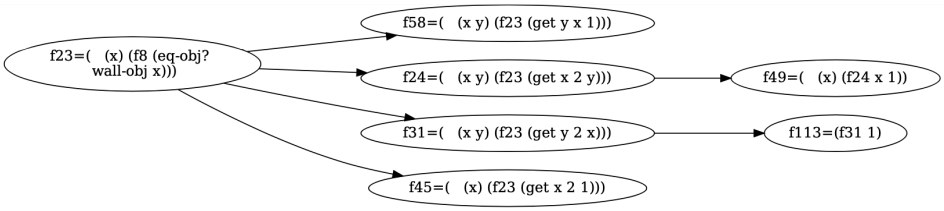

Figure 5: Examples of the extracted functions from LibT5 that are almost identical semantically but different syntactically. The functions f49, f113 and f45 are semantically identical. These function definitions are based on the Lisp language [29].

6.3. Inspecting the Program Library

In this section, we analyze the libraries extracted from the evaluated methods. Appendix G includes the full libraries. Table 1 shows the number of extracted functions for the different program synthesis methods. Our introduced method extracts 114 functions after the training, which is the best method compared to three and seven extracted features from search-based methods.

The use of a language model for program search raises a new problem similar to one previously addressed in Inductive Logic Programming. Cropper analyzed what the perfect library size is and how to forget unnecessary programs in the library [34]. 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.

Figure 5 shows some example functions with the same functionality, i.e., getting the value at a grid position and comparing it to a wall object. The functions f49, f113, and f45 are semantically identical. That is observable also in the AlphaCode system, which clusters synthesized programs before selecting solutions for submission to programming competitions [7]. For enumerative search algorithms, on the other hand, it is more challenging to synthesize many semantically identical programs since the search space is limited to local changes, and the search stops upon finding the k most likely correct programs. We use $k = 5$, similar to the DreamCoder hyperparameters. Depending on the problem one wants to solve, this is advantageous as we avoid the issue of many semantically similar functions. In our case study, however, we need a neural network to find more complex programs in the first place, but also introduce the problem above. To mitigate this, we need to improve how the semantic benefit of extracted functions is evaluated before adding them to the library since we currently use the DreamCoder library learning module without modifications.

Table 1
Number of extracted functions for different program synthesis methods.

Method	Enumerative Search	Neural-guided Search	LibT5
Number of functions	3	7	114

6.4. Discussion & Limitations

In our experiments, LibT5 could learn a library of functions for a navigation task with a discrete state and action space. We could also imitate sub-trajectories of policy rollouts up to a length of 30. 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 analyzing the generated libraries for the given domain and showed that our method could extract 107 more functions than DreamCoder.

While LibT5 showed promising performance for grid-based environments with a small observation space, we must evaluate its fitness for larger observation spaces. Currently, we limit the maximum length of the text prompts that the CodeT5 model can handle. We can mitigate that by using vision encoders, where the state-action pairs are encodable without restrictions on the observation space.

Additionally, for this environment, 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 CodeT5, similar to [35], where an object detection model parses the image 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 for continuous state and action spaces.

7. Conclusion and Future Work

In this paper, we introduced LibT5, a new framework to learn structured and reusable knowledge in grid-based reinforcement learning environments that allows reasoning about the behavior of black-box agents. This is realizable by using a language model as a program synthesizer combined with a library learning module. The main disadvantage of LibT5 is its dependence on an oracle for collecting trajectories, whereas our method does not depend on much background knowledge except for the initial functions 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. 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. Additionally, we want to evaluate our method on more domains to validate that it is domain-agnostic. Games are attractive as they enable a better understanding of the strategies discovered by artificial agents through self-play, such as in AlphaGo [36], where humans struggle to comprehend the reasoning process.

Acknowledgments

The work of S. M. was supported by Grant 01IS20051 from the German Federal Ministry of Education and Research (BMBF).

References

[1] R. Waldinger, R. C. T. Lee, PROW: A Step Toward Automatic Program Writing, 1969.

[2] Z. Manna, R. Waldinger, Knowledge and reasoning in program synthesis, in: Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1, IJCAI'75, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1975, p. 288–295.

