AGENTIC PROVING FOR PROGRAM VERIFICATION

Alessandro Sosso & Akhil Arora & Bas Spitters
Department of Computer Science
Aarhus University Aarhus, Denmark
{sosso,akhil.arora,spitters}@cs.au.dk

ABSTRACT

Agentic systems have recently emerged as state-of-the-art approaches for automated theorem proving in formal mathematics. To assess how far these capabilities extend to *program verification*, we evaluate leading agentic and non-agentic provers on two Lean 4 benchmarks for verifiable code generation, CLEVER and VERINA, which require machine-checkable proofs of functional correctness. Our results show that agentic systems not only achieve near-saturation performance on the *correct* portions of these benchmarks, but also consistently identify errors in specifications and implementations within the benchmarks themselves. In several cases, the agents further propose fixes and successfully prove the corrected statements. These findings highlight a growing mismatch between the difficulty of existing program verification benchmarks and the capabilities of modern agentic provers, and point to the need for more rigorous, bug-resilient evaluation methodologies. More broadly, our results provide empirical evidence that tight compiler-in-the-loop agentic paradigms are currently the most effective approach for foundational program verification.

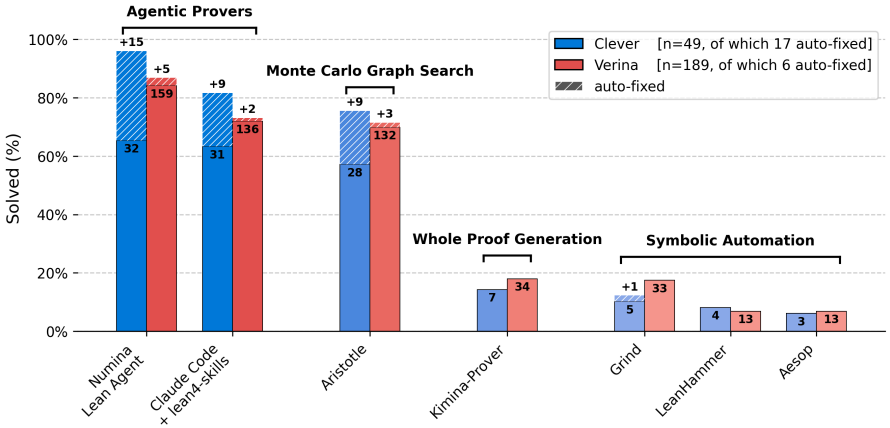

Figure 1: Overview of prover performance on the CLEVER and VERINA benchmarks. Solid bars indicate the number of tests solved on the verified-correct portions of each dataset, while the patterned bar segments above show additional solved instances obtained after correcting erroneous benchmark entries. Numbers annotated with a '+' denote the corresponding count of these additional solves.

1 INTRODUCTION

Large Language Models (LLMs) are known to hallucinate, and even recent "reasoning" models may produce arguments that appear coherent but are logically invalid. In contrast, formal logic offers a long-standing and rigorous alternative: mathematical arguments can be expressed in formal languages and mechanically checked for correctness. This approach is embodied in interactive proof assistants based on type theory (e.g., Rocq (Rocq Development Team, 2026), Lean (Moura & Ullrich, 2021), Agda (Norell, 2007)), higher-order logic (e.g., Isabelle (Nipkow et al., 2002), HOL4 (Slind & Norrish, 2008), HOL-Light (Harrison, 1996)), and set theory (e.g., Mizar (Grabowski et al., 2010), MetaMath (Carneiro et al., 2023)). In this work, we focus on the first category, and in particular on the Lean 4 theorem prover.

Recent years have seen striking progress in applying LLMs to automated proof construction in type-theoretic settings. Notable systems achieve strong performance on formalized mathematics benchmarks in Lean 4 (Hubert et al., 2025; Achim et al., 2025; Chen et al., 2025), as well as in Isabelle (Lin et al., 2024; Xu et al., 2025) and Rocq (Bayazıt et al., 2025; Viennot et al., 2025). These systems employ a range of architectural strategies, including whole-proof generation, proof-state search, and increasingly, agentic designs that integrate planning, tool use, and iterative refinement.

