VerMCTS: Synthesizing Multi-Step Programs using a Verifier, a Large Language Model, and Tree Search

David Brandfonbrener[*] **Simon Henniger**[†] **Sibi Raja**[‡] **Tarun Prasad**[‡]

Chloe Loughridge[‡] **Federico Cassano**[§] **Sabrina Ruixin Hu**[‡] **Jianang Yang**[¶]

William E. Byrd[‖] **Robert Zinkov**[††] **Nada Amin**[‡]

Abstract

Large Language Models (LLMs) can generate useful code, but often the code they generate cannot be trusted to be sound. In this paper, we present VerMCTS, an approach to begin to resolve this issue by generating verified programs in Dafny and Coq. VerMCTS uses a logical verifier in concert with an LLM to guide a modified Monte Carlo Tree Search (MCTS). This approach leverages the verifier to gain intermediate feedback inside the search algorithm by checking partial programs at each step to estimate an upper bound on the value function. To measure the performance of VerMCTS, we develop a new suite of multi-step verified programming problems in Dafny and Coq. In terms of pass@T, a new metric which computes the pass rate given a budget of T tokens sampled from the LLM, VerMCTS leads to more than a 30% absolute increase in average pass@5000 across the suite over repeated sampling from the base language model.

1 Introduction

Large Language Models (LLMs) are increasingly used for generating code, but the code needs to be inspected and possibly re-generated if it doesn't satisfy the user [Zhong and Wang, 2023]. We propose to partially shift the burden of checking code, from the user to the LLM, by generating code in a verification-aware programming language like Dafny or Coq, prompting for specifications and proofs of correctness in addition to code that can then be formally verified. In such a system, the user can focus their attention on the specifications, and less on the code and proofs with the assurance that the generated output has passed the verifier. Our approach couples imprecise generative reasoning from an LLM with logical reasoning from a program verifier. The LLM contributes fruitful suggestions and the verifier ensures soundness.

As a motivating example, consider this prompt: *In Dafny, write an ADT for arithmetic expressions comprising constants, variables, and binary additions. Then write an evaluator taking an expression and an environment (a function that takes a variable name and returns a number) and returning the number resulting from evaluation. Then write an optimizer taking an expression and returning an expression with all additions by 0 removed. Then prove that the optimizer preserves the semantics as defined by the evaluation function.*

To aid a language model to tackle this task, we introduce VerMCTS, an algorithm that combines a verifier and tree search with a language model to synthesize verified programs. An overview of the

[*]Kempner Institute at Harvard University, [‡] Harvard University, [†] TU München, [§] Northeastern University, [¶] Million.js, [‖] University of Alabama at Birmingham, [††] University of Oxford
Correspondence to namin@seas.harvard.edu

MATH-AI Workshop at the 38th Conference on Neural Information Processing Systems (NeurIPS 2024).

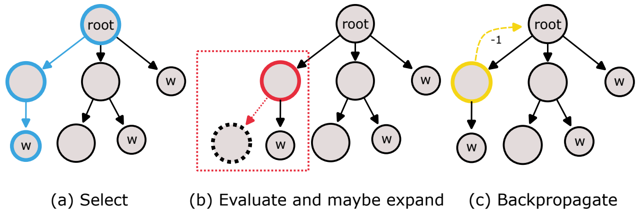


Figure 1: Overview of VerMCTS. The search tree is visualized with "widen" nodes marked with w. (a) A leaf node is selected as in standard MCTS. (b) The selected node is evaluated and maybe expanded. If the selected node is a widen node, then it's parent is selected and maybe expanded (i.e. made wider). See Figure 2 for a detailed description. (c) Once we have a value from the evaluate and maybe expand algorithm, we backpropagate that value up the tree. This figure illustrates the special case where we observed a failure, so no node is added and the score is -1.

algorithm is described in Figure 1 and Figure 2 and the details are presented in Section 2. VerMCTS creates a search tree with progressive widening so it is capable of handling large action spaces defined by lines of code. Within this search tree both expansion and evaluation are guided by the verifier which acts as a computationally cheap (relative to the LLM) upper bound on the value function in the code synthesis MDP, as we show in Section 2.

To evaluate VerMCTS we introduce a suite of 15 challenge problems (9 in Dafny and 6 in Coq). This suite probes essential skills needed for general verified programming like constructing algebraic data types, defining functions, and writing inductive proofs.

On this suite of problems we compare VerMCTS with several baselines including repeated sampling of full programs from the base model, an advanced prompting technique that uses access to the error messages generated by the verifier called Reflexion [Shinn et al., 2023], and a traditional version of MCTS. We quantify performance in terms of pass@T, the pass rate within a budget of T tokens. VerMCTS outperforms the baselines substantially, leading to a 30% absolute average performance improvement over repeated sampling from the base model. Note this repeated sampling is a strong baseline, similar to a pass@k evaluation often used as a skyline in program generation. Moreover, for several problems VerMCTS solves problems that are not solved at all by other methods within the given budget.

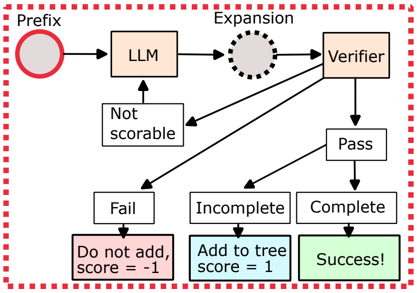


Figure 2: Evaluate and maybe expand. Given a prefix, we query the LLM for expansions until the verifier is able to return a score. If that score is a failure, we do not add the node to the tree, but update the parent with a value of -1. If the score is pass, then we check if the program is complete. If incomplete, we add the expansion to the tree with a score of 1. If complete, we have found a successful program to return.

2 Method: VerMCTS

Our main contribution is to define a search algorithm inspired by MCTS that leverages a verifier and LLM to search for verified programs. We call this method *VerMCTS*. In this section, we first present the Markov Decision Process that we consider as the environment for verified program synthesis and then present VerMCTS in detail. VerMCTS is a variant of traditional MCTS that incorporates the LLM as a prior to generate candidates and the verifier as a heuristic to evaluate partial programs.

