Syntax-guided synthesis of agents for OpenAI Gym

Xiaolu Fu

4th Year Project Report
Artificial Intelligence
School of Informatics
University of Edinburgh

2023

Abstract

Syntax-guided synthesis is a program synthesis method that generates code guaranteed to satisfy a logical specification. Early work of Chasins et al. [15] shows that syntax-guided synthesis can synthesise reactive motion plans for simple benchmarks. But the synthesis time becomes unreasonably long when the depth of a synthesised abstract syntax tree is greater than 5, and the synthesis time increases exponentially as the number of actions in the synthesised strategy increases. The synthesis of motion plans is rarely studied since the paper was published in 2016. Additionally, driven by the popularity of machine learning, the corpus of reactive motion planning benchmarks has expanded quickly. In this dissertation, we aim to explore the capability of a state-of-the-art syntax-guided synthesis solver to solve OpenAI Gym problems and to determine whether the paper's conclusions by Chasins et al. still hold. To achieve this, we modelled 3 grid-world environments and 2 Atari environments from OpenAI Gym using the SyGuS-IF language. The resulting benchmarks were submitted to the SyGuS-Comp repository to facilitate future research. Our Python pipeline, integrated with CVC5, was used to solve some stochastic agent problems; we also proposed generating high-level plans to tackle complex agent problems. While we found that SyGuS is generally not scalable for agent problems and unable to deal with probability and maximisation, it can still generate simple, error-free or high-level agent strategies. This study sheds light on the potential of SyGuS in solving agent problems and identifies areas where it can be improved.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Xiaolu Fu*)

Acknowledgements

I want to thank Elizabeth Polgreen, my supervisor, for suggesting this topic and for her invaluable guidance and support throughout the project. I would also like to thank Andrew Reynolds for reviewing my benchmarks and responding to me on GitHub. Thank all the contributors to the SyGuS-IF language, CVC5, and OpenAI Gym for their accomplishments.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Program synthesis has been a famous problem ever since the first programmable computer was built [22]. With the emergence of advanced deep learning model architectures, modern program synthesizers based on language models have brought great convenience to programmers. However, due to the ambiguity of human intents, even the state-of-the-art model cannot guarantee the correctness of a synthesised program [16]. In contrast, formal synthesis, a classical approach, guarantees correctness but requires programmers to encode the desired behaviour into logical constraints. This property of formal synthesis makes it still irreplaceable nowadays.

Syntax-guided synthesis (SyGuS) is a sub-category of formal synthesis in which a syntactic constraint is also encoded, besides semantic constraints on the desired function. A syntactic constraint is context-free grammar, which restricts the search space of the candidate program. With the syntactic restriction, a synthesizer will ignore the unrelated syntax in a programming language to reduce the synthesis time. The community of formal synthesis has been attempting to solve various problems using SyGuS. Inspired by a previous work [15], this project utilises SyGuS to tackle agent problems.

As an alternative approach in the era of AI, reinforcement learning (RL) is a popularly used approach to tackle agent problems nowadays [32] [30]. RL approaches train agents by rewarding them for specific interactions with an environment [27]. Deep RL approaches utilise the advantages of neural networks; with more than one hidden layer, a model can approximate any continuous functions (which refers to an agent plan) [23]. However, this approximation is not guaranteed to be correct.

Real-life automation tasks are commonly formulated as agent problems. In many automation tasks, the utility is closely related to social safety, where a single error is not tolerated (e.g., driving automation). Formal synthesis techniques fit these tasks, as they can find the exact function that guarantees no error. To investigate the pros and cons of SyGuS, the project uses OpenAI Gym, one of the most popular integrations of agent problems [12].

In this project, we aimed to model OpenAI Gym environments as SyGuS problems and

then solve them using a state-of-the-art SyGuS solver (CVC5). Our research hypothesis is that we can apply SyGuS to solve some OpenAI Gym agent problems.

1.2 Contribution

This project makes the following contributions:

- Modelled 5 OpenAI Gym environments into SyGuS problems and generated multiple SyGuS benchmarks for OpenAI Gym environments with random initialisation.
- Evaluated the performance of the state-of-the-art SyGuS solver on these benchmarks.
- Discussed strengths and weaknesses of SyGuS and types of agent problems it can be applied to.
- Submitted benchmarks to SyGuS-Comp, the main public repository of syntax-guided-synthesis benchmarks. We also reported the performance issues discovered to the CVC5 developers.

1.3 Dissertation Structure

The following content of this thesis is divided into four chapters. Chapter 2 provides a hierarchical overview of SyGuS, starting with a definition of SyGuS problems, followed by a discussion of oracle-guided synthesis and its variant, counterexample-guided inductive Synthesis. At the end of the background chapter, we briefly introduced OpenAI Gym.

Chapter 3 details how five OpenAI Gym environments can be modelled as SyGuS problems. We also mentioned a state-of-the-art solver's performance on our benchmark and the limitations of SyGuS, as this information is crucial for making our design decisions.

Chapter 4 explores why SyGuS cannot be applied to some problems. We evaluated the state-of-the-art solver using our benchmarks and discussed the results of our experiments. Additionally, we discussed the pros and cons of SyGuS and other approaches for solving agent problems.

Chapter 5 serves as the conclusion to the thesis, where we summarise our findings, evaluate our work, discuss potential additional work, and suggest future directions for research.

Chapter 2

Background

2.1 Syntax-Guided Synthesis

Program synthesis is to let artificial intelligence automatically generate programs, and it can potentially change the ways of programming. By using synthesis tools, software engineers only need to think about what they want without knowing how to achieve it. There are many uses for program synthesis tools. The concept of sketching [38][39] allows programmers to sketch the high-level structure of software, and the detailed programs will be automatically constructed; Program synthesis tools can also do automatic refactoring with much lower costs than humans, especially when the code has low readability [17]; Program de-obfuscation is another application, which can help people understanding and dealing with malicious malware [25]; Synthesising a time-consuming part of the program may also help with finding a more efficient version with equivalent functionality [36].

Among the big area of program synthesis, some approaches provide their best guess but do not guarantee correctness (e.g., neural network-related approaches [18]), some approaches guarantee correctness if (and only if) the desired program is specified correctly, formal synthesis belongs to the latter category. The problem of formal synthesis is to find a correct implementation that satisfies a set of given logical specifications [26]. This project focuses on syntax-guided synthesis (SyGuS), a particular case of formal synthesis where syntactic and semantic constraints restrict the potential implementations.

The definition of the SyGuS problem [5] is to prove $\exists f.\forall x.\phi$ by searching a correct function $f \in G$, where x is a set of input of function f, ϕ is a set of given logical specifications that describes the expected behaviours of the desired functions. ϕ is in a background theory T. The theory defines the data type or data structure of variables (e.g., Boolean, String, Array) and types of operation (e.g., comparison, condition, division). To restrict the space of searching, human programmers should provide a context-free grammar G, and G defines a set of implementations. It should contain at least one expression of the desired function; otherwise, a programmer must try a different grammar.

2.1.1 Syntax-Guided Synthesis Example

Assume there is a theory T, all variables under T are real numbers, and the set of operations consists of if-then-else (ITE), logical operations ($\neg$), comparisons ($<$), functions that compute square and square root (Sqrt) of a number. A simple example is to find a function that computes the absolute value of an input, a logical specification of function f could be:

$$\phi\colon ((f(x) = x) \vee (f(x) = -x)) \wedge (f(x) \geq 0) \tag{2.1}$$

A context-free grammar G1 defines the set of functions:

$$\begin{aligned} Exp\colon &= x \mid 0 \mid -Exp \mid ITE(Cond, Exp, Exp) \\ Cond\colon &= Exp \leq Exp \mid Exp < Exp \mid \neg Cond \end{aligned}$$

where ITE means an if-then-else statement. Two expressions are found that satisfy the specification ϕ:

$$\begin{aligned} &ITE(x < 0, -x, x) \\ &ITE(x \leq 0, -x, x) \end{aligned} \tag{2.2}$$

More than two expressions can be found by searching the space defined by the syntax. As the grammar restricts the space of possible implementations, a different grammar may result in a different output. An example of an alternative grammar G2 could be:

$$Exp\colon = x \mid 0 \mid Square(Exp) \mid Sqrt(Exp) \tag{2.3}$$

G2 does not contain f1 and f2; the expected output implementation should differ.

$$Sqrt(Square(x)) \tag{2.4}$$

Grammar is a parameter of SyGuS tools, and providing proper grammar is vital for program synthesis. Users must carefully implement their grammar depending on the programming language and the specific problem to be solved with the program. However, the grammar helps restrict the search space and allows programmers to decide the desired implementations.

2.2 Oracle-Guided Synthesis

Oracle-Guided Synthesis (OGIS) is a typical strategy that searches the space defined by syntactic constraints. It guarantees the synthesised program behaves as intended if the logical specification is correct and satisfiable by the syntactic constraint. Unlike many popular synthesis tools (e.g., speech synthesis, image synthesis) in Artificial Intelligence, the state-of-art OGIS methods do not involve deep learning. Instead, there is a synthesis phase to iteratively search for a candidate solution defined by the grammar and a verification oracle to verify if a candidate program satisfies the formal specification and gives feedback to the next synthesis phase.

The synthesis phase is performed by a search algorithm that searches the space of programs. A vanilla example of a search algorithm is randomly checking every unencountered implementation until a suitable implementation is found. The verification oracle is usually a satisfiability modulo theories (SMT) solver. An SMT solver can check if all inputs and outputs of a function satisfy all specified logical formulas given a background theory [10]. Modern SMT solvers can solve many problems with considerably high efficiency and are still evolving sustainably. There is an annual competition that encourages the innovation of SMT solvers. Among all submissions of the newest competition, an SMT solver called CVC5 [8] has won the most 1st prize and the best average rank of all tracks (see https://smt-comp.github.io/2022/results.html). There is also a competition for SyGuS called SyGuS-Comp [6], but the competition was not held in recent years. The winner of the most recent SyGuS-Comp was CVC4, the previous version of CVC5 (CVC4 could solve SyGuS problems and SMT problems). An executable program can also be a verification oracle if the specification is given as input-output pairs. Sometimes OGIS can be used for refactoring. By providing a program with correct behaviour, an OGIS synthesizer could find an alternative implementation with higher readability or efficiency [36].

The time complexity of the synthesis phase increases exponentially as the depth of the program's abstract syntax tree (AST) increases due to the nature of iterative search. Besides the synthesis phase, the SMT problem is NP-hard, and no known efficient algorithm can solve it under polynomial time. Much recent research in synthesis tackles the time complexity of OGIS, and several variants have been proposed. Innovations in this area can be split into two categories: novel algorithms to search the space of candidate programs in the synthesis phase or new SMT solvers in the verification stage. By choosing a suitable synthesis approach, a lot of problems could be solved within a reasonable time. In the related research, SyGuS solver could solve some simple agent problems within several minutes [15].

2.3 Counterexample-Guided Inductive Synthesis

Counterexample-guided Inductive Synthesis (CEGIS) is a variant of OGIS. It was first introduced in 2008 [37] and is still one of the most used inductive synthesis procedures nowadays. Inductive means the algorithm finds a program that satisfies a part of the logical specification and then checks if the program satisfies the entire specification. Counterexample-guided means the verification oracle learns new input-output pairs produced by the program that contradicts the specification, called counterexamples. These counterexamples are then used to guide the synthesis phase when finding new candidate programs. If a new program produces correct outputs given counterexample inputs, the program satisfies a part of the specification.

As shown in Figure 2.1, the CEGIS accepts inputs from programmers. The necessary parameters are grammar and formal specifications (including the background theory). The output of CEGIS is a piece of code that satisfies the given formal specifications under theory. The CEGIS algorithm will first check the feasibility of the specified SyGuS problem. If no program defined by the syntax satisfies the logical specification, the CEGIS algorithm will terminate and return a message for infeasibility. Some CEGIS

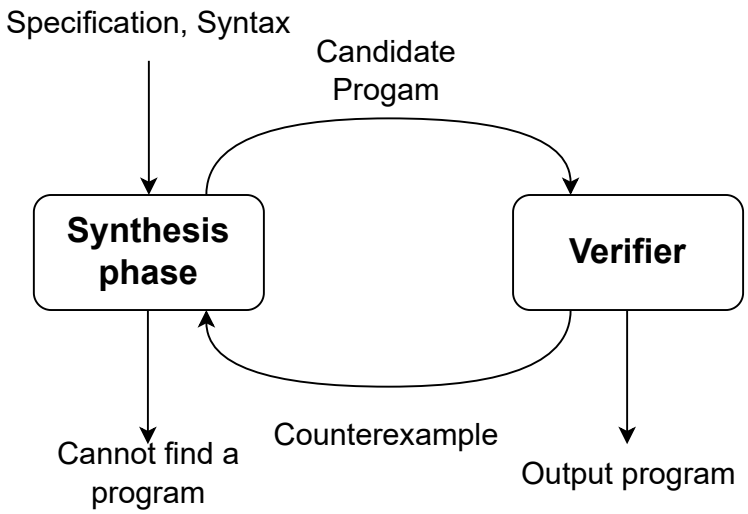

Figure 2.1: Counterexample-Guided Inductive Synthesis Structure

solvers have extra parameters, including input-output pairs, time to terminate, and search strategy. The time to terminate refers to the maximum execution time, and the solver will return a message for timeout if no suitable program is found within the specified time. The search strategy specifies which search algorithm to use in the synthesis phase; it will be explained in section 2.3.2.

