ByteDance | Seed

Seed-Prover 1.5: Mastering Undergraduate-Level Theorem Proving via Learning from Experience

ByteDance Seed AI4Math

Abstract

Large language models have recently made significant progress to generate rigorous mathematical proofs. In contrast, utilizing LLMs for theorem proving in formal languages (such as Lean) remains challenging and computationally expensive, particularly when addressing problems at the undergraduate level and beyond. In this work, we present **Seed-Prover 1.5**, a formal theorem-proving model trained via large-scale agentic reinforcement learning, alongside an efficient test-time scaling (TTS) workflow. Through extensive interactions with Lean and other tools, the model continuously accumulates experience during the RL process, substantially enhancing the capability and efficiency of formal theorem proving. Furthermore, leveraging recent advancements in natural language proving, our TTS workflow efficiently bridges the gap between natural and formal languages. Compared to state-of-the-art methods, Seed-Prover 1.5 achieves superior performance with a smaller compute budget. It solves **88% of PutnamBench** (undergraduate-level), **80% of Fate-H** (graduate-level), and **33% of Fate-X** (PhD-level) problems. Notably, using our system, we solved **11 out of 12 problems** from Putnam 2025 within 9 hours. Our findings suggest that scaling learning from experience, driven by high-quality formal feedback, holds immense potential for the future of formal mathematical reasoning.

Project Page: https://github.com/ByteDance-Seed/Seed-Prover
Model ID: Doubao-Seedprover-1.5

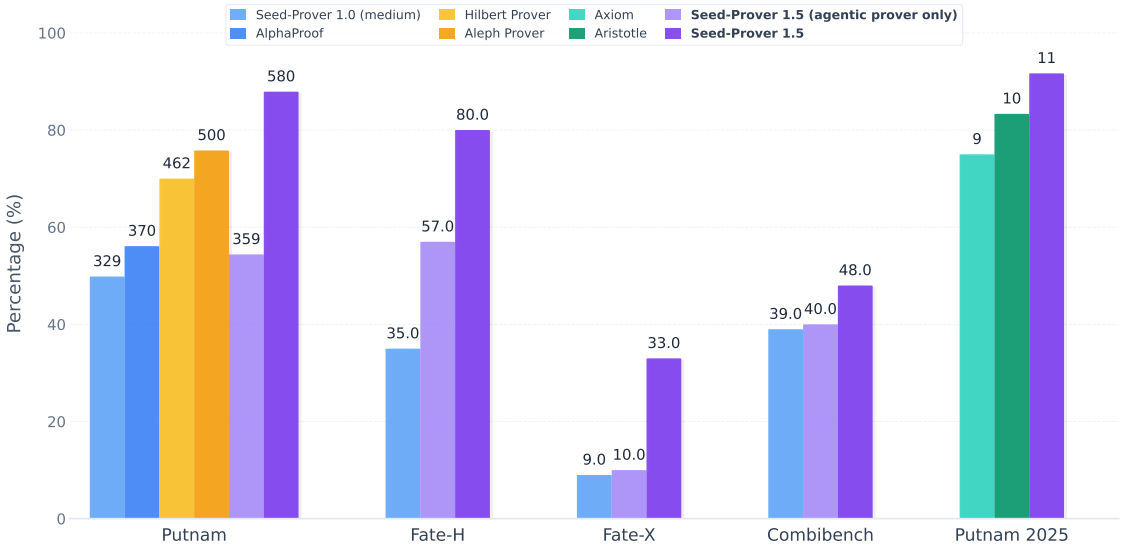


Figure 1 Performance of Seed-Prover 1.5 compare to other state-of-the-art provers.

1 Introduction

Leveraging Lean[15] for mathematical theorem proving offers fully trustworthy verification, effectively eliminating hallucinations and logical errors pervasive in natural language proof [30]. Consequently, formal verification is widely regarded as a potential paradigm shift for mathematical research. However, recent progress in natural language reasoning by LLMs challenges the necessity of this shift. Through specialized RL training and inference workflows, LLMs can not only derive correct answers using natural language, but also ensure a high degree of correctness and rigorousness in the proving process [5, 7, 20]. In stark contrast, a massive gap in capability and efficiency persists between formal and natural language proving. For example, while DeepSeek-Math-V2 [20] achieved near-perfect scores on the Putnam 2024 competition, AlphaProof [8] solved only 56% of the full Putnam benchmark which is on average simpler than the 2024 set, despite consuming massive computational resources (approximately 500 TPU-days per problem). If LLMs can achieve high rigor in natural language proof, while formal proof continues to impose a heavy performance tax, one might question: Is pursuing formal theorem proving with LLMs still a viable and valuable path?

We argue that the answer is yes. In addition to the potential of formal theorem proving on AI4Math, Lean provides a unique advantage: it serves as a fully verifiable environment where models can freely explore, accumulate experience, and undergo large-scale **Agentic RL** training with ground-truth feedback. Distinct from the step-level mode (one interaction per tactic)[1, 8, 27, 28] or the whole-proof mode (long thinking followed by a single interaction)[3, 18, 23], an agentic prover equipped with experiential learning dynamically adjusts its interaction granularity and masters auxiliary tools. Therefore, it represents a superior paradigm in terms of both capability and efficiency. Yet, training such an agentic prover via large-scale RL remains an underexplored frontier area. In this work, we demonstrate the scaling potential of this approach.

Furthermore, given the significant improvement in the ability of LLMs to generate rigorous natural language proofs, it is natural to leverage this capability to assist and accelerate formal proving. We train a sketch model to generate lemma-based Lean sketches, establishing an effective bridge between natural language and formal language. The sketch model acts as a hierarchical problem decomposer, enabling the agentic prover to solve sub-problems in parallel, which underpins our design of an efficient test-time scaling workflow.

In this work, our contributions are as follows:

1. We established an environment integrating Lean with various tools. Through large-scale RL, we trained an agentic prover to learn optimal interaction strategies and tool usage for formal theorem proving.
2. We trained a sketch model using Rubric RL to bridge natural language proofs with formalization. Based on this, we implemented a highly efficient test-time workflow.
3. We present Seed-Prover 1.5, a system that pushes the boundaries of automated formal proving while maintain a moderate compute budget. It achieves state-of-the-art performance across key benchmarks: solving 88% of Putnam, 80% of Fate-H, and 33% of Fate-X problems. Notably, our system successfully proved 11 out of 12 problems from the 2025 Putnam Competition within a 9-hour window. These performances show LLM-based formal proving has narrowed the gap with natural language reasoning in terms of both capability and efficiency at the undergraduate and graduate levels.

2 Related Works

Automated theorem proving is a challenging task in artificial intelligence [17, 24, 32]. Several systems have integrated Lean 4 with large language models to achieve IMO-level mathematical proving capabilities [1, 3, 8]. Broadly, large language model-based provers fall into two categories: step-level interaction[1, 8, 25, 27, 28] and whole-proof generation[3, 9, 12, 13, 18, 19, 23, 26, 33]. Step-level models generate a single tactic for a given proof state and interact with Lean incrementally at each tactic step, while whole-proof models produce complete Lean code and interact with Lean only once via the full code. However, both paradigms suffer from inefficient interaction with Lean: step provers interact too frequently, while whole-proof models interact too sparsely. In contrast to prior works, Seed Prover 1.5 is an agent-based prover that interacts with Lean through lemmas, offering a more balanced and efficient interaction. Recently, Hilbert [22] employs a powerful general reasoning model for informal mathematical proving and a specialized Lean model for formal verification,

achieving strong performance on PutnamBench [21]. In contrast to Hilbert, our approach utilizes rubric reinforcement learning [6] to train a sketch model that bridges natural language proofs and Lean sketches, while training an extremely strong agent-based Lean provers.