2.1 MDP for verified program synthesis

We formulate our multi-step verified synthesis problem as a Markov Decision Process (MDP) $\mathcal{M} := (\mathcal{S}, \mathcal{A}, T, r, H)$ defined by the LLM and the verifier. Here, $\mathcal{S}$ refers to the state space, A refers to the action space, $T\colon \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ refers to the (deterministic) transition dynamics of the environment, $r : \mathcal{S} \to \mathbb{R}$ refers to the reward function, and H is the finite horizon (i.e. a limit on the number of transitions). Defining the MDP just consists of defining these four objects. The state, action, transition dynamics, and reward are defined as follows:

- Each state $s \in \mathcal{S}$ is a string consisting of the initial user prompt and a partial program.
- Actions $a \in \mathcal{A}$ are strings that represent a unit of a program. In Dafny each line is an action. In Coq each "command" (ending with a dot '.') is an action. We also limit the number of tokens in an action.
- The transition dynamics are just defined by string concatentation: $T(s, a) = s + a$.
- The reward function r is defined by the verifier for a given verified programming language and is only defined on complete programs. This terminal reward is 1 if the complete program is accepted and -1 if it is rejected. The reward is 0 for incomplete programs.

With this simple MDP in place, we can define our search algorithm for finding verified programs.

2.2 VerMCTS

Given this MDP with finite actions and deterministic dynamics, it would be possible to run standard MCTS to learn a stochastic policy, but the action space is much too large for this to be practical. Instead, we build a search algorithm inspired by MCTS that can leverage the LLM as a prior for program synthesis and the verifier to evaluate partial programs. Both components are key for a successful search in this large space.

Algorithm 1 Evaluate and (maybe) expand

```
Input: string $s$, depth limit $L$
LLM: string $\rightarrow$ completion
Verifier: string $\rightarrow \{-1, 0, +1\}$
Output: value $v(s)$, (optional) child node
score $\leftarrow 0$
depth $\leftarrow 0$
$a \leftarrow$ ""
while score $= 0$ and depth $< L$ do
    $a \leftarrow a +$ LLM$(s + a)$
    score $\leftarrow$ Verifier $(s + a)$
    depth $\leftarrow$ depth $+ 1$
end while
if score $= -1$ or depth $= L$ then
    return $-1$, None
else
    return $+1, s + a$
end if
```

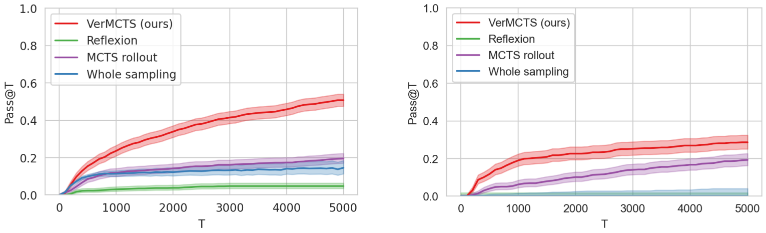
Standard MCTS consists of four steps: select, expand, evaluate, and backpropagate. Our algorithm leaves the select and backpropagate steps essentially unchanged. We modify and combine the expand and evaluate steps to leverage the power of the LLM and the verifier in tandem. Our full algorithm is illustrated in Figure 1. In this section we first discuss progressive widening and then go through each step of VerMCTS in turn.

Progressive widening. To allow for potentially infinite width while still efficiently conducting deep searches, we adapt an idea from classical work on MCTS to progressivly widen nodes in the tree [Chaslot et al., 2008, Couëtoux et al., 2011]. In that work, the number of children available at a given node scales explicitly with the number of visits. In our setting since adding a child node requires an expensive call to the LLM, we instead opt to add a "widen" child to each node that is assigned 0 value and can be selected via the selection procedure described below. This allows the scoring mechanism to prioritize when to expand a node by essentially setting a prior that unexplored branches have 0 value. When the widen node w with parent s is selected, instead of adding a child to w, we add a child to s (i.e. add a sibling to w). In this way, the tree can grow wider over the search process.

Selection: priors and UCT. We use a standard MCTS selection step, but we set a prior for the UCT (upper confidence bound for trees) bonus as in PUCT [Rosin, 2011, Silver et al., 2016]. We choose to let the prior $p = 1.0$ for standard nodes and let $p = p_{widen} < 1.0$ for widen nodes be a hyperparameter that we tune. This basic heuristic gives the model a preference to select the standard nodes which encourages deeper search trees while still allowing for potentially infinite width if needed. With this choice, the score of a node s is:

$$\text{score}(s) = p_s \cdot c_{UCT} \sqrt{\frac{\log N_{parent}}{N_s}} + \frac{\sum_{i=1}^{N} v_i}{N_s} \quad (1)$$

where p_s is the prior at this node, c_{UCT} is a global exploration coefficient, N_{parent} is the number of visits at the parent node, N_s is the number of visits at this node, and v_i is the estimated value at the ith visit to s. Note that this selection procedure has two hyperparameters: c_{UCT} and p_{widen} that encourage selecting more rarely visited nodes and widen nodes respectively.



(a) Dafny

(b) Coq

Figure 3: Average results for pass@T vs. T (the number of tokens) for various baseline methods on our suite of programming problems in Dafny and Coq.

Combining expansion and evaluation. Traditionally, MCTS will first expand a node into children and evaluate it either by simulated rollouts [Chaslot et al., 2008, Zhang et al., 2023a] or a learned value function [Silver et al., 2016]. Neither of these methods is a good fit for our problem because generating rollouts requires many expensive calls to the LLM and learning a value requires large amounts of training data. Moreover, both methods give noisy signal, but in our setting we have access to the ground truth verifier.

Beyond being noiseless, the verifier has one more important property: if a partial program fails the verifier, no subsequent completion can yield success. So, we want to make sure that we never add to the tree any expansion that is a guaranteed failure. Doing this require explicitly linking expansion to evaluation where we evaluate the node and *maybe* expand it, as formalized in Algorithm 1.

In addition to only adding nodes with potential to the tree, we want to leverage the verifier to cheaply evaluate partial programs without extra calls to the LLM. Explicitly, from a node containing the string s we continue to extend a with the LLM until the verifier is able to return a valid score. At this point, we can return the estimated value $v(s)$ of s as follows:

$$v(s) = \texttt{Verifier}(s + a) = \begin{cases} +1 & \text{verified, but may be incomplete.} \\ -1 & \text{verified as a failure.} \end{cases} \tag{2}$$

If $v(s) = +1$, we also add $s + a$ as a child in the tree, while if $v(s) = -1$, we do not add $s + a$ since it is a verified failure. Appendix H gives explicit examples of scoring partial programs.

Backpropagation. The last step of an iteration of MCTS is to backpropagate the observed value from leaf back up to root. We do this in the standard way so that signal is propagated up the tree. The algorithm terminates when it finds a complete solution that verifies or when it exceeds some token limit or time limit.

Appendix A presents theory that VerMCTS optimizes an upper bound on the value function.

3 Results

A full description of the problem suite can be found in Appendix B and experimental methods in Appendix C respectively. Here we present the main results.