2.3.1 Counterexample-Guided Inductive Synthesis Example

In the example SyGuS problem (2.1.1), we wanted to use CEGIS to synthesise a function that calculates the absolute value of a real input number. The grammar G1, specification ϕ is provided to the search algorithm, and we also have two input-output pairs:

$$\{(x = 0, f(x) = 0), (x = 1, f(x) = 1)\} \tag{2.5}$$

The search strategy is to test implementations individually, starting from the easiest. Since the task is simple, we can confidently expect that implementation will be found within 200 attempts.

The first attempt will be $f(x) = 0$, while the function is inconsistent with the second input-output pair when $x = 1$, $f(x)$ should be 1. The second attempt will be $f(x) = x$, and it is consistent with all input-output examples. Thus the candidate program is then passed to the verifier. However, the candidate is not a correct program and cannot satisfy the logical specification. The verifier will generate a counterexample $(x = -1, f(x) = 1)$, and the counterexample will be passed back to the search algorithm and added to the set of input-output pairs. Now the search algorithm has three example pairs in total. The next two attempts will be $f(x) = -0$ and $f(x) = -x$. These functions will be rejected in the synthesis phase as they are inconsistent with all example pairs. The search will continue until it finds a candidate program consistent with all input-output pairs or the number of attempts reaches 200.

The counterexample generated by the verifier can reduce the frequency of querying

the verification oracle, thus reducing the time and cost of unnecessary verifications. Since solving the SMT problem is computationally expensive, reducing the frequency of invoking SMT solvers could significantly boost the synthesis speed.

2.3.2 Search Strategy

The search space of the synthesis phase can be infinite if the grammar is recursive and the expression can be infinitely long. Even if the depth of AST or size of the expression is limited, the search space is still exponential to the depth of AST. If a search algorithm attempts to find a correct implementation from an infinitely large set by a completely random search, it is doubtful to succeed. Therefore, an intelligent search strategy for the synthesis phase is essential to restrict the procedure's time cost.

Related works have proposed various approaches to search efficiently; three approaches will be introduced and discussed. They are constraint-based search [21], stochastic search [36], and enumerative search [40]:

- **Constraint-based search**: The algorithm reduces a formal synthesis problem to a first-order constraint problem and uses an SMT solver to solve the constraint problem efficiently.
- **Stochastic search**: The algorithm assigns a score to an expression and applies biased stochastic sampling using a Markov Chain Monte Carlo sampler.
- **Enumerative search**: The algorithm enumerate all expression by some properties, and the candidate programs are verified in the enumerated order. The search space is pruned by ignoring expressions evaluated to the same value as another expression. For instance, in the grammar G1, f(x) = x and f(x) = -(-x) are equivalent.

Enumerative search is usually the fastest algorithm for various synthesis problems and is used in CVC5. Its intuitive strategy could reduce the number of queries to the oracle and significantly prune the search space for a potential expression.

2.3.3 SyGuS-IF language and Limitations

Computers cannot directly interpret the example logical specification and syntax in 2.1.1 before parsing it into machine-readable code. In practice, we need formalised language to describe logical specifications. To address this requirement and facilitate research in satisfiability modulo theories, several research groups worldwide incepted a community and brought forward the SMT-LIB. Since 2003, numerous researchers have contributed to providing standard rigorous descriptions of background theories and developing common input-output language for SMT solvers [9].

Compared to SMT solvers, a SyGuS solver also needs to parse inputs for the syntax. The SyGuS community proposed the SyGuS-IF language, designed based on the SMT-LIB language [33]. The SyGuS-IF language is used in SyGuS-Comp competitions and is now widely accepted by modern SyGuS solvers.

However, specifying a SyGuS problem in the SyGuS-IF language is not trivial; the language's syntax is simple and easily parseable but makes writing specifications inconvenient. A typical input written in SyGuS-IF language could be longer than the desired program written in high-level programming languages. In 2.1.1, the desired program can be as easy as a function call of an imported package, whereas the example input is an abbreviation of actual input to SyGuS-IF solvers. The output of the SyGuS-IF language is not directly executable. To run the output, users must design their parser and parse it into an existing programming language, and this extra effort is not desired in many software engineering scenarios. Therefore in practice, SyGuS is rarely used to speed up software development nowadays.

2.4 OpenAI Gym

OpenAI Gym (see https://www.gymlibrary.dev/) is an environment library for agent problems. An agent can observe some states of the environment and perform a set of actions to achieve some goals. The environment can be fully observable or partially observable. Agents in real life are impossible to fully observe the environment, while some virtual environments can be fully observed. The agent problem is constructing a set of rules allowing the agent to react to the observed state. The original purpose of OpenAI Gym is to provide benchmarks for reinforcement learning (RL). Modern RL approaches can produce sophisticated agents to interact with complex problems (e.g., autonomous driving), but the agents are not guaranteed to act optimally.

As machine learning is mainly implemented in Python, OpenAI Gym only supports Python interface, and the environment information is not interpretable by any SyGuS solver. Besides the observed environment, RL agents also receive rewards or penalties for achieving specific states. In later iterations or training, RL agents are more likely to perform a sequence of actions that receive rewards and avoid penalties. This reward mechanism is different in SyGuS, a concrete goal is needed instead, and it could be receiving at least some reward or reaching a specific awarded state. In this project, we implemented SyGuS benchmarks corresponding to OpenAI Gym benchmarks from scratch.

Chapter 3

From Agent Problems to SyGuS Problems

3.1 Overview of OpenAI Gym Environments

There are five first-party environments in OpenAI Gym: Atari, Mujoco, Toy Text, Classic Control and Box2D. Each environment is a unique agent problem, specifically, a reactive planning agent problem. Environments within the same set share some properties:

Atari environments are simulated Atari 2600 video games. An observed state is a frame, and each observation consists of pixels. Thus the observation space is discrete. Similar to actual video games, the action space is a set of valid controller inputs, which is also discrete.

MuJoCo stands for Multi-Joint Dynamics with Contact. MuJoCo environments involve sophisticated simulations of real-world physics. In these environments, agents need to manipulate multiple joints simultaneously. The action space is a set of real numbers corresponding to the torque applied on each joint. Each environment is a 3D box of infinite size, and each goal of agents is unbounded. For instance, in the Humanoid environment, an agent needs to make a humanoid with 17 joints run in a direction as fast as possible forever.

Toy Text is the most straightforward set of environments. It consists of 3 grid-world environments and a game of Blackjack. They have small discrete observation spaces and action spaces. Toy Text can be used for debugging in the development of RL algorithms. We expect SyGuS solvers to solve Toy Text benchmarks with the least effort.

Classical Control consists of 5 environments, each with a continuous observation state, and three environments have unbounded goals: reaching a position and staying forever. The other two environments are different versions of the same world, and the goal is to drive a car to the top of a mountain hill. The only difference is the action space. The continuous version takes acceleration as input, whereas the discrete version takes direction (with fixed acceleration) as input.

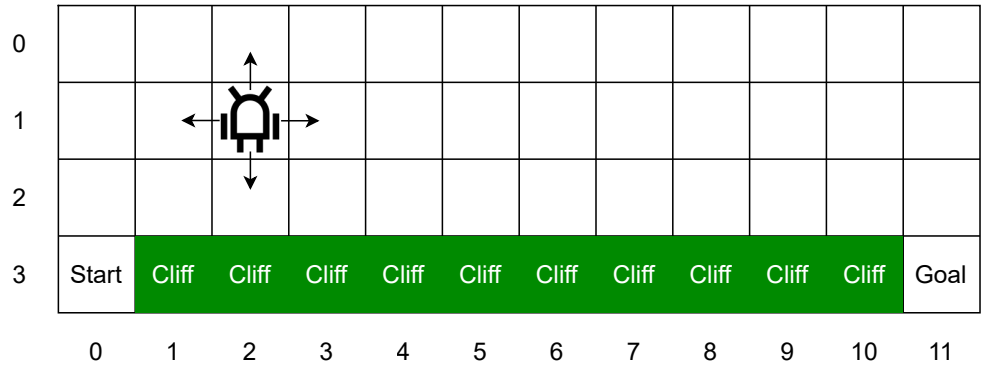

Figure 3.1: Cliff Walking

Box2D is a simpler version of Mujoco, and Box2D environments involve simple physics simulation. Box2D environments have smaller (compared to Mujoco) action space and bounded goals, but randomisation is introduced in every environment.

Solving reactive planning agent problems with SyGuS solvers has not been widely studied. As a contribution to SyGuS research, this project offers formal models for five agent problems in the SyGuS-IF language, and formal models are submitted to the SyGuS-Comp repository (https://github.com/SyGuS-Org/benchmarks). These five models are Cliff Walking, Taxi, Frozen Lake, Atari Pong, and Atari Pitfall. We chose them because they are deterministic and discrete and have finite (and small) observation space. We will discuss each environment in this chapter and explain why we did not choose other environments in the next chapter.

3.2 Cliff Walking

3.2.1 The Environment Overview

Cliff Walking is a deterministic and discrete environment, as shown in Figure 3.1. The gird world is a 4×12 matrix. The agent's initial position is the left bottom square, and the goal position is the right bottom square. At every time step, the agent could move in an orthogonal direction by one square. If the agent steps on a "cliff" square, it will return to the starting point immediately. Besides reaching the goal position, the agent receives a reward of -1 for taking each step and a -100 reward for stepping off the cliff. Therefore, the agent must take the shortest path and never step on a cliff square to maximise the reward. Since an agent that steps on a cliff will return to the starting position and takes more steps to reach the goal position, maximising the reward is the same as minimising the number of steps.

3.2.2 The Formal Model

The observation o_t at each time step t is the position of the agent, and the desired output from a SyGuS solver is a function f, such that $f(o_t) = m_t$, where m_t is the agent's move at time t, m_t equals [0, 1, 2, 3] corresponds to the agent moving [up, right, down, left].

In the OpenAI Gym environment, agent positions are encoded as flattened indexes, and the index of a square is $(y\_coordinate \times 24 + x\_coordinate)$; for example, the starting square is encoded as 36, and the goal square is encoded as 47. In this section, the formal model follows this style. To describe the transition between states, we apply addition or subtraction directly on the flattened index:

$$f_{transition}(o_t,\, m_t) = \begin{cases} o_t, & \text{if } hitBoundary(o_t, m_t) \text{ or } m_t \text{ is invalid} \\ o_t - 12, & \text{if } m_t == 0 \\ o_t + 1, & \text{if } m_t == 1 \\ o_t + 12, & \text{if } m_t == 2 \\ o_t - 1, & \text{if } m_t == 3 \end{cases} \tag{3.1}$$

The *hitBoundary* function checks if the agent is trying to move out of the boundary; for example, if the agent is at the starting position and $m_t = 2$, the next state should still be the starting position. If m_t is invalid, OpenAI Gym will raise an exception. However, when working with the SyGuS-IF language, we cannot define a function that throws exceptions or returns nothing. Therefore, if an invalid action is taken, we leave the agent in its current position. This minor modification will not affect the function if the specified goal is to maximise the reward.

The transition function also allows the agent to stand on a cliff square. But we specified this as a semantic constraint on the desired function, so the function will never move the agent to a cliff square. This design significantly reduces the synthesis time. In the next section, we will discuss the alternative design. In the modified environment, $f_{transition}$ correctly models the transition between states:

$$\forall t, t \geq 0 \implies f_{transition}(o_t, f(o_t)) == o_{t+1} \tag{3.2}$$

To model the environment as a SyGuS problem, we must define a logical specification for all constraints on f. The constraint is in the form of the following:

$$\begin{aligned} &correctStartingPosition \wedge correctTransitions \implies \\ &\quad reachingGoal \wedge correctIntermidiatePositions \end{aligned} \tag{3.3}$$

Suppose the agent's starting position is correct ($o_t = 36$), and each resulting state of an action m_t is correctly defined by $f_{transition}$. In that case, the function should bring the agent to the final position at a time step T ($o_T = 47$) without stepping on the cliff or out of the environment (avoiding penalty).

The *correctTransition* refers to:

$$\bigwedge_{t=1}^{T} (o_t == f_{transition}(o_{t-1}, f(o_{t-1}))) \tag{3.4}$$

The *correctIntermediatePositions* is false only when o_t is not one of the white squares in Figure 3.1:

$$\bigwedge_{t=1}^{T} ((o_t \leq 36 \wedge o_t \geq 0) \vee (o_t == 47)) \tag{3.5}$$

In a typical CEGIS, a counterexample of an undesired candidate program can either be a motion plan that does not end with the agent reaching the goal or some intermediate positions of the agent are not a white square in Figure 3.1.

We also need to prove that a motion plan maximises the reward. However, we should not write this requirement as a constraint:

$$\forall f' \in G, R(f') \leq R(f) \tag{3.6}$$

Where $R(f)$ is the reward received by executing function f, G is the provided syntax. To prove that a function satisfies this constraint, we need to compute the reward received for all possible functions. Proving the above constraint is practically impossible due to the excessive or infinite number of syntax-defined unique functions.