3 Approach

3.1 Agentic Prover

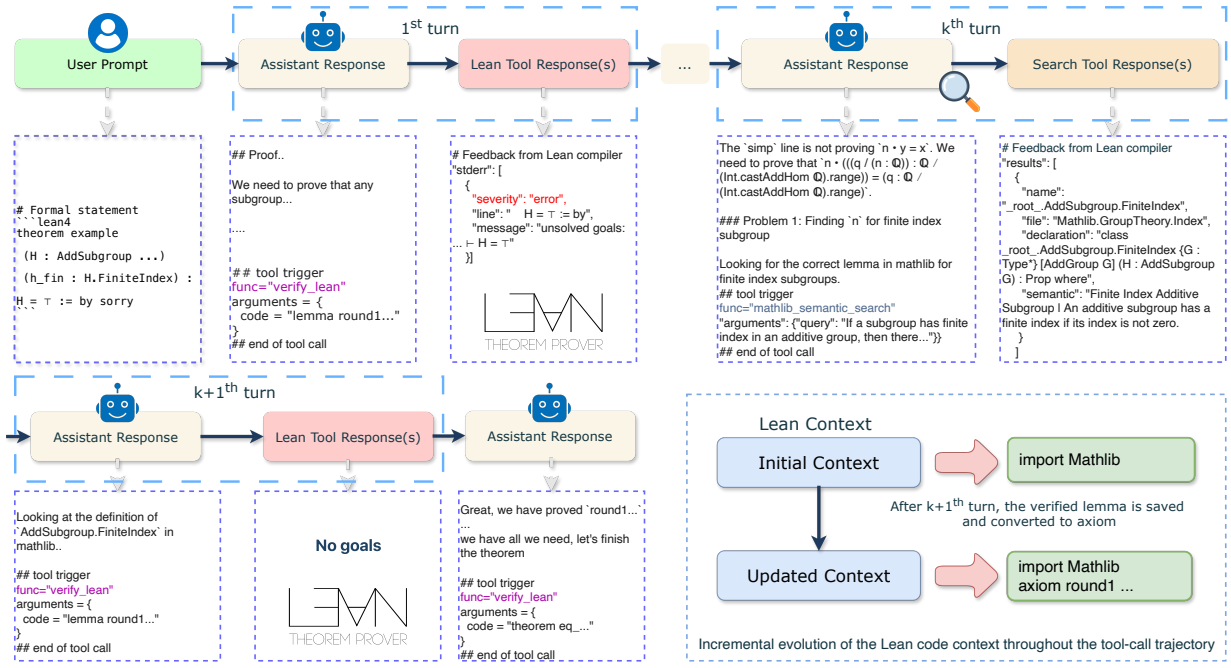


Figure 2 An example of Seed-Prover 1.5 agentic prover running on FATE-H problem.

Unlike the prior agentic prover [19], which generates an entire proof and repeatedly evaluates it using the Lean compiler, we propose a more efficient strategy that incrementally invokes multiple tools to construct formal proofs step by step. Figure 7 illustrates the inference workflow of our agentic prover. Once a lemma is successfully compiled, it is cached in memory and reused in subsequent reasoning steps, eliminating the need to regenerate previously verified code. This incremental caching mechanism enables more efficient utilization of the context window compared to approaches based on whole-proof generation. This representation offers several advantages:

- **Alignment with Modular Proofs**: It seamlessly integrates with our previously proposed lemma-style proof representation [3].
- **Decomposition of Complexity**: It relieves the model of the necessity to generate the whole proof. Instead, the model can focus on resolving the immediate sub-goal.
- **Context Efficiency**: By sequentially caching valid lemmas, we significantly reduce context overhead compared to approaches that must iteratively regenerate the full proof history.
- **Flexible Inference Control**: It enables the implementation of diverse inference strategies, such as pruning irrelevant intermediate steps or backtracking to restart the conversation at specific points.

Tools Our tools can be grouped into three categories: LEAN verification, Mathlib search[1], and Python execution. For LEAN verification, we employ LooKeng [3], a REPL[4]-based Python interface that compiles

[1] https://github.com/leanprover-community/mathlib4

Lean proofs and returns structured feedback to the model. We permit the model input a lemma at each time instead of a whole proof. The statement header and proved lemmas are stored in the running context. For Mathlib search, we use embedding-based retrieval to identify relevant theorems by semantic similarity, calibrated to a fixed Mathlib commit (i.e., v4.22.0) to ensure consistency and reproducibility. The search tools can return the most relevant *theorem*, *lemma*, or *def* declarations whose semantics align closely with the given query. Finally, we provide a Python execution interface that allows the model to generate and run Python scripts, enabling numerical experiments and other computational checks within the proving trajectory.

Inference The input prompt is a Lean formal statement with an optional natural-language proof or other auxiliary instructions. The model begins by reasoning in natural language and calling tools to validate its intermediate proof steps. Multiple tool calls may occur within a single turn, and we impose no restrictions on the number or ordering of such calls. As illustrated in Figure 7, the model may invoke "*Mathlib search*" to explore available theorems and understand how they can be applied. Once sufficient context is gathered, the model proceeds to construct a formal proof of the lemma for Lean verification. Because the formal goal is known in advance, generation terminates either when the final theorem is successfully verified or when the interaction budget (maximum number of turns or maximum sequence length) is exhausted. For all subsequent experiments, we configure our tool-use agent with a maximum sequence length of 64K and a limit of 28 tool calls. This process is viewed as Pass@1 in our setting. We can also apply light inference introduced in Seed-Prover [3] by applying self-summarizing when the interaction budget is exhausted.

3.2 Post-training of Agentic Prover

Cold Start We build upon our prior model in Seed-Prover 1.0 [3], and further post-train it to function as an agentic tool-use model tailored in LEAN environment. To support this transition, we construct in-house synthetic training data and use it to perform supervised fine-tuning (SFT), enabling the model to learn our tool invocation patterns and interactions specific to the proving environment.

RL training Following our prior work [3], we design several task formats for RL training, including proving directly from the formal statement, proving conditioned on a natural-language proof sketch, and proving based on a summary of previous failed attempts. Our training set comprises a mixture of of a combination of publicly available datasets [2, 11, 16, 23, 29] and in-house formalized math textbooks including Graduate Texts in Mathematics. To construct a high-quality RL dataset, we further filter examples by evaluating the SFT model under a light inference setting (Pass@4 $\times$ 8). In this setting, if the model fails to complete a proof within a trajectory, it performs self-summarization over this trajectory and initiates a new trajectory conditioned on the summary. We exclude any example that the model successfully proves more than three times, thereby focusing RL training on sufficiently challenging instances where additional learning signals are most beneficial. We also remove samples that cannot be proved by the SFT model under any prompting strategy. Notably, if a formal statement is provable when conditioned on a summarization prompt but not under the direct proving prompt, we retain such examples under the direct proving prompt, as this sample is provable and potentially could improve the model's efficiency. We implement our RL algorithm based on VAPO [31] and follow similar approach in ReTool [4] to enable tool-integrated reinforcement learning. We adopt a simple outcome-based reward function: the model receives a reward of 1 if a valid proof is completed and verified by the LEAN compiler, and -1 otherwise.

$$\mathcal{L}_{\text{PPO}}(\theta) = -\frac{1}{\sum_{i=1}^{G}|o_i|}\sum_{i=1}^{G}\sum_{t=1}^{|o_i|}\min\left(r_{i,t}(\theta)\hat{A}_{i,t}, \text{clip}\left(r_{i,t}(\theta), 1-\varepsilon_{\text{low}}, 1+\varepsilon_{\text{high}}\right)\hat{A}_{i,t}\right), \tag{1}$$

where G is the training batch size, o_i is the trajectory of the i^{th} sample, $\hat{A}_{\cdot,t}$ is the estimated advantage at time step t, and ε is a hyperparameter for the clipping range. $r_{\cdot,t}(\theta) = \frac{\pi_\theta(a_t|s_t;\,\mathcal{T})}{\pi_{\theta_{\text{old}}}(a_t|s_t;\,\mathcal{T})}$ is the probability ratio and π_θ represents the rollouts with interleaved tool calls and tool responses $\mathcal{T}$. During training, the model will interact with the environment to execute tool calls and obtain the tool response for multi-turn generation. Our model demonstrates substantial improvement after RL training (Table 1), with the single model significantly outperforming the previous medium workflow. We expect that iteratively leveraging the RL-trained model to collect additional data may further improve performance, which we leave for future investigation.

3.3 Sketch Model

To harness the strong natural language proving capabilities, we propose to train a sketch model. This model synthesizes a lemma-style Lean sketch[3] from a formal statement with its natural language proof. By generating auxiliary lemmas (initially admitted via `sorry`) and organizing them into a main proof body, the model decomposes the proposition into N independent sub-goals. Since Lean guaranties the high-level structural soundness, evaluation shifts to the quality of the lemmas: a sketch is valuable if its lemmas are mathematically valid, and the most challenging lemma is strictly easier than the original proposition.