We run VerMCTS and our three baselines across our full suite of problems. The aggregate results are illustrated in Figure 3. In both programming languages VerMCTS convincingly outperforms the baselines. Generally, MCTS rollout is second best, followed by whole sampling and then Reflexion. As previewed in the introduction, we see about a 30% absolute improvement in pass@5000 for VerMCTS relative to whole sampling. Note that Coq is substantially more challenging since the verifier is less automated.

Examining the performance of the baselines more closely, we see that MCTS rollout does outperform whole sampling, even though the verifier is not used to guide the search at intermediate steps. But, using the verifier in VerMCTS provides even better performance. Looking at Reflexion, we see that performance is poor on these tasks. This could be due to many reasons including: (1) the base model is not good at responding to errors in low resource languages like Dafny and Coq, (2) the base model does not do well integrating the long contexts created by the Reflexion prompts, and (3) Reflexion does not make it as easy to backtrack.

Due to space constraints, extended results are in Appendix D, extended related work in Appendix E, and further discussion in Appendix F.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571234. URL `https://doi.org/10.1145/3571234`.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering (TSE)*, 49 (7):3675–3691, 2023a.

Federico Cassano, Ming-Ho Yee, Noah Shinn, Arjun Guha, and Steven Holtzen. Type prediction with program decomposition and fill-in-the-type training, 2023b.

Guillaume Chaslot, Mark H. M. Winands, H Jaap Van Den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 04:343–357, 2008. URL `https://api.semanticscholar.org/CorpusID:1719063`.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022.

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.

Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous upper confidence trees. In *Learning and Intelligent Optimization*, 2011. URL `https://api.semanticscholar.org/CorpusID:13463524`.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE ’16, page 345–356, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884786. URL `https://doi.org/10.1145/2884781.2884786`.

Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1229–1241, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703270. doi: 10.1145/3611643.3616243. URL `https://doi.org/10.1145/3611643.3616243`.

Michael Glass, Gaetano Rossiello, Md Faisal Mahbub Chowdhury, Ankita Naik, Pengshan Cai, and Alfio Gliozzo. Re2G: Retrieve, rerank, generate. In Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz, editors, *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2701–2715, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.194. URL `https://aclanthology.org/2022.naacl-main.194`.

Gabriel Grand, Lionel Wong, Matthew Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code, 2023.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=rpxJc9j04U`.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

ImparaAI. Monte carlo tree search. `https://github.com/ImparaAI/monte-carlo-tree-search`, 2024.

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=SMa9EAovKMC`.

Guillaume Lample, Timothee Lacroix, Marie anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL `https://openreview.net/forum?id=J4pX8Q8cxHH`.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning (ICML'23)*, 2023.

Phind. Beating gpt-4 on humaneval with a fine-tuned codellama-34b. `https://www.phind.com/blog/code-llama-beats-gpt4`, 2023.

Christopher D. Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61:203–230, 2011. URL `https://api.semanticscholar.org/CorpusID:207081359`.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. Refactoring programs using large language models with few-shot examples, 2023.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, L. Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016. URL `https://api.semanticscholar.org/CorpusID:515925`.

Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. A language-agent approach to formal theorem-proving, 2023.

Edwin Bidwell Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22:209–212, 1927. URL `https://api.semanticscholar.org/CorpusID:121572396`.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*, 2023.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Optimal neural program synthesis from multimodal specifications. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 1691–1704, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.146. URL `https://aclanthology.org/2021.findings-emnlp.146`.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations*, 2023a. URL `https://openreview.net/forum?id=Lr8cOOtYbfL`.

Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I. Wang. Coder reviewer reranking for code generation. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023b.

Li Zhong and Zilong Wang. A study on robustness and reliability of large language model code generation. *arXiv preprint arXiv:2308.10335*, 2023.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models, 2023.

A Connecting the partial program score to the MDP

Importantly, while the verifier gives us ground truth information about whether the program verifies so far, it does not give an unbiased estimate of the true value of a state in the MDP defined above. Instead, we can view our use of the verifier as a heuristic that quickly returns an *upper bound* on the value function of a potential child. Recall that the value function V^* of the optimal policy in a deterministic MDP with state-based rewards like ours is defined by the Bellman equation $V^*(s) = \max_a r(s) + V^*(s + a)$. With this definition, we can formally describe the optimism property of our estimates values as follows:

Lemma A.1. *The value $v(s)$ returned by Algorithm 1 satisfies the following:*

$$v(s) \geq \mathbb{E}_{a \sim \texttt{LLM+Verifier}|s}[V^*(s + a)] \tag{3}$$

This is fairly straightforward to prove. If $v(s) = -1$, then we know that the sampled completion a is a failure no matter what happens afterwards, so $v(s) = V^*(s + a) = -1$. On the other hand, if $v(s) = 1$ then we are assigning the maximal possible value in this MDP, so $v(s) \geq V^*(s + a)$.

In this way, our value estimate is explicitly an *optimistic* estimate of the value. This is even beyond the UCT score computed by MCTS. We hypothesize that this encourages deeper exploration of the search trees which can be beneficial in the multi-step problems we consider.

B A problem suite for multi-step verified programming

B.1 Defining the problems

We are not aware of any existing collections of problems that are designed for multi-step program synthesis and checked using verifiers. That is why we have created our own problem suite of nine problems. The problems represent meaningful scenarios in verified programming. They require creating Algebraic Data Types (ADTs), defining functions on them using pattern matching, and proving properties using induction. Compared to prior benchmarks, the problems require more intricate multi-step reasoning and test capabilities that are specifically important for verified programming. The problems are defined as follows:

Factorial asks to define the factorial function and to prove that it is always strictly positive.

Opt0 asks to define an ADT for arithmetic expressions, an optimizer, and to prove that the optimizer preserves semantics.

Opt0 Opt asks to define an ADT for arithmetic expressions, an optimizer, an optimal predicate, and to prove that the optimizer is optimal.

BST asks to define a tree, the binary search tree (BST) property, insertion, and to prove two properties of insertions (membership and BST preservation).

Repeat asks to define a function returning a list with a given element repeated a given number of times, and to prove two properties related to length and membership.

Lights asks to define an ADT for traffic lights, then write a function ensuring that red and green lights are always separated by yellow lights, and then to prove its correctness.

Food asks to define an ADT that represents different foods with toppings, and a predicate about the amount of toppings, and to prove a property of this predicate.

Days asks to define an ADT that represents days of the week, two functions that iterate through business days, and then to prove a property of weekdays.

Reverse asks to define a function that reverses a list, and prove two properties of list reversals (permutation and involution).