[3] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, D. Tarlow, DeepCoder: Learning to Write Programs, 2016. URL: https://openreview.net/forum?id=ByldLrqlx.

[4] M. Nye, L. Hewitt, J. Tenenbaum, A. Solar-Lezama, Learning to Infer Program Sketches, in: Proceedings of the 36th International Conference on Machine Learning, PMLR, 2019, pp. 4861–4870. URL: https://proceedings.mlr.press/v97/nye19a.html, iSSN: 2640-3498.

[5] D. Ritchie, A. Thomas, P. Hanrahan, N. D. Goodman, 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, Curran Associates Inc., Red Hook, NY, USA, 2016, pp. 622–630.

[6] S. Chaudhuri, K. Ellis, O. Polozov, R. Singh, A. Solar-Lezama, Y. Yue, Neurosymbolic Programming, Foundations and Trends® in Programming Languages 7 (2021) 158–243. URL: https://www.nowpublishers.com/article/Details/PGL-049. doi:10.1561/2500000049, publisher: Now Publishers, Inc.

[7] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. d. M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. d. Freitas, K. Kavukcuoglu, O. Vinyals, Competition-level code generation with AlphaCode, Science 378 (2022) 1092–1097. URL: https://www.science.org/doi/abs/10.1126/science.abq1158. doi:10.1126/science.abq1158, _eprint: https://www.science.org/doi/pdf/10.1126/science.abq1158.

[8] A. Odena, C. Sutton, D. M. Dohan, E. Jiang, H. Michalewski, J. Austin, M. P. Bosma, M. Nye, M. Terry, Q. V. Le, Program Synthesis with Large Language Models, in: n/a, n/a, 2021, p. n/a.

[9] Y. Wang, W. Wang, S. Joty, S. C. Hoi, 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, Online and Punta Cana, Dominican Republic, 2021, pp. 8696–8708. URL: https://aclanthology.org/2021.emnlp-main.685. doi:10.18653/v1/2021.emnlp-main.685.

[10] G. Poesia, A. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, S. Gulwani, Synchromesh: Reliable Code Generation from Pre-trained Language Models, 2022. URL: https://openreview.net/forum?id=KmtVD97J43e.

[11] T. Scholak, N. Schucher, D. Bahdanau, 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, Online and Punta Cana, Dominican Republic, 2021, pp. 9895–9901. URL: https://aclanthology.org/2021.emnlp-main.779. doi:10.18653/v1/2021.emnlp-main.779.

[12] L. Hewitt, T. A. Le, J. Tenenbaum, Learning to learn generative programs with Memoised Wake-Sleep, in: Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence

(UAI), PMLR, 2020, pp. 1278–1287. URL: https://proceedings.mlr.press/v124/hewitt20a.html, iSSN: 2640-3498.

[13] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, J. B. Tenenbaum, 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, Association for Computing Machinery, New York, NY, USA, 2021, pp. 835–850. URL: https://doi.org/10.1145/3453483.3454080. doi:10.1145/3453483.3454080.

[14] K. Ellis, L. Morales, M. Sablé-Meyer, A. Solar-Lezama, J. Tenenbaum, Learning Libraries of Subroutines for Neurally– Guided Bayesian Program Induction, in: Advances in Neural Information Processing Systems, volume 31, Curran Associates, Inc., 2018. URL: https://papers.nips.cc/paper/2018/hash/7aa685b3b1dc1d6780bf36f7340078c9-Abstract.html.

[15] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, N. Polikarpova, babble: Learning Better Abstractions with E-Graphs and Anti-Unification, Proceedings of the ACM on Programming Languages 7 (2023) 396–424. URL: http://arxiv.org/abs/2212.04596. doi:10.1145/3571207, arXiv:2212.04596 [cs].

[16] M. Bowers, T. X. Olausson, L. Wong, G. Grand, J. B. Tenenbaum, K. Ellis, A. Solar-Lezama, Top-Down Synthesis for Library Learning, Proceedings of the ACM on Programming Languages 7 (2023) 1182–1213. URL: http://arxiv.org/abs/2211.16605. doi:10.1145/3571234, arXiv:2211.16605 [cs].

