Assertionista: Teaching LLMs to Generate Provably Correct Code

Shourya Bansal
University of Michigan
bansalsh@umich.edu

Christine Gao
University of Michigan
chrisgao@umich.edu

Atharva Gawde
University of Michigan
agawde@umich.edu

Edison Shen
University of Michigan
shene@umich.edu

Abstract

Reinforcement learning post-training has become a widely used approach to improve the performance of large language models in specific domains. During code generation, models frequently produce syntactically correct but semantically incorrect code. We propose Assertionista, a new formal verification-based post-training architecture for code generation with two models: a writer and an annotator. In order to improve robustness, Assertionista uses the annotator as well as the Dafny verifier as a reward function for the writer model's training pipeline. We were able to construct the full training pipeline and finish several epochs of training, but unable to generate definitive results due to hardware and time constraints. We show a proof of concept by applying our work to DafnyBench, a formal software verification benchmark.

1 Introduction

Large language models often produce code that *appears* correct but contains subtle logical errors, precluding reliable use in safety-critical domains. Traditional RLHF introduces subjectivity and cost, while supervised fine-tuning cannot guarantee *provable* correctness. How can we train models to generate code that is mathematically verifiable?

Reinforcement Learning from Verifiable Rewards (RLVR) offers a promising direction. Recent work such as *DeepSeek-R1* [2] has shown that rule-based rewards can improve reasoning without human feedback [5]. Formal verification is particularly well-suited to this approach: instead of heuristic scoring, program correctness can be *proven* through logical verification [3].

We explore this intersection by implementing a dual-model architecture using Qwen2.5-3B-Instruct: separate writer and annotator models trained on Dafny[1] verification tasks with structured rewards derived from the verification pipeline. Building on GRPO for sample-efficient training, we introduce several contributions: a JSON-based annotation method reducing token overhead by 71%, a staged reward function incentivizing verification success, and a systematic investigation of practical challenges in scaling RLVR for formal verification. We also extend the DafnyBench dataset with specification-only examples requiring models to reason purely about Hoare logic [1]. All of our code can be found via a Google Drive link at Appendix C.

[1]We discuss Dafny with a bit more detail in Appendix A.

2 Background

2.1 Related Work

2.1.1 Reinforcement Learning from Verifiable Rewards and Formal Verification.

Reinforcement Learning from Verifiable Rewards (RLVR) replaces human preference signals with programmatic correctness checks. *DeepSeek-R1* [2] and *DeepSeekMath* [5] demonstrated that rule-based rewards can induce complex reasoning in LLMs without human feedback.

These results motivate applying RLVR to formal verification, where correctness is provable rather than merely testable. *DafnyBench* [3] formalizes this task, enabling evaluation using a theorem prover as ground-truth judge. However, formal verification poses unique RL challenges: extremely sparse rewards, expensive verifier feedback, and naive generation yielding syntactically valid but logically vacuous solutions. Our work addresses these through structured reward shaping and a staged, verifier-in-the-loop pipeline.

2.1.2 Optimization Methods for Verifier-Aligned Post-Training.

Most reinforcement learning pipelines for LLMs rely on *Proximal Policy Optimization (PPO)* [4], which stabilizes training via "clipped objectives" and a learned value function. While effective for many tasks, PPO introduces substantial memory and compute overhead due to the critic network, making it particularly costly in verifier-in-the-loop settings where rollouts are already expensive.

Group Relative Policy Optimization (GRPO), introduced in *DeepSeek-R1-Zero* [2], removes the critic by estimating advantages through group-wise reward normalization. This relative comparison framework is well-suited to sparse, verifiable rewards, where absolute value estimates are less informative than within-group rankings. We adopt a GRPO-style optimization scheme for both writer and annotator models, prioritizing memory efficiency and training stability under strict computational constraints.

2.1.3 Positioning of This Work.

Rather than emphasizing large-scale performance gains, our contribution lies in the design and empirical analysis of a verifier-aligned reinforcement learning pipeline under realistic computational constraints. By combining dual-model decomposition, structured action spaces, and group-relative optimization, we clarify the systems-level challenges and design trade-offs involved in scaling RLVR to formal verification tasks.