In this paper, we study how these advances translate to *program verification*, a setting that is both practically relevant and structurally challenging. Unlike informal mathematics, program verification requires reasoning about executable artifacts, explicit specifications, and subtle side conditions, all under strict syntactic and semantic constraints. To this end, we evaluate state-of-the-art agentic and non-agentic provers on two recent Lean 4 benchmarks for verifiable code generation: CLEVER (Thakur et al., 2025) and VERINA (Ye et al., 2025).

A central finding of our study is that modern agentic provers not only achieve very high success rates on the *correct* portions of these benchmarks, but also systematically expose errors in the benchmarks themselves (cf. Fig 1 for an overview). Across both datasets, the strongest provers identify incorrect specifications or implementations, propose fixes, and in many cases successfully prove the corrected statements. This behavior reveals a growing mismatch between the difficulty and robustness of existing program verification benchmarks and the capabilities of contemporary agentic systems.

More broadly, our results suggest that tight compiler-in-the-loop agentic paradigms currently provide the most effective mechanism for scalable proof generation in dependently typed program verification. At the same time, they raise important questions for the design of future benchmarks and evaluation protocols, particularly in settings where verification artifacts themselves may be incomplete or flawed.

2 EXPERIMENTAL SETUP

2.1 PROVERS

In our setup we employed the Claude Code CLI tool (built on the foundational Anthropic models) and its Claude Agent SDK library (Anthropic, 2026) as agentic framework, which we tested in two different configurations.

In our first basic configuration, we equipped it with *lean4-skills*, a package injecting domain-specific instructions, best practices, and workflow patterns into the model prompt, together with *lean-lsp-mcp* (Dressler, 2026), a specialized MCP server that interfaces with the Lean LSP and provides search tools to find relevant lemmas in the project context and Mathlib. This enables an iterative agentic loop in which the model inspects the current proof state, proposes tactics, and refines them based on compiler diagnostics. The package *lean4-skills* provides a wide array of utilities and tools to automate proof development, including experimental subagents for specific tasks. In our testing, we focused however on the `/fill-sorry` command (provided by the *lean4-theorem-proving* plugin inside the package) that systematically targets and resolves sorries, and we prompted the agent simply according to the command syntax and repeated for a total of 3 independent 'one-shot' attempts.

We then tested the recently published *Numina-Lean-Agent* (Liu et al., 2026; Project Numina, 2026b), a wrapper to Claude Code that combines it with *Numina-Lean-MCP* (Project Numina, 2026a), fork of *lean-lsp-mcp* (Dressler, 2026) extending it with other functionalities like *LeanDex*, a semantic search engine for theorems and definitions in standard Lean libraries, an *Informal Prover* (Huang & Yang, 2025), used to generate informal proof sketches, and a *Discussion Partner* tool to querying external LLMs for reasoning and planning. The agent ran with the provided file `prompts/prompt_complete_file.txt` as prompt, and the setting `max_rounds=3`, i.e. allowing up to 3 continuations of the same Claude Code conversation before exiting.

Other than agentic systems, in our testing we considered a range of other provers for Lean 4, serving as baseline for performance:

- **Aristotle** (Achim et al., 2025): a high-level, API-accessible system by Harmonic that achieves SoTA performance on established advanced mathematics benchmarks[1], that relies on a proof-state-aware Monte Carlo Graph Search (MCGS) over a hypertree of proving steps and tactics, together with an informal reasoning system generating lemmas and proof sketches and a specialized geometry solver.
- **Whole-proof generation models** like Kimina-Prover (Wang et al., 2025), DeepSeek-Prover-V2 (Ren et al., 2025), and Goedel-Prover-V2 (Lin et al., 2025): specialized deep learning models trained or fine-tuned for formal verification, that generally adopt chain-of-thought (CoT) and other reasoning strategies interleaved with formal code generation. We tested the Kimina-Prover-72B model as representative of this class, reportetly the highest performing open model of its kind on benchmarks like MINIF2F.
- **Standard tactics** like Grind (The Lean Developers, 2024), LeanHammer (Clune, 2026), Aesop (Limperg & From, 2023), and Canonical (Norman & Avigad, 2025): symbolic automated reasoning tools, integrated directly into Lean as tactics, serving as a baseline for symbolic automation.