[17] Y. Qing, S. Liu, J. Song, M. Song, A survey on explainable reinforcement learning: Concepts, algorithms, challenges, arXiv preprint arXiv:2211.06665 (2022).

[18] A. Verma, H. Le, Y. Yue, S. Chaudhuri, Imitation-Projected Programmatic Reinforcement Learning, in: Advances in Neural Information Processing Systems, volume 32, Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/5a44a53b7d26bb1e54c05222f186dcfb-Abstract.html.

[19] A. Verma, V. Murali, R. Singh, P. Kohli, S. Chaudhuri, Programmatically Interpretable Reinforcement Learning, 2018. URL: https://www.semanticscholar.org/paper/Programmatically-Interpretable-Reinforcement-Verma-Murali/c43bba87b4237a93d96b2a3e91da25d91fd0bb91.

[20] G. Anderson, A. Verma, I. Dillig, S. Chaudhuri, Neurosymbolic Reinforcement Learning with Formally Verified Exploration, in: Advances in Neural Information Processing Systems, volume 33, Curran Associates, Inc., 2020, pp. 6172–6183. URL: https://proceedings.neurips.cc/paper/2020/hash/448d5eda79895153938a8431919f4c9f-Abstract.html.

[21] D. Trivedi, J. Zhang, S.-H. Sun, J. J. Lim, Learning to Synthesize Programs as Interpretable and Generalizable Policies, 2022. URL: https://openreview.net/forum?id=wP9twkexC3V.

[22] W. Qiu, H. Zhu, Programmatic Reinforcement Learning without Oracles, 2022. URL: https://openreview.net/forum?id=6Tk2noBdvxt.

[23] J. P. Inala, O. Bastani, Z. Tavares, A. Solar-Lezama, Synthesizing Programmatic Policies that Inductively Generalize, 2020. URL: https://openreview.net/forum?id=S1l8oANFDH.

[24] T. Silver, K. R. Allen, A. K. Lew, L. Pack Kaelbling, J. Tenenbaum, Few-Shot Bayesian Imitation Learning with Logical Program Policies, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 34, 2020, pp. 10251–10258. URL: https://ojs.aaai.org/index.php/AAAI/article/view/6587. doi:10.1609/aaai.v34i06.6587, iSSN: 2374-3468, 2159-5399

Issue: 06 Journal Abbreviation: AAAI.

[25] O. Bastani, Y. Pu, A. Solar-Lezama, Verifiable Reinforcement Learning via Policy Extraction, in: Advances in Neural Information Processing Systems, volume 31, Curran Associates, Inc., 2018. URL: https://papers.nips.cc/paper/2018/hash/e6d8545daa42d5ced125a4bf747b3688-Abstract.html.

[26] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, Journal of Machine Learning Research 21 (2020) 1–67. URL: http://jmlr.org/papers/v21/20-074.html.

[27] M. Dennis, N. Jaques, E. Vinitsky, A. Bayen, S. Russell, A. Critch, S. Levine, Emergent complexity and zero-shot transfer via unsupervised environment design, in: Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS'20, Curran Associates Inc., Red Hook, NY, USA, 2020.

[28] J. Parker-Holder, M. Jiang, M. Dennis, M. Samvelyan, J. Foerster, E. Grefenstette, T. Rocktäschel, Evolving Curricula with Regret-Based Environment Design, in: Proceedings of the 39th International Conference on Machine Learning, PMLR, 2022, pp. 17473–17498. URL: https://proceedings.mlr.press/v162/parker-holder22a.html, iSSN: 2640-3498.

[29] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, part i, Commun. ACM 3 (1960) 184–195. URL: https://doi.org/10.1145/367177.367199. doi:10.1145/367177.367199.

[30] A. Hussein, M. M. Gaber, E. Elyan, C. Jayne, Imitation learning: A survey of learning methods, ACM Comput. Surv. 50 (2017). URL: https://doi.org/10.1145/3054912. doi:10.1145/3054912.

