PARTIAL SOUNDNESS: NATURAL LANGUAGE THEOREM PROVING WITH MACHINE-CHECKED LOGICAL STRUCTURE

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models are increasingly proficient in mathematical reasoning tasks in natural language, but such outputs can lack formal guarantees, whereas fully formal theorem proving remains difficult due to various factors. We introduce PARTIAL SOUNDNESS, a theorem-proving framework that bridges these settings through machine-checked proof structure. Given a theorem, the framework decomposes it into intermediate subgoals, formalizes the subgoals in Lean, and verifies that they are sufficient to derive the target theorem before any natural language proof is generated. Natural language proofs are then produced and assembled into a complete proof. This yields a partial soundness guarantee: while leaf-level reasoning remains informal, the logical dependency structure of the proof is formally verified. We also introduce INFORMALRA, a dataset of 114 undergraduate real analysis theorems paired with Lean formalizations. Experiments on INFORMALRA and existing benchmarks demonstrate that the proposed framework produces proofs with machine-checked logical structure and more interpretable intermediate reasoning while displaying reasonable proof-generation performance. These results suggest that PARTIAL SOUNDNESS provides a practical middle ground between unconstrained natural language theorem proving and fully formal proof generation.

1 INTRODUCTION

Automated theorem proving (ATP) has become an increasingly important research direction in artificial intelligence and mathematical reasoning. Recent large language models (LLMs) have demonstrated strong performance on mathematical problem solving and logical reasoning tasks (Azerbayev et al., 2023b; Gou et al., 2023), motivating renewed interest in natural language-based theorem proving systems (Welleck et al., 2021a;b). At the same time, formal theorem proving environments such as Lean provide a setting for machine-verifiable proofs and rigorous logical validation, making them attractive frameworks for reliable ATP systems. Recent work on theorem proving with large language models has largely focused on the generation of fully formal proofs (Polu & Sutskever, 2020; Lample et al., 2022; Xin et al., 2024; Wang et al., 2024b; Han et al., 2025).

These two directions, however, involve an inherent tradeoff. Natural language proofs are flexible, intuitive, and closely aligned with existing mathematical writing, but they are difficult to verify automatically. In contrast, formal theorem proving provides machine-checked correctness guarantees, but generating fully formal proofs remains challenging even for state-of-the-art models. Success depends not only on mathematical reasoning ability, but also on formalization-specific details such as exact syntax, theorem names, imported libraries, and proof-assistant conventions. As a result, a proof attempt may fail despite containing the correct mathematical idea, making it difficult to determine if the failure reflects a weakness in mathematical reasoning or merely an inability to navigate the formal language and library infrastructure. Moreover, fully formal proofs are often less accessible to human readers than conventional mathematical arguments.

We adopt two working premises concerning the mathematical problems of primary interest in this paper, not high-profile problems such as famous conjectures or well-known mathematics competition problems:

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

- Current LLMs are better at generating informal proofs than machine-checkable proofs. This premise is supported by reported gaps between informal and formal reasoning in advanced algebra and mathematical analysis (Jiang et al., 2026; Pu et al., 2026).
- LLM-generated natural language proofs are easier to understand than machine-checkable proofs. This premise is motivated in part by work on translating formal proofs into readable natural language through abstraction and summarization (Hattori et al., 2025).

These premises motivate a framework that preserves the strengths of natural language reasoning while introducing formal guarantees at the level of proof structure.

In this work, we propose a PARTIAL SOUNDNESS framework that combines natural language proof generation with intermediate formal verification in Lean (de Moura & Ullrich, 2021). Rather than attempting to formally verify an entire proof end-to-end, our method first decomposes a theorem into intermediate subgoals, formally verifies that these subgoals are logically sufficient to derive the original theorem, and then generates natural language proofs conditioned on this verified structure. This approach aims to preserve the flexibility of natural language reasoning while grounding the overall proof structure in a machine-verified framework.

The primary contribution of this work is the formulation of partial soundness as an explicit guarantee for LLM-based theorem proving. Full proof generation is based on a compilable Lean 4 outline—subgoal proofs deferred via `sorry`—so the final natural language proof inherits a machine-checked logical structure. We also introduce INFORMALRA, a dataset of 114 undergraduate real analysis theorems paired with Lean formalizations, and demonstrate that our framework maintains competitive proof-generation performance while providing more interpretable intermediate reasoning through machine-checked proof structure.

The outline of the rest of the paper is as follows. We briefly cover some preliminary background material, lay out the PARTIAL SOUNDNESS pipeline, and discuss the characteristics of this framework. We also describe our novel dataset INFORMALRA, and detail our experiments. Then we display and analyze the results of our experiments and discuss our findings. The appendix presents examples from INFORMALRA. It also provides the problem statements, solutions, and additional discussions on the case studies presented in the main text, together with the prompts we used.

2 BACKGROUND

2.1 DECOMPOSITION-BASED THEOREM PROVING

Complex mathematical proofs are often easier to construct when decomposed into smaller intermediate goals. Motivated by this observation, several ATP approaches incorporate intermediate reasoning structures or multi-stage proof generation strategies (Jiang et al., 2023; Wang et al., 2025).

Verifiable sketches with deferred subgoals also appear in prior work. Wang et al. (2024a) recursively prove the deferred subgoals in Isabelle, while Varambally et al. (2026) combine informal reasoning, recursive decomposition, and Lean verification to construct formal proofs. Cabral et al. (2026) use dependency graphs and intermediate lemmas to preserve the structure of an existing natural language proof during autoformalization. In our framework, the checked outline instead serves as a gate and a roadmap for generating a natural language proof: a candidate derivation must close the formal target before the model writes separate natural language subproofs and assembles them into the final argument. This exposes the proposed mathematical obligations and checks how their formalizations fit together while keeping the final output in mathematical prose.