Inspired by previous work [15], we applied an iterative bounded check. To find an optimal function f which minimises T, we can prove that for all constraints with $T' < T$ is infeasible. We make iterative queries to a SyGuS solver, starting from $T = 1$. If the solver returns "infeasible", we then add T by one and make another query until a solution is found. Although, in general, proving the feasibility (also known as "unrealizability" in some literature) of a SyGuS problem can be a costly task [24], it is a more viable option than exhaustively comparing all unique functions in our specific scenario. This is because our SyGuS problem is bounded.

The syntax as a part of SyGuS input is recursive:

$$\begin{aligned} Start &:= MoveId \mid ITE(StartBool, Start, Start) \\ MoveId &:= 0 \mid 1 \mid 2 \mid 3 \\ Int &:= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 10 \mid 11 \mid getY(o_t) \mid getX(o_t) \\ Bool &:= Int \leq Int \mid Int < Int \mid Int == Int \mid \neg Bool \mid Bool \vee Bool \mid Bool \wedge Bool \end{aligned}$$

We restrict the width of the search space by only including constants that represent good moves or valid positions. The getY and getX are two functions that transfer a flattened index into Y and X coordinates. Compared to using o_t directly, using coordinates will reduce the search tree's width and speed up the synthesis.

3.2.3 Performance and Alternative Design

As shown by previous work [15], the synthesis time of an agent problem can be significantly affected by the number of actions to reach the goal state and the complexity of the environment. Since the optimal path for the Cliff Walking problem only involves two turning points. We expect a modern SyGuS solver to solve this problem quickly. However, CVC5 took more than 2 hours in the first verification phase of CEGIS (the synthesis phase finds $f(o_t) = 1$ as a candidate program, and then the verifier finds counterexamples of the candidate). If we write the SyGuS problem as a verification problem in which $f(o_t) = 1$, the SMT checker inside CVC5 could solve it within one second. CVC5 wasted time on normalising the conjecture using expensive ITE rewrite rules. This was reported to CVC5 developers, and the problem could be resolved by disabling the ITE rewrites (https://github.com/cvc5/cvc5/issues/9337).

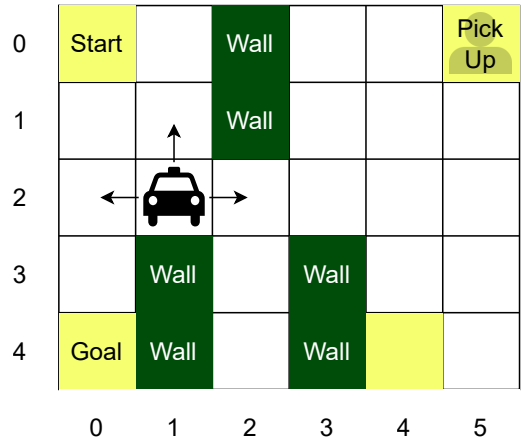

Figure 3.2: Taxi

The benchmark still requires 2-5 minutes (depending on the computational power) to solve, even if ITE rewrites are disabled. CVC5 spends a long time generating counterexamples and enumerating programs. For a simple agent problem like Cliff Walking, generating counterexamples does not significantly help with the synthesis, as the verification is very cheap. A program that produces correct output given an example input will satisfy the logical specification, as only one input-output pair exists.

We have briefly mentioned an alternative design decision of the benchmark. The alternative transition function is:

$$f'_{transition}(o_t, m_t) = \begin{cases} 36, & \text{if } 36 < f_{transition}(o_t, m_t) < 47 \\ f_{transition}(o_t, m_t), & \text{otherwise} \end{cases} \tag{3.7}$$

While using the alternative transition function, the *correctTransition* is no longer needed, and the alternative constraint is:

$$correctStartingPosition \wedge correctTransitions \implies reachingGoal \tag{3.8}$$

An optimal function will never make the agent falls off the cliff, as all synthesised functions are deterministic reactive plans. If the agent falls off the cliff once at a position, it will always fall off it. While checking if a function is correct, an undesired function should be rejected immediately after the agent falls off the cliff. But in the alternative benchmark, a candidate function will not be rejected immediately after the agent falls off the cliff; additional computations are required to check if the agent reaches the goal at time T. The alternative benchmark is sub-optimal because it is more computationally expensive to solve.

3.3 Taxi

3.3.1 The Environment Overview

Like the Cliff Walking problem, the Taxi problem in Toy Text also involves moving the agent from a start square to a goal square and avoiding some states. There are minor

differences:

- The forbidden states are walls instead of cliffs. The agent is not allowed to drive through walls.
- The Taxi now has a random starting state, one of four yellow squares in Figure 3.2, and the goal position is also a randomly decided yellow square.
- Besides moving in four directions, the agent has two more possible actions. It could pick up ($m_t = 4$) or drop a passenger off ($m_t = 5$) at the current location. The passenger is also generated at a yellow square but never its destination.
- The Taxi will not be punished when crushing a wall, but it will be punished when performing pickup or drop-off at the wrong position.

3.3.2 Modelling the Entire Environment

This section discusses the approach to modelling the entire environment, which is not our final implementation of benchmarks but essential for understanding the environment. We follow the same style of naming variables as in the previous section, o_t is the agent's position at time t, and m_t is the agent's action at time t. Besides, Boolean variable $onTaxi_t$ encodes if the passenger is picked up. This information is vital for motion planning, as the taxi needs to drive in the opposite direction immediately after the $onTaxi_t$ is set to true. Let T be the number of steps to complete the task; the logical specification is now slightly different:

$$\begin{aligned} correctStartingState \wedge correctTransitions \implies dropoffAtGoal \wedge \\ correctIntermediatePositions \wedge onTaxi_{T-1} \end{aligned} \tag{3.9}$$

where *correctStartingState* is the conjunction of $(o_0 == s)$ and $\neg\, onTaxi_0$, where s is the taxi's initial position. Let p be the passenger's initial position:

$$onTaxi_t = onTaxi_{t-1} \vee ((o_t == p) \vee (m_t == 4)) \tag{3.10}$$

where 4 is the action ID of pickup.

This means the passenger will always stay in the Taxi after being picked up. In the real OpenAI Gym, a Taxi could drop the passenger at an arbitrary state and receives negative rewards. Similar to stepping off the cliff in Cliff Walking, we could ignore this reward in Taxi to reduce computational complexity. Because we are minimising T to achieve the highest reward, and executing pickup and drop-off requires 1 step, including drop-off in the above equation will not change the synthesised function. Let g be the destination, *dropoffAtGoal* is:

$$((m_T == 5) \wedge (o_{T-1} == g)) \tag{3.11}$$

where 5 is the action ID of drop-off.

3.3.3 Benchmark Generator

A switch-case statement is necessary to create a function that can solve the Taxi problem regardless of the starting, pickup, and goal positions. However, since the SyGuS-IF language does not include a switch-case statement, we must use multiple if-then-else statements instead. This approach can be pretty verbose and may require significant code. There are 48 unique combinations of starting, pickup and goal positions, each requiring unique motion plans. An AST with a depth of 6 is needed to cover all cases, and for each case, an AST with a depth of at least 4 is required to specify the motion plan. Compared to the Cliff Walking benchmark (AST with a depth of 3), given the synthesis time is exponential to the depth of AST, synthesising an AST with a depth of 10 is impractical on modern hardware. Thus we have to create and solve one benchmark for a unique case of starting, pickup, and goal position.

SyGuS-IF is a repetitive language, and implementing a generating program is faster than manually writing benchmarks sometimes. We wrote a Python script that automatically generates and solves Taxi benchmarks. Using Python's built-in read and write functions and the os package, we make minor modifications to a template file, save it as a new benchmark, and then run CVC5 with the subprocess package.

3.3.4 Performance and Benchmark Design

The synthesis time is still unaffordable. In one of the worst cases, the Taxi starts from the top left square (0, 0), the passenger waits at (5, 0), and the destination of the tour is (0,4). The Taxi must execute 20 actions, 4 turning points, and 20 extra variables to specify whether the passenger is in the Taxi. The Taxi problem is more complex than the Cliff Walking problem. In an experiment, the CVC5 took more than 2 hours without successfully synthesising the solution.

To reduce the synthesis time, we decided to split the whole problem into two benchmarks: (1) the Taxi travels from the starting point to the passenger and pickup the passenger, (2) the Taxi travels to the destination and drop-off the passenger. As the length of the solution increases, the time required for synthesis grows exponentially, and the total time spent on synthesising both problems is much less than the synthesis time of the whole problem.

In our final implemented benchmarks, the variable $onTaxi_t$ is deleted, and the synthesised function takes o_t as the only input parameter ($f(o_t) = m_t$). Since the pickup and drop-off tasks are modelled in separate benchmarks, there will be two synthesised functions for the Taxi problem. Each synthesised function will not return different actions given the same position as input.

The logical specification for the pickup task is as follows:

$$(o_0 == s) \wedge correctTransitions \implies correctPickup \wedge correctIntermediatePositions \tag{3.12}$$

Let p be the passenger's initial position, *correctPickup* is:

$$((m_T == 4) \wedge (o_{T-1} == p)) \tag{3.13}$$

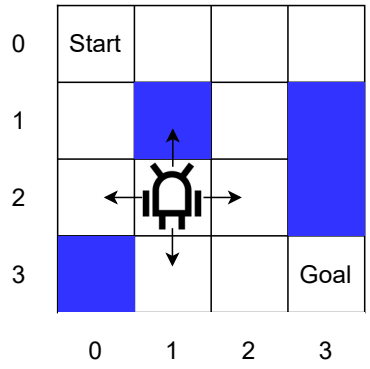

Figure 3.3: Frozen Lake

The logical specification for the drop-off task is as follows:

$$(o_0 == p) \wedge correctTransitions \implies dropoffAtGoal \wedge correctIntermediatePositions \tag{3.14}$$

3.4 Frozen Lake

3.4.1 The Environment Overview

Frozen Lake is another grid world, and there are two versions of it in OpenAI Gym: deterministic and non-deterministic. The deterministic Frozen Lake is very similar to Cliff Walking, except for three minor differences:

- Blue square in Figure 3.3 represents holes in the frozen surface; an agent who falls into a hole will not receive negative rewards.
- The agent will be sent back to the start if fallen into a hole, but the agent will not receive negative rewards for spending extra steps.
- OpenAI Gym offers a function to generate worlds with different arrangements and sizes randomly. A generative Python script is also required to solve different Frozen Lake problems.

3.4.2 The Deterministic Formal Model

We cannot specify "the agent will eventually reach the goal" using SyGuS-IF language, as this is an unbounded property. Proving a function is undesired is equivalent to proving "the agent will never reach the goal". Knowing the agent cannot reach the goal at time T does not mean we can infer the agent cannot reach the goal at $T+1$. Similar to previous problems, we decided to minimise T. Although minimisation is unnecessary in the Gym environment, checking the feasibility of the problem with a small T is faster than solving the problem with a random big T. The formal model of the deterministic Frozen Lake follows the same style as the Cliff Walking model, and we will not discuss it in detail.

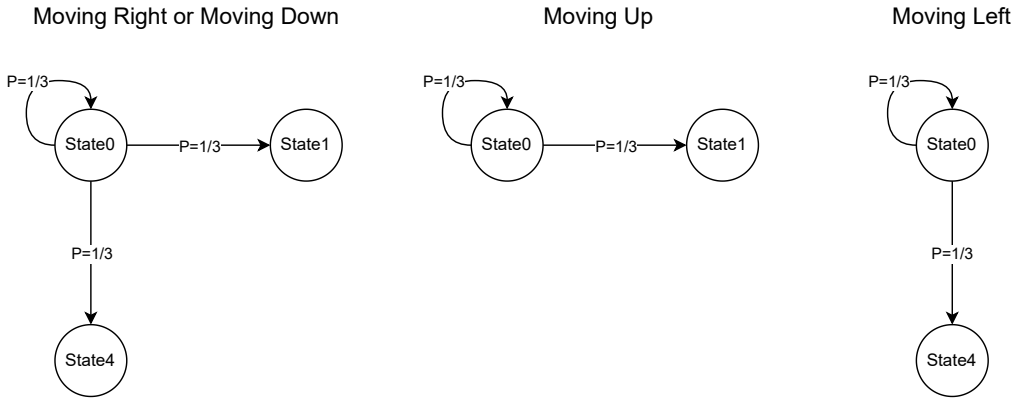

Figure 3.4: Frozen Lake: state transitions from the starting square.

3.4.3 Online Synthesis for a Stochastic Environment

In the non-deterministic Frozen Lake, an agent may not move in its intended direction due to the slippery floor. There is $\frac{2}{3}$ chance the agent moves in an orthogonal direction to its intended direction, and the agent is equally likely to move in each of the two orthogonal directions. For example, if the agent (1, 2) in Figure 3.3 wants to move down, it may end up in one of (0,2), (1, 2), and (2, 2) with an equal probability. Although a sophisticated agent performs equally well as random walking in this environment due to the rewarding mechanism, we still want to synthesise an agent with a reasonable strategy.