All problems are implemented in Dafny, and all but the last three are implemented in Coq, giving a total of 15 problems. Since the Coq verifier has substantially less automation than Dafny which leads to longer proofs and since the model is not always very consistent at Coq syntax, just for Coq we provide some syntax hints in the prompt. The full prompts can be found in Appendix G.

B.2 Criteria for Success

In order to be considered successful, a program must first pass the verifier and some syntactic checks (e.g. the presence of a proof marker and a problem-specific minimum number of lines of code). These initial checks are meant to ensure the model has made a successful attempt to prove a lemma.

A second check ensures that the model has proven the correct lemma: In order to check whether a model has proven a property, we inject a second lemma with it, and prove it by referring to the lemma we asked the model to write. If the model has proven this lemma as directed, this new code including check lemma will verify successfully. If the model has proven an incorrect lemma, a verifier error will be produced. Note that the check lemma is only injected into the verifier input. The model does not get to see it, so this check does not provide additional hints to the model.

A full description of each problem including the prompts and lemmas used for checking success can be found in Appendix G.

C Experimental setup

C.1 Pass@T evaluation metric

We report all of our results in terms of pass@T, which is, to our knowledge, a novel metric inspired by pass@k that is often used in code generation benchmarks [Chen et al., 2021]. While pass@k computes the probability of generating a success when we sample k programs, pass@T computes the probability of success if we allow the model to sample T tokens. Pass@T has several benefits:

1. Pass@T fairly compares methods. One run of MCTS can be much more expensive than sampling one program from a model, so using pass@k is not fair. In contrast pass@T really estimates the dominant cost of generation, namely how many tokens need to be generated to yield success.
2. Pass@T controls for hardware and implementation variability. Compared to using wall-clock time, using pass@T does not depend on the underlying hardware and system-level optimizations.

To estimate pass@T, we generate n runs per problem of up to T_{max} tokens per run (where if the run terminates successfully before T_{max} we stop the run). Then for each $T \leq T_{max}$, we have n binary trials indicating whether that run terminated successfully in $\leq T$ tokens. In the results, we report the mean of these n binary variables and also 95% Wilson intervals [Wilson, 1927].

C.2 Base model

VerMCTS is compatible with any base model and only requires sampling from the model (no training is needed). We opt to use an open-weights model as the base language model and then compare different sampling procedures on top of this base model. Specifically, we use Phind-CodeLLama-34B-v2 [Phind, 2023, Roziere et al., 2023]. This model has been trained explicitly for code generation, but the verified programming languages we use are relatively “low resource” languages, so the models will perform worse than at high-resource languages [Cassano et al., 2023a].

C.3 Baselines

We consider a variety of baseline methods to illustrate the benefits of leveraging the verifier inside of VerMCTS.

- **Whole sampling.** The most naive baseline just samples entire programs from the base model. To compute pass@T we just continue generating new samples until success or until the token limit is reached.
- **Rollout MCTS.** Related work on MCTS uses rollouts to evaluate a node [Chaslot et al., 2008, Zhang et al., 2023a]. We ablate the importance of using the verifier by replacing the “evaluate and maybe expand” step with separate expand and evaluate steps. We expand by sampling a fixed number of actions k from the LLM and evaluate by rolling out with the LLM to a terminal node before querying the reward function.

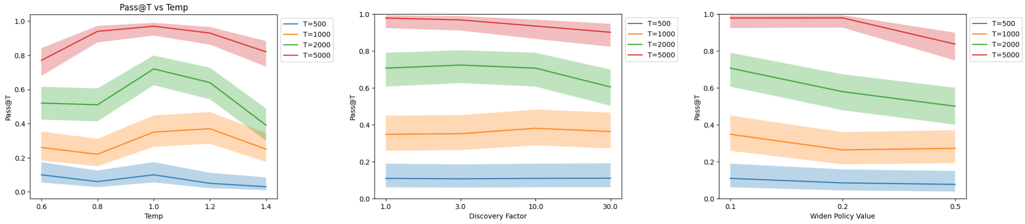


Figure 4: Hyperparameter ablations for VerMCTS on opt0 in Dafny. We find that performance is generally fairly stable to hyperparameter choices.

- **Reflexion.** Finally, to show how VerMCTS is efficient at incorporating information from the verifier we also compare to a Reflexion [Shinn et al., 2023] baseline where the LLM gets to view the errors produced by the verifier on failed attempts.

C.4 Hyperparamters

When sampling from the LLM, we always use nucleus sampling [Holtzman et al., 2019] with $p = 0.95$ following Roziere et al. [2023]. For every method, we sweep over temperature on one representative problem and use that temperature for the rest. Our VerMCTS algorithm also introduces two hyperparameters that govern exploration: c_{UCT} and p_{widen} which we found fairly straightforward to set. We tune hyperparameters on one particular problem (opt0) in Dafny, but only checking for verification and not additionally checking for correctness. Each method has slightly different hyperparameters, but we generally tune temperature of the LLM, the MCTS exploration coefficient, and the MCTS prior for widen nodes. Hyperaparameters are then fixed for all other experiments. Each algorithm's parameters are described below.

VerMCTS. We sweep over temperature in [0.6, 0.8, 1.0, 1.0, 1.4] and find 1.0 to be best, exploration coefficient in [1, 3, 10, 30] and find 3 to be best, and the "widen policy value", i.e. the prior value of the widen nodes in [0.1, 0.2, 0.5] and find 0.1 to be best. See Figure 4.

MCTS rollout. We also sweep over temperature in [0.6, 0.8, 1.0, 1.0, 1.4] and find 0.8 to be best and exploration coefficient in [1, 3, 10, 30] and find 1 to be best. Note, instead of widen nodes, each node has a fixed number of children (3 in our experiments).

Reflexion. We sweep over temperature in [0.2, 0.4, 0.6, 0.8, 1.0] and find 0.4 to be best.

Whole sampling. We sweep over temperature in [0.2, 0.4, 0.6, 0.8, 1.0] and find 0.6 to be best.

We use the Transformers library Wolf et al. [2020] to query the LLMs. For the MCTS, we adapt a generic open-source library ImparaAI [2024].

D Extended results

D.1 Per-problem results

In Figures Figure 5 and Figure 6, we present the per-problem results on our problem suite. There is substantial variation across problems, but across all problems VerMCTS is the best approach or within the margin of error, often exceeding the baselines by a large margin and sometimes solving problems that no baseline solves at all. That said, some problems are clearly challenging: on one problem in Dafny and three in Coq, none of the algorithms find a solution within 5000 tokens.