To train this sketch model, we utilize VAPO [31] with a hybrid reward signal. The Lean compiler ensures the correctness of the sketch structure, while an LLM-as-a-Judge Rubric acts as a semantic value model, employing Long Chain-of-Thought (Long-CoT) to achieve better generalization than scalar-based models. We require the natural language prover to verify each lemma, immediately rejecting the sketch (i.e. natural language quality score is -1) if any lemma is mathematically invalid. We find using the natural language prover to disprove is cheaper than using a theorem prover. Subsequently, we prompt an LLM to evaluate the overall sketch quality, considering factors including alignment with the NL proof, decomposition granularity, difficulty reduction, and Lean junk value analysis. Finally, we fuse these metrics into a strict binary reward $(+1/-1)$: Let N_{lemmas} be the number of generated lemmas, S_{FL} be the lean verification score, and S_{NL} be the natural language quality score. To encourage sufficient decomposition and high quality, we define the reward function R as follows:

$$R = \begin{cases} 1 & \text{if } N_{\text{lemmas}} \geq 3 \wedge S_{\text{FL}} \geq 0 \wedge S_{\text{NL}} \geq 0.7, \\ -1 & \text{otherwise.} \end{cases} \tag{2}$$

Then the sketch model is optimized using VAPO similar to Equation 1. The prompt we used for lemma verification and rubric reward and the sample output of the sketch model is shown in Appendix B.

3.4 Test-Time Workflow

Proving complex mathematical theorems in Lean typically entails thousands of lines of code, rendering monolithic, single-pass generation computationally intractable. To address this, we implement a hierarchical test-time workflow that facilitates multi-agent collaboration. This system orchestrates context management, problem decomposition, and sub-goal assignment to tackle complex proofs effectively.

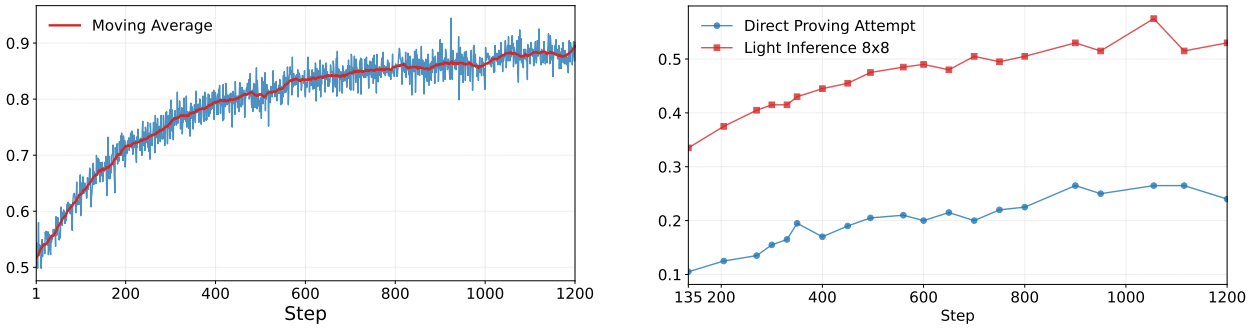
Seed-Prover 1.5 orchestrates three specialized agents:

1. **Natural Language Prover**: An LLM optimized for natural language proving (initialized from Doubao-Seed-1.6). Its role is to generate rigorous, lemma-style natural language proofs to guide the formalization.
2. **Sketch Model**: A translation agent trained to convert natural language proofs into lemma-Style Lean sketches, effectively bridging the gap between informal reasoning and formal syntax.
3. **Agentic Lean Prover**: The tool-integrated agent (described in the previous section) responsible for verifying every individual lemma.

Given a Lean statement, the Natural Language Prover first generates a proof in natural language, which the Sketch Model then converts into a lemma-style Lean sketch. For each unsolved lemma within the sketch, the Agentic Prover attempts to prove or disprove it, under a compute budget of Pass@3 $\times$ 3. If a proof cannot be found, the system recursively performs the "natural language proof $\rightarrow$ Lean sketch" decomposition. If a lemma is disproved, the system reverts to the Sketch Model to refine the sketch. This process repeats until every leaf node (sub-lemma) in the search tree is successfully proved by the Lean Prover, or the maximum search depth is reached.

4 Experiments

We evaluated our agentic prover and test-time workflow using the following benchmarks under Lean v4.22.0:



(a) Batch-level training accuracy.

(b) Accuracy on the test set Putnam-200.

Figure 3 Training dynamics and evaluation metrics over 1200 RL training steps.

- PutnamBench [21]: Consists of 660 problems from the William Lowell Putnam Mathematical Competition (from 1962 to 2024). It evaluates the model's ability to solve undergraduate-level mathematical problems.
- FATE [10]: FATE-H contains 100 problems at the level of honors course exams or graduate-level difficulty. FATE-X contains 100 problems at the level of PhD qualifying exams or beyond. These benchmarks evaluate the model's capacity to solve mathematical problems with graduate level math knowledge and beyond.
- CombiBench [14]: CombiBench is a benchmark specifically centered on combinatorial problems, where the problems often involve newly-defined concepts. We use it to assess the model's capabilities in combinatorics, which is often a shortcoming for LLMs and formal provers. However, we discovered significant formalization issues within this dataset. We list the performance here for reference.
- IMO & Putnam 2025: We use these two competitions to measure our model's performance. These two competitions have no chances of data leakage.
- Erdős[2]: To test if our model is capable of proving frontier math conjectures, we use problems from Erdős problem sets. We collected a subset of Erdős problems from FormalConjecures Project[3] and our in-house expert labeling, removing those with formalization errors (e.g. Erdős-74, 590, 591).

4.1 Scaling Behavior of the Agentic Prover Training

We monitor the performance of our agentic prover during large-scale RL training.

RL Training Dynamics As shown in Figure 3a, RL training accuracy increases from approximately 50% at initialization to nearly 90% at more than 1000 steps. We believe this improvement is enabled by two key factors: (i) a curated training dataset that aligns closely with the proof tasks, and (ii) the accurate reward signal from Lean verification. Consequently, a reward approaching 90% suggests that reinforcement learning effectively helps the model extract maximal benefit from the training data. Figure 4 provides a deeper view into the agent's behavioral change during RL training. Figure 4a and 4b reveal a significant optimization in efficiency: the average number of function calls drops from approximately 15 to 10, which correlates with a consistent reduction in average sequence length (from ∼28k to ∼17k tokens, see Figure 4b). This suggests the model is learning to use tools more strategically, avoiding redundant or "trial-and-error" invocations. Despite the above reduction, the model's reasoning capability improves. Figure 4c and 4d track the scoring metrics for samples with longer response lengths (16K–64K). The improvement over optimization steps indicates that the agent is also increasingly capable of managing complex, long-horizon problems. However, the persistence of negative scores within the 32K–64K range suggests that effectively reasoning over extremely long contexts remains a challenge.

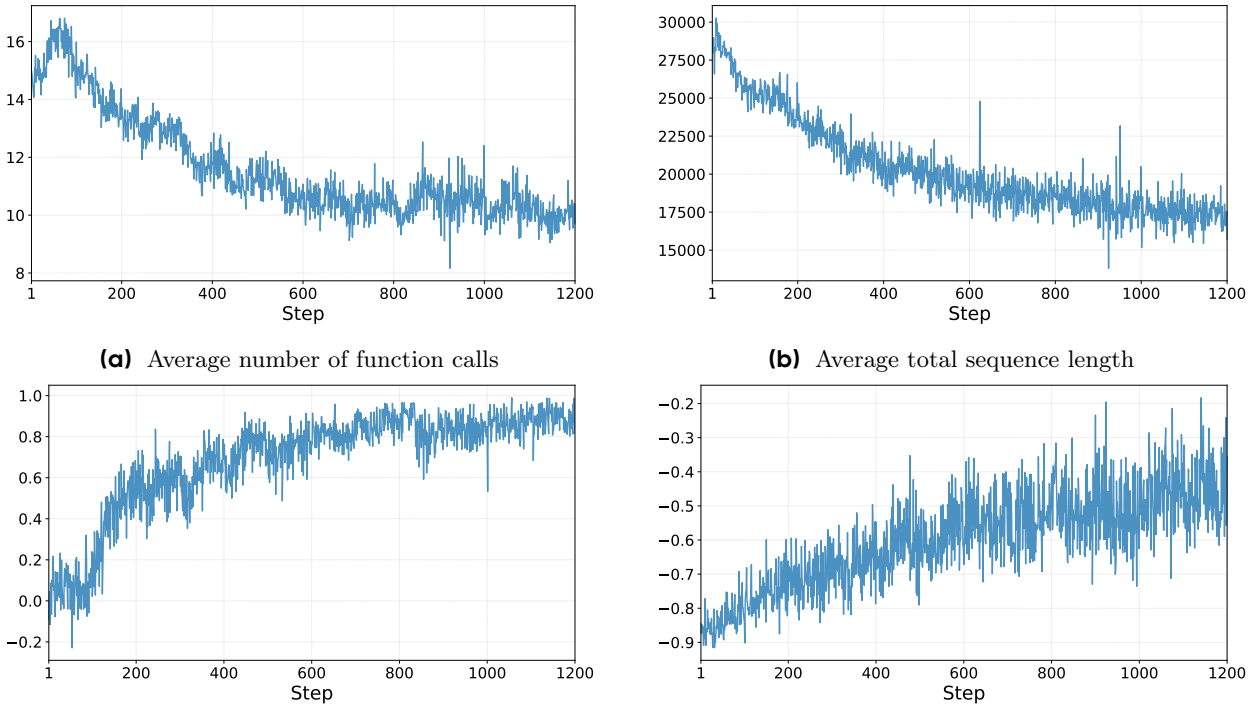
[2] https://www.erdosproblems.com/

[3] https://github.com/google-deepmind/formal-conjectures/tree/main/FormalConjectures