2.2 DIVERSIFYING MATHEMATICAL DATASETS

Recent mathematical benchmarks for large language models have predominantly been constructed from contest problems (Tsoukalas et al., 2024; Luong et al., 2025). Such problems provide a useful testbed for evaluating mathematical reasoning, and they will likely remain an important component of future benchmarks. However, contest problems are not the only natural source of mathematical data. Datasets based on undergraduate textbooks, course materials, and subject-specific mathematical sources can provide complementary ways to evaluate mathematical reasoning in different

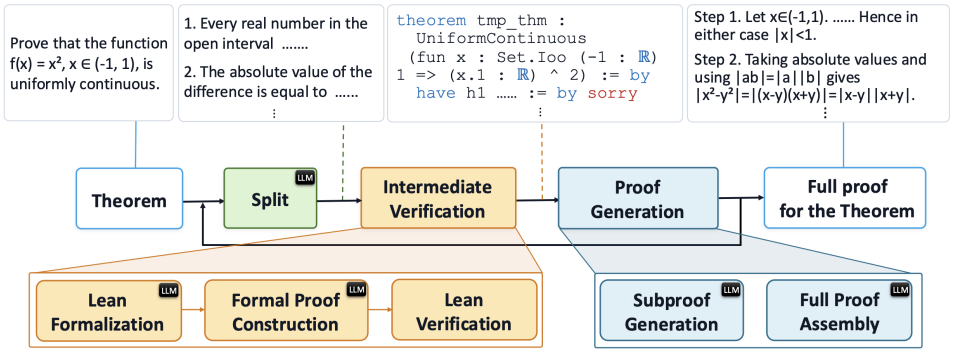


Figure 1: Overview of the PARTIAL SOUNDNESS pipeline. A theorem is decomposed into subgoals, Lean verifies that the theorem follows from them, and proofs for the subgoals are then generated and assembled into a full proof. This ordering provides partial soundness by machine-checking the derivation of the theorem from the subgoals.

mathematical contexts. Our dataset follows this broader direction by focusing on undergraduate real analysis, illustrating one possible approach to diversifying mathematical datasets.

3 METHOD

Our PARTIAL SOUNDNESS framework decomposes a theorem into intermediate subgoals, formally verifies that these subgoals are logically sufficient to derive the original theorem, and only then generates natural language proofs for the subgoals and assembles them into a full proof. This ordering is the core design choice: the model never attempts the full proof until the skeleton connecting its pieces has already been machine-checked. The PARTIAL SOUNDNESS pipeline is illustrated in Figure 1. The pipeline consists of six modules organized into three stages:

(1) The **Split Stage** consists of the *Split Module*, which oversees theorem decomposition.

(2) The **Intermediate Verification Stage** consists of three modules:
- *Lean Formalization Module:* autoformalizes subgoals into Lean propositions,
- *Formal Proof Construction Module:* constructs a candidate Lean derivation of the theorem from the (unproved) subgoals,
- *Lean Verification:* compiles and checks the candidate derivation.

(3) The **Proof Generation Stage** consists of two modules:
- *Subproof Generation Module:* proves each subgoal in natural language,
- *Full Proof Assembly Module:* integrates subproofs into a single natural language proof.

More details on the modules are provided below. All modules rely on the use of an LLM except for the Lean Verification Module. Here, the workflow is predefined rather than autonomously determined by an LLM. If the framework fails to generate a proof for a theorem or the proof is invalid, an LLM generates a gap report listing its shortcomings. This report is referred to on a reattempt at proof generation through the PARTIAL SOUNDNESS framework, and no further reattempts are allowed.

3.1 MODULES

Split Module. Given a theorem statement, the Split Module decomposes the theorem into a set of intermediate subgoals. These subgoals serve as the proof outline for the remainder of the pipeline, providing the inputs for formal verification and natural language proof generation. The objective is to identify intermediate statements that capture the theorem's logical structure while reducing the complexity of the proof generation task. The resulting decomposition provides a roadmap for constructing the final proof as well as the collection of statements whose logical sufficiency is verified in the Intermediate Verification Stage.

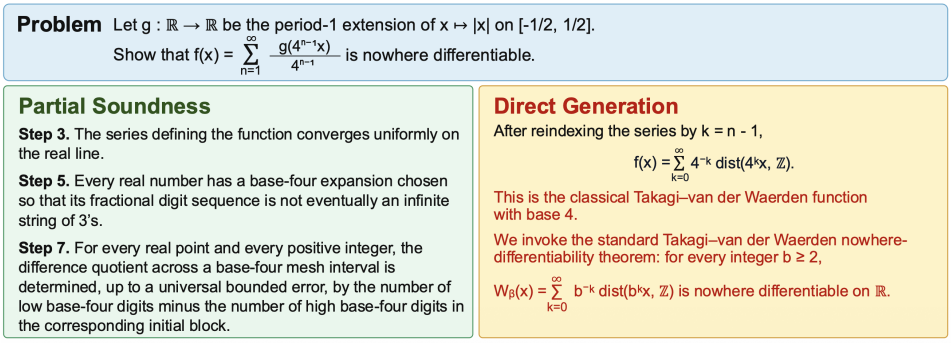


Figure 2: Proof excerpts for an INFORMALRA theorem. Our pipeline (left) states and proves intermediate subgoals, whereas direct generation (right) closes by citing the Takagi–van der Waerden theorem, which already covers the problem.

Lean Formalization Module. The Lean Formalization Module autoformalizes each natural language subgoal into a Lean 4 proposition. This step produces statements only; no proofs are attempted here. The module also takes as input a Lean statement of the original theorem, rather than autoformalizing the theorem itself at this stage.

Formal Proof Construction Module. The Formal Proof Construction Module takes the formalized subgoal propositions together with the Lean statement of the theorem and constructs a candidate derivation: each subgoal is introduced as a hypothesis via a `have` statement whose proof is deferred with `sorry`, and the module then attempts to prove the theorem's Lean statement based on these hypotheses. The output is a single Lean code snippet (a proof outline) that is conditionally complete. It is missing the proofs of the leaves (the subgoals), but is a complete claim about how those leaves combine.

Lean Verification. The candidate outline from the Formal Proof Construction Module is type-checked by Lean 4. Verification checks two things: (1) that the formalized subgoals in the outline are well-formed and type-correct, and (2) that the theorem's proof term successfully closes given the subgoals as hypotheses, i.e., that Lean accepts the code up to the declared `sorry`s. This step does *not* establish that the subgoals are true—only that, *if* they hold, the theorem follows. This conditional guarantee is what we call partial soundness. A trivial decomposition can also pass this check. For example, consider a single subgoal that simply restates the theorem. We therefore apply an additional screening step to rule out such degenerate outlines before entering the Intermediate Verification Stage.

Subproof Generation Module. Once a decomposition passes verification, the Subproof Generation Module generates a natural language proof for each subgoal separately. Because each subgoal's role in the overall argument has already been fixed by the verified outline, the model is not required to also discover *how* the subgoals fit together—only to prove each one.

Full Proof Assembly Module. The Full Proof Assembly Module integrates the generated subgoal proofs into a single natural language proof of the original theorem, rather than concatenating them. It follows the logical structure fixed by the verified outline—connecting assumptions, intermediate claims, and the final conclusion—while producing output intended to read as a single coherent mathematical argument, not a sequence of labeled lemmas.