[31] M. Chevalier-Boisvert, L. Willems, S. Pal, Minimalistic gridworld environment for gymnasium, 2018. URL: https://github.com/Farama-Foundation/Minigrid.

[32] A. Banburski, A. Gandhi, S. Alford, S. Dandekar, S. Chin, tomaso a poggio, Dreaming with ARC, in: Learning Meets Combinatorial Algorithms at NeurIPS2020, 2020. URL: https://openreview.net/forum?id=-gjy2V1ko6t.

[33] B. Ryder, Constructing the call graph of a program, IEEE Transactions on Software Engineering SE-5 (1979) 216–226. doi:10.1109/TSE.1979.234183.

[34] A. Cropper, Forgetting to Learn Logic Programs, Proceedings of the AAAI Conference on Artificial Intelligence 34 (2020) 3676–3683. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5776. doi:10.1609/aaai.v34i04.5776, number: 04.

[35] K. Yi, J. Wu, C. Gan, A. Torralba, P. Kohli, J. B. Tenenbaum, Neural-symbolic vqa: Disentangling reasoning from vision and language understanding, in: Neural Information Processing Systems, 2018.

[36] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis, Mastering the game of Go with deep neural networks and tree search, Nature 529 (2016) 484–489. doi:10.1038/nature16961.

A. Algorithm

Algorithm 1 summarizes the steps for discovering a library of functions and training the CodeT5 model. The assessment if the library is improved, only checks if the improved library contains more functions as the library from the last iteration. If no more functions are extracted in this iteration, we will stop the training process.

Algorithm 1: The complete training algorithm for LibT5

Input : imitation data D, initial grammar G, environments E
Output : library L, model θ

```
improved_library = True;
sequence_length = 5
number_of_programs = 50000
epochs = 5
program_ast_depth = 6
model = CodeT5()
while(improved_library) {
    train_dataset = generate_random_training_dataset(number_of_programs, E, G,
  program_ast_depth)
    model = train_model(model, train_dataset, epochs)
    test_tasks = generate_test_tasks_from_data(D, sequence_length)
    programs = synthesize_programs_for_test_tasks(test_tasks)
    programs = filter_correct_programs(programs, test_tasks)
    new_G = extract_library(programs)
    improved_library = compare_libraries(new_G, G)
    if (improved_library) {
        G = new_G
    }
    sequence_length++
}
return model, L;
```

B. Domain-specific Language

Table 2 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` → `direction` → `action`, so a random program is always defined from two input parameters of type `map` and `direction` and returns a value of type `action`.

Table 2
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.

Primitive Function/Values	Description	Type
left, right, forward	possible actions	action
0, 1, 2, 3 and 4	integer values	int
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
empty, wall, goal	possible objects on the map	object
if	standard if-clause	bool → func → func → func
eq-direction?	checks if two directions are equal	direction → direction → bool
eq-obj?	checks if two objects are equal	object → object → bool
get	get a object on the map for two coordinates	map → int → int → 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

C. Text Prompts

Text prompts are generated by converting the agent's partial 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 6 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.