2.2 BENCHMARKS

In our testing, we have considered the two Lean program verification benchmarks CLEVER (Thakur et al., 2025), and VERINA (Ye et al., 2025). Unlike traditional coding benchmarks that assess functional correctness via test cases, these datasets require provers to provide formal proofs of correctness (i.e. of adherence to a given formal specification). These benchmarks were originally designed to test code generation and verification systems end-to-end, starting from natural language (NL) descriptions, and they generally employ a staged reasoning pipeline to isolate specific capabilities within the verification loop:

SPECGEN	Translating NL intent into formal logical specifications (pre- and post-conditions).
CODEGEN	Synthesizing the functional implementations in Lean.
PROOFGEN	Constructing formal proofs that implementations adhere to their specification.

To facilitate granular analysis of automated provers, we focused on proof generation to isolate the model's ability to construct logical arguments. This resulted in specific derivative datasets (notated as -P variants, e.g. CLEVER-P, VERINA-P), obtained by providing ground-truth specifications and implementations taken from the original benchmarks, and admitted proofs for the provers to complete. NL descriptions are not included in these versions and thus not provided to the provers during testing. For additional details, please see Appx. B.

3 RESULTS

Table 1 presents the results. The evaluation on CLEVER and VERINA reveals a clear performance hierarchy: the agentic systems based on Claude Code currently define the state-of-the-art, with Aristotle trailing by a small margin; proof generation models like Kimina-Prover display much lower pass rates than reported on more standard benchmarks as MINIF2F, while symbolic provers provide a stable but significantly lower baseline.

Both Claude Code and Artistotle were able to identify bugs within the benchmark datasets themselves, Aristotle through its goal negation search mechanism, Claude through its answers returned in NL (sometimes already providing fixes and solving the fixed statements, without being specifically prompted). In particular, the provers were cumulatively able to identify 17 wrong entries in CLEVER[2], and 6 wrong entries[3] in the current version of VERINA[4].

[1] most notably having achieved gold-medal performance on the IMO 2025

[2] reported publicly on December 18th 2025, link removed for anonymity.

[3] We found a total of 16 wrong entries in the original version of VERINA, some of which were later fixed by authors on January 5th 2026 (commit `e9926ecbc6d203b8e2ca008b500ccda4d09a8c6e`). Those errors were already reported by Aristotle (`https://harmonic.fun/news#blog-post-verina-bench-sota`) and identified independently by us on November 16th 2025.

[4] reported publicly on January 26th 2026, link removed for anonymity.

Table 1: Number of solved tests by each provers over the correct entries in the tested benchmarks. Numbers added in brackets represent the number of solved entries among the auto-fixed ones. The second column is explained in appendix B.

		CLEVER				VERINA			
Prover		examples		human_eval		basic		advanced	
Numina Lean Agent	P	5	(+1)	27	(+14)	102	(+2)	57	(+3)
Claude Code + lean4-skills	P	5	(+1)	26	(+8)	96		40	(+2)
Aristotle	PM	5		23	(+9)	89	(+1)	43	(+2)
Kimina-Prover	P	4	N/A	3	N/A	31	N/A	3	N/A
Grind	PU	2		3	(+1)	31		2	
LeanHammer	PU	1		3		13		0	
Aesop	PU	1		2		13		0	
Canonical	PU	0		0		0		0	
number of entries		5	(+1)	27	(+16)	106	(+2)	77	(+4)

Table 2: Number of fixes per benchmark by kind.