Standard (commonly used) SyGuS solvers are not good at dealing with probabilities. There are SyGuS solvers that specialise in probabilistic synthesis, PAYNT is a famous one among them [7], but these solvers require the problem to be modelled in a different language, PRISM [35]. This project only attempted to model Gym environments in the SyGuS-IF language. This subsection will explain why SyGuS-IF-language-based approaches cannot deal with probabilities and what is the compromised solution.

Standard SyGuS solvers can deal with non-determinism by treating the environment as a finite state machine. A transition from a state will reach a set of states. If the agent moves down from the starting position (state0), it will reach state0, state1 and state4. However, this approach to modelling non-determinism ignores the probability and suffers from the risk of state-space explosion.

No function guarantees to bring the agent to the goal within some steps. When the agent is in the starting position, executing each valid move will make the agent stay in the same position with a chance, as shown in Figure 3.4. The logical specification "reach the goal at time T" is infeasible (T is non-infinity), as there is at least $\frac{1}{3}^T$ chance that the agent is not going anywhere.

The best motion strategy will minimise the expected number of steps. The goal of maximising the expectation could be formulated in the SyGuS-IF language. Assigning a utility to each non-deterministic transition, a better action should result in high-utility transitions with a bigger chance. Minimising the expected number of steps becomes maximising the utility. To make the specification correct, the utility of reaching a

position must be the expected number of steps to travel from that position to the goal.

However, this game is unbounded. There can be infinitely many steps to travel from a neighbouring square to the goal. In this case, computing the average number of steps requires a set of mathematical calculations not included in any SyGuS-IF-supported logical theory (e.g., integration and limit theories). Maximising the expectation is infeasible without manually assigning the utility for each transition. However, SyGuS can optimise the expectation when the maximum number of actions is restricted. We will discuss a finite and stochastic problem in 4.1.3.

There is a different area of research in formal synthesis called strategy synthesis. It uses Markov Decision Processes (MDP) and probabilistic checks to find a motion strategy with some expectations. The found strategy is a direct mapping from states to actions [20].

The only solution is not to formulate the entire problem as a SyGuS problem. By performing online synthesis, each time, we generate a function that only solves a part of the agent problem and re-synthesis a function when the previously synthesised function is no longer working. Each synthesised function assumes the agent travels from its current position to the goal position without slipping. A new motion plan will be synthesised when the agent moves in an unintended direction and cannot reach the goal position using the previous plan (assuming no slips). In this project, the online synthesis is accomplished with a Python script. This is the pseudocode for the script:

Algorithm 1 Online Synthesis

Require: *Env* generated by the Gym API
  $state \leftarrow Env.start$
  $goal \leftarrow Env.goal$
  $CVC5\_output \leftarrow synthesis(Env, state)$
  $plan \leftarrow parse(CVC5\_output)$
  while *state* is not *goal* **do**
    if $planNotWorking(plan, Env, state)$ **then**
      $CVC5\_output \leftarrow synthesis(Env, state)$
      $plan \leftarrow parse(CVC5\_output)$
    end if
    state = plan.next_state(state) ▷ execute the plan
  end while

Our online synthesis is named after [43] and is different from receding horizon approaches [42], the motion plan is synthesised over the entire fully-observable environment, and there are no fixed time steps before the next re-planning.

3.4.4 Performance of Online Synthesis

The system's performance is hard to evaluate due to high randomness in the environment. A Frozen Lake problem with a bigger size or more holes tends to invoke CVC5 more times. Moreover, CVC5 takes longer to solve a Frozen Lake problem with a bigger

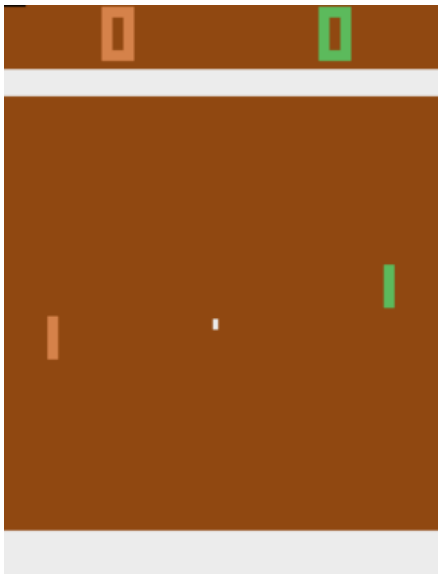

Figure 3.5: Atari Pong, the image is the output of the OpenAI Gym API

size or more holes. This stochasticity in the environment makes the computational cost explodes even faster. This solution of the system is still not optimal, but better than random walking in most scenarios.

3.5 Atari Pong

3.5.1 The Environment Overview

The Atari Pong is one of the most famous video games in history, and the gym environment is a simulation of the player against AI mode in the original game (Figure 3.5). The game is played like tennis, with a paddle on both sides of the screen, and a player moves a paddle vertically to reflect the ball. If the player or computer misses the ball, the opponent will get one score and be able to serve in the next round. The reward is negative when the opponent gets one score. Besides serving and moving the paddle, the player can choose the direction to reflect the ball: up or down.

Three versions of the simulated environment differ from the original game by including some noise. There are 4 versions of the Pong environment:

- One version deployed a stickiness mechanism, which forces the agent to repeat the previous action.
- Another version deployed a stochastic frame-skipping mechanism, which makes the skipped frame unobservable and the next frame unpredictable.
- The third version deployed both random mechanisms.
- The last version does not have any noise.

The noise was proposed as it benefits RL training [31] but can be detrimental to a

synthesised agent. The synthesised agent must find strategies that work for all possible scenarios, and the noise makes that impossible (same as the stochasticity). The frame-skipping mechanism is especially devastating since it forces the agent not to choose repetitive actions [28], and synthesised SyGuS agents are prone to act repetitively due to the restriction in running time and the depth of AST. The stickiness is pseudo-random, so it is still unfeasible to brute-force the random generator. In contrast, an RL agent finds imperfect strategies that work for as many scenarios as possible.

Unlike other first-party environments, Atari environments are black boxes, and the latest OpenAI Gym version no longer supports them. These environments are simulated via the arcade learning environment (ALE) [11], and the ALE is built based on Atari 2600 emulator Stella and runs ROM images of games directly. Since Atari 2600 production has been ceased 30 years ago, and most games were programmed in assembly language for an old processor, modern binary analysis tools cannot interpret the ROM image of an Atari game. Getting and understanding any Atari game source code nowadays is a great challenge. We performed brute-force experiments to learn the environment. However, we still cannot specify the exact environment using a function other than switch cases.

The deterministic agent always misses the ball from a specific direction, but the synthesised agent cannot exploit its weakness without knowing the exact adversarial strategy. The best strategy is to return the paddle to the middle position immediately after reflecting the ball; predict the ball trajectory immediately after the opponent reflects the ball; the paddle is then moved to where the trajectory crosses the x-coordinate of the paddle. The entire process is complex and will make the synthesis time unreasonably long.

We separate the process into three parts: predicting the trajectory, moving the paddle, and returning to the middle. We did not further split each part into multiple more straightforward problems, as we wanted each synthesised function to be meaningful.

3.5.2 Predicting the Trajectory

The trajectory-predicting algorithm takes the current and previous position of the ball as input and returns a y-coordinate (where the paddle should move). This is a problem to find a linear function that passes two points and then predicts the reflection.

We used x and y-coordinates instead of flattened indexes to model the world, as there are an excessive number of states. There is no velocity decay throughout the game. The ball takes an extra frame to reflect on the upper or lower boundary. There is no vertical displacement during the reflection step, but the ball moves horizontally. Sometimes, the ball moves vertically in that step, but the vertical displacement is compensated in the next step, which results in no difference in the final intersection. Thus given the velocity (vx_t, vy_t) the ball position (x_t, y_t) at time t should be:

$$(x_t, y_t) = \begin{cases} (x_{t-1} + vx_{t-1},\ y_{t-1}), & \text{if } \mathit{isReflect}(t) \\ (x_{t-1} + vx_{t-1},\ y_{t-1} + vy_{t-1}), & \text{otherwise} \end{cases} \tag{3.15}$$

The horizontal component of the ball's velocity vx_t is always 2 pixels per frame, so we

did not define a variable for it. The synthesised function takes the ball position (x_t, y_t) and vy_t as input parameters and returns $yCoordinate$.

For simplicity, we use $correctTransition_t$ to represent the above specification, and $correctVelocityUpdate_t$ refers to:

$$(vx_t, vy_t) = \begin{cases} (vx_{t-1}, \ -vy_{t-1}), & \text{if } isReflect(t) \\ (vx_{t-1}, \ vy_{t-1}), & \text{otherwise} \end{cases} \tag{3.16}$$

The function $isReflect$ checks if the ball moves out of the boundary in the next step:

$$isReflect(t) = ((y_t + vy_t) < 0) \vee ((y_t + vy_t) > 79) \tag{3.17}$$

The game board height was 160 pixels, but due to pixel duplication, the board can be down-sampled into 80×80 pixels without affecting any game decisions.

The specification for where the paddle should move to can be less strict. After down-sampling, the surface of the paddle consists of 4 pixels, and a contact of a single pixel is enough to reflect the ball. The function $isPredicted$ checks if the paddle has contacted the ball at time step t:

$$\begin{aligned} isPredicted(yCoordinate, (x_t, y_t)) = (x_t \geq 69) \wedge (x_t \leq 71) \\ \wedge (yCoordinate \leq y_t) \wedge (yCoordinate + 3 \geq y_t) \end{aligned} \tag{3.18}$$

The synthesised function should return a $yCoordinate$, which is the y-coordinate of the top pixel of the paddle, and 69 to 71 is the range of the x-coordinate when the ball hits the paddle. The logical specification is that correct ball movement and correct velocity updates imply the paddle hits the ball before time step T:

$$\begin{aligned} &(\bigwedge_{t=1}^{T} correctTransition_t \wedge correctVelocityUpdate_t) \\ &\quad \Longrightarrow (\bigvee_{t=1}^{T} isPredicted(yCoordinated, (x_t, y_t))) \end{aligned} \tag{3.19}$$

We applied a top-down approach, where we first implemented the arithmetic equation for predicting the trajectory and then minimise the grammar. The syntactic constraint only includes expressions that exist in our implemented function:

$$\begin{aligned} Start: &= RealNum \mid Start + RealNum \mid Start - RealNum \mid Start \times RealNum \\ RealNum: &= 69 \mid 0.5 \mid x_0 \mid y_0 \mid vy_0 \end{aligned}$$

This top-down approach is not applicable when a desired function is unknown, and this is not a practical use of SyGuS. The reason for doing so is to test the capability of a SyGuS solver and provide benchmarks for research of new SyGuS solvers.

3.5.3 Controlling the Paddle

The game has a deterministic inertia mechanism, making the paddle hard to manipulate. The function to calculate the inertia is not intuitive and disobeys real-world physics. We could find a function other than switch cases to model the inertia in our experiments.

We then calculated the average inertia, implemented a deterministic Python agent based on it, and tested it in the gym environment. Since every part of the experiment is deterministic, the agent traps in a loop that always reflects the ball in the same position. Given a time bound T, the agent's total reward will be 0.

We did not write a benchmark for controlling the paddle for two reasons: CVC5 cannot solve the trajectory-predicting problem within a reasonable time; We cannot model the inertia correctly, and an approximation is verified to be sub-optimal.

3.5.4 Performance

While synthesising the function that predicts the trajectory, CVC5 stopped on the first verification stage. If we write the SyGuS problem as a verification problem, the verifier spent more than 30 minutes without returning a result. However, this performance was expected.

Although the logical specification is straightforward, the formal model is much more complex. Firstly, using real numbers is inevitable in this model, as we need at least one division to formulate the function passing through two points (the denominator is a constant). Arithmetic theories for real numbers are computationally expensive for an SMT solver. Secondly, there are much more states and variables compared to previous models.

Since a SyGuS solver could not solve the first part of the desired strategy, it cannot synthesise the whole function. Atari Pong is an example that shows the limitation of SyGuS.

3.6 Atari Pitfall

3.6.1 The Environment Overview

Atari Pitfall is a deterministic but very complex environment. The environment consists of 255 boxes. The top level of the environment is a maze, and each cell in the maze is a box, and each box is connected to 4 other boxes. The agent could travel to other boxes by moving toward the boundary of the current box. It could move left/right above/under the ground. Every time the agent travels in the underground tunnel, it moves 3 boxes forward. As demonstrated in Figure 3.6, the agent starts in Box0. If it moves left above the ground, it will end up in Box254 or reach Box252 by moving left under the ground. Underground walls stop players from using the underground shortcut.