D.2 Examining the VerMCTS search trees

In Figure 7 we provide an experiment to probe for a mechanistic understanding of how VerMCTS works in Dafny. We consider the number of nodes (excluding widen nodes), the depth and the width of the search trees as the number of tokens generated increases. Note that since we do not add

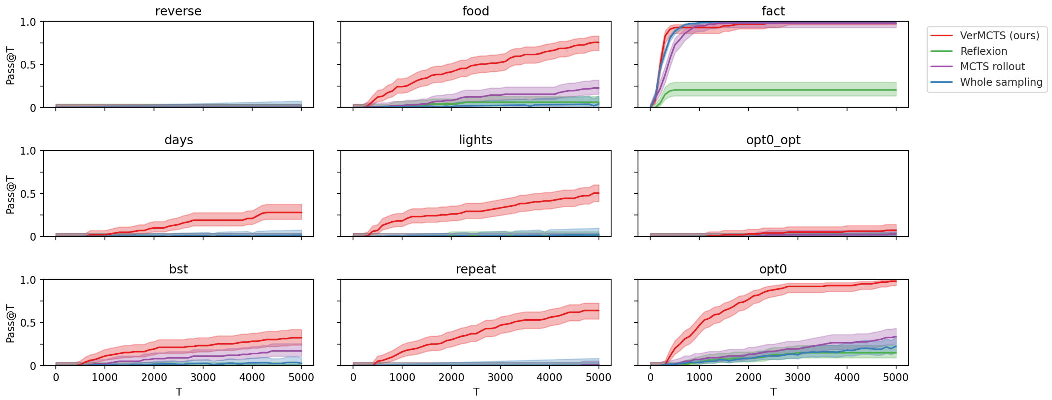


Figure 5: Pass@T results for all algorithms on our suite of problems in Dafny.

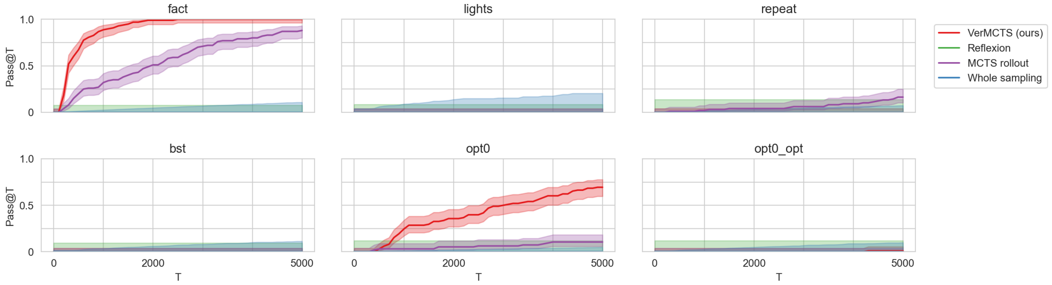


Figure 6: Pass@T results for all algorithms on our suite of problems in Coq.

failed expansions to the tree, sometimes more tokens are generated without adding nodes to the tree. Generally, we observe that the more challenging problems (with lower pass rates) tend to lead to larger search trees, indicating that the algorithm is successfull. We also notice that while the number of nodes grows fairly linearly across time for most problems, the depth grows earlier and then flattens out. This suggests that the VerMCTS search is closer to "depth first", first pushing an expansion branch to a terminal node before going back and widening the tree.

E Related Work

Neural Program Synthesis with Large Language Models Austin et al. [2021] and Chen et al. [2021] demonstrated that Large Language Models (LLMs) can generate correct Python programs from natural language descriptions. These studies introduced the MBPP and HumanEval datasets, respectively, which are widely used for evaluating LLMs in program synthesis tasks. Cassano et al. [2023a] extended this concept by showing that LLMs can also generate programs in over 20 languages other than Python. This was achieved by translating the MBPP and HumanEval datasets using their system, MultiPL-E. Their findings indicate that generating accurate programs in lower resource languages is more challenging compared to higher resource languages, such as Python. In our experiments, for proof synthesis, we have another dimension of challenge: some languages (Coq) are inherently more challenging than others (Dafny), depending on how much automation the verifiers provide. However, none of these works explored the generation of programs that are correct by construction.

Symbolic Algorithms for Neural Program Synthesis Grand et al. [2023] integrated a classic symbolic top-down synthesis algorithm for library learning Bowers et al. [2023] with LLMs. Cassano et al. [2023b] employed program decomposition and a bottom-up tree-search algorithm to infer missing TypeScript types. Zhou et al. [2023] used Monte Carlo Tree Search (MCTS) to create single-function programs in Python. Zhang et al. [2023a] applied a tree-based planning algorithm for decoding LLM token sequences, which were then evaluated for correctness using a test suite. Lample et al. [2022] adapted MCTS for neural theorem proving by employing a tree-based search

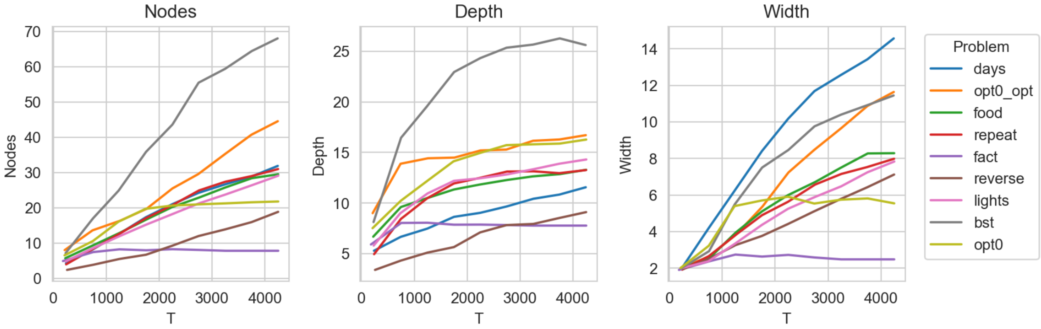


Figure 7: Average number of nodes, depth, and width of the VerMCTS search tree as the number of tokens increases across the full suite of Dafny problems. Recall that failed expansions are not added to the tree. Harder problems tend to lead to larger trees.

algorithm to generate proof trees in Lean. Different from these closely related works, we (1) focus on verified program synthesis in Dafny and Coq, and (2) leverage the verifier inside the loop of the search algorithm to efficiently guide the search.