Adaptive Search Behavior During RL training, the average number of search tool calls per trajectory remained generally low (< 3). However, we observed a distinct difference between datasets during inference: on Fate-H, the model averaged approximately 10 search calls per trajectory, whereas on Putnam, it averaged only 1–2. This suggests that the model is capable of adapting its tool-call strategy to different scenarios, particularly given that Fate relies heavily on Mathlib search to derive proofs. We also observed that the number of search calls decreased in later checkpoints, while performance continued to improve (see Figure 5). This trend suggests that RL training not only improves reasoning capabilities but also enables the model to internalize knowledge from search results. For example, we found that training cases with significantly decreased response lengths are often able to locate the key lemma faster and identify the correct theorem to complete the proof.

Approach	Budget	Putnam	Fate-H	Fate-X
Goedel-Prover-V2-32B	pass@64	86/660	2/100	0/100
Seed-Prover 1.0 (medium)	18 H20 days / problem	331/660	35/100	9/100
Seed-Prover 1.5 (agentic prover only)	pass@8 $\times$ 8	**359/660**	**57/100**	**10/100**

Table 1 Performance between Seed-Prover 1.5 agentic prover and Seed-Prover 1.0 medium workflow.



(a) Average number of function calls

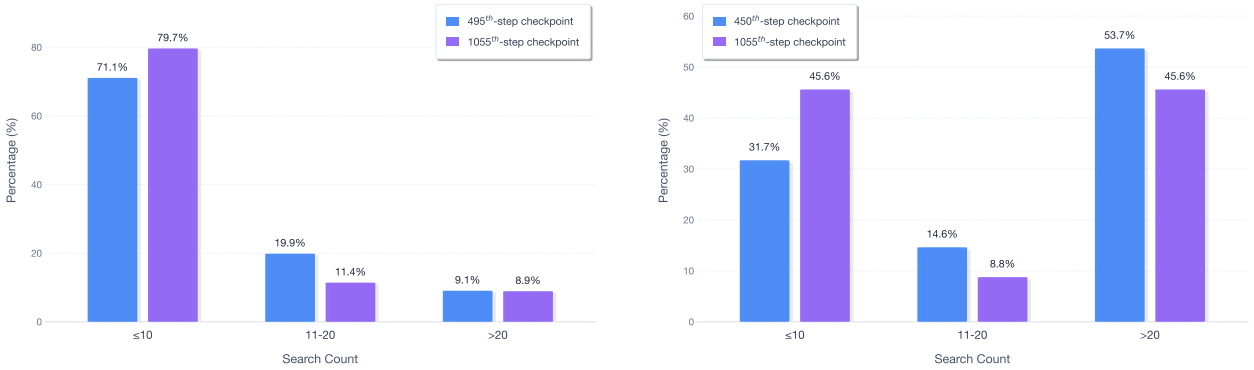
(b) Average total sequence length

(c) Score of sample responses with lengths 16K to 32K

(d) Score of sample responses with lengths 32K to 64K

Figure 4 Other training metrics in our agentic RL training.

Test-set Performance To enable fast development, we select 200 problems from PutnamBench named Putnam-200 to evaluate different RL model steps. Figure 3b shows that accuracy on the Putnam-200 subset continues to increase as the training reward increases, for both direct solving (Pass@8x1) and the light inference (Pass@8×8) setting. This result represents a substantial improvement over our previous configurations, indicating that continued scaling of reinforcement learning leads to sustained gains in both training and test performance. As a comparison with Seed-Prover 1.0 [3], we select the best-performing checkpoint at 1055^{th} training step and evaluate its light inference performance on the full PutnamBench and



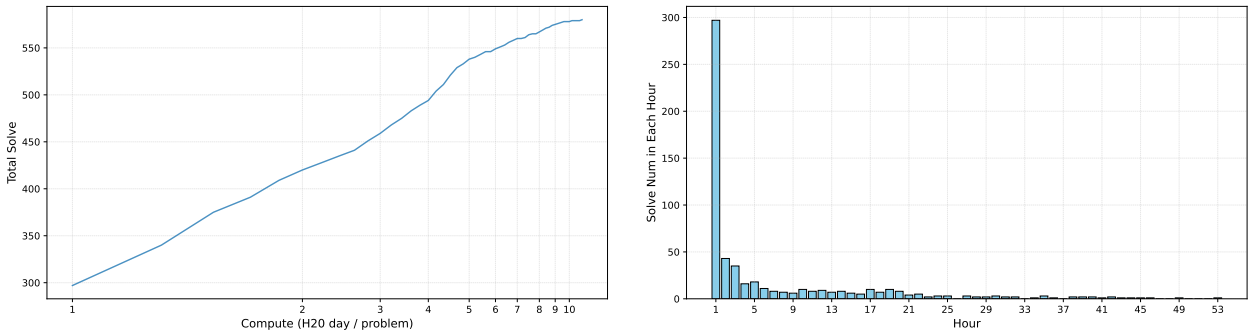
(a) Putnam **(b)** Fate-H

Figure 5 Distribution of search calls for proved samples using different checkpoints. The charts illustrate the percentage of successfully proved samples stratified by the number of search calls (≤ 10, $11-20$, and > 20) for the (a) Putnam and (b) Fate-H benchmarks. The number of search tool calls per sample here include all trajectories (i.e., summary-based trajectory)

Fate benchmarks, as reported in Table 1. All evaluations for the agent prover are conducted with a maximum sequence length of 64K tokens and a maximum of 28 tool calls. Under a compute budget of Pass@8 $\times$ 8 the proposed agent significantly outperforms Seed-Prover 1.0 using the medium workflow, despite the latter consuming substantially more computational resources than light inference. Note that the reported results do not leverage longer sequence lengths or additional inference-time strategies such as error pruning, which enable further performance improvements.

4.2 Evaluation and Scaling Behavior of the Test-Time Workflow

Scaling We present the test-time scaling behavior of Seed-Prover 1.5 workflow on PutnamBench. For each problem, we set an initial maximum search depth of 4. If a problem reaches the limit without resolution, we incorporate the lemmas proven during the search into the context and restart the search from scratch. Consequently, this extends the maximum search depth to 8 for each problem. As illustrated in Figure 6a, investing more compute (including search width and search depth) leads to a log-linear increase in Seed-Prover 1.5's solve rate. Figure 6b demonstrates that a significant majority of problems are solved within the first few hours, while a long tail of more challenging problems is discovered as the search duration extends up to the 53rd hour.



(a) Scaling of solved problems with compute. The number of solved problems on PutnamBench increases log-linearly with respect to the computational budget.

(b) Problem distribution by solution time. The histogram shows the number of problems solved within each specific hour of search.

Figure 6 Test-time Scaling of Seed-Prover 1.5 on PutnamBench.

PutnamBench & FATE Table 2 shows that Seed-Prover 1.5 demonstrates a significant advantage over AlphaProof [8], Hilbert Prover [22], and Aleph Prover on PutnamBench. Furthermore, compared to Seed-Prover 1.0, Seed-Prover 1.5 achieves significantly better performance across various benchmarks with a reduced compute budget. Our system solves 87.9% of problems on PutnamBench and 80% on Fate-H, demonstrating its proficiency in handling formal proofs for undergraduate and graduate-level mathematics. However, the system still faces challenges with PhD-level problems and beyond, primarily due to the increased complexity and limitations in Mathlib support.

Name	Compute Budget	Putnam	Fate-H	Fate-X	Combibench
Seed-Prover 1.0 (medium)	18 H20 days / problem	50.4%	35%	9%	39%
AlphaProof	500 TPU days / problem	56.1%	-	-	-
Hilbert	avg pass@1840	70.0%	-	-	-
Aleph Prover	avg 1834 tool calls	75.8%	-	-	-
Seed-Prover 1.5	10 H20 days / problem	**580/87.9%**	**80%**	**33%**	**48%**

Table 2 Performance comparison of Seed-Prover 1.5 against other methods.