Each scene has various hazards, like rolling logs, lakes, pits, holes, hostile creatures, etc. There are 32 treasures spread over the 255 boxes, and the agent is awarded (with a score) if it finds and picks up a treasure. On the other hand, the agent will be punished for not avoiding the hazard, scores are reduced for being harmed by non-lethal hazards (logs and holes), and the agent will lose one life for being harmed by other hazards. There are 3 lives in total. The game will terminate if the agent runs out of time or all lives are lost. There are 20 minutes for operation before the timeout. The game aims to maximise the score by avoiding hazards and finding treasures. Because the environment

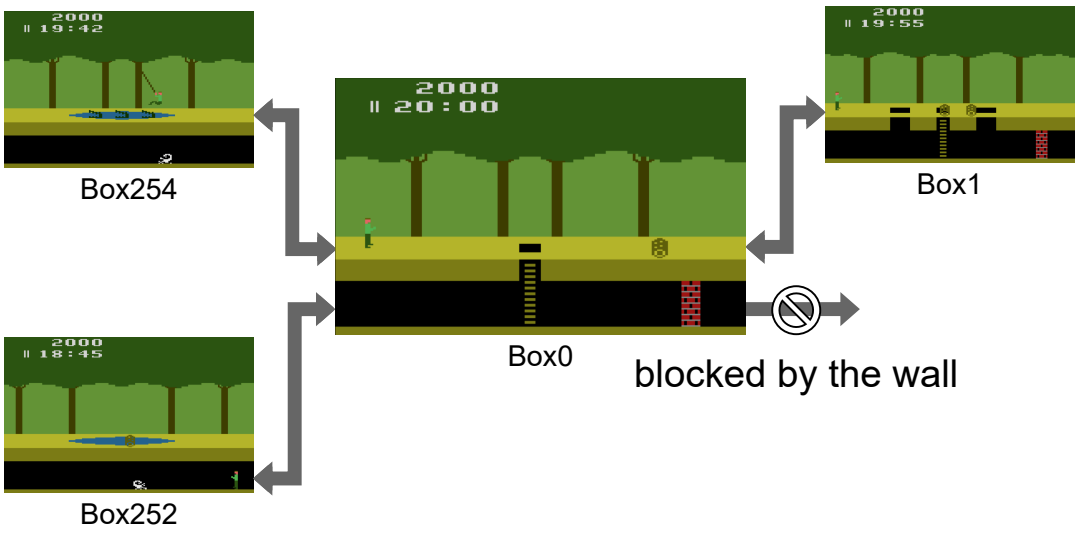

Figure 3.6: Atari Pitfall

is complicated, the reward is sparse, and agents are penalised quickly, many classic RL models cannot be adequately trained. Their best score is the default score a the beginning of the game (they used 0 in the paper) [41], a simple strategy that waits until timeout is no worse than these RL agents.

To find a better solution, we only have to set the goal as finding the nearest treasure and then waiting for the timeout at a safe position. Synthesising a pixel-and-frame level solution is impossible, as we do not have source code to model the exact transition between states. The environment complexity has exceeded SyGuS solvers' capability. We only need to generate a high level that tells the agents which path to go and how it could deal with some hazards.

3.6.2 Pathfinding

The synthesised agent should travel between boxes and find the nearest treasure. Like a grid world, the agent can move in one of 4 directions. However, there are also some minor differences:

- Pitfall boxes are arranged and enumerated linearly. We do not use flattened indexes.
- Besides o_t to encode which box the agent is in at time t, we also need $isUnder_t$ to encode whether the agent is in the underground tunnel, the agent can only switch $isUnder_t$ by using a ladder or hole (which always co-exists with a ladder).
- There is no forbidden state, but specific transitions between states are prohibited, as there might be a wall or no ladder.

We first define 4 functions [wallRight, wallLeft, ladder, treasure]. They return true if the current box (given as an Int parameter) contains a wall (left/right), ladder or treasure. Each function is a disjunction of the current box index equals every box index that contains a wall, ladder or treasure. For example, the function checks if there is a wall at

the right end of the underground tunnel:

$$\begin{aligned} wallRight(currPosition) = (currPosition == -5) \vee (currPosition == -13) \\ \vee (currPosition == 0) \vee (currPosition == 1) \end{aligned} \tag{3.20}$$

Since we aim to find the nearest treasure, we can only model a small part of the environment. The agent is not expected to travel through a complete loop (255 boxes) of the environment, so we do not have to define the environment as a circle. By using negative numbers (-5 and -13) instead of large positive numbers (250 and 242) as box indexes, we can simplify transition functions into:

$$\begin{aligned} &nextPosition(o_t,\ isUnder_t,\ m_t) = ITE(m_t == 0,\ o_t, \\ &\quad ITE((isUnder_t \implies ladder(currPosition)) \wedge (m_t == 1),\ o_t + 1, \\ &\quad ITE(\neg isUnder_t \implies ladder(currPosition) \wedge \neg wallRight(currPosition) \\ &\quad \wedge (m_t == 2),\ o_t + 3, \\ &\quad ITE(\neg isUnder_t \implies ladder(currPosition) \wedge \neg wallLeft(currPosition) \\ &\quad \wedge (m_t == 3),\ o_t - 3, \\ &\quad ITE((isUnder_t \implies ladder(currPosition)) \wedge (m_t == 4),\ o_t - 1,\ o_t))))) \end{aligned} \tag{3.21}$$

$$\begin{aligned} &nextState(o_t,\ isUnder_t,\ m_t) = ITE(m_t == 0,\ isUnder_t, \\ &\quad ITE((isUnder_t \implies ladder(currPosition)) \wedge (m_t == 1),\ false, \\ &\quad ITE(\neg isUnder_t \implies ladder(currPosition) \wedge \neg wallRight(currPosition) \\ &\quad \wedge (m_t == 2),\ true, \\ &\quad ITE(\neg isUnder_t \implies ladder(currPosition) \wedge \neg wallLeft(currPosition) \\ &\quad \wedge (m_t == 3),\ true, \\ &\quad ITE((isUnder_t \implies ladder(currPosition)) \wedge (m_t == 4),\ false,\ isUnder_t))))) \end{aligned} \tag{3.22}$$

where m_t is the action performed at time step t, and it is the output of the synthesised function. The action id [1, 2, 3, 4] refers to the direction the agent travels [right, right underground, left underground, left].

The synthesised function takes o_t and $isUnder_t$ as input parameters, so $f(o_t, isUnder_t) = m_t$. The logical specification of the synthesised function f is to find a treasure at time T:

$$correctPosition \wedge correctIsUnder \implies treasure(o_T) \wedge (m_T == 0) \tag{3.23}$$

where $m_T == 0$ is an abstract expression of waiting until timeout after finding the treasure. The specification $correctPosition$ refers to the agent always reaching the expected position:

$$\bigwedge_{t=1}^{T} o_{t+1} == nextPosition(o_t, isUnder_t, m_t) \tag{3.24}$$

The specification $correctIsUnder$ refers to:

$$\bigwedge_{t=1}^{T} isUnder_{t+1} == nextState(o_t, isUnder_t, m_t) \tag{3.25}$$

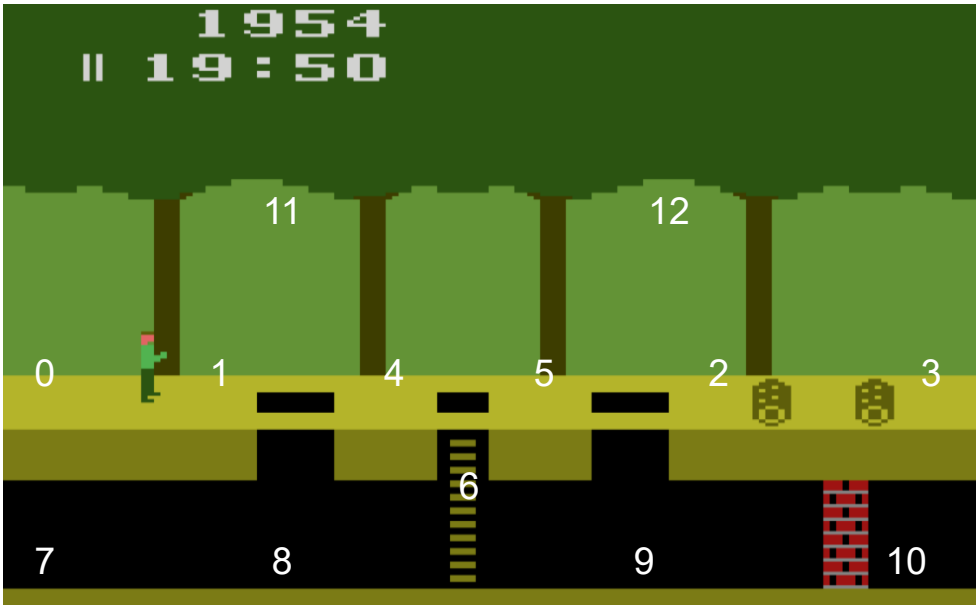

Figure 3.7: Atari Pitfall, enumerated intermediate states, agents may transition to other states by performing the correct actions.

The grammar is very similar to a grid-world problem, which we will not discuss in this chapter, but instead, we will put the whole benchmark in Appendix A.

3.6.3 Avoiding Hazards

We cannot model the environment with pixel-and-frame level details, but we could abstract the environment and synthesis a high-level plan. For example, in Figure 3.6, Box254, there is a lake with 3 crocodiles and a swinging vine above the ground. The agent will lose a life if falling into the lake or the mouth of a crocodile, to travel across the dangerous lake, the agent could jump onto the swinging vine and jump off after being swung to the opposite side of the lake, or it could timing jump 3 times when crocodiles closed their mouth (the agent could stand on a crocodile with mouth closed). The high-level plan does not include when or how long the agent should jump. It only needs to know the agent is near the dangerous lake.

We investigated the game and found that each unique interactive hazard or challenge always appears in a fixed position. Thus, we can reduce the 210×160 pixels game into a 13-states environment. The agent only needs to do a monotonous action to reach a connected state. The resulting state of acting is dependent on the specific box. For instance, the agent will reach state 4 in Figure 3.7 by jumping rightwards, but it could reach state 11 instead if there is a swinging vine.

There are 11 interactive objects listed in Table 3.1, most of which are hazards. The agent could interact with the object and travel across the object. Please note that all interactions (transitions) in the table are bi-directional, the agent can jump (leftwards) over a snake from state 2 and reach state 3, and it can also jump reversely from state 3 and reach state 2. Besides these transitions, the agent can also travel between states if

Interactive Objects	Positions (Between 2 States)	How to Interact
Scorpion	8-9	Jump
Wall	7-8, 9-10	None
Pit	1-2	None
Quicksand	1-2	Walk
Crocodile + Lake	1-2	3 Jumps
Ladder	4-5, 4-6, 6-5, 8-6, 9-6, 6-5, 8-9	Jump, Climb, Walk
2 Holes	1-4, 5-2	Jump
Rolling Log(s)	0-3	Jump
3 Fixed Logs	0-1, 2-3, 2-3	Jump
Snake	2-3	Jump
Swinging Vine	1-11, 11-12, 12-2	Jump, Wait

Table 3.1: A Pitfall box may contain several of the listed Objects, and the agent could perform some actions to interact with them.

there is no object between them.

The number of transitions between states is minimised, and transitions with negative rewards are removed. For example, the agent could fall into a hole and enter the underground tunnel (a transition between state 1 and state 8). The transitions using the ladder are also simplified, the agent could choose to reach state 4 or state 5 after climbing up the ladder, but we force the agent to choose state 5 in our formal model.

A few objects in the environment are precisely the same as others except demonstrated with a different image, and they are not listed in the table. For example, Box252 in Figure 3.6 contains a lake with tidal (its size varies and may disappear regularly), which is equivalent to a quicksand except it is blue. There are patterns that some objects always co-exist: a pit (or a lake with fixed size) always appears with a swinging vine; if there are 2 holes, there must be a ladder as well; a ladder always co-exists with one wall. Moreover, there must be an object located between state 1 and state 2, if the object is not a crocodile, there must be another object generated on the ground (state 0-1 or state 2-3), and the underground tunnel must contain either a scorpion or a ladder (with a wall).

Rolling Logs are ignored in the formal model as they are dynamic. The agent must know when exactly it should jump to deal with rolling logs. Moreover, interactions with a rolling log do not result in state transitions, as the agent could dodge the rolling log by simply jumping in place.

Due to the excessive number of cases in the environment, we had to hardcode a transition function for each unique box, e.g., the agent could climb a ladder in Box0 but could not do so in Box254. We will not discuss the hardcoded function in detail but list an example benchmark in Appendix B. Let $correctTransition$ specify that each motion choice of the function always results in the correct transition, and the logical specification for the synthesised function is:

$$correctStartingState \wedge correctTransitions \implies reachingDestination \tag{3.26}$$

where *correctStartingState* and *reachingDestination* refer to where the agent enters and leaves the box, this is decided by the top-level plan (output of pathfinding). For instance, the agent enters the box_x from box_{x-1} and travels to box_{x+1} can be specified as $o_0 == 0$ and $o_T == 3$. In this agent problem, we do not have to minimise the number of steps, and the agent has sufficient time. We let T be 12 (number of states minus 1) for every box.

The syntactic constraint is also different because of the excessive number of cases and limited number of states in the reactive plan. A switch-case statement is necessary for this problem, with 13 cases in each box. We utilised 13 ITE statements to replicate the functionality of a switch-case statement. Thus, the synthesizer must only map a motion to each specific state.