4 CHARACTERISTICS OF PARTIAL SOUNDNESS

Unlike end-to-end approaches, our framework does not ask the model to generate a complete proof in a single step. Instead, proof generation is conditioned on a proof outline that is verified for logical sufficiency of the proposed subgoals. Only after this intermediate structure has been validated does the model generate natural language proofs for the individual subgoals. The result is precisely the

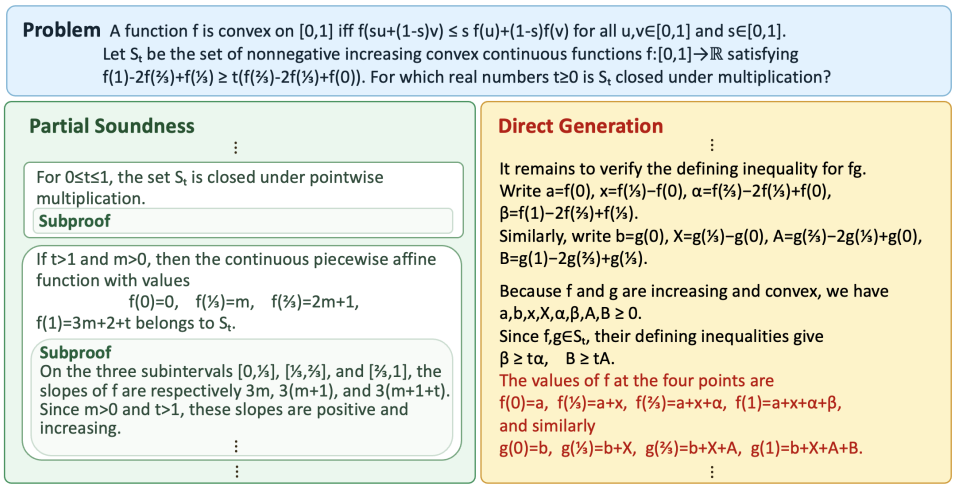


Figure 3: Proof excerpts for a Putnam problem. In our pipeline (left), explicit steps outline the proof structure, each with its own subproof, making the flow clear at a glance. Direct generation (right) yields a monolithic proof, so locating the error (red) and tracing its effects requires reading it in full.

partial soundness guarantee of our title: the leaf-level reasoning remains natural language, but the logical structure connecting it is machine-checked—implicit hypotheses surface as explicit subgoal statements, and errors in individual subgoals can be identified, not buried in a monolithic proof.

Retaining natural language proofs for the subgoals is a deliberate part of this design. Recent mathematical reasoning benchmarks report substantial gaps between models' informal mathematical reasoning and their ability to produce complete machine-checkable proofs; errors in Lean and library use can account for failures even when the underlying mathematical argument is sound (Jiang et al., 2026; Pu et al., 2026). By checking in Lean how the formalized intermediate propositions compose to establish the provided theorem statement, while allowing the corresponding subproofs to be written in ordinary mathematical language, our framework applies formal verification to the structure of the assembled argument without requiring every leaf proof to depend on Lean implementation details. The resulting proof presents the mathematical reasoning in a form that readers can follow without reading a tactic script, while retaining an inspectable record of the intermediate claims and their role in the overall argument.

4.1 Reduced Reliance on External Theorems

Figure 2 shows an example of a theorem statement from our new dataset that asks for a proof that a given function is nowhere differentiable. We refer to proof generation without our pipeline as *direct generation*. In this case, direct generation gives a proof that identifies the function as a Takagi–van der Waerden function and closes by invoking an existing result for the function, as shown in red. The conclusion may hold, but the reasoning specific to the problem ends there. Our Partial Soundness pipeline instead derives the same conclusion from a machine-checked decomposition and the necessary subgoals that state the intermediate content.

The distinction lies not in mathematical correctness, but in how the reasoning is represented. Direct generation compresses the argument into a citation, whereas Partial Soundness exposes and verifies the intermediate mathematical structure. Consequently, a citation-based proof provides only limited evidence that a model can reconstruct the underlying derivation, since success may depend on retrieving a sufficiently strong theorem. By exposing the intermediate derivation, Partial Soundness makes this reasoning directly observable.

4.2 Stepwise Proofs

The Partial Soundness pipeline produces proofs that are organized as sequences of steps or subgoals, which reveal the overall proof structure. These subgoals alone describe how the proof

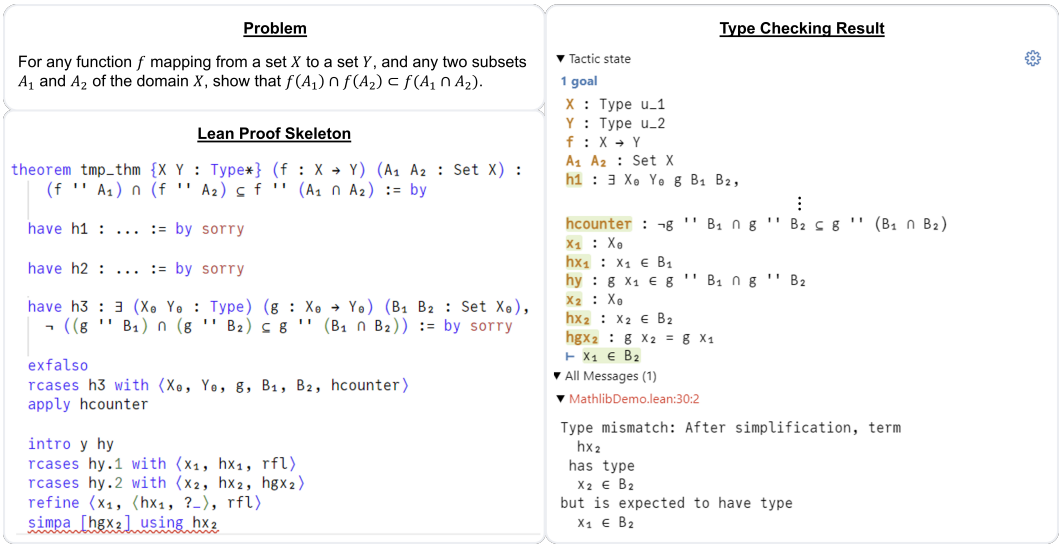


Figure 4: Controlled example illustrating the effect of Lean verification. The left panels present a deliberately flawed problem obtained by omitting the injectivity assumption and the generated Lean proof skeleton, while the right panel shows the corresponding type-checking result.

proceeds, so its structure can be illuminated without reading the subgoal proofs, or subproofs. The individual subgoals can then be parsed separately to understand the subproofs for each subgoal. On the other hand, direct generation produces a single monolithic proof in which the intermediate claims are not clearly identified.