Theorem Proving with Large Language Models Han et al. [2022] demonstrated that LLMs can be trained to generate proofs in Lean through self-supervision. Yang et al. [2023] presented that Retrieval-Augmented Generation (RAG) Glass et al. [2022] models significantly enhance LLMs' performance in theorem proving tasks. First et al. [2023] employed a methodology akin to that of Han et al. [2022] to generate and repair complete proofs in Isabelle/HOL. Jiang et al. [2023] introduced methods to first map natural language proofs to formal proof sketches in Isabelle and then fill in the gaps using an automated prover. These studies predominantly used LLMs to iteratively generate individual proof steps, which were then verified using a theorem prover. Thakur et al. [2023] propose a language-agent approach to formal theorem-proving, alternating selection and execution steps. In contrast, we focus on verified program synthesis and developing a method that effectively integrates a verifier and LLM without any additional training.

Scoring Partial Programs Desai et al. [2016], one of the first to effectively tackle the problem of program synthesis using natural language, used a scoring function to rank candidate partial programs. Cassano et al. [2023b] similarly used a scoring function to rank candidate partial programs based on their types in order to aid the tree search process, and provided multiple solutions to the user ranked by their score. Ye et al. [2021] used abstract interpretation to rule out partial programs that do not satisfy some constraints, typically on input/output examples. Chen et al. [2022] used LLM-generated unit tests suites and their pass rates to score candidate programs, and provided the user with the top-scoring program. Ni et al. [2023] further utilized execution information to rank candidate programs. Shirafuji et al. [2023] used a scoring function to rank example refactoring programs generated by an LLM before applying them to the given code. Zhang et al. [2023b] studies using scoring functions to rank candidate partial programs in-depth, and proposes the use of a *reviewer* model to score candidate programs based on how closely they match the given instruction. Most of these works have scored partial programs specified as grammatical programs with holes as opposed to our left-to-right generation of partial programs, and have not considered verified programming languages.

F Discussion

We have demonstrated that relatively weak language models can reliably produce verified code by guiding a search process that verifies partial programs at each step. Our technique shines on multi-step problems, made of dependent sub-problems. Our technique can be adapted to a setting where the interfaces and specifications are given, and the code is verified at each step by additional code containing assertions or proofs.

Limitations. A key aspect of our approach resides in the scoring of partial programs. However, the scoring is limited by coarse granularity and lack of lookahead in the scoring function. The granularity of the verification step is a whole unit, e.g. a function in Dafny and a command in Coq. For Dafny, the coarse granularity means we have to wait multiple lines to get feedback. For Coq, the fine granularity doesn't help much with bigger proofs, which require planning.

Future work. What we find most interesting and promising about our approach is that so much is possible by a "blind" search that only uses scalar reward signal. In future work, it would be fruitful to find ways of allowing the search to rely on richer feedback while maintaining the efficiency of leveraging the verifier to avoid doing costly rollouts or reflection steps. Moreover, it will be interesting to see if the basic idea of VerMCTS, using a cheap and provable upper bound on the value function to guide search, can be applied beyond the verified programming setting.

G Prompts

G.1 Repeat Prompt

Coq. *In Coq: (1) Write a function ‘repeat‘ that takes an integer ‘x‘ and a natural number ‘n‘ as inputs, and returns a list of length ‘n‘ in which every element is ‘x‘. (2) Then write a lemma ‘repeat_correct‘ that checks that for any ‘x‘ and ‘n‘, ‘repeat‘ returns a list of length ‘n‘ and that every element of the list is ‘x‘.*

Dafny. *In Dafny: (1) Write a function ‘repeat‘ that takes an integer ‘x‘ and a natural number ‘n‘ as inputs, and returns a list of length ‘n‘ in which every element is ‘x‘. (2) Then write a lemma ‘repeat_correct‘ that checks that for any ‘x‘ and ‘n‘, ‘repeat‘ returns a list of length ‘n‘ and that every element of the list is ‘x‘.*

Hints for Coq.

### Hint: Start with ‘Require Import List. Import ListNotations.‘

Check lemma for Coq.

```
Lemma CHECK_repeat_correct: ∀ (x: int) (n: nat),
      length (repeat x n) = n /       ∀ i, 0 ≤ i -> i < n -> nth (repeat x n) i = x.
    Proof.
      intros.
      eapply repeat_correct; eauto.
    Qed.
```

Check lemma for Dafny.

```
lemma CHECK_repeat_correct(x: int, n: nat)
      ensures |repeat(x, n)| = n
      ensures ∀ i • 0 ≤ i < n ⟹ repeat(x, n)[i] = x
    {
      repeat_correct(x, n);
    }
```

G.2 Opt0 Opt Prompt

Coq. *In Coq, write an ADT ‘Expr‘ for arithmetic expressions comprising constants, variables and binary addition. Then write a predicate ‘optimal‘ that holds on an expression if it has no additions by 0. Then write an optimizer ‘optimize‘ that removes all additions by 0. Then write a lemma ‘OptimizerOptimal‘ that ensures ‘optimal(optimize(e))‘ for all expressions ‘e‘.*

Dafny. *In Dafny, write an ADT ‘Expr‘ for arithmetic expressions comprising constants, variables and binary addition. Then write a predicate ‘optimal‘ that holds on an expression if it has no additions by 0. Then write an optimizer ‘optimize‘ that removes all additions by 0. Then write a lemma ‘OptimizerOptimal‘ that ensures ‘optimal(optimize(e))‘ for all expressions ‘e‘.*

Hints for Coq.

### Hint: In the addition case, the ‘optimize‘ function should recursively optimize the sub-expressions and then match on the optimized sub-expressions.
### Hint: You can import the ‘string‘ datatype with the line ‘Require Import Coq.Strings.String.‘
### Hint: Use Fixpoint instead of Definition for recursive functions.
### Hint: If you do induction on ‘e‘ with sub-expressions ‘e1‘ and ‘e2‘, the two inductive hypotheses are called ‘IHe1‘ and ‘IHe2‘.