3.6.4 Performance

The entire Atari Pitfall environment is split into 8 SyGuS benchmarks. One benchmark is to find the shortest path to the nearest treasure, and other benchmarks correspond to each box along the shortest path. CVC5 with a default parameter could solve these benchmarks within a half minute in total. The problem takes less time than grid-world problems. It could be explained for 2 reasons: the pathfinding is similar to a grid-world problem, but only with 6 steps; the specified ITE syntax in hazard avoidance benchmarks has significantly reduced the search space in the synthesis phase.

According to no free lunch theorem (NFL) [3], a stochastic optimisation algorithm cannot outperform an utterly random algorithm on every optimisation problem. Atari Pitfall is a challenging problem for many classic reinforcement learning agents [41]. However, we think a SyGuS solver could help RL agents by generating high-level plans. Compared to SyGuS, an RL agent cannot learn the high-level plan with 30 seconds of training, whereas SyGuS cannot find the pixel and frame level strategy. Combining SyGuS for high-level plans and RL for low-level actions could address the sparse reward problem.

A similar framework has been proposed recently [29], whereas their method utilised planning domain definition language (PDDL) [4]. Their experiments provide evidence that combining logical specifications and reinforcement learning is effective.

Chapter 4

Discussion of SyGuS and Experiment Results

This chapter will first discuss what OpenAI Gym environments we did not model and explain our decisions. Then we will discuss the experiment on benchmarks in Chapter 3. The last section of this chapter listed the pros and cons of SyGuS compared to other approaches.

4.1 Non-SyGuS-Compatible Problems

We have discussed why solving some agent problems involving stochasticity or optimisation without re-synthesising is not feasible. We did not model other OpenAI Gym environments due to the limitation of the commonly used SyGuS approaches, and this section discusses these environments.

4.1.1 Other Atari Environments

Many Atari games were designed to be non-deterministic so that players could have a different experience in each game. However, the SyGuS-IF language does not support arithmetic for probabilities, and finding a strategy that never fails is infeasible in many cases. For example, in the famous Pac-Man, an adversarial ghost moves faster than Pac-Man. The ghost moves in random directions. No matter where the Pac-Man is, there is always a chance for the ghost to reach Pac-Man. A human player or reinforcement learning strategy would try to minimise that probability, whereas a SyGuS solver will find the problem infeasible.

Most games aim to maximise the score, whereas this is impossible to prove when context-free grammar defines a recursive and unbounded search space. A game compatible with the standard SyGuS must not be stochastic and does not involve optimisation.

Modelling SyGuS-compatible Atari Environments is still challenging since all Atari games are black boxes. We cannot guarantee the specification is correct without knowing the source code. Thus, the synthesised function is not guaranteed to be correct.

4.1.2 All MuJoCo Environments

MuJoCo environments pose a massive challenge for solving and modelling due to their continuous nature. The torque applied to each robot joint in each coordinate is a floating number. Unlike agent problems with a discrete action space, we cannot include an infinite action space in our syntactic constraint. As an example, in cliff_walking, we included the action of moving left, which is unnecessary for the solution. Still, in a MuJoCo model, we cannot include every float32 number because most are unnecessary for the goal. For each floating number, we need a mathematical computation that returns it. However, we do not know the mathematical computation unless the reactive strategy is known.

Besides the problem of continuous action space, there is no bounded goal in all MuJoCo environments. We cannot prove an agent is "running as fast as possible forever", even if we could model them in the SyGuS-IF language. Moreover, Mujoco environments have random noises. In conclusion, MuJoCo environments possess almost every property that SyGuS cannot handle easily.

4.1.3 Toy Text: Blackjack

We did not model one Toy Text environment, Blackjack, a classical casino banking game. The environment is stochastic and bounded, and we think SyGuS can solve this problem by reducing it into a mathematical problem but doing so is unnecessary. If we specify the desired function to maximise the expectation of winning, the synthesised function is equivalent to a part of the logical specification.

In this game, the dealer has an infinite deck, and the game begins with the dealer having one card face-up (observed by all) and one card face-down (hidden). Meanwhile, the agent has two face-up cards. The agent can request more cards or stop at any point as long as the total sum of their cards does not exceed 21. The dealer will continue drawing cards until their sum reaches 17 or higher. If the dealer or the agent's sum exceeds 21, they lose the game. However, if neither exceeds 21, the winner is determined based on who has a sum closer to 21. If both of them exceed 21, the dealer wins.

Unlike the non-deterministic Frozen Lake problem, Blackjack is finite. The agent or the dealer cannot draw infinite times. In the luckiest scenario, an agent will have at most 21 Aces (Ace is 1 point or 10 points), whereas the agent in the Frozen Lake could always move in an unintended direction and not reach the goal. We could only define the accurate expectation when the stochastic problem is bounded.

Given the observed card and their fixed behaviour, we could compute the expectation $E(Sum_d)$ of the dealer's sum. We could also compute the probability P_r of the agent's sum greater than $E(Sum_d)$ given the agent requests extra cards. The agent's probability of exceeding 21 after requesting one extra card can be expressed using P_a, and the probability of the dealer exceeding 21 is P_d. Let the probability of winning if the agent requests an extra card be P_{win1}:

$$P_{win1} = (1 - P_a) \times (1 - (1 - P_d) \times (1 - P_r)) \tag{4.1}$$

If the agent stops to request extra cards, the probability of the agent's current sum greater than $E(Sum_d)$ is P_s, and the probability of winning is P_{win0}:

$$P_{win0} = 1 - (1 - P_d) \times (1 - P_s) \tag{4.2}$$

The synthesised function must choose an action that maximises the probability of winning given the current observation o, so the logical specification should be:

$$f(o) == ITE(P_{win1} \geq P_{win0}, 1, 0) \tag{4.3}$$

where 1 is requesting extra cards, and 0 is stopping. However, the desired function is the same as $ITE(P_{win1} \geq P_{win0}, 1, 0)$. In this case, the formal model is not meaningful in solving the problem.

4.1.4 All Classic Control Environments

Synthesis control is different from other agent problems. Proving an agent can reach a position is straightforward, whereas proving the agent will stay in that position forever is challenging. Modern commonly used SyGuS solvers cannot prove the reach-and-stay specification.

Researchers have been investigating the formal synthesis of control problems and have made significant advances in solving such problems using CEGIS, as evidenced in recent studies [1] [2]. In the literature, researchers used a special CEGIS algorithm, and the problem was split into two Single Input, Single Output (SISO) models. Generally, we consider synthesising control problems as a different research topic that is already well-studied.

4.1.5 All Box2D Environments

Box2D environments are highly configurable, and we could remove all random noises. Each environment is randomly initialised, and we still need a benchmark generator. Modelling Box2D environments is possible, whereas solving them using a generally used SyGuS solver is practically impossible. They are much more complex environments than Atari Pong: each environment's action space has at least 5 dimensions, there are more intermediate states before achieving the goal, and each state encodes more information (requires more variables to model). The state explosion is inevitable. Moreover, trigonometry was involved in every Box2D environment, which forced the solver to use an arithmetic theory that is harder to solve. We decided not to model Box2D environments because CVC5 failed to solve a similar but simpler problem.

4.2 CVC5 Performance on Benchmarks

4.2.1 Comparison with a Related Work

The relationship between the execution time of the SyGuS solver and the agent problem is studied and well explained in [15]. SyGuS solvers suffer from the state-space

explosion, and SyGuS for agent problems are generally not scalable. The synthesis time is exponential to the depth of AST and the number of time steps before reaching the goal.

Although the previous work used CVC4 in their experiment, our experiment results are consistent with the previous study's conclusion. Cliff Walking is intuitively simpler than Taxi, but this is not true for a SyGuS solver. The Cliff Walking agent must travel more steps to reach the goal, and the agent must change direction 2 times. In contrast, the agent in Taxi will change direction at most 2 times (depending on the random initialisation) and perform at most 10 actions. In a reactive agent strategy, the number of turning points in the path is proportional to the depth of AST. As shown in Figure 4.1, CVC5 takes more time to solve the Cliff Walking benchmark than the average synthesis time of multiple randomly generated Taxi benchmarks.

4.2.2 Comparison of Enumeration Strategies

In this project, we further investigated how the enumeration strategy of a CEGIS solver affects the synthesis time. We experiment on an Intel Core i7-9750H CPU 2.60GHz with 16 GB of RAM and Windows 10 OS. Only up to 20% of CPU power was occupied throughout running, but the RAM could be fully occupied when there is a state-space explosion.

5 enumeration strategies can be configured as a command line parameter: smart, fast, random, var-agnostic, and auto. Candidate programs are enumerated in different orders by different strategies. They are detailed explained in a thesis for CVC4 [34], here is a brief summary:

- **smart** enumeration sorts candidates based on datatype constraints. It rules out many redundant solutions.
- **fast** enumeration sorts all candidates purely based on their AST. It only rules out a few clearly redundant solutions.
- **random** enumeration is a pseudo-randomly generated sequence.
- **var-agnostic** enumeration is a hybrid of smart and fast enumerations. The technique is advantageous when there are many variables in the grammar.
- **auto** enumeration is the default option. It lets the solver decide the best enumerator for each SyGuS problem.

The grammar for the Cliff Walking, Taxi and Frozen Lake benchmarks are recursive, and searching a candidate randomly from an infinite search space is not desired. In our first experiment, the random enumeration spent more than 10 hours without solving a Taxi benchmark, so we decided not to include the random strategy in the latter experiments.

We only used deterministic Frozen Lake in our experiments, and there are two reasons: the experiment aims to compare CVC5's enumeration strategies, and online synthesis requires additional script; the execution time of our online synthesis is highly stochastic, and we cannot guarantee a high statistical significance.

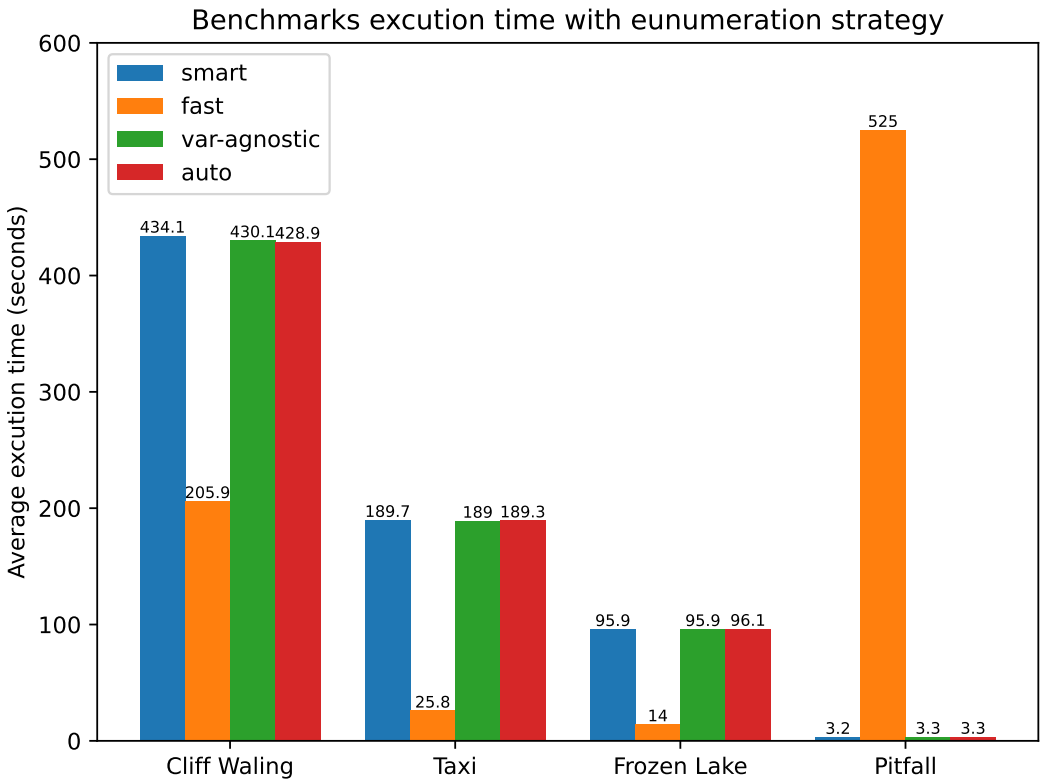

Figure 4.1: Experiment results: average running time on benchmarks.

We found that CVC5 cannot solve a grid world with 3 or more turning points in the optimal path. For example, the agent in Figure 3.3 must change direction 3 times to travel from the starting square to the goal square. This is due to the state-space explosion. The minimum depth of AST is directly related to the number of turning points, and the synthesis time rises quickly as the depth of AST increases. In the experiment, we found the smart enumeration strategy could not solve the specified environment shown in Figure 3.3 within 10 hours. Moreover, we found the fast enumeration strategy could not solve any Pitfall hazard avoidance benchmarks within a reasonable time. Thus, we set a 10 minutes time limit on all Frozen Lake and Pitfall experiments. All Cliff Walking and Taxi benchmarks have at most 2 turning points in their optimal paths. Thus we did not restrict the execution time in Cliff Walking and Taxi experiments.

The experiment results show that the smart, var-agnostic and auto enumeration strategies have close results. The fast enumeration strategy is significantly faster with grid-world problems and much slower than others in the Pitfall problem. A Pitfall problem combines one pathfinding benchmark with multiple hazard avoidance benchmarks, and the fast enumeration could solve the pathfinding benchmark quickly.