Figure 3 illustrates this distinction. In PARTIAL SOUNDNESS, each subgoal is followed by its own subproof, so checking a step requires reading only the subproof placed under it. Direct generation yields one monolithic proof, but contains an error marked in red text. Fixing this proof requires not just the identification of errors, but also an understanding of how the errors may have permeated into later parts of the proof. The stepwise proof offered by the PARTIAL SOUNDNESS framework addresses both needs: an error inside a subproof affects its target subgoal, and the role of that step in the overall proof is clear from the verified theorem decomposition.

4.3 EFFECT OF LEAN VERIFICATION

Generating a natural language proof in a single pass can make it difficult to identify flawed logical reasoning in its intermediate steps. Indeed, through manual inspection of proofs generated across multiple datasets, we found cases in which the overall argument appeared plausible but contained errors or gaps in recurrence relations, boundary cases, algebraic derivations, or other details. Specific examples can be found in the appendix. These observations highlight the need to examine whether the logical flow of an argument adequately justifies the target conclusion before generating or accepting the final natural language proof.

In our PARTIAL SOUNDNESS algorithm, we aim to perform this examination through the Intermediate Verification Stage. To check its role, we apply the PARTIAL SOUNDNESS pipeline to a controlled example from which a necessary assumption has been deliberately removed: for an arbitrary $f : X \to Y$ and $A_1, A_2 \subseteq X$, the statement claims $f(A_1) \cap f(A_2) \subseteq f(A_1 \cap A_2)$, which is false without injectivity of f. Figure 4 shows the generated Lean proof skeleton and the corresponding type-checking result. Among the subgoals produced by the Split Stage, *h3* asserts the existence of a counterexample to the inclusion, and the skeleton attempts to derive a contradiction by proving the inclusion that is denied in *hcounter*. After elaboration, the remaining goal is $x_1 \in B_2$, given $hx_2 : x_2 \in B_2$ and $hgx_2 : g(x_2) = g(x_1)$; without injectivity, $x_2 = x_1$ cannot be derived, so hx_2 fails to have the required type and verification fails. The decomposition thus remains uncertified.

This demonstrates that intermediate formal verification can block the indiscriminate generation of natural language proofs. However, Lean does not determine that the statement is false, identify the

Table 1: Manual evaluation for PS and DG, averaged over two trials.

Method	Dataset	Number of Problems	No Proof	Incorrect	Correct
PS	INFRA	114	0	0	114
	Putnam	220	1.5	2.5	216
	Atlas	70	0.5	1.5	68
	ProofNet	331	0	2	329
DG	INFRA	114	0.5	0	113.5
	Putnam	220	2.5	2.5	215
	Atlas	70	0	0.5	69.5
	ProofNet	331	4	0.5	326.5

missing assumption, or judge the subgoal steps semantically incorrect. Since a verification failure may also stem from limitations of proof generation, a formalization mismatch, or incorrect use of library lemmas, it should be read as non-certification rather than as evidence of falsity. The Intermediate Verification Stage nonetheless serves as a conservative certification gate. By rejecting candidate outlines that fail Lean type-checking, it prevents them from proceeding to natural language proof generation, thereby reducing the risk that a failed formal composition guides the final proof.

5 THE INFORMALRA DATASET

As recent evaluation efforts move beyond competition problems, we aim to construct a new dataset that evaluates domain-specific expertise in real analysis. Existing publicly available datasets do not provide a course-centered collection spanning the core topics of undergraduate real analysis. The problems in our dataset were adapted primarily from the exercises in *Elementary Classical Analysis* by Jerrold E. Marsden and Michael J. Hoffman. We opted for this textbook because it is a well-established undergraduate textbook on real analysis and provides a coherent treatment of the subject. We curated a subset of diverse and representative problems to attain a final dataset that would cover a broad range of proof patterns and concepts.

Our dataset, which we call INFORMALRA, consists of 114 problems, each comprising a natural language statement along with its corresponding Lean formalization. To construct the dataset, we paraphrased the selected textbook exercises while preserving their original mathematical meaning. Each problem contains exactly one proof objective. Any problems involving multiple objectives were decomposed into separate problems. For example, each "if and only if" statement was split into two separate theorem statements. This allows each generated proof to be evaluated against a single mathematical objective and avoids excessively complex formal theorem statements. We formalized each problem in Lean, generated with the help of LLMs and then reviewed by humans to ensure their semantic alignment with the corresponding natural language problem, and finally type-checked in Lean. Since mathematical concepts may admit multiple equivalent definitions, and the definitions used in our chosen textbook may differ from those in other textbooks or formal libraries, we also identified and refined the relevant definitions from the textbook to guide formalization and to provide context during natural language proof generation.

6 EXPERIMENTAL SETTINGS

We evaluate PARTIAL SOUNDNESS against the direct generation baseline on INFORMALRA and three external datasets, excluding problems containing errors. We selected these datasets for their representation of complementary sources of mathematical reasoning. We use 220 analysis problems from PutnamBench, a benchmark consisting of problems from an undergraduate-level mathematics competition (Tsoukalas et al., 2024), as well as 70 problems from the *Real Analysis* component of Atlas, an LLM-autoformalized Lean 4 library of mathematics textbooks (Rammal et al., 2026). We also use 331 problems from ProofNet, an undergraduate-level mathematics benchmark (Azerbayev et al., 2023a). Evaluating across all three allows us to assess PARTIAL SOUNDNESS beyond a single style of theorem-proving task.

For both PARTIAL SOUNDNESS and direct generation, proof generation is performed using GPT-5.5, while intermediate verification, used only in PARTIAL SOUNDNESS, is carried out in Lean 4. At the

Table 2: LLM-based evaluation under two protocols: (a) from DeepSeek, and (b) from DeepTheorem. Cm., Cr., and Cl. are used for Completeness, Correctness, and Clarity, respectively.