Check lemma for Coq.

```
lemma CHECK_OptimizerOptimal(e: Expr) ensures optimal(optimize(e)) { OptimizerOptimal(e); }
```

Check lemma for Dafny.

```
lemma CHECK_OptimizerOptimal(e: Expr) ensures optimal(optimize(e)) { OptimizerOptimal(e); }
```

G.3 Lights Prompt

Coq. *In Coq: (1) Write a datatype ‘light‘ for traffic lights with cases ‘Red‘, ‘Yellow‘, ‘Green‘. (2) Write a function ‘activation‘ which takes two lights, source and target, and returns a list of lights, the first element being the source and the last element being the target. If the source and target are not yellow and are distinct, then the returned list has a middle element of yellow. (3) Write a helper ‘adjacent_ok‘ that takes two lights, and checks that they are not one red and the other green. (4) Write a helper ‘all_adjacent_ok‘ that takes a list of lights, and checks that all adjacent elements are ‘adjacent_ok‘. (5) Write a lemma ‘check_activation‘ to prove that forall source and target lights, a returned list never has adjacent elements that are distinct and red or green. The proposition should be ‘all_adjacent_ok (activation source target)‘.*

Dafny. *In Dafny: (1) Write a datatype ‘light‘ for traffic lights with cases ‘Red‘, ‘Yellow‘, ‘Green‘. (2) Write a function ‘activation‘ which takes two lights, source and target, and returns a list of lights, the first element being the source and the last element being the target. If the source and target are not yellow and are distinct, then the returned list has a middle element of yellow. (3) Write a helper ‘adjacent_ok‘ that takes two lights, and checks that they are not one red and the other green. (4) Write a helper ‘all_adjacent_ok‘ that takes a list of lights, and checks that all adjacent elements are ‘adjacent_ok‘. (5) Write a lemma ‘check_activation(source: light, target: light)‘ to prove that a returned list never has adjacent elements that are distinct and red or green. The ‘ensures‘ clause should be ‘all_adjacent_ok(activation(source, target))‘.*

Hints for Coq.

### Hint: Start with ‘Require Import List. Import ListNotations.‘

Check lemma for Coq.

```
Lemma CHECK__check_activation: ∀ (source: light) (target: light),
    all_adjacent_ok(activation(source  target).
    Proof.
      intros.
      eapply check_activation; eauto.
    Qed.
```

Check lemma for Dafny.

```
lemma CHECK__check_activation(source: light, target: light)
    ensures all_adjacent_ok(activation(source, target))
    {
      check_activation(source, target);
    }
```

G.4 BST Prompt

Coq. *In Coq, (1) write an ADT for a tree of natural numbers. Call it ‘Tree‘. Then (2) write a predicate ‘IsBST‘ that checks whether a given tree is a binary search tree (BST). Then (3) write a function ‘insert‘ that inserts an element into a binary search tree while preserving the BST property. Then (4) write a predicate ‘Contains‘ that checks whether a given tree contains a given element. Then (5) write a lemma ‘InsertContains‘ about the insert function that ensures that the tree resulting from inserting an element contains that element (without requiring nor ensuring the BST property). Then (6) write another lemma ‘InsertPreservesBST‘ about the insert function that checks the BST property continues to hold after insertion. This lemma should take bounds on the BST, and require that the element to be inserted is within those bounds.*

Dafny. *In Dafny, (1) write an ADT for a tree of natural numbers. Call it ‘Tree‘. Then (2) write a predicate ‘IsBST‘ that checks whether a given tree is a binary search tree (BST). Then (3) write a function ‘insert‘ that inserts an element into a binary search tree while preserving the BST property. Then (4) write a predicate ‘Contains‘ that checks whether a given tree contains a given element. Then (5) write a lemma ‘InsertContains‘ about the insert function that ensures that the tree resulting from inserting an element contains that element (without requiring nor ensuring the BST property). Then (6) write another lemma ‘InsertPreservesBST‘ about the insert function that checks the BST property continues to hold after insertion. This lemma should take bounds on the BST, and require that the element to be inserted is within those bounds.*

Hints for Coq.
### Hint: Start with ‘Require Import List. Import ListNotations.‘

### Hint: Use Fixpoint instead of Definition for recursive functions.
### Hint: Use ‘l‘ and ‘r‘ for variable names instead of ‘left‘ and ‘right‘ to avoid name clashes.

Check lemma for Coq.

```
// (5) Lemma about the insert function that ensures the tree resulting
//      from inserting an element contains that element
Lemma CHECK_InsertContains: ∀ (t: Tree) (x: nat),
  Contains (insert t  x) x.
Proof.
  intros.
  eapply InsertContains; eauto.
Qed.

// (6) Lemma about the insert function that checks the BST property
//      continues to hold after insertion
lemma CHECK_InsertPreservesBST: ∀ (t: Tree) (x: nat) (min: nat) (max: nat),
  (IsBST t min max) -> min ≤ x ≤ max ->
  IsBST (insert t x) min max.
Proof.
    intros.
    eapply InsertPreservesBST; eauto.
Qed.
```

Check lemma for Dafny.

```
// (5) Lemma about the insert function that ensures the tree resulting from
//      inserting an element contains that element
lemma CHECK_InsertContains(t: Tree, x: nat)
  ensures Contains(insert(t, x), x)
{
  InsertContains(t, x);
}

// (6) Lemma about the insert function that checks the BST property continues
//      to hold after insertion
lemma CHECK_InsertPreservesBST(t: Tree, x: nat, min: nat, max: nat)
  requires IsBST(t, min, max) ∧ min ≤ x ≤ max
  ensures IsBST(insert(t, x), min, max)
{
    InsertPreservesBST(t, x, min, max);
}
```

G.5 Opt0 Prompt

Coq. *In Coq, write an ADT for arithmetic expressions (called ‘Expr‘) comprising constants, variables and binary additions. Then write an evaluator (called ‘Eval‘) taking an expression and an environment (a function that takes a variable name and returns a number) and returning the number resulting from evaluation. Then write an optimizer (called ‘Optimize‘) taking an expression and returning an expression with all additions by 0 removed. Then prove that the optimizer preserves the semantics as defined by the evaluation function. Do so by proving the lemma ‘OptimizePreservesSemantics: forall (e: Expr) (env: string -> nat), Eval(Optimize(e), env) = Eval(e, env)‘.*

Dafny. *In Dafny, write an ADT for arithmetic expressions (called ‘Expr‘) comprising constants, variables and binary additions. Then write an evaluator (called ‘Eval‘) taking an expression and an environment (a function that takes a variable name and returns a number) and returning the number resulting from evaluation. Then write an optimizer (called ‘Optimize‘) taking an expression and returning an expression with all additions by 0 removed. Then prove that the optimizer preserves the semantics as defined by the evaluation function. Do so by proving the lemma ‘OptimizePreservesSemantics(e: Expr, env: string -> int) ensures Eval(Optimize(e), env) == Eval(e, env)‘.*

Hints for Coq.
### Hint: In the optimizer, recursively optimize the sub-expressions.
### Hint: You can import the ‘string‘ datatype with the line ‘Require Import Coq.Strings.String.‘.

### Hint: Use Fixpoint instead of Definition for recursive functions.
### Hint: With tactics like ‘induction‘ and ‘destruct‘, _avoid_ naming with ‘as‘ and let Coq pick the names for you. For example, use ‘induction e.‘ but _not_ ‘induction e as [...]‘.

