InnateCoder: Learning Programmatic Options with Foundation Models

Rubens O. Moraes[1] , **Quazi Asif Sadmine**[2,3] , **Hendrik Baier**[4,5] and **Levi H. S. Lelis**[2,3]
[1]Departamento de Informática, Universidade Federal de Viçosa
[2]Department of Computing Science, University of Alberta
[3]Alberta Machine Intelligence Institute (Amii)
[4]Information Systems, Eindhoven University of Technology
[5]Centrum Wiskunde & Informatica, Amsterdam

Abstract

Outside of transfer learning settings, reinforcement learning agents start their learning process from a clean slate. As a result, such agents have to go through a slow process to learn even the most obvious skills required to solve a problem. In this paper, we present INNATECODER, a system that leverages human knowledge encoded in foundation models to provide programmatic policies that encode "innate skills" in the form of temporally extended actions, or options. In contrast to existing approaches to learning options, INNATECODER learns them from the general human knowledge encoded in foundation models in a zero-shot setting, and not from the knowledge the agent gains by interacting with the environment. Then, INNATECODER searches for a programmatic policy by combining the programs encoding these options into larger and more complex programs. We hypothesized that INNATECODER's way of learning and using options could improve the sampling efficiency of current methods for learning programmatic policies. Empirical results in MicroRTS and Karel the Robot support our hypothesis, since they show that INNATECODER is more sample efficient than versions of the system that do not use options or learn them from experience.

1 Introduction

Deep reinforcement learning (DRL) agents typically begin their training process with randomly initialized neural networks. As a result, they must learn from scratch even the most basic skills required to solve a problem. Due to the high sample complexity of DRL algorithms, such *tabula rasa* learning can be inefficient and time-consuming, which has inspired several research directions aiming at reusing prior computation [Zhu *et al.*, 2023; Khetarpal *et al.*, 2022].

In this paper, we harness the general human knowledge encoded in foundation models to endow agents with helpful skills before they even start interacting with the environment. This is achieved by using programmatic representations of policies [Trivedi *et al.*, 2021]—programs written in a domain-specific language encoding policies—and the foundation models' ability to write computer programs. Depending on the language used, programmatic policies can generalize better to unseen scenarios and be human-interpretable [Verma *et al.*, 2018; Bastani *et al.*, 2018]. In addition to these advantages and loosely inspired by the innate abilities of animals [Tinbergen, 1951], we show that these representations of policies allow us to harness helpful "innate skills" from foundation models.

Given a natural-language description of the problem that the agent needs to learn to solve, our system, which we call INNATECODER, queries a foundation model for programs that encode policies to solve the problem. Although the programs the model generates are unlikely to encode policies that solve the problem, we hypothesize that the set of sub-programs we obtain from these programs can encode helpful temporally extended actions, or options [Sutton *et al.*, 1999]. We consider options as functions the agent can call and that will tell it how to act for a number of steps [Precup *et al.*, 1998].

Options can improve the agent's learning in different ways. They can allow the agent to better explore the problem space [Machado *et al.*, 2017] or transfer knowledge between different tasks [Konidaris and Barto, 2007]. This paper presents a novel way of learning options with foundation models. We also present a novel way of using the learned options, which is inspired by recent work on learning semantic spaces of programming languages [Moraes and Lelis, 2024]. In the semantic space, neighbor programs encode similar but different agent behavior, which can be conducive to algorithms searching for programmatic policies [Trivedi *et al.*, 2021]. Instead of searching only in the space of programs induced by the syntax of the language, INNATECODER also searches in the semantic space induced by options. While previous methods can benefit from up to hundreds of options [Eysenbach *et al.*, 2019], INNATECODER's use of programmatic options allows it to benefit from thousands of them.

INNATECODER's approach to harnessing options from foundation models contrasts with previous approaches to automatically learning them, e.g., [Tessler *et al.*, 2017; Bacon *et al.*, 2017; Igl *et al.*, 2020; Klissarov and Machado, 2023]. This is because options are harnessed from the general knowledge encoded in a foundation model, as opposed to the knowledge the agent gains by interacting with the environment. This zero-shot approach to learning options is enabled by the use of a domain-specific language to bridge the gap between the high-level knowledge encoded in foundation models and the low-level knowledge required at the sensorimotor

$\rho :=$ `if` h `then` a
$h :=$ `frontIsClear` | `markersPresent`
$a :=$ `move` | `putMarker` | `pickMarker`

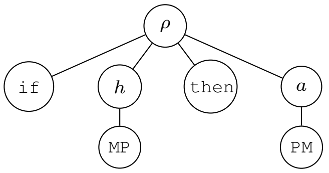

Figure 1: Top: Context-free grammar specifying a simplified version of the domain-specific language for Karel the Robot. Bottom: Abstract syntax tree for `if markersPresent then pickMarker`, where `MP` and `PM` stand for `markersPresent` and `pickMarker`, respectively. Karel is a robot acting on a grid, where it needs to accomplish tasks such as collecting and placing markers on different locations of the grid. In this program, Karel will pick up a marker if one is present in its current location on the grid.

control level of the agent [Klissarov *et al.*, 2024]. For example, foundation models trained on Internet data likely encode the knowledge that, to win a match of a real-time strategy game, the player must collect resources and build structures, which will allow for the training of the units needed to win the game. However, the model cannot issue low-level actions in real time to control dozens of units to accomplish this plan. INNATECODER helps bridge this gap by distilling the knowledge of the model into options that can be executed in real time.