(a) DeepSeek evaluation

Method	Dataset	No Proof	0	0.5	1
PS	INFRA	0	0.00	0.25	113.75
	Putnam	1.5	2.50	3.25	212.75
	Atlas	0.5	1.75	3.50	64.25
	ProofNet	0	0.00	3.50	327.50
DG	INFRA	0.5	0.00	0.75	112.75
	Putnam	2.5	2.00	6.50	209.00
	Atlas	0	2.25	8.00	59.75
	ProofNet	4	0.25	16.00	310.75

(b) DeepTheorem evaluation

Method	Dataset	Overall	Validity	Cm.	Cr.	Cl.
PS	INFRA	0.9846	0.9985	0.9934	1.0000	0.9460
	Putnam	0.9814	0.9883	0.9856	0.9988	0.9476
	Atlas	0.9550	0.9648	0.9510	0.9944	0.9312
	ProofNet	0.9868	0.9981	0.9947	0.9997	0.9484
DG	INFRA	0.9797	0.9971	0.9722	1.0000	0.9740
	Putnam	0.9651	0.9775	0.9433	0.9989	0.9582
	Atlas	0.9433	0.9612	0.9093	0.9970	0.9499
	ProofNet	0.9769	0.9942	0.9615	1.0000	0.9703

framework level, both methods allow a single reattempt within each trial (as outlined in Section 3). For each theorem, PARTIAL SOUNDNESS sends a candidate proof outline to Lean for verification. If verification fails, a new candidate is generated using Lean feedback, with up to five retries per theorem within each attempt. Both PARTIAL SOUNDNESS and direct generation are independently executed twice on each dataset, and all results are averaged across two trials.

The generated Lean code is automatically checked by the Lean 4 compiler to verify syntactic and type correctness. Final correctness of the natural language proofs, however, is determined through manual evaluation by graduate students in mathematics. In addition, Table 2 reports LLM-based evaluations using two complementary grading protocols based on DeepSeekMath-V2 (Shao et al., 2025), which we call DeepSeek, and DeepTheorem (Zhang et al., 2025). To mitigate position bias, the LLM-based comparison follows an A-B/B-A evaluation protocol in which the two generated solutions are evaluated under both presentation orders.

7 RESULTS

We report the performance of PARTIAL SOUNDNESS, denoted by *PS*, and compare it with the direct generation baseline, denoted by *DG*. We use *INFRA* to denote INFORMALRA. All reported results are averaged over two trials; consequently, the values in Tables 1-3 may be fractional. Our objective is not primarily to increase the accuracy of proof generation. Rather, the contribution of PARTIAL SOUNDNESS is to introduce a formally verified and reliable logical structure while maintaining competitive proof quality. The following experiments therefore evaluate not only end-task correctness but also the structural characteristics of the generated proofs.

7.1 MANUAL EVALUATION OF PROOF CORRECTNESS

Table 1 summarizes the results of manual evaluation for PS and DG. Each generated proof is classified as Correct, Incorrect, or No Proof, where No Proof denotes cases in which the model fails to generate a proof. Across all four datasets, PS achieves proof correctness comparable to DG. Despite introducing theorem decomposition and intermediate verification before proof generation, PS maintains essentially the same level of mathematical correctness as DG. These results indicate that machine-checking the logical structure can be incorporated without sacrificing overall proof quality.

Another observation is that, across most datasets, PS exhibits fewer proof generation failures than DG, although the differences are small. One plausible explanation is that DG must understand the problem and construct a complete solution in a single pass, whereas PS decomposes the problem into smaller subgoals and solves them sequentially. This step-by-step process may reduce the reasoning burden on the model and thereby lower the likelihood of failure.

7.2 LLM EVALUATION OF PROOF CORRECTNESS AND QUALITY

Table 2 reports automatic evaluations using two complementary LLM-based grading protocols: a categorical grading scheme based on DeepSeek and a rubric-based evaluation adapted from DeepTheorem. Across both evaluation protocols, PS exhibits trends consistent with the manual

Table 3: Proof-length and internal diagnostic metrics for PS and DG.

Dataset	Detailed Step (PS only)		Extra Step (PS only)		Extra Sentence (PS / DG)		Average Sentence (PS / DG)
	# of Problems	Average	# of Problems	Average	# of Problems	Average	
INFRA	58	2.23	1.5	1.00	13.5 / 1	2.19 / 1.50	48.39 / 13.11
Putnam	105.5	2.33	8.5	1.24	17.5 / 41	2.80 / 1.54	64.60 / 18.69
Atlas	35	2.10	0	0	6.5 / 0.5	3.15 / 1.00	50.33 / 15.36
ProofNet	161.5	3.26	7	1.57	50 / 0.5	1.08 / 1.00	55.00 / 14.75

evaluation in Table 1. In particular, PS maintains proof quality comparable to DG. The agreement between manual and LLM-based evaluations suggests that the observed performance of PS is not specific to a single evaluation methodology. Instead, both forms of evaluation indicate that introducing a machine-checked logical structure via Lean preserves the quality of generated natural language proofs while providing an additional formal guarantee unavailable in direct generation. The DeepTheorem evaluation further provides insight into different aspects of proof quality. Across all four datasets, PS achieves validity and correctness scores comparable to those of DG while generally obtaining higher completeness scores.

7.3 Proof Length and Structural Analysis

Table 3 reports a structural analysis of the generated proofs, and examines the structure of proofs generated through PS. The results are given in terms of three characteristics. *Detailed Step* refers to a subgoal that is mathematically correct and necessary in a proof, but deemed to be overly fine-grained. On the other hand, *Extra Step* denotes a subgoal that is mathematically incorrect or unnecessary, and *Extra Sentence* refers to a non-contributory sentence in a proof. These features are identified manually. For each feature, we report the average number of problems exhibiting it and the average number of occurrences per affected proof. As DG does not produce explicit subgoals, we report only Extra Sentence for it. The average number of sentences in a proof is reported for PS and DG.

Proofs by PS are longer than those by DG, reflecting the explicit decomposition of proofs into intermediate subgoals. Detailed steps occur in roughly half of generated proofs across the datasets. In contrast, mathematically incorrect or unnecessary intermediate steps arise in only a small fraction of generated proofs. Furthermore, while extra sentences appear more frequently in PS proofs, when they occur, they account for a smaller share of proof length than in DG. This pattern suggests that the additional structure introduced by decomposition is generally meaningful rather than redundant.

8 Discussion

We presented Partial Soundness, a theorem-proving framework that combines natural language proof generation with intermediate machine-checked verification in Lean. We also introduced InformalRA, a dataset of undergraduate real analysis theorems paired with human-reviewed Lean formalizations, and evaluated Partial Soundness on both this dataset and existing theorem-proving benchmarks. Our results demonstrate that Partial Soundness maintains competitive proof quality while producing more explicit and interpretable proof structures.