D. Experimental Setup & Hardware Resources

Table 3 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

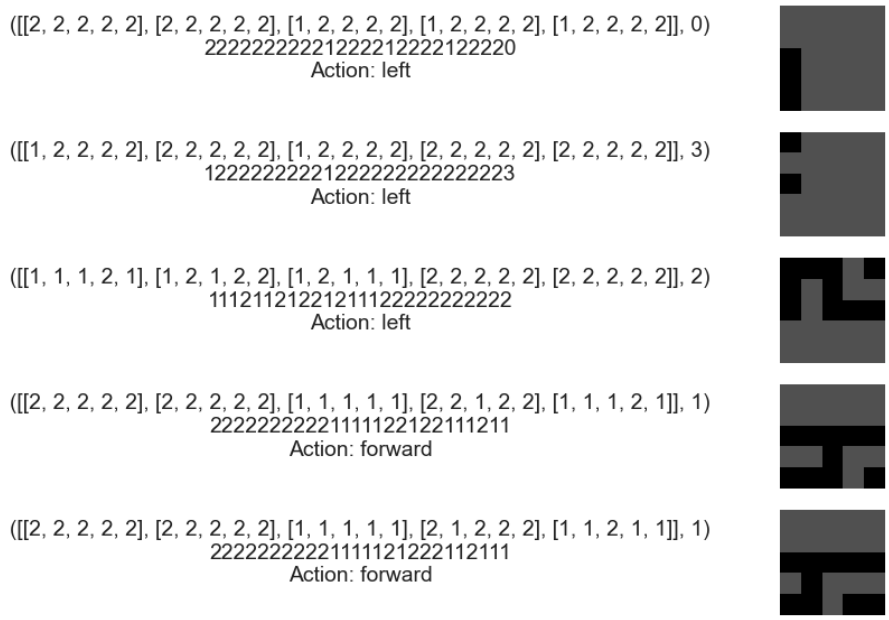

Figure 6: Text prompts are created by converting the 2D array into a string were all values are concatenated without spaces.

and tried out a few timeouts; Nevertheless, we did not find any improvements in increasing the search timeout.

For the neural-guided search, we train an encoder-decoder neural network. The encoder for the state observations was adapted from the ACCEL agent [28]. The decoder, which predicts which functional primitives are used in the program to be synthesized, is generated automatically from the DreamCoder system [13]. We train the neural-guided search model for 5000 update steps. In addition, before training a neural network, the DreamCoder system first performs an enumerative search to obtain examples for generating random training programs. The synthesized programs from the enumerative search are also added to the training dataset. The library learning module is restricted to an arity of three, which means that extracted functions can have up to three input parameters.

We tried out different numbers of training programs and also encoders for DreamCoder, since in Figure 3, DreamCoder and the enumerative search algorithm both resulted in the same performance. Using the default DreamCoder parameters with 500 random training programs achieved the same accuracy as using 5000 or 50000 programs. We concluded that our problem to solve is not well-suited for DreamCoder compared to the hand-crafted tasks that [13] uses to evaluate the DreamCoder system.

E. Ablation Study: Curriculum

In this section, we justify our decision to use the curriculum based on an ablation study using the method without a curriculum. Table 4 shows the number of solved tasks. None of the

Table 3
The hyperparameters for the different methods.

Hyperparameters	Enumerative Search	DreamCoder	CodeT5	LibT5
Search Timeout	720 seconds	720 seconds	-	-
Epochs	-	50	5	5
Training Programs	-	50000	50000	50000
Arity	3	3	-	3

Table 4
Ablation study of the method without using a curriculum.

	Enumerative Search	DreamCoder	CodeT5	LibT5
Solved Tasks	0/14	0/14	0/14	0/14

Table 5
The sequence lengths of the collected trajectories.

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Length	131	75	73	61	112	63	145	80	96	80	50	92	87	18

methods could solve a task; Thus, we analyzed the sequence lengths of the trajectories, which are displayed in Table 5. Since the tasks are limited and none of them were solved, no method could extract a library from solved tasks. To build a library so that more tasks are solvable in later iterations, we introduce a curriculum based on the sequence length. Using that, we can generate 26 tasks with a sequence length of 5 from the first task with a sequence length of 131. By converting all complete trajectories into sub-trajectories, the number of tasks increases from 14 to 229, i.e., sufficient to extract functions from solved tasks that form a library.

F. Synthesized Programs & Visualization of the Decision Making Process

We show two more examples of the agent's decision-making process. The agent's position is colored in blue. Grey and black show walls and empty cells, respectively. The cells that the agent attends are yellow. In Figure 7-left, we see a synthesized program. Numbers with a leading dollar sign are variables which are input parameters of a function. Each lambda in a synthesized program corresponds to an input function parameter. So two consecutive lambdas represent two input parameters. The number sign with brackets #(...) denotes extracted functions discovered in a previous iteration.