3 Methods

3.1 High-Level Overview

Assertionista's goal is to turn a programmer's *intention* into code that is provably correct. As long as the programmer is able to specify their intention using Hoare logic in Dafny [1], Assertionista should generate Dafny code that satisfies the specification, without needing any other guidance or prodding.

In a more mathematical sense, fix a Hoare triple $\{P\}\, S\, \{Q\}$[2]. Assertionista is a black box A that takes a specification and spits out a correct implementation:

$$A : (P, Q) \to S.$$

Here, we call (P, Q) the specification and S the provable implementation.

Now we will discuss the strategy of being able to create that transformation.

3.2 Assertionista Breakdown

Building a model that is able to solve the $(P, Q) \to S$ equation directly is very hard. A single correct Dafny program has many components, from the implementation correctness to the verification

[2]For readers less familiar with Hoare triples, they can be thought of as "Precondition, Implementation, Postcondition" tuples.

correctness. It turns out that we can treat the implementation and the verification as independent tasks, because in reality they *are* independent.

Assertionista is thus broken down into two components: the Annotator and the Writer.

- **The Annotator:** The annotator takes a Dafny program with a specification and implementation but *without any verification annotations*. It is then able to, using a combination of code structure, code semantics, and overall model understanding, determine where to place which annotations so that the Dafny verifier is able to prove that the program is correct (or incorrect).[3]
- **The Writer:** The writer is more high-level. Given just a function specification, the Writer generates Dafny code *without verification annotations*. This code needs to compile, but does not need to verify out of the box because it contains no verification annotations to help the solver to start off.

This split is motivated by:

1. **systems constraints**: full-program regeneration during RL rollouts is token-heavy and increases verification latency, and
2. **optimization structure**: algorithm synthesis and proof-hint synthesis behave differently under verifiable reward signals.

The writer is encouraged to produce plausible executable code, while the annotator is forced to engage directly with formal proof obligations via localized edits that are inexpensive to generate and apply.

Ultimately, the pipeline looks like:

$$\text{Assertionista}(\text{Spec}) = \text{Annotator}(\text{Writer}(\text{Spec})).$$

And to verify that out output is correct, we can use the Dafny verifier as the automated judge, playing into the "verifiable rewards" of RLVR that we discussed earlier.

3.3 Improving the DafnyBench Dataset

To train Assertionista, we used the open-source Huggingface DafnyBench dataset. However, the dataset was insufficient for our purposes: While it contained "ground truth," and "hints removed" columns, which contained a fully working Dafny program and the program without verification hints respectively, this wasn't going to helpful for training the Writer Model: it only needed Dafny specifications.

Thus, we introduced a new dimension to each data point: the "specification," that we dubbed `unimplemented_body`. This removed all method and lemma bodies to create a skeleton version of each program. Now, each data point yielded training triples

$$(\texttt{unimplemented\_body},\ \texttt{body} = \texttt{hints\_removed},\ \texttt{annotated\_body} = \texttt{ground\_truth}),$$

enabling a staged curriculum in which a model can learn:

1. implementation synthesis from skeletons, and
2. verification-guided annotation recovery.

3.4 Annotator Interface: JSON Patch Action Space

The annotator's action space is a structured annotation patch, rather than free-form code:

$$J = [a_1, \ldots, a_K], \qquad a_k = (\texttt{line}_k, \texttt{content}_k).$$

Each element specifies a line number and a single-line Dafny annotation string to be inserted before that line. The repository enforces this interface via a strict prompting contract:

[3]To save tokens and accelerate model output, the annotator outputs JSON "patches" instead of the modified Dafny code. We will discuss this strategy later in the paper.

- Reasoning must appear inside `<think>...</think>`, and the patch inside `<json>...</json>`.
- The `<json>` block must contain only a JSON array of objects with fields `line` and `content`.
- No prose, code fences, or multi-line annotation content is permitted inside `<json>`.
- Line numbers are anchors into the current unannotated Dafny source.