### Hint: For the proof, do ‘induction e.‘. Do NOT name the hypotheses with ‘as‘.
### Hint: The simple cases are by ‘simpl. reflexivity.‘.
### Hint: The addition case is by ‘simpl. rewrite <- IHe1. rewrite <- IHe2. destruct (optimize e1); destruct (optimize e2); try destruct n; try destruct n0; eauto using PeanoNat.Nat.add_0_r.‘.
### Hint: You’ll need ‘Require Import Arith‘.

Check lemma for Coq.

```
Lemma CHECK_OPS: ∀ (e: Expr) (env: string -> nat), Eval (Optimize e) env = Eval e env.
    Proof.
    intros.
    apply OptimizePreservesSemantics; eauto.
    Qed.
```

Check lemma for Dafny.

```
lemma CHECK_OPS(e: Expr, env: string -> int)
    requires true
    ensures Eval(Optimize(e), env) = Eval(e, env)
{
    OptimizePreservesSemantics(e, env);
}
```

G.6 Factorial Prompt

Coq. *In Coq, write a factorial function, called ‘fac‘, and prove (in a lemma ‘FacPositive: forall (n: nat), fac n > 0‘) that the factorial is always strictly positive.*

Dafny. *In Dafny, write a factorial function, called ‘fac‘, and prove (in a lemma called ‘FacPositive(n: nat)‘) that the factorial is always strictly positive.*

Hints for Coq.

### Hint: Don’t forget to import the Arith module.
### Hint: use ‘Nat.lt_0_1‘ in the base case of the proof.
### Hint: use ‘Nat.lt_lt_add_r‘ in the inductive case of the proof.

Check lemma for Coq.

```
Lemma CHECK_FacPositive: ∀ (n: nat), fac n > 0. Proof. intros. apply FacPositive; eauto. Qed.
```

Check lemma for Dafny.

```
lemma CHECK_FacPositive(n: nat) ensures fac(n) > 0 { FacPositive(n); }
```

G.7 Food Prompt

In Dafny: (1) Write a datatype for ‘Food‘: ‘Pasta‘ or ‘Pizza‘. Each Pasta or Pizza has a list of toppings. Each ‘Topping‘ is one of: ‘tomato‘, ‘cheese‘, ‘olive‘, ‘broccoli‘, ‘mushroom‘, ‘pepper‘. (2) Write a predicate ‘ok‘ that accepts any pizza with five toppings or fewer, and any pasta with two toppings or fewer. (3) Write a lemma ‘ok3_pizza‘ that proves that an accepted food with three or more toppings must be a pizza.

Hints for Dafny.

### Hint: The length of a list or sequence ‘s‘ is ‘|s|*‘.*

Check lemma for Dafny.

```
lemma CHECK_ok3_pizza(x: Food)
    requires ok(x)
    requires |x.toppings| ≥ 3
    ensures match x { case Pizza(_) ⇒ true case _ ⇒ false }
    {
      ok3_pizza(x);
    }
```

G.8 Reverse Prompt

In Dafny: (1) Write a function ‘reverse‘ that takes a list as input and reverses it. (2) Then write a lemma ‘reverse_permutes‘ that checks that for any list ‘l‘, an element exists in ‘l‘ if and only if it exists in the result of calling ‘reverse‘ on ‘l‘. (3) Then write a lemma ‘reverse_involutes‘ that checks that for any list ‘l‘, calling ‘reverse‘ twice on ‘l‘ yields ‘l‘.

Hints for Dafny.

### Hint: The length of a list or sequence ‘s‘ is ‘|s|‘.
### Hint: Use a plain ‘function‘ to define ‘reverse‘, not a ‘function method‘ or a ‘method‘.

Check lemma for Dafny.

```
lemma CHECK__reverse_permutes(l: seq<int>)
    // TODO
    {
    }
    lemma CHECK__reverse_involutes(l: seq<int>)
    ensures reverse(reverse(l)) = l;
    {
      reverse_involutes(l);
    }
```

G.9 Days Prompt

In Dafny: (1) Write an ADT ‘Day‘ for the days of the week: ‘Sunday‘ to ‘Saturday‘. (2) Write a function ‘next_biz_day‘ that gives the next business day. (3) Write a function‘iter_biz_day(d: Day, n: nat): Day‘ that iterates the next business day function, for an arbitrary number n of business days. (4) Write a lemma ‘iter5_biz_day_idempotent‘ that ensures that starting with a business day, taking the next five business days is idempotent.

Check lemma for Dafny.

```
lemma CHECK_iter5_biz_day_idempotent(d: Day)
    requires d ≠ Saturday
    requires d ≠ Sunday
    ensures iter_biz_day(d, 5) = d
    {
      iter5_biz_day_idempotent(d);
    }
```

H Examples of Scoring Partial Programs

Partial program with a score of 0:

```
datatype Expr =
```

Partial program with a score of $+1$:

```
datatype Expr =
    | Const(val: int)
```

Partial program with a score of -1:

```
datatype Expr =
    | Const(val: int)
    | Var(name: string)
    | Add(e1: Expr, e2: Expr)

function Evaluate(e: Expr,
    env: string -> int): int
    reads env
{
    match e
    case Const(val) ⇒ val
    case Var(name) ⇒ env(name)
    case Add(e1, e2) ⇒
      Evaluate(e1, env) +
      Evaluate(e2, env)
}
```

The negative score is due to the `reads` clause, which shouldn't be there. Unfortunately, we only confirm the error once the whole function is generated.

I Broader Impacts

The development of algorithms that allow generation of verified code using smaller models has notable broader impacts on both machine learning and society. We increase the efficiency per token in code language model usage, and allow for the usage of smaller models. This further reduces energy consumption and allows for a usage of cheaper hardware, thereby democratizing access to this technology. Our approach, which is asking models to prove their work is correct, and then immediately and externally checking whether the proof is correct, can mitigate some of the open issues with trusting LLMs.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and intro clearly state the key results directly.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations is a subsection of our Discussion section.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: Our paper does not include theoretical results.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We discuss our algorithms in detail with pseudocode in section 2 and provide all used hyperparameters in Appendix C.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We provide our code, including instructions on how to run it and reproduce our experimental results.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: Yes, we specify all hyperparameters, algorithms, and and used models.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: The error bars report Wilson intervals as described in C.1.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Experiments are explicitly reported in terms of token counts which can be directly converted to compute requirements on your hardware. We use an internal cluster with A100 and H100 GPUs.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The paper conforms with the code of ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We discuss broader impacts in Appendix I.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: This paper is about optimizing results from existing models, and does not introduce new models. Hence, we believe our paper poses no such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The model we use is correctly cited in section C.2.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: The paper does not release new assets.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: Our paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: Our paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.