	CLEVER	VERINA	Total
Wrong Specification Logic	7	2	9
Implementation Bugs	4	2	6
Missing Preconditions	3	2	5
Off-by-One/Boundary Errors	3		3
Total	17	6	23

When given the NL descriptions of the problems as well as the results of previous proof attempts where bugs were identified, Claude Code was furthermore able provide some fixes for all of them, and prove some of the resulting statements. Most bugs concerned wrong preconditions: table 2 provides an overview of the types of fixes made by Claude Code. Appendix A provides an example of erroneous implementation fixed by Claude Code.

Considering only the correct portions of the dataset, the CLEVER benchmark was completely saturated by the Numina-Lean-Agent that solved all 32 entries, while Claude Code with lean4-skills and Aristotle solved 28 and 31 respectively. Out of the 183 correct entries in VERINA instead, the three provers solved 159, 136 and 132 respectively. Numina-Lean-Agent further managed to prove 15/17 and 5/6 of the fixed statements of CLEVER and VERINA respectively, while Claude Code with lean4-skills proved 9/17 and 2/6, and Aristotle 9/17 and 3/6.

This high level of preformance is likely due to their direct access to the compiler feedback provided by their search and agentic capabilities, as this is the main documented difference in implementation.

The specialized proof-gen models perform much worse on this test suite, struggling to produce certified proofs without the error-correcting feedback loops available to agentic systems. Note, however, that in our experimental setup the models were tasked with a 'one-shot' attempt, without successive refinement of the proofs. Performance of whole-proof generation models may be greatly improved by allowing multiple iterations, where compiler feedback on previous attempts is fed back to the model as new context. We plan to test this hypothesis in future developments.

Finally, symbolic provers, like Grind and LeanHammer display significantly lower performance than other provers. This is not surprising, as they were designed to fill in gaps between assertions, not complete entire proofs. They also display a high degree of sensitivity to problem formulation and particularly to term unfolding, as shown in the difference in performance over the -P and -PU versions of both datasets.

4 CONCLUSIONS

Our experiments show that agentic provers achieve near-perfect performance on the Clever and Verina benchmarks, outperforming specialized provers. This calls for new benchmarks. We plan to create those in the future using tools for Rust verification, such as Aeneas (Ho & Protzenko, 2022) and Hax (Bhargavan et al., 2025).

REFERENCES

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, October 2025.

Anthropic. Claude-agent-sdk-python. *GitHub repository*, 2026. `https://github.com/anthropics/claude-agent-sdk-python`.

Barış Bayazıt, Yao Li, and Xujie Si. A case study on the effectiveness of llms in verification with proof assistants, 2025.

Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. hax: Verifying security-critical rust software using multiple provers. IACR ePrint Report 2025/142, 2025.

Mario Carneiro, Chad E. Brown, and Josef Urban. Automated theorem proving for metamath. In Adam Naumowicz and René Thiemann (eds.), *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 9:1–9:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.ITP.2023.9.

Wenhu Chen, Shuo Wei, Hao Wang, et al. Seed-prover: Deep and broad reasoning for automated theorem proving, 2025.

Joshua Clune. LeanHammer. *GitHub repository*, 2026. `https://github.com/JOSHCLUNE/LeanHammer`, accessed on Jan 21st 2026.

Oliver Dressler. Lean-lsp-mcp. *GitHub repository*, 2026. `https://github.com/oOo0oOo/lean-lsp-mcp`, accessed on Feb 3rd 2026.

Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar: State of the art and beyond. *Journal of Automated Reasoning*, 44(3):347–377, 2010. doi: 10.1007/s10817-009-9149-0.

John Harrison. Hol light: A tutorial introduction. In *Formal Methods in Computer-Aided Design (FMCAD)*, 1996.

Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. Proceedings of the ACM on Programming Languages (PACMPL), ICFP, 2022.

Yichen Huang and Lin F. Yang. Winning Gold at IMO 2025 with a Model-Agnostic Verification-and-Refinement Pipeline, September 2025.

Thomas Hubert, Harsh Mehta, Laurent Sartran, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 2025. doi: 10.1038/s41586-025-09833-y.