We evaluated our hypothesis that foundation models can generate helpful programmatic options in the domains of MicroRTS, a challenging real-time strategy game [Ontañón, 2017], and Karel the Robot [Pattis, 1994], a benchmark for program synthesis and reinforcement learning algorithms [Chen *et al.*, 2018; Trivedi *et al.*, 2021]. The results in both domains support our hypothesis, since INNATECODER was more sample-efficient than versions of the system that do not use options or learn options from experience. We also show that the policies INNATECODER learns are competitive and often outperform the current state-of-the-art algorithms. INNATECODER is inexpensive because it uses the foundation model a small number of times as a pre-processing step, making it an accessible system to smaller labs and companies.[1]

2 Problem Definition

We consider sequential decision-making problems formulated as Markov decision processes (MDPs) $(S, A, p, r, \mu, \gamma)$. Here, S is the set of states and A the set of actions. The function $p(s_{t+1}|s_t, a_t)$ is the transition model, which gives the probability of reaching state s_{t+1} given that the agent is in s_t and takes action a_t at time step t. The agent observes a reward value of R_{t+1} when transitioning from s_t to s_{t+1}. The reward value the agent observes is returned by the function r. μ is the distribution of the initial states of the MDP; states sampled from μ are denoted s_0. γ in $[0, 1]$ is the discount factor. A policy π is a function that receives a state s and returns a probability distribution over actions available at s. The goal is to learn a policy π that maximizes the expected sum of discounted rewards for π starting in an s_0: $\mathbb{E}_{\pi,p,\mu}[\sum_{k=0}^{\infty} \gamma^k R_{k+1}]$. $V^\pi(s) = \mathbb{E}_{p,\pi}[\sum_{k=0}^{\infty} \gamma^k R_{k+1}|s_0 = s]$ is the value function that measures the expected return for policy π starting from s. In this work, we approximate the value function of a policy π and state s with Monte Carlo roll-outs and denote the approximation as $\hat{V}^\pi(s)$.

We consider programmatic representations of policies, which are policies written in a domain-specific language (DSL). The set of programs a DSL accepts is defined through a context-free grammar (M, N, R, I), where M, N, R, and I are the sets of non-terminals, terminals, the production rules, and the grammar's initial symbol, respectively. Figure 1 shows a DSL for a simplified version of the language we use for Karel the Robot (the complete DSL is shown in Appendix H of an extended version of this paper [Moraes *et al.*, 2025]). In this DSL, the set M is composed of symbols ρ, h, a, while the set N includes the symbols `if`, `frontIsClear`, `markersPresent`, `move`, `putMarker`, `pickMarker`. R are the production rules (e.g., $h \rightarrow$ `frontIsClear`), and ρ is the initial symbol. We denote programmatic policies with letters p and n and their variations such as n' and n^*.

We represent programs as abstract syntax trees (AST), where each node n and its children represent a production rule if n represents a non-terminal symbol. For example, the root of the tree in Figure 1 represents the non-terminal ρ, while node ρ and its children represent the production rule $\rho \rightarrow$ `if` h `then` a. Leaf nodes in the AST represent terminal symbols. Figure 1 shows an example of an AST for the program `if markersPresent then pickMarker`. A DSL D defines the possibly infinite space of programs $[\![D]\!]$, where in our case each program p in $[\![D]\!]$ represents a policy.

Given a domain-specific language D, our task is to find a programmatic policy $p \in [\![D]\!]$ that maximizes the expected sum of discounted rewards for a given MDP.

3 INNATECODER

Figure 2 shows a schematic view of INNATECODERwith its three components: one for learning options, another that uses the options to induce an approximation of the semantic space of the DSL, and a component to search in such a space. Next, we explain these components.

3.1 Learning Options

An option is a program encoding a policy that the agent can invoke in specific states. Once invoked, the program tells the agent what to do for a number of steps. Once completed, the option "returns" the control to the agent. An option ω is defined with a tuple $(I_\omega, \pi_\omega, T_\omega)$, where I_ω is the set of states in which the option can be initiated; π_ω is the policy the agent follows once ω starts; T_ω is a function that returns the probability in which ω terminates at a given state. INNATECODER learns programs, written in a given DSL, encoding options. The programs receive a state of the MDP and return the action the agent should take at that state, thus encoding π_ω.

We assume that the set I_ω is the set S of all states of the MDP, which means that the program can be invoked in any

[1] INNATECODER is available at https://github.com/rubensolv/InnateCoder.

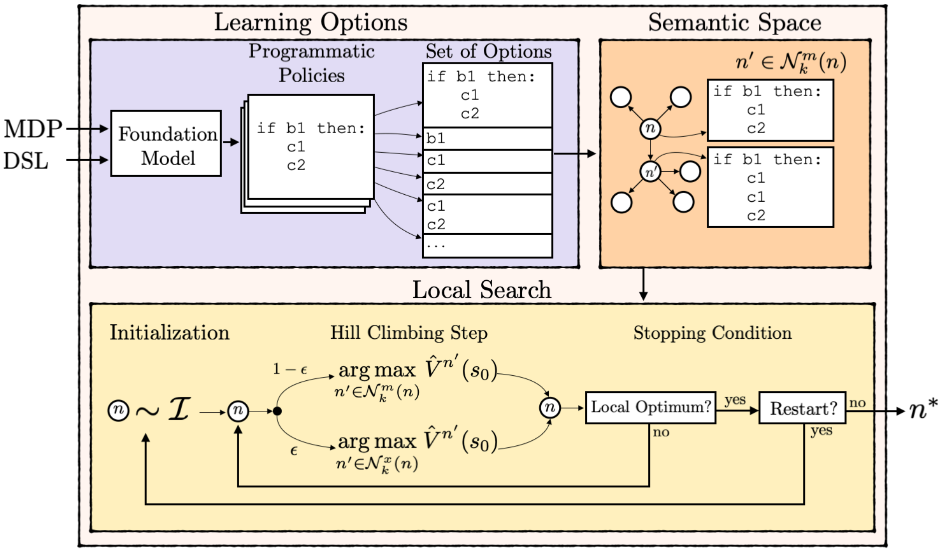

Figure 2: InnateCoder has three parts. "Learning Options" harnesses options from a foundation model. The model generates a set of programmatic policies that are broken into a set of options (Section 3.1). "Semantic Space" uses the options to induce the semantic space of the DSL (Section 3.2). "Local Search" searches in a mixture of the syntax and semantic spaces for a policy n^* (Section 3.3).