However, our results depend on the capabilities of the underlying LLMs. Assessing decomposition quality and formalization fidelity still requires problem-specific human judgment, and our evaluations do not yet establish when decomposition is most beneficial. Furthermore, the Partial Soundness framework shows that the formalized subgoals are logically sufficient to derive the theorem, but the natural language proofs of these subgoals remain unverified.

Ultimately, our Partial Soundness framework illustrates that intermediate formal verification provides a practical middle ground between unconstrained natural language theorem proving and fully formal proof generation. We hope this framework motivates future work on theorem-proving systems that combine human-readable mathematical reasoning with machine-checked logical guarantees. Our future work will explore reducing dependence on LLM capabilities, enabling broader access through open-source models, and developing more systematic methods to assess and improve decomposition quality, formalization fidelity, and the soundness and readability of generated proofs.

AI USE STATEMENT

In this work, we used generative AI tools for research execution and to aid in both writing and generating the datasets. In particular, all modules in our PARTIAL SOUNDNESS framework except the Lean Verification Module rely on the use of an LLM. We also used LLMs for direct generation (proof generation without our pipeline) as well as for evaluating proof correctness and quality. Additionally, LLMs were used in preparing the INFORMALRA dataset, specifically to paraphrase the statements and produce initial, partial formalizations of those statements in Lean. We also used LLMs in this work to proofread, polish, and streamline the text, and to aid in code writing and debugging. We have not used generative AI tools for ideation or to find research inspiration, nor in drafting whole sections of this work. We have reviewed all AI-assisted work. Three graduate students in mathematics tested and verified the INFORMALRA dataset and manually evaluated proofs generated by our PARTIAL SOUNDNESS framework and those generated directly by LLMs. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Our source code and the INFORMALRA dataset will be available on GitHub. All other datasets used in this work are publicly available. Exact prompts and select examples are provided in the Appendix.

REFERENCES

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics, 2023a. eprint: arXiv:2302.12433.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen Marcus McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics, 2023b. eprint: arXiv:2310.10631.

Rafael Cabral, Tuan Manh Do, Xuejun Yu, Wai Ming Tai, Zijin Feng, and Xin Shen. ProofFlow: A dependency graph approach to faithful proof autoformalization. In *International Conference on Learning Representations*, 2026.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe (eds.), *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Minlie Huang, Nan Duan, Weizhu Chen, et al. ToRA: A tool-integrated reasoning agent for mathematical problem solving, 2023. eprint: arXiv:2309.17452.

Sangjun Han, Taeil Hur, Youngmi Hur, Kathy Sangkyung Lee, Myungyoon Lee, and Hyojae Lim. Simplifying formal proof-generating models with ChatGPT and basic searching techniques. In *Intelligent Computing: Proceedings of the 2025 Computing Conference, Volume 2*, volume 1424 of *Lecture Notes in Networks and Systems*, pp. 205–222. Springer, 2025. doi: 10.1007/978-3-031-92605-1_14.

Seiji Hattori, Takuya Matsuzaki, and Makoto Fujiwara. Natural language translation of formal proofs through informalization of proof steps and recursive summarization along proof structure. In *Proceedings of the 18th International Natural Language Generation Conference*, pp. 376–389, Hanoi, Vietnam, October 2025. Association for Computational Linguistics.

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations*, 2023.

Jiedong Jiang, Wanyi He, Yuefeng Wang, Guoxiong Gao, Yongle Hu, Jingting Wang, Nailin Guan, Peihao Wu, Bryan Dai, Liang Xiao, and Bin Dong. FATE: A formal benchmark series for frontier algebra of multiple difficulty levels. In *International Conference on Learning Representations*, 2026.

Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. In *Conference on Neural Information Processing Systems*, volume 35, pp. 26337–26349, 2022.

Thang Luong, Dawsen Hwang, Hoang H. Nguyen, et al. Towards robust mathematical reasoning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020. eprint: arXiv:2009.03393.

Lushi Pu, Weiming Zhang, Xinheng Xie, Zixuan Fu, Bingxiang He, Hongya Lyu, Xin Li, Jie Zhou, and Yudong Wang. MA-ProofBench: A two-tiered evaluation of LLMs for theorem proving in mathematical analysis. *arXiv preprint arXiv:2606.13782*, 2026. doi: 10.48550/arXiv.2606.13782.

Ahmad Rammal, Niket Patel, Fabian Gloeckle, Amaury Hayat, Julia Kempe, Rémi Munos, Charles Arnal, and Vivien Cabannes. Formalizing mathematics at scale, 2026. eprint: arXiv:2605.29955.

Zhihong Shao, Yuxiang Luo, Chengda Lu, Z.Z. Ren, Jiewen Hu, Tian Ye, Zhibin Gou, Shirong Ma, and Xiaokang Zhang. DeepSeekMath-V2: Towards Self-Verifiable Mathematical Reasoning, 2025. eprint: arXiv:2511.22570.

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 *Advances in Neural Information Processing Systems*, volume 37, 2024.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. In *International Conference on Learning Representations*, 2026.

Haiming Wang, Huajian Xin, Zhengying Liu, Wenda Li, Yinya Huang, Jianqiao Lu, Zhicheng Yang, Jing Tang, Jian Yin, Zhenguo Li, and Xiaodan Liang. Proving theorems recursively. In *Conference on Neural Information Processing Systems*, 2024a.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. LEGO-Prover: Neural theorem proving with growing libraries. In *International Conference on Learning Representations*, 2024b.

Haiming Wang, Mert Unsal, Xiaohan Lin, et al. Kimina-Prover Preview: Towards large formal reasoning models with reinforcement learning, 2025. eprint: arXiv:2504.11354.

Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. NaturalProofs: Mathematical theorem proving in natural language. In *Conference on Neural Information Processing Systems*, 2021a.

Sean Welleck, Jiacheng Liu, Jesse Michael Han, and Yejin Choi. Towards grounded natural language proof generation. In *NeurIPS 2021 Workshop on Math AI for Education (MATHAI4ED)*, 2021b.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, 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, 2024. eprint: arXiv:2408.08152.

Ziyin Zhang, Jiahao Xu, Zhiwei He, Tian Liang, Qiuzhi Liu, Yansi Li, Linfeng Song, Zhenwen Liang, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. DeepTheorem: Advancing LLM Reasoning for Theorem Proving Through Natural Language and Reinforcement Learning, 2025. eprint: arXiv:2505.23754.