Given a base program P (either the writer output or the dataset `body`) and an edit list J, we construct

$$\widehat{P} = \mathsf{ApplyEdits}(P, J).$$

Insertions are applied in descending order of line number to avoid index shifting. The source is split into lines, each `content` line is inserted at index $\max(0, \texttt{line} - 1)$, and a trailing newline is restored if present. This design ensures that the mapping from model output to executable artifact is fully deterministic, and that all rewards are computed on a concrete $\widehat{P}$.

3.5 Verifier-Centric Reward Function for the Annotator

The annotator is optimized directly against a verifier-aligned reward computed on $\widehat{P}$ after parsing, patching, compilation, and verification. The reward is stage-gated with early exits. **Format validity.** If the response is missing `<json>...</json>` tags or contains invalid JSON (fails schema validation), the reward is 0. If the patch parses and validates, we grant a fixed format reward

$$r_{\text{fmt}} = 0.3.$$

Anti-cheating constraint (`assume` penalty). If any inserted annotation begins with `assume`, we apply a negative reward and terminate evaluation:

$$r_{\text{assume}} = \begin{cases} -1.0 & \text{if any } \texttt{content} \text{ starts with } \texttt{assume}, \\ 0 & \text{otherwise.} \end{cases}$$

Compilation. If $\widehat{P}$ fails Dafny parsing or compilation (checked via `compile_no_verify`), the trajectory receives no further reward. Otherwise,

$$r_{\text{cmp}} = 1.0.$$

Verification. If Dafny verifies $\widehat{P}$ (checked via `verify`), we grant

$$r_{\text{ver}} = 3.0,$$

and 0 otherwise.

Total annotator reward. Let R_{ann} denote the total annotator reward:

$$R_{\text{ann}} = r_{\text{fmt}} + r_{\text{assume}} + r_{\text{cmp}} + r_{\text{ver}},$$

with evaluation short-circuiting at the first failing stage. This yields a dense learning signal even when full verification is rare, while preserving a strong incentive for verifier success.

3.6 Writer Model: From `unimplemented_body` to `hints_removed`-Style Implementations

The writer is conceptually upstream of the annotator and is trained to *implement* missing method and lemma bodies without producing proof annotations. In the full pipeline, the writer's input is `unimplemented_body` and its output is a complete Dafny program P_{write} that compiles and resembles the dataset's `body` distribution (i.e., `hints_removed`: implemented code with minimal or no proof hints). This matches the repository's dataset construction, in which each row contains both `unimplemented_body` and `body`.

Writer RL objective via downstream verification. The writer is not rewarded directly on its raw text. Instead, it is evaluated *through* the annotator and verifier. Given writer output P_{write}, we run the annotator to produce a patch J, form $\widehat{P} = \mathsf{ApplyEdits}(P_{\text{write}}, J)$, and compute verifier outcomes on $\widehat{P}$. The writer is optimized using a reward that reflects the *existence of a small verification repair*:

$$R_{\text{write}} = \mathbf{1}_{\{\mathsf{Verify}(\widehat{P})=1\}} + \lambda_{\text{cmp}}\, \mathbf{1}_{\{\mathsf{Compiles}(P_{\text{write}})=1\}},$$

optionally augmented with auxiliary terms that encourage a "good substrate" for annotation (e.g., discouraging gratuitous control-flow complexity or degenerate returns). Intuitively, this trains the writer to produce implementations that are close, in proof space, to verifiable programs and that admit small localized repairs by the annotator.

3.7 Policy Optimization: Group-Relative Updates (GRPO-Style Normalized REINFORCE)

We optimize policies using the repository's GRPO-style training loop, which implements group-relative reward normalization followed by a policy-gradient update over generated tokens.

For each prompt x in a minibatch, we sample M completions from the current policy $\pi_\theta(\cdot \mid x)$ via multinomial sampling from the next-token distribution. This yields groups $\{(x, y_i)\}_{i=1}^{M}$ that share the same prompt.