state. However, note that the program may not return any actions in a given state s, which is equivalent to s not being in I_ω. For example, "if b_1 then c_1" returns the action given in c_1 only if condition b_1 is satisfied in the state in which the option was queried; it returns no action for states that do not satisfy b_1. The option termination criterion T_ω is also determined by the program; the option terminates when the program terminates. This termination criterion means that, depending on the DSL, options have an internal state, representing the line in the program in which the execution will continue the next time the agent interacts with the environment. For example, if the option "c_1 c_2" is invoked for state s_t and c_1 returns an action, then the agent's action in s_{t+1} is determined by c_2.

Programmatic options are harnessed from a foundation model as follows. We provide a natural language description of the MDP and the Backus-Naur form of the DSL to the model. The model then provides a set of m programs written in the DSL encoding policies for the MDP. While it is only rarely that the model generates policies that maximize the expected return of the MDP, we hypothesize that these programmatic policies can be broken up into sub-programs that can encode options. Each program p is broken into one sub-program for each sub-tree rooted at a non-terminal symbol in the AST of p. For example, for the program "if b_1 then c_1 c_2" we obtain the sub-programs "if b_1 then c_1 c_2", "b_1", "c_1", "c_2", and "c_1 c_2". These sub-programs form a set of options O, which InnateCoder uses to approximate the semantic space of the DSL. Note that this set of options is generated zero-shot, before the agent starts interacting with the environment.

3.2 Inducing Semantic Spaces with Options

Methods for searching for programmatic policies traditionally search in the space of programs defined by the context-free grammar of the DSL [Verma *et al.*, 2018; Carvalho *et al.*, 2024]. We refer to this type of space as the syntax space, since it is based on the syntax of the language.

Definition 1 (Syntax Space)**.** *The syntax space of a DSL D is defined by $(D, \mathcal{N}_k^x, \mathcal{I}, \mathcal{E})$. With $[\![D]\!]$ defining the set of candidate programs, or solutions, $\mathcal{N}_k^x$ (x is for "syntax") is the syntax neighborhood function that receives a candidate and returns k candidates from $[\![D]\!]$. $\mathcal{I}$ is the distribution of initial candidates. Finally, $\mathcal{E}$ is the evaluation function, which receives a candidate in $[\![D]\!]$ and returns a value in $\mathbb{R}$.*

We define the distribution of initial candidates $\mathcal{I}$ through a procedure that starts with a string that is the initial symbol of the grammar and iteratively, and uniformly at random, samples a production rule to replace a non-terminal symbol in the string. In the example of Figure 1, we replace the initial symbol ρ with `if(`h`) then `a with probability 1.0, since this is the only rule available; then, h is replaced with `frontIsClear` or `markersPresent` with probability 0.5 each. This iterative process stops once the string contains only terminal symbols.

The syntax neighborhood function $\mathcal{N}_k^x$ defines the structure of the search space, as it determines the set of candidate solutions that the search procedure can evaluate from a given candidate n. Given a candidate n, $\mathcal{N}_k^x(n)$ returns a set of k neighbors of n. These neighbors are generated by selecting, uniformly at random, a node that represents a non-terminal symbol in the AST of n, and then replacing the sub-tree rooted at the selected node by another randomly generated sub-tree. This process is repeated k times, to generate k possibly different neighbors of n. Finally, $\mathcal{E}$ is an approximation of the value function of the policy encoded in n from a set of initial states s_0, $\hat{V}^n(s_0)$; we obtain $\hat{V}^n(s_0)$ by averaging the returns after rolling n out from states s_0.

Moraes and Lelis [2024] showed that searching in the syntax space can be inefficient because often the neighbors n' of a candidate n encode policies that are semantically identical to n; the programs differ in terms of syntax, but encode the same behavior. As a result, the search process wastes time evaluating the same agent behavior. They approximate the underlying semantic space of the DSL, where neighbor programs are syntactically similar but differ in terms of behavior. In their setting, the agent learns programs for a set of tasks, which are used to induce the semantic space, and the induced space is used in downstream tasks. INNATECODER overcomes the requirement to operate on a stream of problems by using a foundation model to learn the options in a zero-shot setting.

Definition 2 (Semantic Space)**.** *The semantic space of a DSL D is defined by $(D, \mathcal{N}_k^m, \mathcal{I}, \mathcal{E})$, where $\mathcal{I}$ and $\mathcal{E}$ are identical to the syntax space (Definition 1). The function $\mathcal{N}_k^m$ (m is for "semantics") is a semantic neighborhood function that also receives a candidate and returns k candidates from $[\![D]\!]$.*

We define the function $\mathcal{N}_k^m$ with a set of options Ω, where each option ω in Ω represents a different agent behavior. A neighbor of candidate n is obtained by selecting, uniformly at random, a node c in the AST of n that represents a non-terminal symbol. Then, we replace the sub-tree rooted at c with the AST of an option ω in Ω. The option ω is selected, uniformly at random, among those in Ω whose AST root represents the same non-terminal symbol c represents. By matching the non-terminal symbols when selecting ω, we match ω with the type of the sub-tree that is removed from n. Similarly to $\mathcal{N}_k^x$, $\mathcal{N}_k^m$ generates k possibly different neighbors by repeating this process k times. We require the options in Ω to encode different agent behaviors to increase the chance of sampling neighbors with different behaviors.