The program first checks the direction the agent faces and then determines a position on the grid and compares it to a wall object. The position on the grid depends on the agent's direction, so the cell the agent compares changes in the third row of pictures. Figure 8 shows another example of the agent's reasoning process. The program first checks if the position (1,0) is empty and then turns 90° to the left if that is the case. If it is not, the agent checks the cell in front of it (2,1) and moves forward if the position in front of the agent is empty, otherwise it turns left.

```
(lambda (lambda (#(lambda
  (lambda (#(lambda
    (if $0 left-action forward-action))
      (eq-obj? wall-obj (get $0 2 $1)))))
      (if (eq-direction? direction-2 $0) 1 3) $1)))
```

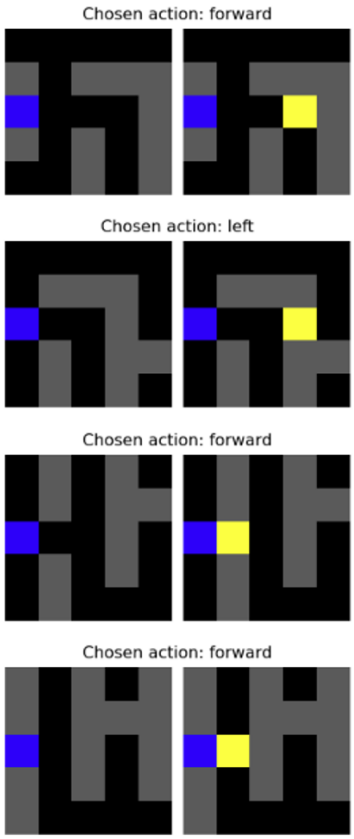

Figure 7: Left: The program for a given sub-trajectory. Right: The decision-making process, when executing the program on the state-action sequence. We show explanations for the first four states of the sub-trajectory. The synthesized program first checks the direction the agent faces, then determines a position on the grid and compares it to a wall object.

G. Extracted Libraries

In this section, we present the full libraries for the three different methods. The enumerative search was only able to extract three functions (Figure 9), the neural-guided search builds a library of seven functions (Figure 10), and our method was able to extract 114 functions (Figure 11).

```
(lambda (lambda
  (if (eq-obj? wall-obj (get $1 1 0))
  (#(lambda (#(lambda
    (if $0 forward-action left-action))
      (eq-obj? ($0 #(lambda (get $0 2 1)))
          empty-obj)))
      (lambda ($0 $2))) (#(lambda (if (eq-obj? $0
          empty-obj)
      forward-action left-action)) wall-obj))))
```

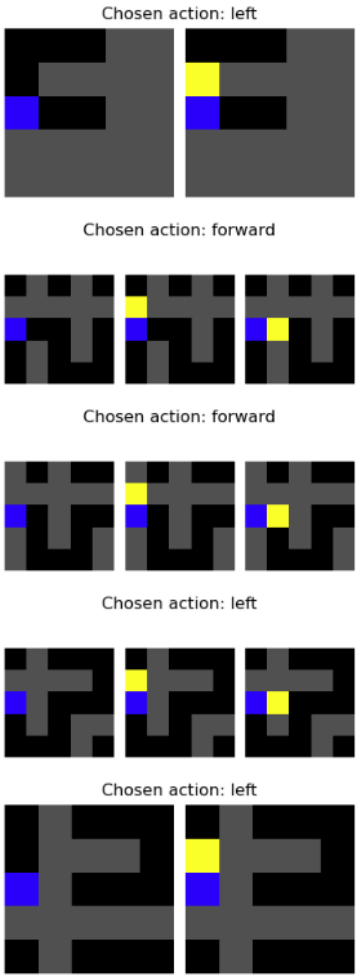

Figure 8: Left: The program for a given sub-trajectory. Right: The decision-making process, when executing the program on the state-action sequence. We show explanations for the first five states of the sub-trajectory. This program first checks if the position (1,0) is an empty cell. If it is empty, the agent turns 90° to the left, otherwise it checks the position (2,1) in front of it. If (2,1) is empty the agent moves forward, if there is a wall the agent turns to the left.

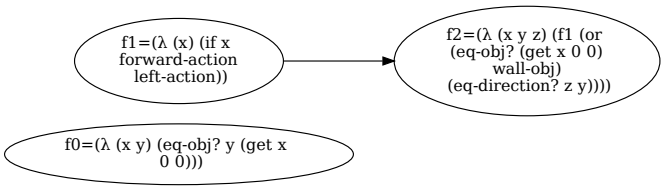

Figure 9: The extracted functions from programs found by using an enumerative search algorithm.