Before computing gradients, rewards are normalized within each prompt group:

$$\widetilde{R}_i = \frac{R_i - \mu_x}{\sigma_x + \varepsilon}, \qquad \mu_x = \frac{1}{M}\sum_{i=1}^{M} R_i,\;\; \sigma_x^2 = \frac{1}{M}\sum_{i=1}^{M}(R_i - \mu_x)^2,$$

where ε is a small constant for numerical stability. Updates depend on whether a completion outperforms its siblings for the same prompt, rather than on an absolute baseline.

We compute log-probabilities of generated tokens (masking out prompt tokens) and optimize a standard REINFORCE objective using $\widetilde{R}_i$ as an advantage-like scalar:

$$\max_\theta\; \mathbb{E}\left[\widetilde{R} \cdot \sum_{t \in \text{gen}} \log \pi_\theta(y_t \mid x, y_{<t})\right].$$

Gradients are accumulated over microbatches and globally clipped.

Although GRPO is often presented with an explicit KL penalty against a reference policy, the current repository implementation performs group-relative normalization and on-policy token-gradient updates without an explicit KL term. This choice is driven by simplicity and memory constraints; a KL penalty can be incorporated by maintaining a frozen SFT reference model and adding a per-token KL term to the objective.

3.8 End-to-End Evaluation Protocol

All reported metrics are computed using the same deterministic pipeline employed during training:

$$P \xrightarrow{\text{(optional) writer}} P_{\text{write}} \xrightarrow{\text{annotator}} J \xrightarrow{\mathsf{ApplyEdits}} \widehat{P} \xrightarrow{\text{Dafny compile}} \xrightarrow{\text{Dafny verify}} \text{score}.$$

We log both scalar rewards and component-wise indicators (format validity, compilation success, verification success, and `assume` violations), enabling fine-grained failure attribution (e.g., "invalid JSON" vs. "compiles but does not verify") during analysis.

4 Evaluations

4.1 Evaluation Baseline

We evaluate the effectiveness of our training by comparing the model against the baseline in generating correct annotations. All experiments were run on an AWS g6e.2xlarge instance with one A40 GPU (48 GB VRAM), 8 vCPUs, and 64 GB of memory.

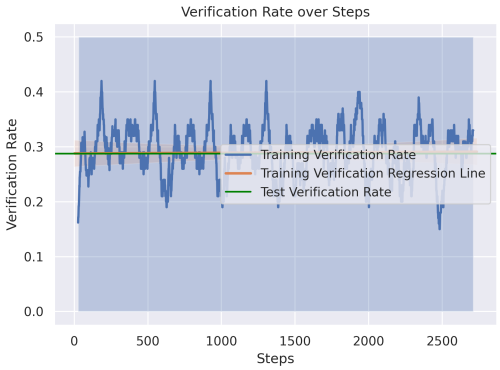

Figure 1: Verification Rate over Iterations

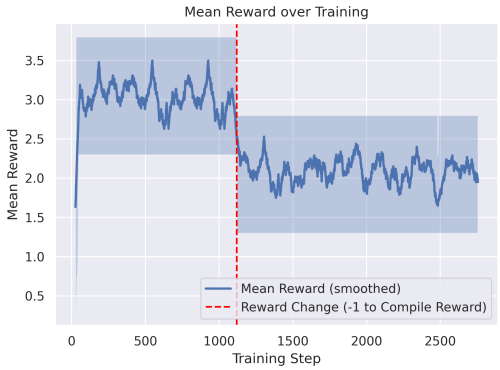

Figure 2: Mean Reward Over Time

4.2 Results

4.2.1 JSON-based Patching

In order to save computation and memory bandwidth, we changed our annotator's action space from raw code to a diff-based JSON schema, allowing it to edit existing code rather than rewrite it. We observed that this reduced the average number of tokens generated per sample in the DafnyBench dataset from 411 to approximately 120 tokens—a 71% reduction in output length. Our model achieved consistent format compliance after several training iterations due to the formatting reward in our pipeline.