IMO & Putnam 2025 We evaluated Seed-Prover 1.5 on the IMO 2025 (Table 3) and Putnam 2025 competition (Table 4). In these evaluations, we capped the maximum search depth at 4 while increasing the parallel width to accelerate problem resolution. While Seed-Prover 1.0 required "Heavy" mode to solve 5 out of 6 problems, Seed-Prover 1.5 achieved the same solve rate using compute resources comparable to the 1.0 "Medium" setting (20 H20-days/problem), with a significantly shorter runtime than Seed-Prover 1.0 (Heavy). We also tested on the 12 problems from Putnam 2025; utilizing a maximum compute budget of 40 H20-days/problem, we successfully solved 11 of them within 9 hours[4].

IMO 2025	P1	P2	P3	P4	P5	P6
Solve Hour	16.5	0.01	5	8	1	X

Table 3 Time taken (in hours) for Seed Prover 1.5 to solve IMO 2025 problems. (P2 is solved by Seed-Geometry)

Putnam 2025	A1	A2	A3	A4	A5	A6	B1	B2	B3	B4	B5	B6
Solve Hour	1	0.5	2	4	X	4	9	6	0.5	2	4	3

Table 4 Time taken (in hours) for Seed Prover 1.5 to solve Putnam 2025 problems.

Erdős Seed-Prover 1.5 solved problem numbers 124, 198, 303, 316, 350, 370, 379, 418, 449, 493, 499, 645, 728 and 958. However, based on our observations, these problems are mathematically relatively simple, or we proved a trivial or simplified version of them due to mis-formalization (some mis-formalized problems are not listed here). Our systems, whether using natural language or formal language, are still some distance away from truly helping to advance research on frontier open mathematical problems, which motivates our pursuit of further research.

[4]To be noticed, our prover is not using any 'native_decide' in Putnam, which is unsafe under Lean.

5 Conclusion

In this paper, we propose Seed-Prover 1.5, a high-performance agentic Lean prover integrates with a sketch model which serves as a bridge between natural language proofs and formal Lean code. Seed-Prover 1.5 shows strong performance on competitive mathematics problems (e.g., Putnam Competition) and graduate-level mathematical tasks (e.g., FATE dataset), thereby laying a solid foundation for proving frontier mathematical problems. Nevertheless, our system currently cannot make significant mathematical contributions comparable to human experts, and this limitation stems from a critical "dependency issue" inherent to frontier mathematical research. A key distinction lies in the nature of mathematical tasks: IMO or Putnam problems are deliberately designed such that solvers do not require knowledge of specific prior research papers while driving significant progress in mathematics typically hinges on synthesizing insights across a multitude of related papers. Achieving such progress requires addressing three interconnected challenges: first, identifying the most influential and relevant papers; second, conducting natural language proofs grounded in these works; third, developing scalable approaches to formalizing both the papers themselves and the results derived from them. Solving these three challenges would enable the large-scale generation of formal mathematical research—an advancement that could ultimately contribute to resolving certain open mathematical conjectures.

References

[1] Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. Aristotle: Imo-level automated theorem proving, 2025. URL https://arxiv.org/abs/2510.01346.

[2] Alon Albalak, Duy Phung, Nathan Lile, Rafael Rafailov, Kanishk Gandhi, Louis Castricato, Anikait Singh, Chase Blagden, Violet Xiang, Dakota Mahan, and Nick Haber. Big-math: A large-scale, high-quality math dataset for reinforcement learning in language models, 2025. URL https://arxiv.org/abs/2502.17387.

[3] Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, Cheng Ren, Jiawei Shen, Wenlei Shi, Tong Sun, He Sun, Jiahui Wang, Siran Wang, Zhihong Wang, Chenrui Wei, Shufa Wei, Yonghui Wu, Yuchen Wu, Yihang Xia, Huajian Xin, Fan Yang, Huaiyuan Ying, Hongyi Yuan, Zheng Yuan, Tianyang Zhan, Chi Zhang, Yue Zhang, Ge Zhang, Tianyun Zhao, Jianqiu Zhao, Yichi Zhou, and Thomas Hanwen Zhu. Seed-prover: Deep and broad reasoning for automated theorem proving. arXiv preprint arXiv:2507.23726, 2025.

[4] Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. arXiv preprint arXiv:2504.11536, 2025.

[5] Songyang Gao, Yuzhe Gu, Zijian Wu, Lingkai Kong, Wenwei Zhang, Zhongrui Cai, Fan Zheng, Tianyou Ma, Junhao Shen, Haiteng Zhao, Duanyang Zhang, Huilun Zhang, Kuikun Liu, Chengqi Lyu, Yanhui Duan, Chiyu Chen, Ningsheng Ma, Jianfei Gao, Han Lyu, Dahua Lin, and Kai Chen. Long-horizon reasoning agent for olympiad-level mathematical problem solving, 2025. URL https://arxiv.org/abs/2512.10739.

[6] Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains, 2025. URL https://arxiv.org/abs/2507.17746.

[7] Yichen Huang and Lin F. Yang. Winning gold at imo 2025 with a model-agnostic verification-and-refinement pipeline, 2025. URL https://arxiv.org/abs/2507.15855.

[8] Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. Nature, pages 1–3, 2025.

[9] Xingguang Ji, Yahui Liu, Qi Wang, Jingyuan Zhang, Yang Yue, Rui Shi, Chenxi Sun, Fuzheng Zhang, Guorui Zhou, and Kun Gai. Leanabell-prover-v2: Verifier-integrated reasoning for formal theorem proving via reinforcement learning, 2025. URL https://arxiv.org/abs/2507.08649.

[10] Jiedong Jiang, Wanyi He, Yuefeng Wang, Guoxiong Gao, Yongle Hu, Jingting Wang, Nailing Guan, Peihao Wu, Chunbo Dai, Liang Xiao, et al. Fate: A formal benchmark series for frontier algebra of multiple difficulty levels. arXiv preprint arXiv:2511.02872, 2025.

[11] Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. Numinamath. [https://github.com/project-numina/aimo-progress-prize](https://github.com/project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf), 2024.

[12] Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, et al. Goedel-prover: A frontier model for open-source automated theorem proving. arXiv preprint arXiv:2502.07640, 2025.

[13] Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction, 2025. URL https://arxiv.org/abs/2508.03613.

[14] Junqi Liu, Xiaohan Lin, Jonas Bayer, Yael Dillies, Weijie Jiang, Xiaodan Liang, Roman Soletskyi, Haiming Wang, Yunzhou Xie, Beibei Xiong, et al. Combibench: Benchmarking llm capability for combinatorial mathematics. arXiv preprint arXiv:2505.03171, 2025.

[15] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-79875-8. doi: 10.1007/978-3-030-79876-5_37. URL https://doi.org/10.1007/978-3-030-79876-5_37.

[16] Zhongyuan Peng, Yifan Yao, Kaijing Ma, Shuyue Guo, Yizhe Li, Yichi Zhang, Chenchen Zhang, Yifan Zhang, Zhouliang Yu, Luming Li, et al. Criticlean: Critic-guided reinforcement learning for mathematical formalization. arXiv preprint arXiv:2507.06181, 2025.

[17] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. arXiv preprint arXiv:2202.01344, 2022.

[18] Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition, 2025. URL https://arxiv.org/abs/2504.21801.

[19] Shijie Shang, Ruosi Wan, Yue Peng, Yutong Wu, Xiong-hui Chen, Jie Yan, and Xiangyu Zhang. Stepfun-prover preview: Let's think and verify step by step, 2025. URL https://arxiv.org/abs/2507.20199.

[20] Zhihong Shao, Yuxiang Luo, Chengda Lu, ZZ Ren, Jiewen Hu, Tian Ye, Zhibin Gou, Shirong Ma, and Xiaokang Zhang. Deepseekmath-v2: Towards self-verifiable mathematical reasoning. arXiv preprint arXiv:2511.22570, 2025.

[21] George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. In The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track.

[22] Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. arXiv preprint arXiv:2509.22819, 2025.

[23] Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. 2025. URL http://arxiv.org/abs/2504.11354.

[24] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. Advances in Neural Information Processing Systems, 35: 32353–32368, 2022.

[25] Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Zheng Yuan, Dahua Lin, and Kai Chen. Internlm2.5-stepprover: Advancing automated theorem proving via expert iteration on large-scale lean problems, 2024. URL https://arxiv.org/abs/2410.15700.