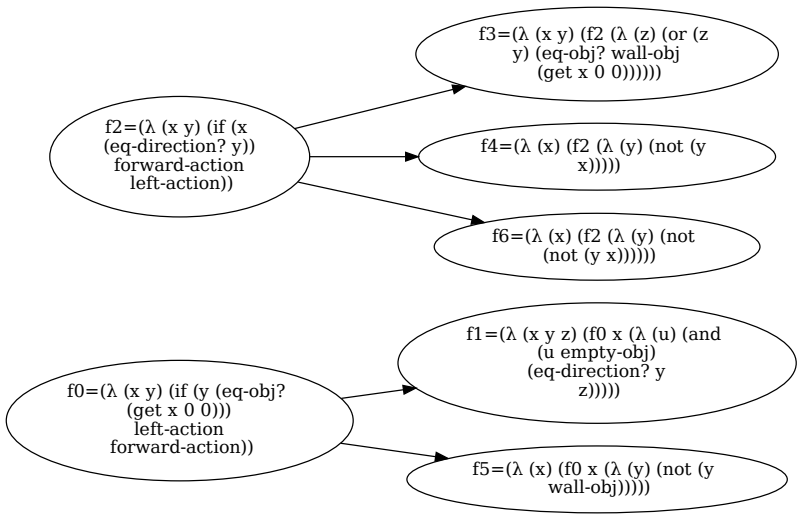

Figure 10: The extracted functions from programs found by using the neural-guided search algorithm.

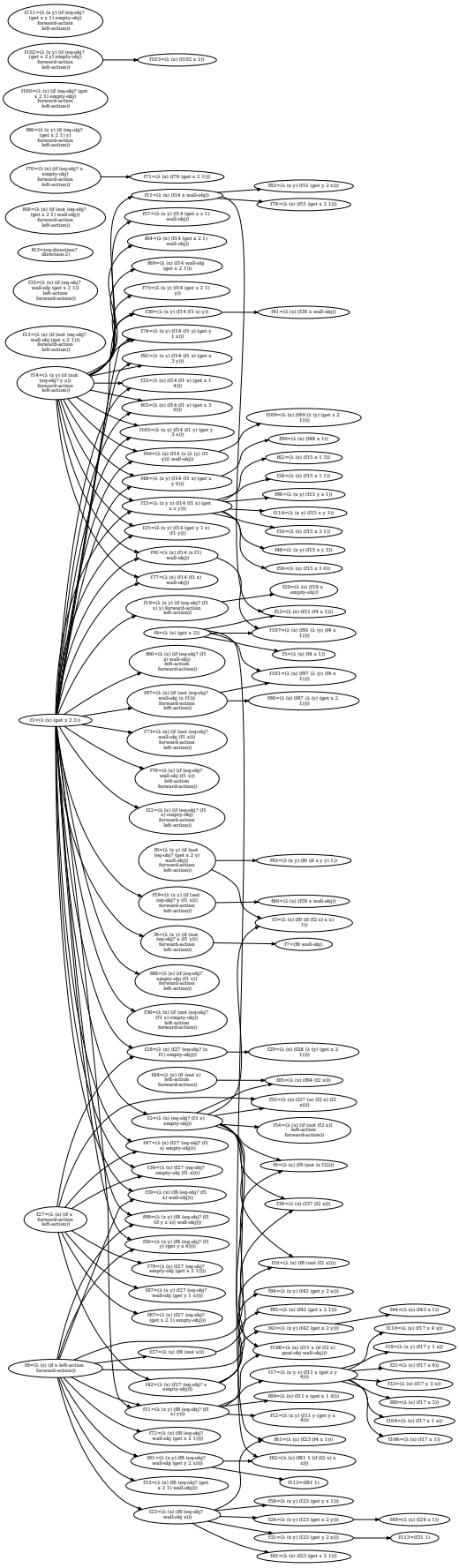

Figure 11: The extracted functions from programs found by using our introduced method LibT5 (zoom in for better visibility).