4.2.2 Training Dynamics

We executed approximately 4.4 epochs of training over 2,735 optimization steps on the DafnyBench dataset. Figure 1 shows the verification rate over training, while Figure 2 displays the smoothed mean reward trajectory.

Verification Performance. The training verification rate fluctuated substantially between 0.05 and 0.45 throughout training, with a mean of 29.0% ($\sigma = 0.313$). Linear regression analysis reveals an essentially flat trend (slope $\approx 10^{-6}$, $R^2 < 0.001$, $p = 0.88$), indicating no statistically significant improvement in verification success over the course of training. The test verification rate remained constant at approximately 29%, closely matching the training mean. This suggests that while the model maintained its initial capability, the RL signal was insufficient to drive measurable gains in proof generation.

Reward Function Refinement. At approximately step 1,200, we modified the reward function to reduce the weight assigned to compilation success, shifting emphasis toward verification outcomes. This change is visible in Figure 2 as a sharp drop in mean reward. Prior to the change, the model received a mean reward of 2.93 ± 1.14; afterward, mean reward decreased to 2.09 ± 0.99. Crucially, this reward adjustment did not affect verification rates—mean verification remained at 29% both before and after the change—confirming that the compilation signal was contributing to reward inflation without driving the model toward better proof annotations.

Compilation and Format Success. Despite the flat verification trajectory, the model demonstrated strong performance on intermediate objectives. After the reward refinement at step 1,200, compilation success rate reached 94.5%, indicating that the model reliably generates syntactically valid Dafny code. Format compliance (valid JSON patch generation) remained near 100% throughout training, confirming that the format reward successfully shaped the model's output structure. These results validate that our staged reward function effectively shaped behavior for format adherence and compilation, even though the final verification objective proved more elusive.

Exploration Behavior. Policy entropy decreased from 0.37 in the first 100 steps to 0.24 in the final 100 steps, indicating that the model became increasingly deterministic in its annotation strategies over training. While reduced entropy can signal convergence to a stable policy, in the context of

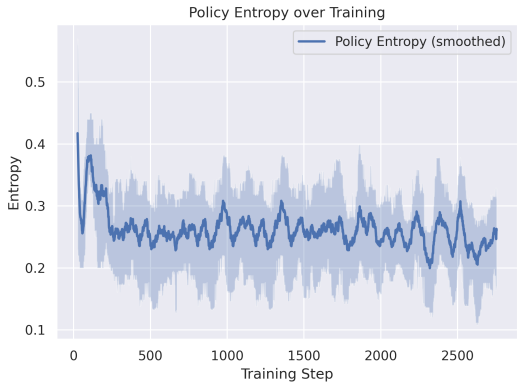

Figure 3: Entropy over Time

sparse verification rewards, this may also reflect premature exploitation of local optima—generating patches that compile but do not verify, as seen in Figure 3.

4.2.3 Interpretation

The experimental results reveal a characteristic challenge of RLVR in sparse-reward domains: without sufficient positive signal, policy gradient methods struggle to discover improved behaviors. The model learned to satisfy the denser intermediate rewards (format validity, compilation success) but failed to translate these into higher verification rates. This aligns with our hypothesis that supervised fine-tuning on verified examples is a necessary precursor to effective RL, providing a warm start from which the model can refine, rather than discover, proof strategies.

The flat verification curve, combined with high compilation rates, suggests the model converged to generating plausible-looking annotations that satisfy syntactic constraints without capturing the semantic requirements of Dafny proofs. This mode of failure—producing compilable but unverifiable outputs—underscores the difficulty of learning formal reasoning purely from sparse binary feedback.

4.3 Challenges

Given the ambitious nature of our dual-model design, our experimental execution was constrained by significant practical challenges. These limitations directly impacted the scope and outcomes of our study.