Enumeration strategies are designed for pruning the search space and reducing the number of invocations of the verifier [5]. By evaluating some properties of candidate functions (datatype or relation with variables), many expressions will be ignored for sharing the same property as a rejected candidate function. However, this enumeration process is computationally expensive sometimes, and it may be harmful to the synthesis

time, especially when the verification is computationally cheap, and the search space is narrow (so the pruning is less effective).

The fast enumeration strategy is a brute-force sort of all expressions by their syntactic complexity. In grid-world problems, all synthesised functions are uncomplicated, and only a few expressions are eliminated by other strategies. Hence, the most straightforward sorting method proves to be the swiftest in grid-world problems, and the it should be the preferred choice for such benchmarks in the auto enumeration strategy. The latter was created to solve most benchmarks faster on average. However, due to a shortage of resources in the field of reactive agent synthesis, the auto enumeration strategy may not always make the best decision in such situations.

The performance of the fast enumeration strategy is significantly worse in the pitfall benchmarks. Remember the specified syntactic structure in pitfall is a switch-case statement, and the depth of AST is fixed. The fast enumeration strategy will check at most a^{13} expressions, where a is the number of actions. In contrast, pruning the search space using other enumeration strategies significantly reduces this upper limit.

4.3 SyGuS and Reinforcement Learning

Many studies with OpenAI Gym have compared their results with previously studied RL methods. However, we will not compare the running time and performance with RL methods in this project. Instead, we will focus on explaining why SyGuS is different.

SyGuS cannot deal with black boxes, which makes the performance or the execution time of SyGuS and RL methods incomparable. All environments in OpenAI Gym are expected to be used as black boxes. During RL training, the agent was not supposed to know the semantics of the observations, and the semantics should be learned after several attempts and receiving rewards. For example, in the Atari Pong environment, the observation is a set of pixels, where the agent does not know which pixel indicates the ball or the paddle at the beginning, nor does it not know what will happen when the paddle hits the ball. In contrast, the background knowledge is learned by humans and specified as input to SyGuS solvers. SyGuS is a different type of problem, which is not comparable with OpenAI Gym problems. A rigorous comparison should be carried out with an algorithm that automatically rewrites an agent problem in OpenAI Gym into a SyGuS problem and then solves it with a SyGuS solver.

SyGuS guarantees to return a correct output, which differs from RL approaches. However, in most cases, the agent problem involves stochasticity and maximisation, and how to specify the synthesised function becomes a problem. Without specifying that the synthesised function should maximise the reward or expectation, the synthesised function is not guaranteed to outperform an RL solution.

In terms of the execution time, SyGuS problems are undecidable if the syntax contains an ITE statement [13]. However, an ITE statement is necessary for reactive agent problems. The efficiency of modern SyGuS solvers restricts the depth of AST. Any functions with a deep AST cannot be synthesised within a reasonable time. As a comparison, RL methods with deep neural networks can approximate any continuous

function [23]. SyGuS solvers are not expected to outperform RL methods on complex agent problems.

4.4 SyGuS and Genetic Programming

Genetic programming is an evolutionary algorithm that could also synthesise programs. Applying genetic programming on Atari games is also studied [41], whereas this approach is similar to RL approaches. An evolved program is not guaranteed to be correct but satisfies most input-output examples. The approach could deal with black boxes, as it only requires input-output samples, and it has the power to express very complex functions, including loops.

4.5 SyGuS and Automated planning

SyGuS is more similar to automated planning, with both methods specifying the background knowledge as a formal language. There are many solvers only focusing on automated planning based on PDDL. Because a SyGuS solver is designed to tackle a wide range of problems, the verification algorithm needs to perform better on average for all problems. This makes a SyGuS solver unlikely to outperform a PDDL solver in solving agent problems.

PDDL solvers usually return a sequence of actions, which differs from the function in SyGuS. Our benchmarks cannot show this difference. It can be demonstrated by randomised Cliff Walking, in which the agent's initial position is randomly generated. To model this environment, we only have to change the *correctStartingPosition* in the logical specification (3.3) into $(o_0 \leq 36 \wedge o_0 \geq 0) \vee (o_0 == 47)$, and the synthesised function would be the same as before. In contrast, this different version requires a sophisticated technique to specify in PDDL [19]. A more rigorous comparison requires experiments with PDDL models.

Chapter 5

Conclusions and Future works

In this project, we modelled 5 OpenAI Gym environments using the SyGuS-IF language and implemented a Python script for generating new benchmarks. While designing grid-world benchmarks, we followed the style of a previous work [15]. Besides reproducing the achievements in the previous work, we have also demonstrated the approach of applying SyGuS to stochastic problems, designed new benchmarks for problems that were not studied, and investigated why SyGuS is incompatible with other OpenAI Gym environments. Last, we evaluated the state-of-the-art SyGuS solver (CVC5) using our benchmark and studied why the enumeration strategy in a CEGIS solver affects the synthesis time.

5.1 Summary of Findings

SyGuS solvers can solve some simple agent problems in OpenAI Gym, which guarantees the correctness of the solution. If the logical specification is equivalent to maximising the reward, the synthesised function is guaranteed to be the best strategy that maximises the reward; RL and genetic programming are not guaranteed to be the best. PDDL approaches are more similar to SyGuS approaches in modelling the environment in formal language and guaranteeing it to be the best. But PDDL approaches focus more on sequential motion strategy. Applying SyGuS to agent problems can be beneficial, but only in minor cases.

If the specification is not guaranteed to be correct, then the synthesised function is not guaranteed to be correct. Generating specifications is difficult: nowadays, there is no practical approach to dealing with black boxes using SyGuS. Also, the SyGuS-IF language does not support arithmetic theory for probabilities and limits, and modelling stochastic agent problems is challenging.

One of the most significant limitations of modern SyGuS solvers is their efficiency. Although we can split an agent problem into several SyGuS problems, deciding the complexity of each sub-problem and where to split becomes a dilemma. Because of the undecidability of SyGuS problems with ITE statements, the synthesis time rapidly increases as the depth of AST increases. A complex function will not be synthesised

within a reasonable time, and a simple function could be equivalent to the logical specification sometimes (like the Blackjack environment). A SyGuS solver could synthesise a function quickly where there is no ITE in the syntactic constraint or the exact number of ITE is explicitly specified (like the hazard avoidance benchmark), whereas this is rarely the case for reactive agent problems.

In some complex agent problems, the decisions can be synthesised hierarchically, and a high-level plan can sometimes be expressed with a simple function. SyGuS could be combined with other probabilistic approaches and improve performance by generating a high-level plan that is guaranteed to be correct. In our Pitfall models, the high-level plan includes which direction the agent should go and what action should be performed to tackle each hazard. With this information, an RL agent only has to learn the exact pixel and frame number to act, and the difficulty of training is significantly reduced.

5.2 Limitations and Future Works

SyGuS is a vibrant research field, there are numerous recent signs of progress in improving SyGuS, and there will be more in the future. In this section, we will discuss what we can or could have done to improve the performance of SyGuS on reactive agent problems and suggest potential future works.

Knowing that the enumeration strategy affects the synthesis time significantly, a straightforward enumeration strategy that does not prune the search space effectively speeds up the synthesis in some grid-world problems. We could further improve the performance by designing a "state" enumeration strategy that prunes more search space and is computationally cheap. There are unsafe states in agent problems, where reaching these states indicates failure. By checking if the agent reaches an unsafe state in the first few steps, a large number of expressions could be ignored in the synthesis phase, and this checking process could be fast if it only focuses on the first few steps. As a piece of supporting evidence for our proposal, CVC5 spent more time solving the alternative benchmark in section 3.2.2, where stepping off the cliff was not linked directly to failure. A desired enumeration strategy would automatically establish the logical connection between stepping off the cliff and failure, resulting in the solver spending equivalent time on the benchmark in sections 3.2.1 and 3.2.2.

A recent study proposed an approach that synthesis probabilistic programs [14]. The approach is based on the PRISM language, which we did not consider until the late stage of the project. We could have modelled the non-deterministic Cliff Walking environment using PRISM and solved it with a probabilistic synthesis tool (e.g., PAYNT). The program synthesiser using PRISM may also suffer from the state-space explosion. This relationship between the agent problem and the execution time requires further examination. The probabilistic theory could be added to the SyGuS-IF language as a suggested future work. Developing new solvers dealing with the new theory will then become another challenge. This is a long-term research topic.

Combining SyGuS with modern deep-learning algorithms is another research topic full of potential. In the previous chapter, we discussed generating high-level plans with SyGuS and making detailed decisions using reinforcement learning approaches. We

could also apply human-in-the-loop (HITL) approaches to extract logical specifications from a black box environment. Specifying the background logic is a great challenge, and AI-generated logical specifications are not guaranteed to be correct, whereas the HITL system addresses both issues.

In Atari games, a machine-learning approach could learn a logical specification from gaming samples of human players. After that, the logical specification is checked and modified by a human user so that the logical specification is more likely to be correct. The system will then solve the specified SyGuS problem using a SyGuS solver. An RL approach could generate detailed decisions if the synthesised function is a high-level plan. This approach could potentially improve the usability of SyGuS, as well as mitigate the error rate in RL approaches.

Bibliography

[1] Alessandro Abate, Iury Bessa, Dario Cattaruzza, Lucas Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Automated formal synthesis of digital controllers for state-space physical plants. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pages 462–482. Springer, 2017.

[2] Alessandro Abate, Iury Bessa, Lucas Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Automated formal synthesis of provably safe digital controllers for continuous plants. *Acta Informatica*, 57(1-2):223–244, 2020.

[3] Stavros P Adam, Stamatios-Aggelos N Alexandropoulos, Panos M Pardalos, and Michael N Vrahatis. No free lunch theorem: A review. *Approximation and Optimization: Algorithms, Complexity and Applications*, pages 57–82, 2019.

[4] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. Pddl— the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

[5] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[6] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: Results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

[7] Roman Andriushchenko, Milan Češka, Sebastian Junges, Joost-Pieter Katoen, and Šimon Stupinskỳ. Paynt: a tool for inductive synthesis of probabilistic programs. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*, pages 856–869. Springer, 2021.

[8] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I*, pages 415–442. Springer, 2022.

[9] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

[10] Clark Barrett and Cesare Tinelli. *Satisfiability modulo theories*. Springer, 2018.

[11] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[12] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[13] Benjamin Caulfield, Markus N Rabe, Sanjit A Seshia, and Stavros Tripakis. What's decidable about syntax-guided synthesis? *arXiv preprint arXiv:1510.08393*, 2015.

[14] Milan Češka, Christian Hensel, Sebastian Junges, and Joost-Pieter Katoen. Counterexample-driven synthesis for probabilistic program sketches. In *Formal Methods–The Next 30 Years: Third World Congress, FM 2019, Porto, Portugal, October 7–11, 2019, Proceedings*, pages 101–120. Springer, 2019.

[15] Sarah Chasins and Julie L Newcomb. Using sygus to synthesize reactive motion plans. *arXiv preprint arXiv:1611.07620*, 2016.

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[17] Cristina David, Pascal Kesseli, and Daniel Kroening. Kayak: Safe semantic refactoring to java streams. *arXiv preprint arXiv:1712.07388*, 2017.

[18] Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, et al. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences*, 119(32):e2123433119, 2022.

[19] Vladimir Estivill-Castro and Jonathan Ferrer-Mestres. Path-finding in dynamic environments with pddl-planners. In *2013 16th International Conference on Advanced Robotics (ICAR)*, pages 1–7. IEEE, 2013.

[20] Ruben Giaquinta, Ruth Hoffmann, Murray Ireland, Alice Miller, and Gethin Norman. Strategy synthesis for autonomous agents using prism. In *NASA Formal Methods: 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings 10*, pages 220–236. Springer, 2018.

[21] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46(6):62–73, 2011.

[22] Sumit Gulwani, Alex Polozov, and Rishabh Singh. *Program Synthesis*, volume 4. NOW, August 2017.

[23] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[24] Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. Proving unrealizability for syntax-guided synthesis. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*, pages 335–352. Springer, 2019.

[25] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224, 2010.

[26] Susmit Jha and Sanjit A Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54:693–726, 2017.

[27] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[28] Shivaram Kalyanakrishnan, Siddharth Aravindan, Vishwajeet Bagdawat, Varun Bhatt, Harshith Goka, Archit Gupta, Kalpesh Krishna, and Vihari Piratla. An analysis of frame-skipping in reinforcement learning. *arXiv preprint arXiv:2102.03718*, 2021.

[29] Junkyu Lee, Michael Katz, Don Joven Agravante, Miao Liu, Geraud Nangue Tasse, Tim Klinger, and Shirin Sohrabi. Hierarchical reinforcement learning with ai planning models, 2022.

[30] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[31] Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[33] Saswat Padhi, Elizabeth Polgreen, Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. The sygus language standard version 2.1, 2014.

[34] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. cvc 4 sy: smart and fast term enumeration for syntax-guided synthesis. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*, pages 74–83. Springer, 2019.

[35] Taisuke Sato and Yoshitaka Kameya. Prism: a language for symbolic-statistical modeling. In *IJCAI*, volume 97, pages 1330–1339, 1997.