We filter the set of options O harnessed from the foundation model into a set Ω of behaviorally different options. First, we repeatedly sample an option $o \in O$, roll it out once in the environment (by playing itself in MicroRTS) from an initial state s_0 sampled from μ, and collect all states observed in which the agent gets to act into a vector of states $\mathcal{S}$. We do this until we have at least 300 and at most 700 states in $\mathcal{S}$.[2] Second, we test the options' behaviors by calling every option $o \in O$ once for each state $s \in \mathcal{S}$. We then collect the actions o returns into a vector called an *action signature* A_o for the option o. The i-th entry of A_o corresponds to the action o chooses for the i-th state in $\mathcal{S}$. We use these action signatures to characterize the option's behavior. Finally, in order to form a set Ω of behaviorally different options, we keep only one option from O, arbitrarily selected, for each observed A_o. Note that this assumes discrete action spaces. Future research will investigate different ways of measuring behavior in continuous action spaces. Also, filtering O into Ω uses less than 1% of the computation in our experiments. Programs that cannot be rolled out (e.g., Boolean expressions cannot issue actions) are not included in the set of options.

3.3 Searching in Semantic Space

INNATECODER uses stochastic hill-climbing (SHC) to search in the semantic space of a given DSL D for a policy that maximizes the agent's return. SHC starts its search by sampling a candidate program n from $\mathcal{I}$. In every iteration, SHC evaluates all k neighbors of n in terms of their $\mathcal{E}$-value. The search then moves on to the best neighbor of n in terms of $\mathcal{E}$, and this process is repeated from there. SHC stops if none of the neighbors has an $\mathcal{E}$-value that is better than the current candidate, that is, it reaches a local optimum. SHC uses a restarting strategy: once SHC reaches a local optimum, if SHC has not yet exhausted its search budget, it restarts from another initial candidate sampled from $\mathcal{I}$. SHC returns the best solution, denoted n^*, encountered in all restarts of the search.

INNATECODER does not search solely in the semantic space, but mixes both syntax and semantic spaces in the search. This is because the set of options might cover only a part of the space of programs the DSL induces. To guarantee that INNATECODER can access all programs in $[\![D]\!]$, with probability ϵ, SHC uses the syntax neighborhood function in the search, and with probability $1 - \epsilon$, it uses the semantic one. We use $\epsilon = 0.4$ in our experiments. We chose this value because it performed better in preliminary experiments.

4 Empirical Evaluation

Although foundation models are unlikely to generate programs that encode strong policies, we hypothesize that the programs they generate can be broken into smaller programs encoding helpful options. We evaluated our hypothesis by measuring the sampling efficiency of algorithms searching in the semantic spaces induced by them. We evaluated INNATECODER on MicroRTS [Ontañón, 2017] and Karel the Robot [Pattis, 1994].

MicroRTS We use the following maps from the MicroRTS repository,[3] with the map size in brackets: NoWhereToRun (9×8), basesWorkers (24×24), and BWDistantResources (32×32), and BloodBath (64×64). We use these maps because they differ in size and structure. Since MicroRTS is a multi-agent problem, we use 2L, a self-play algorithm, to learn programmatic policies [Moraes *et al.*, 2023]. In the context of 2L, INNATECODER is required to solve an MDP in every iteration of self-play (see Appendix M of the extended paper [Moraes *et al.*, 2025]). We use a new version of the MicroLanguage as the DSL [Mariño *et al.*, 2021]. The language offers specialized functions and an action-prioritization scheme through for-loops, where nested for-loops allow for higher priority of actions. We provide a detailed explanation of the MicroLanguage, as well as images of the maps used, in Appendices H and K [Moraes *et al.*, 2025].

Karel We use the following Karel problems, from previous works [Trivedi *et al.*, 2021; Liu *et al.*, 2023]: StairClimber, FourCorners, TopOff, Maze, CleanHouse, Harvester, DoorKey, OneStroke, Seeder, and Snake. The problems are described in Appendix I [Moraes *et al.*, 2025]. We use the more difficult version of the environment known as "crashable" [Carvalho *et al.*, 2024], where an episode terminates with a negative reward if Karel bumps into a wall. We use the same DSL used in previous work [Trivedi *et al.*, 2021], which we describe in Appendix H [Moraes *et al.*, 2025].

Baselines The current state-of-the-art methods for both MicroRTS and Karel use programmatic representations of policies, where the policies are written in the DSLs we use in our experiments [Moraes *et al.*, 2023; Trivedi *et al.*, 2021].

[2] We chose these values so that we could approximate the behavior of the options well while being computationally reasonable.

[3] https://github.com/Farama-Foundation/MicroRTS/

Therefore, we focus on methods that use programmatic representations as baselines. However, we provide comparisons of INNATECODER with deep reinforcement learning baselines in Appendices B.1 (MicroRTS) and B.2 (Karel) of the extended paper [Moraes *et al.*, 2025]. For both MicroRTS and Karel we use SHC searching in the syntax space as a baseline, as it represents state-of-the-art performance in both domains in our non-transfer setting (**SHC**). We also use two variants of INNATECODER where the options are learned without the help of a foundation model. These variants can be seen as implementations of the Library-Induced Semantic Spaces (LISS) [Moraes and Lelis, 2024] for non-transfer settings. In the first variant, LISS learns the options as it learns to solve the problem. In terms of the scheme shown in Figure 2, we skip the "Learning Options" step and build the set of options from the programs returned in every complete search of SHC. When we reach the box "Restart?", we use the sub-programs of the best program encountered in that search to augment the set of options. We call this baseline **LISS-o**, where "o" stands for "online". In the second variant, we sample programs from $\mathcal{I}$ and use their sub-programs as options. We call this baseline **LISS-r**, where "r" stands for "random". We also use the best program the foundation model generated from all programs used to create the set of options as a baseline called **FM**. LISS-o and LISS-r allow us to evaluate the effectiveness of learning options from a foundation model, while FM allows us to evaluate the foundation model as an alternative to solve the problem directly. We also use the Cross Entropy Method (**CEM**), which outperformed DRL algorithms in Karel [Trivedi *et al.*, 2021].