Computational Bottlenecks. The most substantial challenge was the high computational and memory cost of the RLVR training loop on our target dataset. Even with access to an NVIDIA A40 GPU (48 GB VRAM), parallel processing of rollouts for multiple prompts quickly exhausted available memory. To avoid memory failures,[4] we were forced to run the environment and reward computations sequentially, leading to an approximate runtime of two minutes per Dafny sample. With 620+ samples per epoch, a single epoch required roughly 20 hours while evaluation spanned over 4 hours. This severely limited our capacity for extensive hyperparameter tuning and the multi-epoch training typically required for policy convergence. Consequently, we were only able to complete the RLVR training pipeline for the *annotator* model for 4 epochs, and the *writer* model training was not feasible within our computational and time budget.

Sparse Reward and Insufficient Pre-Training. The Dafny verifier provides a definitive, binary signal, creating an environment with extremely sparse rewards. A more effective strategy would be to conduct *Supervised Fine-Tuning (SFT)* on verified Dafny examples *before* initiating RL. However, we could not fully implement this enhancement due to time constraints after realizing that Reinforcement Learning was not an effective starting point. This addition would provide the model with a stronger initial policy that already generates syntactically valid and often correct annotations, drastically

[4]It took us weeks of debugging to realize that our constant "out of memory" issues were not because of faulty GRPO implementations, but rather simply the hardware not being strong enough to support the level of training we needed.

increasing the frequency of positive rewards at the start of RL and providing a much richer learning gradient. Our lack of SFT meant the RL policy began from a suboptimal starting point, making it difficult to overcome the exploration challenge inherent in the sparse reward space.

Environment Complexity. Integrating the Dafny verifier as a training environment introduced latency and complexity. Each reward computation required spawning a subprocess, compiling code, and running the verifier, a non-trivial overhead that compounded our runtime issues. Furthermore, transient system issues or timeouts in the SMT solver occasionally produced ambiguous reward signals, requiring significant error handling in the training loop.

Ultimately, our results should be interpreted as a proof-of-concept validation of the integrated RLVR-for-verification pipeline rather than a demonstration of converged model performance. Over the four epochs of training we completed, we observed minimal improvement in verification success rate, a rather expected outcome given the combination of sparse rewards, limited training duration, and the absence of pre-training via supervised fine-tuning. This suggests that while the pipeline is functional, achieving meaningful performance gains likely requires both a stronger initial policy and a much longer training horizon.

4.4 Next Steps

Given the challenges outlined, future work can address these limitations through a more structured training pipeline and scaled computational resources.

A Structured Two-Phase Training Pipeline. Our experience underscores that RLVR in formal verification cannot succeed from a cold start, especially when hardware constrained. Future efforts must adopt a deliberate two-phase strategy: (1) extensive *Supervised Fine-Tuning (SFT)* to mastery on verified Dafny examples, establishing a strong initial policy π_{SFT}; followed by (2) RL fine-tuning with the verifier as a reward source. This sequence mirrors curriculum learning—first imitating correct solutions, then optimizing for verifiability. We hypothesize that SFT would raise the initial reward baseline high enough for RL to receive a meaningful learning signal, transforming an intractable sparse-reward problem into a tractable policy refinement task.

Scaled Infrastructure for Exploration and Stability. With a competent π_{SFT} in place, the RL phase would still require significant compute to explore the space of provably correct variants. Multi-GPU or TPU-based rollout collection would enable larger sample groups ($G > 4$) and more epochs, improving advantage estimation in GRPO and allowing the policy to discover more robust annotations. Under these conditions, we conjecture the RL-tuned model could surpass the SFT baseline not just in final success rate, but in *generalization*—learning to synthesize correct invariants for problems not seen during SFT, as the reward signal directly reinforces logical soundness rather than pattern matching.

5 Conclusion

In this work, we presented Assertionista, a dual-model architecture for training large language models to generate provably correct code. While we were not able to produce definitive results due to computational complexity and hardware limitations, we outlined a proof of concept that can be utilized in further research. We were able to prove the validity of our JSON-based patching method, the calculation of our verifier-based rewards, and the policy gradient updates to our model.

We believe that Assertionista proves the viability of formal verification-based systems within language models. In the future, we plan to experiment with supervised fine-tuning to augment the reinforcement learning training pipeline, and run our system end-to-end on more performant hardware.