APPENDIX

A INFORMALRA EXAMPLES AND PROOFS GENERATED BY PARTIAL SOUNDNESS

This section presents some problems from INFORMALRA and their complete proofs generated by PARTIAL SOUNDNESS.

Example 1: Problem Used in Figure 1 of the Main Text.

Source: INFORMALRA.

The first example is the problem used in Figure 1 of the main text to illustrate the overall pipeline. While the figure shows only selected intermediate outputs and proof excerpts, the complete generated proof is provided below.

Natural Language Statement

Prove that the function $f(x) = x^2, \quad x \in (-1, 1)$ is uniformly continuous.

Lean Formalization

```
theorem tmp_thm :
  UniformContinuous (fun x : Set.Ioo (-1 : ℝ) 1 => (x.1 : ℝ)^2)
    := by sorry
```

PARTIAL SOUNDNESS Proof

Step 1. Every real number in the open interval $(-1, 1)$ has absolute value less than 1.
Let $x \in (-1, 1)$. Then $-1 < x < 1$. If $x \geq 0$, then $|x| = x < 1$. If $x < 0$, then $|x| = -x$, and from $-1 < x$ we get $-x < 1$. Hence in either case $|x| < 1$.

Step 2. For all real numbers x and y,

$$|x^2 - y^2| = |x - y|\,|x + y|.$$

For arbitrary real numbers x, y, the difference of squares factors as

$$x^2 - y^2 = (x - y)(x + y).$$

Taking absolute values and using $|ab| = |a|\,|b|$ gives

$$|x^2 - y^2| = |(x - y)(x + y)| = |x - y|\,|x + y|.$$

Step 3. For all $x, y \in (-1, 1)$, one has $|x + y| \leq 2$.
Let $x, y \in (-1, 1)$. By Step 1,

$$|x| < 1 \quad \text{and} \quad |y| < 1.$$

The triangle inequality gives

$$|x + y| \leq |x| + |y|.$$

Thus

$$|x + y| \leq |x| + |y| < 1 + 1 = 2.$$

In particular, $|x + y| \leq 2$.

Step 4. For every $\epsilon > 0$, there exists $\delta > 0$ such that whenever $x, y \in (-1, 1)$ and $|x - y| < \delta$, one has $|x^2 - y^2| < \epsilon$.
Let $\epsilon > 0$. Set

$$\delta = \frac{\epsilon}{2}.$$

Then $\delta > 0$. Now take $x, y \in (-1, 1)$ with $|x - y| < \delta$. By Step 2,

$$|x^2 - y^2| = |x - y|\,|x + y|.$$

By Step 3, $|x+y| \le 2$. Therefore

$$|x^2 - y^2| = |x-y|\,|x+y| \le 2|x-y|.$$

Since $|x-y| < \delta$, it follows that

$$|x^2 - y^2| < 2\delta = 2 \cdot \frac{\epsilon}{2} = \epsilon.$$

Thus the stated δ works for this ϵ.

Step 5. The function $f(x) = x^2$ on $(-1,1)$ is uniformly continuous.
By Step 4, for every $\epsilon > 0$, there exists $\delta > 0$ such that for all $x, y \in (-1,1)$, if $|x-y| < \delta$, then

$$|x^2 - y^2| < \epsilon.$$

Since $f(x) = x^2$ and $f(y) = y^2$, this is exactly

$$|f(x) - f(y)| < \epsilon.$$

Hence for every $\epsilon > 0$, there is a $\delta > 0$ such that all $x, y \in (-1,1)$ with $|x-y| < \delta$ satisfy $|f(x) - f(y)| < \epsilon$. Therefore, by the definition of uniform continuity, f is uniformly continuous on $(-1,1)$.

Example 2: Problem Used in Figure 2 of the Main Text.

Source: INFORMALRA.

The second example is the problem used in Figure 2 of the main text. Its natural language statement and Lean formalization appear below. The complete proof generated by PARTIAL SOUNDNESS is omitted here, as it appears in the *Reduced Reliance on External Theorems* subsection below.

Natural Language Statement

Let $g : \mathbb{R} \to \mathbb{R}$ be the period-1 extension of $x \mapsto |x|$ on $[-1/2, 1/2]$. Show that

$$f(x) = \sum_{n=1}^{\infty} \frac{g(4^{n-1}x)}{4^{n-1}}$$

is nowhere differentiable.

Lean Formalization

```
theorem tmp_thm
    (g f : ℝ → ℝ)
    (hg_abs : ∀ x : ℝ,
      x ∈ Set.Icc (-(1 : ℝ) / 2) ((1 : ℝ) / 2) → g x = |x|)
    (hg_periodic : Function.Periodic g 1)
    (hf_series :
      ∀ x : ℝ,
        Summable (fun n : ℕ =>
          g ((4 : ℝ) ^ n * x) / ((4 : ℝ) ^ n)) ∧
        f x = Σ' n : ℕ,
          g ((4 : ℝ) ^ n * x) / ((4 : ℝ) ^ n)) :
    ∀ x : ℝ, ¬ DifferentiableAt ℝ f x := by sorry
```

B COMPLETE EXAMPLES DISCUSSED IN THE MAIN TEXT

This section presents the complete problems and generated proofs for the examples discussed in the main text. Due to space constraints, only selected excerpts are shown in the main text, whereas the corresponding outputs are provided here in full.

B.1 Reduced Reliance on External Theorems

Figure 2 of the main text shows excerpts from the proofs by Partial Soundness and direct generation; the problem statement and both proofs are reproduced in full below. Our pipeline decomposes the problem into subgoals, making the main parts of the argument explicit and allowing them to be examined separately. Direct generation reaches the same conclusion by invoking a nowhere-differentiability result for the Takagi–van der Waerden family. By exposing the intermediate derivation, Partial Soundness makes the reasoning behind the conclusion directly observable.

Source: InformalRA.

Problem

Let $g : \mathbb{R} \to \mathbb{R}$ be the period-1 extension of $x \mapsto |x|$ on $[-1/2, 1/2]$. Show that

$$f(x) = \sum_{n=1}^{\infty} \frac{g(4^{n-1}x)}{4^{n-1}}$$

is nowhere differentiable.