Foundation Models We use OpenAI's API for GPT 4o, whose training cut-off date is October 2023. We also perform tests, for MicroRTS, using the LLama 3.1 model with 405 billion parameters, whose training cut-off is December 2021. We used the GPT model in both the MicroRTS and Karel experiments, while the Llama model was used in the MicroRTS experiments. There were no MicroRTS programs available online prior to the Llama cut-off date, so the Llama evaluations on MicroRTS did not suffer from data leakage. The GPT model might have trained on the MicroRTS and Karel programs that were available online prior to its training cut-off date. We attempt to measure how much a possible data leakage can influence our results by using the FM baseline. If the model can simply retrieve the solutions seen in training, one would expect this baseline to perform well.

Other Specifications All experiments were run on 2.6 GHz CPUs with 12 GB of RAM. We use $k = 1,000$ in the neighborhood function. In MicroRTS, SHC is run with a restarting time limit of $2,000$ seconds for each self-play iteration. In Karel, since we are solving a single MDP, SHC restarts as many times as possible within the computational budget. For MicroRTS, we query the foundation models 120 times to generate the same number of programs; for Karel, we use 100 programs. We use the same number of programs as the LISS-r baseline. We perform 30 independent runs (seeds) of each system, including the generation of the programs by the model.

Metrics of Performance For MicroRTS, performance is measured in terms of winning rate: we sum the number of victories and half the number of draws and divide this sum by the total number of matches played [Ontañón, 2017]. For Karel, performance is measured in terms of episodic return [Trivedi *et al.*, 2021]. We use prompts where we briefly describe each problem and provide a formal description of the DSL used. The prompts used in our experiments are given in Appendices L (MicroRTS) and J (Karel) [Moraes *et al.*, 2025]. Both MicroRTS and Karel are deterministic, so the value of $\mathcal{E}$ for policies can be computed with a single roll-out. We report average performance and 95% confidence intervals.

Efficiency Experiment We verify the sampling efficiency of INNATECODER, LISS-o, LISS-r, and SHC. Similarly to previous work, we present learning curves, where for MicroRTS, we plot the winning rate by the number of games played (Figure 3), and for Karel, we plot episodic return by the number of episodes (Figure 4). For MicroRTS, the winning rate is computed for a system by having the policy the system generated, after a given number of games played, play against the policies each of the other systems generated after the maximum number of games played (rightmost point of each plot).

Competition Experiment We evaluate INNATECODER against COAC, Mayari, and RAISocketAI, the winners of the previous three MicroRTS competitions. We randomly select 9 from the 30 programs generated in the "Efficiency Experiment" and evaluate them against the competition winners. We report the average results of the 9 programs against each opponent in the four maps we use.

Size and Information Experiments We also evaluate the effect of the size of the set of options on the sample efficiency of INNATECODER. We evaluated sets Ω with 300, 600, 1400, 5000, 7000 and 30000 options on the LetMeOut map (16 × 8); all options were generated with the Llama 3.1 model. In Appendix C [Moraes *et al.*, 2025], we evaluate INNATECODER using prompts with more or less information and in Appendix D with GPT 3.5, to verify if performance decreases if using a smaller model. Using more information was never worse than using less information, and switching from GPT 4 to 3.5 did not reduce performance.

4.1 Learning Curve Results

Figures 3 and 4 show the learning curves for MicroRTS and Karel, respectively, where INNATECODER is denoted as IC-GPT or IC-Llama, depending on the model it uses to learn the options. INNATECODER is often much more sample-efficient than all baselines and, in many cases, by a large margin. We did not observe significant differences between IC-GPT and IC-Llama. LISS-o and LISS-r perform worse than INNATECODER and SHC in MicroRTS. However, LISS-o was competitive with SHC in Karel and LISS-r could outperform SHC (DoorKey and Seeder). The LISS-o and LISS-r results on MicroRTS suggest that the semantic space can be less conducive to search than the syntax space, depending on the quality of the options used to induce it. LISS-r performs better in Karel than in MicroRTS, probably because it uses a distribution $\mathcal{I}$ that uses a handcrafted probability distribution over the production rules of the language [Trivedi *et al.*, 2021]. The resulting grammar allows for the generation of helpful options.

FM performs poorly in all experiments. The results of FM and INNATECODER support our hypothesis that INNATE-

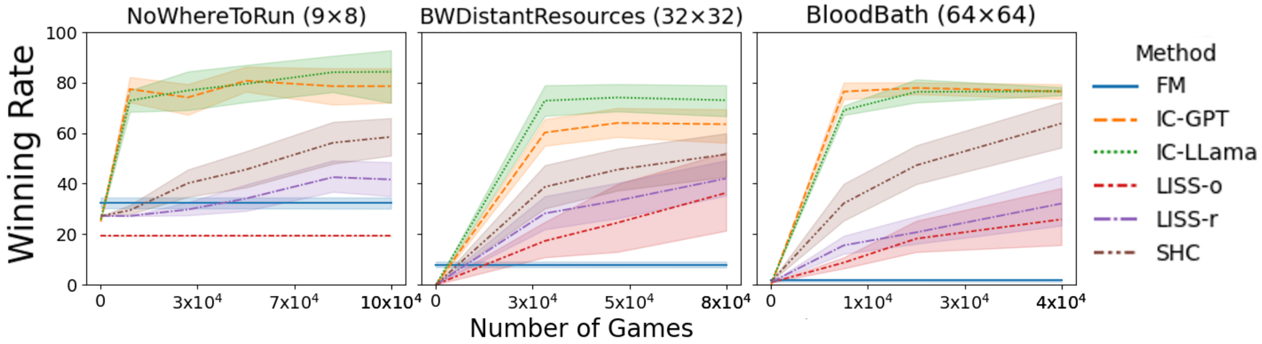

Figure 3: Winning rate (maximum is 100) per number of games played. The winning rate of the policies each system generates for a given number of games played is computed considering as opponents the policies all systems generate at the end of the learning process. The plots show the average winning rate of 30 independent runs (seeds) and the 95% confidence interval.