Jannis Limperg and Asta Halkjær From. Aesop: White-Box Best-First Proof Search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, pp. 253–266, New York, NY, USA, January 2023. Association for Computing Machinery. ISBN 979-8-4007-0026-2. doi: 10.1145/3573105.3575671.

Xiaohan Lin, Qingxing Cao, Yinya Huang, Haiming Wang, Jianqiao Lu, Zhengying Liu, Linqi Song, and Xiaodan Liang. Fvel: Interactive formal verification environment with large language models via theorem proving, 2024.

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, August 2025.

Junqi Liu, Zihao Zhou, Zekai Zhu, Marco Dos Santos, Weikun He, Jiawei Liu, Ran Wang, Yunzhou Xie, Junqiao Zhao, Qiufeng Wang, Lihong Zhi, Jia Li, and Wenda Li. Numina-Lean-Agent: An Open and General Agentic Reasoning System for Formal Mathematics, January 2026.

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*, pp. 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.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

Chase Norman and Jeremy Avigad. Canonical for Automated Theorem Proving in Lean, April 2025.

Project Numina. Lean-lsp-mcp. *GitHub repository*, 2026a. https://github.com/project-numina/lean-lsp-mcp, accessed on Feb 3rd 2026.

Project Numina. Numina-lean-agent. *GitHub repository*, 2026b. https://github.com/project-numina/numina-lean-agentn, accessed on Feb 3rd 2026.

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, April 2025.

Rocq Development Team. The rocq prover. *Project website*, 2026. https://rocq-prover.org.

Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *Lecture Notes in Computer Science*, pp. 28–32. Springer, 2008. doi: 10.1007/978-3-540-71067-7_6.

Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. CLEVER: A Curated Benchmark for Formally Verified Code Generation, October 2025.

The Lean Developers. *The Lean Language Reference: The grind tactic*. Lean FRO, 2024. URL https://lean-lang.org/doc/reference/latest/The--grind--tactic/. Lean version 4.15.0 or later.

Jules Viennot, Guillaume Baudart, Emilio Jesùs Gallego Arias, and Marc Lelarge. Minif2f in rocq: Automatic translation between proof assistants – a case study, 2025.

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, April 2025.

Qiyuan Xu, Renxi Wang, Peixin Wang, Haonan Li, and Conrad Watt. A minimalist proof language for neural theorem proving over isabelle/hol, 2025.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking Verifiable Code Generation, October 2025.

A EXAMPLE OF FIXED ERROR

```
17  def implementation (xs: List Rat) : Rat :=
18    let rec poly (xs: List Rat) (x: Rat) := xs.reverse.foldl (λ acc a ⇒ acc * x + a) 0;
  -   let rec poly' (xs: List Rat) (x: Rat) := (xs.drop 1).reverse.foldl (λ acc a ⇒ acc * x + a) 0;
19    let rec eps := (1: Rat) / 1000000;
  -   let rec find_zero (xs: List Rat) (guess: Rat) (fuel: Nat) :=
  -   let eval := poly xs guess;
  -   let eval' := poly' xs guess;
  -   if eval ≤ eps ∨ fuel = 0 then (guess, fuel)
  -   else
  -   let guess' := (eval' * guess - eval) / eval';
  -   find_zero xs guess' (fuel - 1);
  -   (find_zero xs 1.0 1000000).1
20+   let lc := |xs.getLast!|;
21+   let initBound : Rat := if lc = 0 then 1
22+     else 1 + xs.dropLast.foldl (fun acc a ⇒ acc + |a|) 0 / lc;
23+   let rec findBound (xs: List Rat) (b: Rat) (fuel: Nat) : Rat :=
24+     match fuel with
25+     | 0 ⇒ b
26+     | fuel' + 1 ⇒
27+       if poly xs b * poly xs (-b) ≤ 0 then b
28+       else findBound xs (b * 2) fuel';
29+   let rec bisect (xs: List Rat) (lo hi: Rat) (fuel: Nat) : Rat :=
30+     match fuel with
31+     | 0 ⇒ (lo + hi) / 2
32+     | fuel' + 1 ⇒
33+       let mid := (lo + hi) / 2;
34+       if |poly xs mid| ≤ eps then mid
35+       else if poly xs lo * poly xs mid ≤ 0 then bisect xs lo mid fuel'
36+       else bisect xs mid hi fuel';
37+   let bound := findBound xs initBound 100;
38+   let coeffBits := xs.foldl (fun acc a ⇒ acc + a.num.natAbs.size + a.den.size) 0;
39+   let fuel := 1000000 + coeffBits * (xs.length + 1) * 40;
40+   bisect xs (-bound) bound fuel
41
```