[26] Huajian Xin, Z.Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Haowei Zhang, Qihao Zhu, Dejian Yang, Zhibin Gou, Z.F. Wu, Fuli Luo, and Chong Ruan. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. In The Thirteenth International Conference on Learning Representations, 2025. URL https://openreview.net/forum?id=I4YAIwrsXa.

[27] Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Kai Shen. Bfs-prover: Scalable best-first tree search for llm-based automatic theorem proving, 2025. URL https://arxiv.org/abs/2502.03438.

[28] Ran Xin, Zeyu Zheng, Yanchen Nie, Kun Yuan, and Xia Xiao. Scaling up multi-turn off-policy rl and multi-agent tree search for llm step-provers. arXiv preprint arXiv:2509.06493, 2025.

[29] Huaiyuan Ying, Zijian Wu, Yihan Geng, Zheng Yuan, Dahua Lin, and Kai Chen. Lean workbook: A large-scale lean problem set formalized from natural language math problems, 2024. URL https://arxiv.org/abs/2406.03847.

[30] Huaiyuan Ying, Shuo Zhang, Linyang Li, Zhejian Zhou, Yunfan Shao, Zhaoye Fei, Yichuan Ma, Jiawei Hong, Kuikun Liu, Ziyi Wang, Yudong Wang, Zijian Wu, Shuaibin Li, Fengzhe Zhou, Hongwei Liu, Songyang Zhang,

Wenwei Zhang, Hang Yan, Xipeng Qiu, Jiayu Wang, Kai Chen, and Dahua Lin. Internlm-math: Open math large language models toward verifiable reasoning, 2024. URL https://arxiv.org/abs/2402.06332.

[31] Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, TianTian Fan, Zhengyin Du, Xiangpeng Wei, Xiangyu Yu, Gaohong Liu, Juncai Liu, Lingjun Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Ru Zhang, Xin Liu, Mingxuan Wang, Yonghui Wu, and Lin Yan. Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks, 2025. URL https://arxiv.org/abs/2504.05118.

[32] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In International Conference on Learning Representations.

[33] Yichi Zhou, Jianqiu Zhao, Yongxin Zhang, Bohan Wang, Siran Wang, Luoxin Chen, Jiahui Wang, Haowei Chen, Allan Jie, Xinbo Zhang, et al. Solving formal math problems by decomposition and iterative reflection. arXiv preprint arXiv:2507.15225, 2025.

Appendix

A Contributors

The names are sorted alphabetically. An asterisk * indicates a member who left Seed.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, Zhihong Wang, Mingxuan Wang, Chenrui Wei, Shufa Wei, Huajian Xin, Fan Yang*, Weihao Gao, Zheng Yuan, Tianyang Zhan, Zeyu Zheng, Tianxi Zhou, Thomas Hanwen Zhu

B Prompts of Rubric RL and Example of Sketch

Prompt 1: Atomic Lemma Verification

```
**Role and Goal**
* **Role:** You are a rigorous mathematician and an expert Lean 4 formalization engineer.
* **Overall Objective:** Determine if a specific **"Formal Statement"** is **Mathematically Correct** and
      **Provable**.
* **Key Philosophy:** Treat the "Formal Statement" as an independent proposition. You must verify it
     using logical reasoning and standard mathematical axioms.

**Context Usage (The Sketch)**
* **Purpose of Sketch:** The provided "Lean Code Sketch" is strictly a **Dictionary for Definitions**.
* **When to use it:** Only refer to the Sketch to resolve the definitions of unknown symbols (e.g.,
     custom `def`, `structure`, `class`, or `instance`).
* **What to IGNORE:** **Ignore all other `lemma` or `theorem` entries inside the Sketch.** They may be
     hallucinated or incorrect. Do NOT assume they are true, and do NOT cite them in your proof.

**Task Description**

1.  **Symbol Resolution:**
    * Read the **Formal Statement**.
    * If standard math symbols are used (e.g., `Nat`, `Real`, `List`), apply standard mathematical
     semantics (considering Lean's implementation details).
    * If custom symbols are used, look up their precise definitions in the **Lean Code Sketch**.

2.  **Independent Verification (The Core Task):**
    * **Construct a Proof or Counter-Example:** Attempt to prove the statement from first principles (
     definitions) or standard mathematical theorems.
    * **Do NOT Circularly Reference:** Never prove the statement by saying "it is already listed in the
     sketch."
    * **Check for Counter-Examples:** Actively try to break the statement. If a counter-example exists (e.
     g., "bounded increasing sequence is not necessarily eventually constant"), the statement is **
     Incorrect**.
    * **Special Check for Infinite Operators:**
        * **Integrals ∫(``):** Check for `IntegrableOn` or `Integrable` hypotheses. If missing,
     especially on non-compact domains (e.g., `Set.Ici`, `Set.univ`), the integral is likely undefined (
     junk value). Note: Compact domains with continuous functions are safe.
        * **Infinite Sums Σ(`'`):** Check for `Summable` hypotheses. If missing, Σ`'` evaluates to `0` (
     junk value). Verify if this causes a contradiction (e.g., `Positive_LHS ≤ Σ' (divergent) = 0`). If
     so, the statement is **Incorrect**.

3.  **Junk Value Analysis:**
    * **Consult the Reference:** rigorousy apply the rules from the provided **"Lean 4 Corner Cases and
     Junk Values"** document.
    * **Check Edge Cases:** Verify if the statement fails due to Lean's total functions (e.g., check `0`
     denominators, empty lists, infinite sums, or `Nat` subtraction).
    * **Verdict:** If the theorem holds in standard math but fails because Lean returns a specific junk
```

```
    value (e.g., `1/0 = 0` makes the equation false), it is **Incorrect**.

**Assessment Criteria**

* **Correct (Provable):**
    * The statement holds true based on the definitions and standard logic.
    * A valid, step-by-step natural language proof exists.

* **Incorrect (Unprovable):**
    * **Mathematical Falsehood:** A counter-example exists.
    * **Missing Hypothesis:** The statement is not universally true because it lacks a necessary
     precondition (e.g., missing `Integrable` or `Summable`).
    * **Logic Gap:** The statement cannot be derived from the definitions.
    * **Junk Value Failure:** The statement fails due to Lean's total function handling (e.g., `1/0` or
     `∑' divergent = 0`).

**Output Format**

Return a single valid JSON object:

{{
  "correctness": "Correct" or "Incorrect",
  "reason": "Concise explanation. If 'Correct', briefly summarize the proof logic. If 'Incorrect',
     provide the counter-example, the specific logical flaw, or the missing hypothesis (e.g., 'The
     statement is incorrect because it requires a `Summable` hypothesis, otherwise the sum is 0.').",
  "proof_sketch": "A rigorous Natural Language proof or a concrete counter-example construction. Do not
     use 'sorry'."
}}

Note: correctness must be "Correct" or "Incorrect".
----------------------------sketch((Definitions)---------------------------------------
{sketch}

----------------------------formal statement to evaluate-------------------------------------
{formal_statement}


--------lean4 doc string maybe helpful for you to understand the theorem--------
{doc_string}


--------docs about Lean 4 Corner Cases and Junk Values  that maybe helpful--------
Lean 4 Corner Cases and Junk Values
This document catalogs important corner cases in Lean 4's mathematical library (Mathlib) where
     definitions use "junk values" or default values for edge cases. This is not an exhaustive list.
What are Junk Values?
In Lean 4, many mathematical operations are total functions (defined for all inputs) rather than partial
     functions. When an operation is mathematically undefined or doesn't make sense for certain inputs,
     Lean assigns a default "junk value" instead of leaving it undefined.
Common Examples
Arithmetic Operations
 * Natural number subtraction: (2 : ℕ) - (3 : ℕ) = 0
   * Natural numbers cannot be negative, so subtraction returns 0 when the result would be negative
   * Notation: ℕ represents natural numbers (0, 1, 2, ...)
 * Division by zero: x / 0 = 0 in many contexts
   * Instead of being undefined, division by zero returns 0
Cardinality
 * Extended cardinality of infinite sets: Nat.encard s = 0 when s is infinite
   * Nat.encard returns the cardinality as a natural number
   * For infinite sets, it returns 0 as a junk value (since infinite cardinality cannot be represented as
      ℕ)
Real-valued Functions
 * Square root of negative numbers: Real.sqrt x = 0 when x < 0
```

```
   * The real square root is only defined for non-negative numbers
   * For negative inputs, it returns 0 as a junk value
 * Logarithm of non-positive numbers: Real.log x = 0 when x ≤ 0
   * The real logarithm is only defined for positive numbers
   * For non-positive inputs, it returns 0 as a junk value
Calculus
 * Derivative of non-differentiable functions: deriv f = 0 when f is not differentiable
   * If a function isn't differentiable at a point, its derivative is defined as 0
Series and Products
 * Divergent infinite sums: ∑' i, f i = 0 when the series doesn't converge
   * The notation ∑' represents an infinite sum (tsum)
   * When a series diverges, it evaluates to 0 by default
   * Note: Summable and HasSum typically mean absolutely summable in Lean, which is a stronger condition
     than conditional convergence
 * Divergent infinite products: prod' i, f i = 1 when the product doesn't converge
   * The notation prod' represents an infinite product (tprod)
   * When a product diverges, it evaluates to 1 by default
Working with Junk Values
When assessing provability:
 * CRITICAL CHECK: If the Doc String describes a general mathematical fact (e.g., "The derivative of log
     is 1/x") but the Formal Statement lacks necessary preconditions (e.g., x is not 0 or
     DifferentiableAt), it is Unprovable.
 * If the statement fails due to a junk value, mark it as Unprovable and explain the specific edge case (
     e.g., "Fails when x=0 because 0/0=0, not 1").

 Please evaluate the Formal Statement above based on the Sketch and Reference provided. Return the JSON
     verdict.
```