[36] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[37] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[38] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 281–294, 2005.

[39] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415, 2006.

[40] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 48(6):287–296, 2013.

[41] Dennis G Wilson, Sylvain Cussat-Blanc, Hervé Luga, and Julian F Miller. Evolving simple programs for playing atari games. In *Proceedings of the genetic and evolutionary computation conference*, pages 229–236, 2018.

[42] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M Murray. Receding horizon temporal logic planning. *IEEE Transactions on Automatic Control*, 57(11):2817–2830, 2012.

[43] Eiichi Yoshida, Kazuhito Yokoi, and Pierre Gergondet. Online replanning for reactive robot motion: Practical aspects. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5927–5933. IEEE, 2010.

Appendix A

Pathfinding Benchmark

```
(set-logic LIA)

(define-fun wall-left ((curr-position Int)) Bool
    (or (= curr-position (- 6)) (= curr-position (- 9))
      (= curr-position (- 10)) (= curr-position (- 11))
      (= curr-position (- 12)) (= curr-position (- 14))
      (= curr-position 4) ))

(define-fun wall-right ((curr-position Int)) Bool
    (or (= curr-position (- 5)) (= curr-position (- 13))
      (= curr-position 0) (= curr-position 1) ))

(define-fun ladder ((curr-position Int)) Bool
    (or (= curr-position (- 6)) (= curr-position (- 9))
      (= curr-position (- 10)) (= curr-position (- 11))
      (= curr-position (- 12)) (= curr-position (- 14))
      (= curr-position 4) (= curr-position (- 5)) (=
      curr-position (- 13)) (= curr-position 0) (=
      curr-position 1) ))

(define-fun treasure ((curr-position Int)) Bool
    (or (= curr-position (- 17)) (= curr-position 6)))

(define-fun next-position ((curr-position Int) (
   underground Bool) (way Int)) Int
    (ite (= way 0) curr-position
    (ite (and (=> underground (ladder curr-position)) (=
      way 1)) (+ curr-position 1)
    (ite (and (=> (not underground) (ladder curr-position
      )) (not (wall-right curr-position)) (= way 2)) (+
      curr-position 3)
    (ite (and (=> (not underground) (ladder curr-position
```

```
        )) (not (wall-left curr-position)) (= way 3)) (+
           curr-position (- 3))
      (ite (and (=> underground (ladder curr-position)) (=
         way 4)) (+ curr-position (- 1))
      curr-position))))))

(define-fun next-state ((curr-position Int) (underground
   Bool) (way Int)) Bool
    (ite (= way 0) underground
    (ite (and (=> underground (ladder curr-position)) (=
       way 1)) false
    (ite (and (=> (not underground) (ladder curr-position
       )) (not (wall-right curr-position)) (= way 2))
       true
    (ite (and (=> (not underground) (ladder curr-position
       )) (not (wall-left curr-position)) (= way 3)) true
    (ite (and (=> underground (ladder curr-position)) (=
       way 4)) false
    underground))))))

(synth-fun move ((curr-position Int) (underground Bool))
   Int
    ((Start Int) (MoveId Int) (CondInt Int) (StartBool
       Bool))
    ((Start Int (
        MoveId
        (ite StartBool Travel Travel)
        ))
    (MoveId Int (0
        1
        2
        3
        4
        ))
    (CondInt Int (
        curr-position
        (- 1) (- 2) (- 3) (- 4) (- 5) (- 6) (- 7) (- 8)
           (- 9) (- 10)
        (- 11) (- 12) (- 13) (- 14) (- 15) (- 16) (- 17)
        0 1 2 3 4 5
    ))
    (StartBool Bool ((and StartBool StartBool)
        underground
        (wall-left curr-position)
        (wall-right curr-position)
        (ladder curr-position)
```

```
        (or StartBool StartBool)
        (not StartBool)
        (<= CondInt CondInt)
        (= CondInt CondInt)))))

(declare-var pos0 Int)
(declare-var under0 Bool)
(declare-var mov0 Int)
(declare-var pos1 Int)
(declare-var under1 Bool)
(declare-var mov1 Int)
......
(declare-var pos20 Int)
(declare-var under20 Bool)
(declare-var mov20 Int)

(constraint
(=>
(and (= pos0 0) (not under0) (= mov0 (move pos0 under0))
     (= pos1 (next-position pos0 under0 mov0)) (= under1
        (next-state pos0 under0 mov0)) (= mov1 (move pos1
         under1))
     (= pos2 (next-position pos1 under1 mov1)) (= under2
        (next-state pos1 under1 mov1)) (= mov2 (move pos2
         under2))
     (= pos3 (next-position pos2 under2 mov2)) (= under3
        (next-state pos2 under2 mov2)) (= mov3 (move pos3
         under3))
     ......
     (= pos20 (next-position pos19 under19 mov19)) (=
        under20 (next-state pos19 under19 mov19)) (=
        mov20 (move pos20 under20))
) (and (= mov6 0) (treasure pos6))))


(check-synth)
```

The *move* function is the synthesised function f in Chapter 3, pos0, pos1, pos2... refers to o_t, mov0, mov1, mov2... refers to m_t, and under0, under1, under 2 refers to *isUnder_t*.

Appendix B

Hazard Avoidance Benchmark

```
(set-logic LIA)

; up left end
(define-fun state0 ((three-logs Bool) (move Int)) Int
    (ite (or (and three-logs (= move 5)) (and (not
       three-logs) (= move 1))) 1 0))

; up left
(define-fun state1 ((three-logs Bool) (pit Bool) (
   quicksand Bool) (crocodile Bool) (ladder Bool) (
   ladder-and-holes Bool) (swinging-vine Bool) (move Int)
   ) Int
    (ite (or (and (not pit) ladder (= move 5))
    (and (or crocodile (and (not pit) ladder-and-holes))
       (= move 7))
    (and (not pit) quicksand (= move 1))) 2
    (ite (and swinging-vine (= move 5)) 11
    (ite (or (and three-logs (= move 6)) (and (not
       three-logs) (= move 3))) 0
    (ite (or (and ladder (= move 1)) (and
       ladder-and-holes (= move 5))) 4 1)))))

; up right
(define-fun state2 ((three-logs Bool) (snake Bool) (pit
   Bool) (quicksand Bool) (crocodile Bool) (ladder Bool)
   (ladder-and-holes Bool) (swinging-vine Bool) (move Int
   )) Int
    (ite (or (and (not pit) ladder (= move 6))
    (and (or crocodile (and (not pit) ladder-and-holes))
       (= move 7))
    (and (not pit) quicksand (= move 3))) 1
    (ite (and swinging-vine (= move 6)) 12
```

```
    (ite (or (and (or three-logs snake) (= move 5)) (and
       (not (or three-logs snake)) (= move 1))) 3
    (ite (or (and ladder (= move 3)) (and
       ladder-and-holes (= move 6))) 5 2)))))

; up right end
(define-fun state3 ((three-logs Bool) (snake Bool) (move
   Int)) Int
    (ite (or (and (or three-logs snake) (= move 6)) (and
       (not (or three-logs snake)) (= move 3))) 2 3))

; up left middle
(define-fun state4 ((ladder Bool) (ladder-and-holes Bool)
    (move Int)) Int
    (ite (and (or ladder ladder-and-holes) (= move 2)) 6
    (ite (and (or ladder ladder-and-holes) (= move 5)) 5
    (ite (or (and ladder (= move 3)) (and
       ladder-and-holes (= move 6))) 1 4))))

; up right middle
(define-fun state5 ((ladder Bool) (ladder-and-holes Bool)
    (move Int)) Int
    (ite (and (or ladder ladder-and-holes) (= move 2)) 6
    (ite (or (and ladder (= move 1)) (and
       ladder-and-holes (= move 5))) 2
    (ite (or (and ladder (= move 3)) (and
       ladder-and-holes (= move 6))) 4 5))))

;ladder
(define-fun state6 ((ladder Bool) (ladder-and-holes Bool)
    (scorpion Bool) (move Int)) Int
    (ite (and (or ladder ladder-and-holes) (= move 4)) 5
    (ite (and (not scorpion) (= move 1)) 9
    (ite (and (not scorpion) (= move 3)) 8 6))))

;down left end
(define-fun state7 ((wall-left Bool) (move Int)) Int
    (ite (and (not wall-left) (= move 1)) 8 7))

;down left
(define-fun state8 ((wall-left Bool) (ladder Bool) (
   ladder-and-holes Bool) (scorpion Bool) (move Int)) Int
    (ite (and (not wall-left) (= move 3)) 7
    (ite (or (and scorpion (= move 5)) (and (not scorpion
       ) (= move 1))) 9
    (ite (and (or ladder ladder-and-holes) (= move 4)) 6
```

```
        8))))

;down right
(define-fun state9 ((wall-right Bool) (ladder Bool) (
   ladder-and-holes Bool) (scorpion Bool) (move Int)) Int
    (ite (and (not wall-right) (= move 1)) 10
    (ite (or (and scorpion (= move 6)) (and (not scorpion
       ) (= move 3))) 8
    (ite (and (or ladder ladder-and-holes) (= move 4)) 6
       9))))

;down right end
(define-fun state10 ((wall-right Bool) (move Int)) Int
    (ite (and (not wall-right) (= move 3)) 9 10))

; swinging-vine left
(define-fun state11 ((swinging-vine Bool) (move Int)) Int
    (ite (and swinging-vine (= move 1)) 1
    (ite (and swinging-vine (= move 0)) 12 11)))

; swinging-vine right
(define-fun state12 ((swinging-vine Bool) (move Int)) Int
    (ite (and swinging-vine (= move 2)) 2
    (ite (and swinging-vine (= move 0)) 11 12)))

(define-fun interpret-move ((curr-state Int) (three-logs
   Bool) (snake Bool) (pit Bool) (quicksand Bool) (
   crocodile Bool) (ladder Bool) (ladder-and-holes Bool)
   (swinging-vine Bool) (scorpion Bool) (wall-left Bool)
   (wall-right Bool) (move Int)) Int
    (ite (= curr-state 0) (state0 three-logs move)
    (ite (= curr-state 1) (state1 three-logs pit
       quicksand crocodile ladder ladder-and-holes
       swinging-vine move)
    (ite (= curr-state 2) (state2 three-logs snake pit
       quicksand crocodile ladder ladder-and-holes
       swinging-vine move)
    (ite (= curr-state 3) (state3 three-logs snake move)
    (ite (= curr-state 4) (state4 ladder ladder-and-holes
        move)
    (ite (= curr-state 5) (state5 ladder ladder-and-holes
        move)
    (ite (= curr-state 6) (state6 ladder ladder-and-holes
        scorpion move)
    (ite (= curr-state 7) (state7 wall-left move)
    (ite (= curr-state 8) (state8 wall-left ladder
```

```
        ladder-and-holes scorpion move)
    (ite (= curr-state 9) (state9 wall-left ladder
        ladder-and-holes scorpion move)
    (ite (= curr-state 10) (state10 wall-right move)
    (ite (= curr-state 11) (state11 swinging-vine move)
    (ite (= curr-state 12) (state12 swinging-vine move)
        (- 1)
    ))))))))))))))

(synth-fun move ((curr-state Int)) Int
    ((Start Int) (MoveId Int))
    ((Start Int (
        (ite (= curr-state 0) MoveId
        (ite (= curr-state 1) MoveId
        ......
        (ite (= curr-state 12) MoveId (- 1)
        ))))))))))))))
        (MoveId Int (0 ; stay
        1 ; walk right
        2 ; walk left
        3 ; climb down
        4 ; climb up
        5 ; jump right
        6 ; jump left
        7 ; 3 jumps (bidirectional)
        ))
    ))

(declare-var pos1 Int)
(declare-var mov1 Int)
(declare-var pos2 Int)
(declare-var mov2 Int)
......
(declare-var pos12 Int)
(declare-var mov12 Int)

(declare-var snake Bool)
(declare-var swinging-vine Bool)
(declare-var scorpion Bool)
(declare-var crocodile Bool)
(declare-var pit Bool)
(declare-var ladder Bool)
(declare-var ladder-and-holes Bool)
(declare-var wall-left Bool)
(declare-var wall-right Bool)
(declare-var quicksand Bool)
```

```
(declare-var three-logs Bool)

(constraint
(=>
(and (= pos1 0) (= mov1 (move pos1))
     (= pos2 (interpret-move pos1 three-logs snake pit
        quicksand crocodile ladder ladder-and-holes
        swinging-vine scorpion wall-left wall-right mov1)
        ) (= mov2 (move pos2))
     (= pos3 (interpret-move pos2 three-logs snake pit
        quicksand crocodile ladder ladder-and-holes
        swinging-vine scorpion wall-left wall-right mov2)
        ) (= mov3 (move pos3))
    ......
     (= pos12 (interpret-move pos11 three-logs snake pit
        quicksand crocodile ladder ladder-and-holes
        swinging-vine scorpion wall-left wall-right mov11
        )) (= mov12 (move pos12))
     snake (not swinging-vine) (not scorpion) (not
        crocodile) (not pit) ladder (not ladder-and-holes
        ) (not wall-left) wall-right (not quicksand) (not
         three-logs)
) (= pos12 3)))


(check-synth)
```