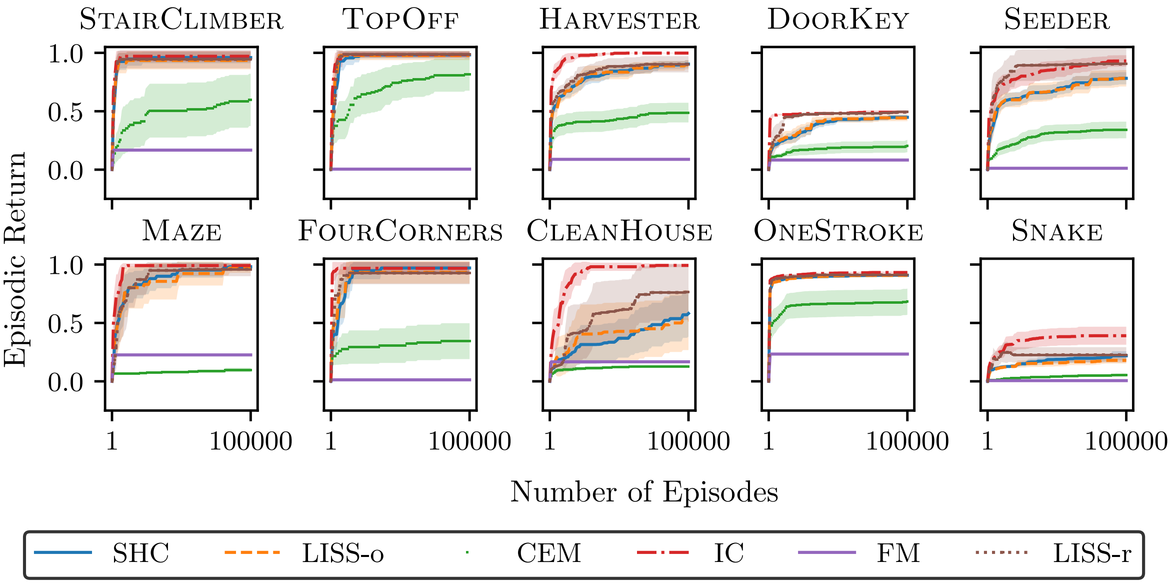

Figure 4: Average episodic return (maximum is 1.0 for all tasks) per number of episodes. The plots show average episodic return of 30 independent runs (seeds) and the 95% confidence interval.

InnateCoder	COAC	RAI AI	Mayari	Average
GPT-4o	53.75	36.25	71.25	53.75
Llama 3.1	43.79	70.00	58.17	57.32
Llama 3.1 + GPT-4o	70.14	72.92	46.39	63.15

Table 1: Winning rate of InnateCoder against winners of previous competitions, averaged across all 4 maps used in our experiments.

Coder can extract helpful options from foundation models even if the programs the model generates do not encode strong policies. In MicroRTS, some of the options allowed the agent to allocate units to collect resources and train other units. Other systems had to learn such skills from scratch, while InnateCoder's agent had them "innately available". The FM results also suggest that data contamination was not an issue, as the model performed poorly on all tasks.

4.2 Competition results

Table 1 shows the results of InnateCoder against the winners of previous MicroRTS competitions, averaging its winning rate across 4 maps. COAC and Mayari are human-written policies, and RAISocketAI is a DRL agent [Goodfriend, 2024]. RAISocketAI was trained with a larger computational budget than InnateCoder, thus giving it an advantage. We evaluated GPT-4o and Llama 3.1 while generating 120 programs. We also evaluate InnateCoder with the union of the programs generated by both GPT-4o and Llama (Llama 3.1 + GPT-4o in the table). The combination of GPT-4o and Llama 3.1 resulted in the best winning rate.

The combination of programs written by Llama 3.1 and GPT-4o does not lead to "monotonic improvements", as evidenced by the drop in performance against Mayari. This happens because none of the competition winners is constrained by the DSL we use in our experiments. As a result, the opti-

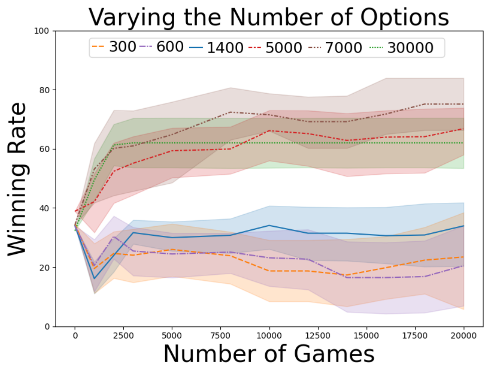

Figure 5: Average winning rate of InnateCoder policies for different sizes of the option set over 10 independent runs (seeds) of each version. We also present the 95% confidence intervals.

mization done in self-play might not be specific for the opponents evaluated in Table 1, but to policies written in the DSL and encountered during the self-play process.

4.3 Evaluating Number of Options

Figure 5 presents the learning curves for different versions InnateCoder, where we vary the size of the set of options Ω. The three lines with the highest winning rate are for option sets of sizes 5000, 7000, and 30000. The versions of InnateCoder with option sets of sizes 300, 600, and 1400 perform worse. These results demonstrate that InnateCoder can benefit from thousands of options. This is possible due to InnateCoder's way of using options through the induction of the language's underlying semantic space.