Prompt 2: Proof Strategy Alignment

```
### **1. ROLE & GOAL**

You are a Lean 4 proof strategy evaluator. Your primary goal is to assess if an AI-generated "proof
     sketch" represents a sound and effective proof plan for a given theorem. A good plan decomposes a
     complex problem into simpler, independent, and solvable sub-problems (lemmas) that facilitate
     automated theorem proving. You must be concise and follow a strict "fail-fast" workflow.

### **2. GUIDING PRINCIPLE**

Think like a senior mathematician mentoring a student on how to simplify a problem for a computer solver.
      Is the student's plan (the sketch) logical and does it reduce the search complexity? A VETO
     corresponds to a fundamentally flawed plan that needs to be re-thought from scratch. A low-scoring
     PASS corresponds to a plan that works but fails to significantly simplify the problem.

### **3. INPUTS**

You will be given three inputs:
1.  **`Formal Statement`**: The final theorem to be proven in Lean 4.
2.  **`Natural Language Proof`**: A human-written, high-level description of the proof strategy.
3.  **`Proof Sketch`**: An AI-generated plan containing:
    *   A set of `lemma` statements (without proofs).
    *   A `main_proof` body that uses these lemmas to prove the `Formal Statement`.

### **4. EVALUATION WORKFLOW**

1.  **Phase 1: Foundational Checks.** This is a mandatory first step. You will check for fatal strategy
     misalignment and validate **every** helper lemma statement.
2.  **Phase 2: Report Generation.** Based on the results of Phase 1:
```

````
    *   **If any check fails:** The evaluation is `VETOED`. You will output a minimal diagnostic report
     and a score of -10.0.
    *   **If all checks pass:** The evaluation is `PASSED`. You will proceed to generate a full rubric
     and scoring analysis.

### **5. EVALUATION CRITERIA**

#### **Phase 1: Foundational Checks (Veto Triggers)**

*   **Fatal Misalignment:** The sketch's core strategy (e.g., induction on `n`) completely contradicts
     the NL proof's stated strategy (e.g., casework on `x`).

*   **Proof by Delegation:** The proof body of the top-level theorem statement is hollow and fails to
     demonstrate the high-level assembly of its lemmas. The main theorem's proof body has a crucial role:
      **it must explicitly showcase how the lemmas (the proven sub-problems) are logically combined to
     reach the final conclusion.** This assembly process should be transparent and reflect the strategy
     outlined in the `Natural Language Proof`. A sketch is vetoed if it abdicates this responsibility by
     hiding the assembly logic inside a single "wrapper" lemma.

    * **Bad Pattern (to VETO):** The main theorem body shows no *actual* reasoning. It merely calls a "
     wrapper" lemma (e.g., `main_proof`) that shares the **same goal as the main theorem**. This wrapper
     takes the helper lemmas as arguments and hides the crucial synthesis step in its own `sorry`. This
     pattern makes the plan's logic impossible to evaluate.

      * Be careful:** This bad pattern can be disguised using `have` statements, but it is still a veto.

        ```lean
        -- 'lemma_P' solves a sub-problem P.
        lemma lemma_P : P := by sorry

        -- 'main_proof' is a "wrapper" lemma.
        -- ***Its goal (R) is identical to the main theorem's goal.***
        -- It hides the logic of how P is used to prove R.
        lemma main_proof (hP : P) : R := by sorry

        -- VETO (Verbose variant): This proof is hollow.
        -- It looks like it's doing work, but all assembly logic
        -- is delegated to 'main_proof'.
        theorem T_verbose : R := by
          have hP : P := lemma_P
          -- The next line is the delegation. Its goal 'R' is T_verbose's goal.
          have h_main : R := main_proof hP
          exact h_main

        -- VETO (Direct variant): This is the same bad pattern, just shorter.
        theorem T_direct : R := by
          exact main_proof (lemma_P)
        ```

    *   **Good Pattern (to PASS):** The main theorem body acts as the "glue," using tactics (`have`, `
     apply`, `constructor`, `intro`, etc.) to orchestrate the lemmas. **It explicitly demonstrates the
     reasoning for how the sub-goals are put together.** This makes the high-level structure of the
     argument clear and evaluatable.
        ```lean
        -- These lemmas solve the sub-problems.
        lemma lemma_P : P := by sorry
        lemma lemma_Q : Q := by sorry

        -- The main proof body SHOWS the assembly logic.
        theorem T : P and Q := by
          -- The logic is: to prove a conjunction, prove each part.
          have hP : P := lemma_P
          have hQ : Q := lemma_Q
````
````

```
        -- Then, use the constructor for 'And' to combine them.
        exact ⟨hP, ⟩hQ -- PASS: This transparently assembles the results.
      ```
*   **Clarification on Single-Line Proofs:** A single-line proof in the main theorem is **NOT** a "Proof by Delegation" if it uses a **transparent constructor** to assemble the results. The key difference is transparency:
    *   **GOOD (Transparent):** `exact ⟨lemma_P, ⟩lemma_Q` is good because ⟨⟩`...` is a standard, understandable constructor for `And`. The assembly logic is fully visible.
    *   **BAD (Opaque):** `exact main_proof lemma_P lemma_Q` is bad because `main_proof` is an opaque, user-defined function whose inner workings are hidden in a `sorry`.

*   **Invalid Lemma:** A lemma is invalid if it fails to make **substantive progress** or is **structurally broken**. This includes:
*   **False/Unprovable:** The statement is mathematically false. This is the most severe flaw and an immediate VETO trigger. Check one by one.
*   **Missing Context / Not Self-Contained:** Since each lemma is searched as an independent theorem, it **must** define all its variables. A lemma that uses variables (e.g., `n`, `x`, `h`) without taking them as arguments (unless they are globally defined in the provided header) is INVALID. It relies on the local context of the main proof, which will not be available during search.
*   **Trivial:** The lemma offers no simplification and its proof is self-evident.
    *   **What IS Trivial:** A statement that is a simple logical tautology (`p → p`), a direct application of a hypothesis (`h → G` when `h : G` exists), or something solvable by purely mechanical rewriting with no mathematical insight (`rfl`, `ring`). The goal is to VETO lemmas that do no real work.
    *   **What IS NOT Trivial (Crucial Exception):** A lemma is **NOT** trivial just because its proof is a short call to a powerful library theorem or a single tactic (`norm_num`). **Isolating and naming a key, reusable mathematical fact is a valuable decomposition step.**

*   **Circular or Redundant:** A lemma is invalid under this rule **only if** it meets one of these specific conditions:
    1. It is a literal restatement of a hypothesis.
    2. It restates the *entire* theorem's goal without any simplification or decomposition.
    ** The Principle of Valid Decomposition:**
    You MUST distinguish the above from **valid top-down decomposition**. Breaking a complex goal into its primary logical components is a **highly desirable strategy**.

* **Invalid because Junk Value:** In Lean 4, some definitions will have junk value. Like (2:N)-(3:N)=0, deriv of a non-derivable function equals 0, tsum equals 0 when not converge. Please reason and check if the lemma will be invalid due to the junk value.

Rubric Construction Guide (For PASSED Sketches)

Before scoring, you must first define the ideal proof plan in your decomposition_rubric. This rubric serves as your 'gold standard' for the evaluation.

Your rubric should reflect a hierarchy of decomposition quality:

* Tier 1 (Strategic Decomposition): At a minimum, the problem should be broken down along its main logical structure (e.g., proving conjuncts separately).
* Tier 2 (Search Simplification): A more ideal plan goes deeper. It identifies and isolates core mathematical steps that make the problem **easier to search**. This matches the conceptual steps in the Natural Language Proof.