Contributions by Author

This project was deeply collaborative, with all authors contributing substantively to every stage of the work, including design, implementation, experimentation, analysis, and writing:

1. **Shourya Bansal.** Integrated the Dafny verifier, cleaned the dataset, graphed results, and did a lot of debugging.

2. **Atharva Gawde.** Wrote the JSON patching logic, including applying the edits and constructing the reward function.
3. **Christine Gao.** Wrote the training pipeline and ran the experiments, including parameter tuning and logging.
4. **Edison Shen.** Wrote the GRPO logic, including advantage calculation, policy gradient updates, and minibatching.

Everyone contributed equally to the writing of the project proposal and final paper, as well as the original ideation.

References

[1] Jonathan Aldrich. *Lecture Notes: Hoare Logic 17-654/17-754: Analysis of Software Artifacts*. 1BC. URL `https://www.cs.cmu.edu/~aldrich/courses/654-sp09/notes/3-hoare-notes.pdf`.

[2] Wenfeng Liang, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and et al. Deepseek-r1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. doi: 10.48550/arXiv.2501.12948.

[3] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. Dafnybench: A benchmark for formal software verification. *arXiv preprint arXiv:2406.08467*, 2024. doi: 10.48550/arXiv.2406.08467.

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. doi: 10.48550/arXiv.1707.06347.

[5] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Yu Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. doi: 10.48550/arXiv.2402.03300.

A A Dafny Overview

For some additional context to Assertionista, we will give a brief overview of the Dafny programming language, motivating its choice in Assertionista.

Per the official Dafny documentation, "Dafny is a *verification-aware programming language that has native support for recording specifications and is equipped with a static program verifier. By blending sophisticated automated reasoning with familiar programming idioms and tools, Dafny empowers developers to write provably correct code (w.r.t. specifications)*."

Given a set of invariants and assumptions, Dafny uses sophisticated solvers to ensure that, for all possible code execution paths, the program provably maintains all given invariants, assumptions, preconditions, postconditions, and the like.

Because Dafny balances using a powerful solver (minimizing the amount of intervention needed in annotating code) with an accessible syntax and ease-of-use, it was the perfect way to start post-training a Large Language Model to write correct code. Using the tools that Dafny has already created, the model's job would be much easier when compared to more low-level formal verification languages.

B The Annotator Prompt

We used the following curated prompt, after much iteration, to have the Annotator model generate correct patches.

````
An interaction between a user and an expert mathematician / Dafny programmer. The
user provides a Dafny program with no or incomplete annotations that fails to verify
. The expert explains what annotations are missing and provides a JSON "patch" to
fix the program. The JSON patch is an array of objects with the following fields:
JSON SCHEMA:
{
  "$defs": {
    "DafnyAnnotation": {
      "properties": {
        "line": {
          "title": "Line",
          "type": "integer"
        },
        "content": {
          "title": "Content",
          "type": "string"
        }
      },
      "required": [
        "line",
        "content"
      ],
      "title": "DafnyAnnotation",
      "type": "object"
    }
  },
  "items": {
    "$ref": "#/$defs/DafnyAnnotation"
  },
  "title": "DafnyAnnotationList",
  "type": "array"
}
STRICT RULES:
* No prose, comments, code fences, or backticks inside <json>...</json> tags.
* Each annotation's `content` field is a single Dafny annotation line (no newlines,
no indentation).
* Use 1-based line numbers and specify the line *before* which the annotation is
inserted.
* Multiple annotations may share the same line number (apply in listed order).

The reasoning process and json output are enclosed within <think> </think> and <json
> </json> tags, respectively, i.e., <think> reasoning process here </think> <json> [
JSON array ONLY] </json>. The annotations must be necessary and sufficient to make
the program verify. After you close the </json> tag, the end token follows
immediately.

User:
```
{dafny_body}
```

Expert Mathematician/Dafny Programmer: <think>
````

C Code

Original Code:

https://drive.google.com/file/d/1UpNhp49-U7nIQ7MFcCTz-1MMFF3avWUb/view