These results also show that InnateCoder's sample efficiency plateaus at 5000 options, since the use of 7000 and 30000 options does not increase performance. Interestingly, performance does not degrade either as we increase the set size. We conjecture that this occurs because many of the options will encode different and yet similar behaviors that do not affect the agent's winning rate. Although the set of distinct behaviors encoded in the set options grows with larger sets, the relative number of options with behaviors that affect the winning rate remains roughly the same. As a result, the function $\mathcal{N}_k^m$ that uses option sets of sizes 5000, 7000, or 30000 induces spaces that are similarly conducive to search.

5 Related work

Programmatic Policies One of the key challenges in generating programmatic policies is that the search space is discontinuous and gradient-based optimization cannot be used. Some previous work relied on imitation learning to guide the search for policies [Verma *et al.*, 2018; Verma *et al.*, 2019; Bastani *et al.*, 2018]. The issue of this imitation learning approach is known as *representation gap* [Qiu and Zhu, 2022; Medeiros *et al.*, 2022], where the space of programmatic policies does not include the oracle policy that the system tries to imitate. As a result, the oracle might guide the search to unpromising parts of the space. Previous work tried to learn latent spaces of programming languages that are conducive to search [Trivedi *et al.*, 2021; Liu *et al.*, 2023], which was shown to be outperformed by the syntax space with SHC [Carvalho *et al.*, 2024]. Semantic spaces were shown to be more conducive to search than syntax spaces, but required a sequence of tasks, where the agent learns the space in one task and reuses it in others [Moraes and Lelis, 2024]. Our work does not require an oracle nor a sequence of tasks to learn the semantic space, which is learned in a zero-shot setting.

Options Options were shown to improve the sampling efficiency of learning agents through faster credit assignment [Mann and Mannor, 2014], better exploration [Baranes and Oudeyer, 2013; Bellemare *et al.*, 2020], and transfer of knowledge across tasks [Konidaris and Barto, 2007; Alikhasi and Lelis, 2024]. However, previous methods for learning options require the user to design them before learning starts [Sutton *et al.*, 1999] or to provide considerable information as input to the process, such as the option duration [Frans *et al.*, 2017; Tessler *et al.*, 2017] or the number of options learned [Bacon *et al.*, 2017; Igl *et al.*, 2020]. Other methods rely on the agent interaction with the current environment [Achiam *et al.*, 2018; Machado *et al.*, 2018; Jinnai *et al.*, 2020] or with other earlier environments, as in transfer learning approaches [Konidaris and Barto, 2007; Alikhasi and Lelis, 2024]. We present a novel way of learning options as they are not learned from the agent's experience nor designed by the user, but harnessed from foundation models. While we use options to define a search space, future work will explore their use as functions neural policies can call.

6 Conclusions

In this paper, we introduced InnateCoder, a system that equips learning agents with skills, in the form of programmatic options, before the agent starts to interact with the environment. This is achieved by extracting programmatic options from foundation models. We hypothesized that even if the model is unable to write programs encoding strong policies for a problem, sub-programs of the generated program could encode helpful options. We tested our hypothesis in MicroRTS and Karel, two domains in which programmatic policies represent the current state of the art. The policies InnateCoder generated outperformed, often by a large margin, a baseline that did not attempt to learn options; a baseline that learned the options while learning how to solve the problem; a baseline that learned the options from programs sampled directly from the domain-specific language; and the foundation model that attempted to generate programmatic policies directly. We also showed that some of the policies InnateCoder generated were competitive or outperformed the winners of previous MicroRTS competitions, including programmatic policies written by human programmers and a deep reinforcement learning agent that used a larger computational budget than we allowed InnateCoder to use. These results place InnateCoder as the current state-of-the-art in both Karel and MicroRTS. Our experiments also showed that InnateCoder's scheme of using programmatic options to induce semantic spaces allows it to benefit from thousands of options, while most previous work can benefit only from dozens or at most hundreds of options [Eysenbach *et al.*, 2019].

Acknowledgments

This research was supported by Canada's NSERC, and the CIFAR AI Chairs program, and Brazil's CAPES. The research was carried out using computational resources from the Digital Research Alliance of Canada and the UFV Cluster. This research has also received funding from the project ALIGN4Energy (NWA.1389.20.251) of the research programme NWA ORC 2020 which is (partly) financed by the Dutch Research Council (NWO), and from the European Union's Horizon Europe Research and Innovation Programme, under Grant Agreement number 101120406. The paper reflects only the authors' view and the EC is not responsible for any use that may be made of the information it contains. We thank the anonymous reviewers for their feedback.

References

[Achiam *et al.*, 2018] Joshua Achiam, Harrison Edwards, Dario Amodei, and Pieter Abbeel. Variational option discovery algorithms. *arXiv preprint arXiv:1807.10299*, 2018.

[Alikhasi and Lelis, 2024] Mahdi Alikhasi and Levi H. S. Lelis. Unveiling options with neural network decomposition. In *The Twelfth International Conference on Learning Representations*, 2024.

[Bacon *et al.*, 2017] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[Baranes and Oudeyer, 2013] Adrien Baranes and Pierre-Yves Oudeyer. Active learning of inverse models with intrinsically motivated goal exploration in robots. *Robotics and Autonomous Systems*, 61:49–73, 2013.

[Bastani *et al.*, 2018] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Proceedings of the International Conference on Neural Information Processing Systems*, pages 2499–2509, 2018.

[Bellemare *et al.*, 2020] Marc G. Bellemare, Salvatore Candido, Pablo S. Castro, Jun Gong, Marlos C. Machado, Subhodeep Moitra, Sameera Ponda, and Ziyu Wang. Autonomous navigation of stratospheric balloons using reinforcement learning. *Nature*, 588:77–82, 2020.