Phase 2: Scoring Criteria (For PASSED Sketches)

*   **Structural Alignment (50%):** How well does the sketch's lemma structure align with the optimal, context-specific breakdown that you define in the rubric?
Score 0-1 (Poor): The sketch passes the VETO checks but is fundamentally incomplete or useless.
Score 3-5 (Tier 1 - Strategic): The sketch achieves a Tier 1 decomposition. It correctly breaks the problem into its main logical parts but doesn't create deeper, simplifying lemmas.
Score 8-10 (Tier 2 - Search Simplification): The sketch achieves a Tier 2 decomposition. It

```
     successfully identifies and lemmatizes the core underlying mathematical steps, closely mirroring the
      ideal plan in the rubric.

*   **Lemma Value (50%):** Are the individual valid lemmas high-quality for **search and solving**?
    Score 0-1 (Low value): The proposed valid lemmas are **not self-contained** (missing necessary
     arguments/context) or are so **overly broad (monolithic)** that they are as hard to prove as the
     original theorem.
    Score 3-5 (Moderate value): The lemmas are correct and self-contained, but they may be slightly
     redundant or don't significantly reduce the difficulty of the proof (e.g., just renaming variables
     or minor shuffling).
    Score 8-10 (High value): The lemmas act as excellent checkpoints. They are **self-contained** (carry
     all necessary context) and isolate a concrete, solvable sub-problem (even if specific to the
     instance). They effectively absorb the "essence" of the Natural Language Proof steps.

### **6. FINAL SCORE CALCULATION**

*   If `VETOED`: `final_score: -10.0`.
*   If `PASSED`:
    1.  `weighted_score = (alignment * 0.4)  + (value * 0.6)`
    2.  `utilization_factor = (number of lemmas USED in main_proof) / (total number of VALID lemmas
     proposed)`
    3.  `final_score = round(weighted_score * utilization_factor, 1)`

### **7. REQUIRED OUTPUT FORMAT**

You MUST use the following JSON format. The `rubric_and_scoring` object is **conditional**.

```json
{{
  "evaluation_status": "[VETOED or PASSED]",
  "veto_reason": {{
    // CONDITIONAL: Fill only if status is "VETOED".
    "type": "[FATAL_MISALIGNMENT or PROOF_BY_DELEGATION or INVALID_LEMMA]",
    "analysis": "[A one-sentence explanation of the veto reason.]"
  }},
  "lemma_diagnostics": [
    // CONDITIONAL: This list should ONLY contain INVALID lemmas. If all lemmas are valid, this list
     should be empty.
    {{
      "lemma_name": "[Name of the INVALID lemma]",
      "reason": "[A brief reason explaining why it is INVALID. Use specific terms like 'Trivial', '
     Circular', 'False', 'Missing Context', 'Junk Value'.]"
    }}
  ],
  "rubric_and_scoring": {{
    // CONDITIONAL: Include this entire object ONLY if status is "PASSED".
    "decomposition_rubric": {{
      "strategy":  "[Brief description of the optimal strategy, derived from the NL proof.]",
      "lemmas": [
        "[Concept for ideal lemma 1, e.g., 'Property of list reversal']",
        "[Concept for ideal lemma 2, e.g., 'Sum of an arithmetic series']"
      ]
    }},
    "justification": {{
      "alignment": "[ Justify the score. First, classify the sketch as **Tier 1 (strategic breakdown)**
     or **Tier 2 (search simplification)**. Then, explain why it fits that tier by comparing it to the
     ideal rubric.]",
      "value": "[Justify the score based on lemma quality. Reward self-contained lemmas that make the
     search easier.]",
      "utilization": "[Report the raw numbers, e.g., 'The sketch used 2 out of 3 valid proposed lemmas
     .']"
    }},
    "scores": {{
```
```

```
      "alignment": "[0.0-10.0]",
      "value": "[0.0-10.0]",
      "utilization_factor": "[0.0-1.0]"
    }}
  }},
  "final_score": "[-10.0 or 0.0-10.0]"
}}```
----below are your inputs
hints: You will receive a Lean Doc str field. This Lean Doc str is an official Lean docstring, which is
     provided as a key reference to help you accurately understand the context, meaning, and signature of
      this Lean entry.
Formal Statement

{}
Natural Language Proof
----------beg of natural language proof----------
{}
----------end of natural language proof----------

Lean Sketch

{}

Lean Doc str
{}
```

```
import Mathlib
import Aesop

/--
Let p(x) be a polynomial that is nonnegative for all real x.
Prove that for some k, there are polynomials f1(x), ..., fk(x) such that
p(x) = ∑_{j=1}^k (fj(x))^2.
-/
lemma zero_poly_to_exist_form_lemma
  (p : Polynomial ℝ)
  (hp_zero : p = 0)
  (hp_nonneg : ∀ x, p.eval x ≥ 0) : -- added hp_nonneg for consistency, though it's redundant when p = 0
  ∃* (k) (f : Fin k → Polynomial ℝ), k > 0 ∧ ∀ x : ℝ, p.eval x = ∑ j : Fin k, ((f j).eval x) ^ 2 := by
  sorry

lemma factor_nonzero_nonneg_as_h_sq_q_positive_lemma
  (p : Polynomial ℝ)
  (hp : ∀ x, p.eval x ≥ 0)
  (hp_ne_zero : p ≠ 0) :
  ∃ (h q : Polynomial ℝ), p = h * h * q ∧ ∀ (x : ℝ), q.eval x > 0 := by
  sorry

lemma positive_poly_as_sum_two_squares_lemma
  (q : Polynomial ℝ)
  (hq : ∀ (x : ℝ), q.eval x > 0) :
  ∃ (f1 f2 : Polynomial ℝ), ∀ (x : ℝ), q.eval x = (f1.eval x) ^ 2 + (f2.eval x) ^ 2 := by
  sorry

lemma p_h_sq_q_two_squares_to_p_is_two_squares_lemma
  (p h q f1 f2 : Polynomial ℝ)
  (h1 : p = h * h * q)
  (h2 : ∀ (x : ℝ), q.eval x = (f1.eval x) ^ 2 + (f2.eval x) ^ 2) :
  ∃ (g1 g2 : Polynomial ℝ), ∀ (x : ℝ), p.eval x = (g1.eval x) ^ 2 + (g2.eval x) ^ 2 := by
  sorry

lemma two_squares_to_existential_form_lemma
  (p : Polynomial ℝ)
  (g1 g2 : Polynomial ℝ)
  (h : ∀ (x : ℝ), p.eval x = (g1.eval x) ^ 2 + (g2.eval x) ^ 2) :
  ∃* (k) (f : Fin k → Polynomial ℝ), k > 0 ∧ ∀ (x : ℝ), p.eval x = ∑ j : Fin k, ((f j).eval x) ^ 2 := by
  sorry

theorem putnam_1999_a2
  (p : Polynomial ℝ)
  (hp : ∀ x, p.eval x ≥ 0) :
  ∃* (k) (f : Fin k → Polynomial ℝ), k > 0 ∧ ∀ x : ℝ, p.eval x = ∑ j : Fin k, ((f j).eval x) ^ 2 := by
  by_cases h_p : p = 0
  · -- Case 1 : p = 0
    exact zero_poly_to_exist_form_lemma p h_p hp
  · -- Case 2 : p ≠ 0
    have h1 : ∃ (h q : Polynomial ℝ), p = h * h * q ∧ ∀ (x : ℝ), q.eval x > 0 := by
      exact factor_nonzero_nonneg_as_h_sq_q_positive_lemma p hp h_p
    rcases h1 with ⟨ h, q, h_eq, hq_pos ⟩
    have h3 : ∃ (f1 f2 : Polynomial ℝ), ∀ (x : ℝ), q.eval x = (f1.eval x) ^ 2 + (f2.eval x) ^ 2 := by
      exact positive_poly_as_sum_two_squares_lemma q hq_pos
    rcases h3 with ⟨ f1, f2, h31 ⟩
    have h4 : ∃ (g1 g2 : Polynomial ℝ), ∀ (x : ℝ), p.eval x = (g1.eval x) ^ 2 + (g2.eval x) ^ 2 := by
      exact p_h_sq_q_two_squares_to_p_is_two_squares_lemma p h q f1 f2 h_eq h31
    rcases h4 with ⟨ g1, g2, h41 ⟩
    exact two_squares_to_existential_form_lemma p g1 g2 h41
```

Figure 7 An example of sketch generated by Seed-Prover 1.5.