Here's a summary of the fix and why it's correct:

Root cause: Newton-Raphson is not globally convergent. Even with a correct derivative, it can enter cycles (as shown by [2, -2, 0, 1] cycling between 0 and 1). No starting point is safe for all polynomials.

Fix: Replaced Newton-Raphson with the bisection method, which is globally convergent:
1. findBound: Finds a bound b (starting from 1, doubling up to 2^{100}) such that poly(b) and poly(-b) have opposite signs. This is guaranteed to exist for odd-degree polynomials (even-length coefficient list) with non-zero leading coefficient -- exactly the docstring's preconditions.
2. bisect: Standard bisection on [-bound, bound] with 1,000,000 iterations. Each step halves the interval containing the root. After n steps, the interval has width 2·bound / 2^n, giving astronomically precise convergence -- far more than needed for eps = 10^{-6}.

Verification against test cases:
- [1, 2] (1 + 2x): bisect finds mid = -0.5 where poly = 0 exactly in 2 steps
- [-6, 11, -6, 1] ($x^3-6x^2+11x-6$): bisect converges to a value near root 1 with |poly| ≤ eps

The fix has two changes to the implementation (spec unchanged):

Fix 1 - Cauchy bound initialization (lines 20-22): Instead of findBound xs 1 100, we compute the Cauchy root bound $1 + \sum|a_i|/|a_i|$ as the starting value. This guarantees the initial bound is already past all roots, regardless of coefficient magnitude.

Fix 2 - Adaptive bisection fuel (lines 38-39): The bisection fuel scales with the total bit-size of the coefficients (coeffBits * (xs.length + 1) * 40 + 1000000). This ensures enough precision even for polynomials with large/steep coefficients.

Figure 2: Fix of CLEVER entry `human_eval/problem_32.lean` generated by Claude Code, and an extract of Claude's replies explaining the edits it made.

B ADDITIONAL BENCHMARK DETAILS

Other variants were also created for each benchmark to investigate sensitivity to problem formulation displayed by some provers. This includes the following two variants:

- -PU, where an initial unfolding and simplification of specifications and implementation definitions (inspired by existing solutions in each dataset) is added at the beginning of proofs before passing them to provers;
- -PM, where Mathlib is always impored as dependency (required by Aristotle);
- -PF, with auto-generated fixed to wrong statements in the datasets.

CLEVER (Thakur et al., 2025) Adapts 161 problems from the classic HumanEval dataset, with particular attention to preventing shortcuts that models may take (e.g., by avoiding specifications that look like code and tha could be directly "copy-pasted" into a solution). For proof-focused evaluation, we selected only the entries with complete ground-truth specifications and implementations, for a total of 49 entries out of 161.

VERINA (Ye et al., 2025) Composed of 189 manually curated tasks split into two difficulty tiers: (1) VERINA-A (basic): 108 tasks derived from existing human-written Dafny benchmarks (MBPP-DFY, CloverBench) and translated to Lean; and (2) VERINA-B (advanced): 81 more complex algorithmic problems adapted from platforms like LeetCode and challenging datasets like LiveCodeBench. The ground-truth specs and implementations were provided for all entries, while only 46 entries of the basic dataset had solutions to the proof-gen step provided, the remainder was left as sorrys. This allowed us to employ the whole dataset in our testing.