[Carvalho *et al.*, 2024] Tales Henrique Carvalho, Kenneth Tjhia, and Levi H. S. Lelis. Reclaiming the source of programmatic policies: Programmatic versus latent spaces. In *The Twelfth International Conference on Learning Representations*, 2024.

[Chen *et al.*, 2018] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

[Eysenbach *et al.*, 2019] Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. In *International Conference on Learning Representations (ICLR)*, 2019.

[Frans *et al.*, 2017] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. *arXiv preprint arXiv:1710.09767*, 2017.

[Goodfriend, 2024] Scott Goodfriend. A competition winning deep reinforcement learning agent in microrts, 2024.

[Igl *et al.*, 2020] Maximilian Igl, Andrew Gambardella, Jinke He, Nantas Nardelli, N Siddharth, Wendelin Böhmer, and Shimon Whiteson. Multitask soft option learning. In *Conference on Uncertainty in Artificial Intelligence*, pages 969–978, 2020.

[Jinnai *et al.*, 2020] Yuu Jinnai, Jee W. Park, Marlos C. Machado, and George Konidaris. Exploration in reinforcement learning with deep covering options. In *International Conference on Learning Representations*, 2020.

[Khetarpal *et al.*, 2022] Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. Towards continual reinforcement learning: A review and perspectives. *J. Artif. Intell. Res.*, 75:1401–1476, 2022.

[Klissarov and Machado, 2023] Martin Klissarov and Marlos C. Machado. Deep laplacian-based options for temporally-extended exploration. In *International Conference on Machine Learning*, 2023.

[Klissarov *et al.*, 2024] Martin Klissarov, Pierluca D'Oro, Shagun Sodhani, Roberta Raileanu, Pierre-Luc Bacon, Pascal Vincent, Amy Zhang, and Mikael Henaff. Motif: Intrinsic motivation from artificial intelligence feedback. In *The Twelfth International Conference on Learning Representations*, 2024.

[Konidaris and Barto, 2007] George Konidaris and Andrew Barto. Building portable options: Skill transfer in reinforcement learning. In *International Joint Conference on Artificial Intelligence*, pages 895–900, 2007.

[Liu *et al.*, 2023] Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-Yi Lee, and Shao-Hua Sun. Hierarchical programmatic reinforcement learning via learning to compose programs. In *Proceedings of the International Conference on Machine Learning*, 2023.

[Machado *et al.*, 2017] Marlos C. Machado, Marc G. Bellemare, and Michael Bowling. A laplacian framework for option discovery in reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2017.

[Machado *et al.*, 2018] Marlos C. Machado, Clemens Rosenbaum, Xiaoxiao Guo, Miao Liu, Gerald Tesauro, and Murray Campbell. Eigenoption discovery through the deep successor representation. In *International Conference on Learning Representations*, 2018.

[Mann and Mannor, 2014] Timothy Mann and Shie Mannor. Scaling up approximate value iteration with options: Better policies with fewer iterations. In *Proceedings of the International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 127–135, 2014.

[Mariño *et al.*, 2021] Julian R. H. Mariño, Rubens O. Moraes, Tassiana C. Oliveira, Claudio Toledo, and Levi H. S. Lelis. Programmatic strategies for real-time strategy games. In

Proceedings of the AAAI Conference on Artificial Intelligence, pages 381–389, 2021.

[Medeiros *et al.*, 2022] Leandro C. Medeiros, David S. Aleixo, and Levi H. S. Lelis. What can we learn even from the weakest? Learning sketches for programmatic strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 7761–7769. AAAI Press, 2022.

[Moraes and Lelis, 2024] Rubens O. Moraes and Levi H. S. Lelis. Searching for programmatic policies in semantic spaces. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2024.

[Moraes *et al.*, 2023] Rubens O. Moraes, David S. Aleixo, Lucas N. Ferreira, and Levi H. S. Lelis. Choosing well your opponents: How to guide the synthesis of programmatic strategies. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 4847–4854, 2023.

[Moraes *et al.*, 2025] Rubens O. Moraes, Quazi Asif Sadmine, Hendrik Baier, and Levi H. S. Lelis. Innatecoder: Learning programmatic options with foundational models (extended version). 2025.

[Ontañón, 2017] Santiago Ontañón. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, 58:665–702, 2017.

[Pattis, 1994] Richard E Pattis. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, 1994.

[Precup *et al.*, 1998] Doina Precup, Richard S. Sutton, and Satinder Singh. Theoretical results on reinforcement learning with temporally abstract options. In Claire Nedellec and Céline Rouveirol, editors, *Machine Learning: ECML-98, 10th European Conference on Machine Learning*, volume 1398 of *Lecture Notes in Computer Science*, pages 382–393. Springer, 1998.

[Qiu and Zhu, 2022] Wenjie Qiu and He Zhu. Programmatic reinforcement learning without oracles. In *The Tenth International Conference on Learning Representations*, 2022.

[Sutton *et al.*, 1999] Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[Tessler *et al.*, 2017] Chen Tessler, Shahar Givony, Tom Zahavy, Daniel Mankowitz, and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, pages 1553–1561, 2017.

[Tinbergen, 1951] Nikolaas Tinbergen. *The Study of Instinct*. Oxford University Press, 1951.

[Trivedi *et al.*, 2021] Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J. Lim. Learning to synthesize programs as interpretable and generalizable policies. In *Advances in Neural Information Processing Systems*, pages 25146–25163, 2021.

[Verma *et al.*, 2018] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri. Programmatically interpretable reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pages 5052–5061, 2018.

[Verma *et al.*, 2019] Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[Zhu *et al.*, 2023] Zhuangdi Zhu, Kaixiang Lin, Anil K. Jain, and Jiayu Zhou. Transfer learning in deep reinforcement learning: A survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 45(11):13344–13362, 2023.