Partial Soundness Proof

Step 1. The function g is the distance from a real number to the nearest integer.
Let $x \in \mathbb{R}$. Choose an integer n such that

$$-\frac{1}{2} \le x - n < \frac{1}{2}.$$

Since g is the period-1 extension of $t \mapsto |t|$ on $[-1/2, 1/2]$,

$$g(x) = g(x - n) = |x - n|.$$

For any integer m, if $m = n$ then $|x - m| = |x - n|$. If $m \neq n$, then $\ell = m - n$ is a nonzero integer, so $|\ell| \ge 1$, and the reverse triangle inequality gives

$$|x - m| = |(x - n) - \ell| \ge |\ell| - |x - n| \ge 1 - |x - n|.$$

Because $|x - n| \le 1/2$, we have $1 - |x - n| \ge |x - n|$. Thus $|x - m| \ge |x - n|$ for every $m \in \mathbb{Z}$. Therefore

$$g(x) = \min_{m \in \mathbb{Z}} |x - m|,$$

so $g(x)$ is precisely the distance from x to the nearest integer.

Step 2. The function g is continuous, bounded, and Lipschitz with Lipschitz constant 1.
By Step 1,

$$g(x) = \min_{m \in \mathbb{Z}} |x - m|$$

for every $x \in \mathbb{R}$. Choosing an integer n with $-1/2 \le x - n < 1/2$, we get

$$0 \le g(x) \le |x - n| \le \frac{1}{2}.$$

Hence g is bounded, with $0 \le g(x) \le 1/2$ for all x.
To prove the Lipschitz estimate, let $x, y \in \mathbb{R}$. Choose an integer k such that $g(y) = |y - k|$. Then

$$g(x) \le |x - k| \le |x - y| + |y - k| = |x - y| + g(y),$$

so $g(x) - g(y) \le |x - y|$. Interchanging x and y gives $g(y) - g(x) \le |x - y|$. Therefore

$$|g(x) - g(y)| \le |x - y|$$

for all $x, y \in \mathbb{R}$. Thus g is Lipschitz with constant 1, and consequently continuous.

Step 3. The series defining f converges uniformly on $\mathbb{R}$.
For $n \ge 1$, set

$$u_n(x) = \frac{g(4^{n-1}x)}{4^{n-1}}.$$

By Step 2, $0 \le g(t) \le 1/2$ for every $t \in \mathbb{R}$. Hence

$$|u_n(x)| \le \frac{1}{2 \cdot 4^{n-1}}$$

for every $x \in \mathbb{R}$. Since

$$\sum_{n=1}^{\infty} \frac{1}{2 \cdot 4^{n-1}} = \frac{1}{2} \sum_{n=1}^{\infty} 4^{-(n-1)} = \frac{1}{2} \cdot \frac{1}{1 - 1/4} = \frac{2}{3} < \infty,$$

the Weierstrass M-test shows that

$$\sum_{n=1}^{\infty} \frac{g(4^{n-1}x)}{4^{n-1}}$$

converges uniformly on $\mathbb{R}$.

Step 4. Every real number has a base-four expansion whose fractional digit sequence is not eventually an infinite string of 3's.
Let $x \in \mathbb{R}$. Choose an integer part and reduce to a fractional part $y \in [0, 1)$. Define recursively

$$r_0 = y, \qquad a_n = \lfloor 4r_{n-1} \rfloor, \qquad r_n = 4r_{n-1} - a_n \quad (n \ge 1).$$

Then each $a_n \in \{0, 1, 2, 3\}$ and $0 \le r_n < 1$. From

$$r_{n-1} = \frac{a_n}{4} + \frac{r_n}{4},$$

iteration gives, for every $N \ge 1$,

$$y = \sum_{n=1}^{N} \frac{a_n}{4^n} + \frac{r_N}{4^N}.$$

Since $0 \le r_N/4^N < 4^{-N} \to 0$, it follows that

$$y = \sum_{n=1}^{\infty} \frac{a_n}{4^n}.$$

The digit sequence cannot be eventually all 3's. Indeed, if $a_n = 3$ for every $n \ge N$, then applying the same expansion from r_{N-1} and letting the remainder tend to zero would give

$$r_{N-1} = \sum_{q=1}^{\infty} \frac{3}{4^q} = 1,$$

contradicting $0 \le r_{N-1} < 1$. Thus every real number admits a base-four expansion with fractional digits not eventually equal to 3.

Step 5. Tails of the defining series are unchanged under shifts by integral multiples of a reciprocal base-four scale.
Fix $x \in \mathbb{R}$, a positive integer m, and an integer p. For $n \ge m + 1$,

$$4^{n-1}\left(x + \frac{p}{4^m}\right) = 4^{n-1}x + p4^{n-1-m}.$$

Since $p4^{n-1-m}$ is an integer and g has period 1,

$$g\Big(4^{n-1}\left(x + \frac{p}{4^m}\right)\Big) = g(4^{n-1}x).$$

Therefore the corresponding summands in the two tails are equal:

$$\frac{g\big(4^{n-1}\left(x + \frac{p}{4^m}\right)\big)}{4^{n-1}} = \frac{g(4^{n-1}x)}{4^{n-1}}.$$

Since the series converges by Step 3, passing to the limit in the tail sums yields

$$\sum_{n=m+1}^{\infty} \frac{g\big(4^{n-1}\left(x + \frac{p}{4^m}\right)\big)}{4^{n-1}} = \sum_{n=m+1}^{\infty} \frac{g(4^{n-1}x)}{4^{n-1}}.$$

Step 6. Across a base-four mesh interval, the difference quotient is the number of low digits minus the number of high digits in the initial block.
Let

$$x = A + \sum_{j=1}^{\infty} \frac{a_j}{4^j}$$

be a base-four expansion as in Step 4, with $A \in \mathbb{Z}$, $a_j \in \{0, 1, 2, 3\}$, and the fractional digits not eventually all 3. For $m \geq 1$, define

$$K_m = 4^m A + \sum_{j=1}^{m} a_j 4^{m-j} \in \mathbb{Z}.$$

Then x lies in the mesh interval

$$\left[\frac{K_m}{4^m}, \frac{K_m + 1}{4^m}\right].$$

Set

$$Q_m = \frac{f\left(\frac{K_m+1}{4^m}\right) - f\left(\frac{K_m}{4^m}\right)}{4^{-m}}.$$

By Step 5, the terms of the defining series with index $n \geq m + 1$ have equal values at the two endpoints of this interval, so they cancel in the difference. Hence

$$Q_m = 4^m \sum_{n=1}^{m} \frac{g\left(4^{n-1}\frac{K_m+1}{4^m}\right) - g\left(4^{n-1}\frac{K_m}{4^m}\right)}{4^{n-1}}.$$

Writing $r = m - n + 1$, the n-th contribution is

$$4^r \left[g\left(\frac{K_m + 1}{4^r}\right) - g\left(\frac{K_m}{4^r}\right)\right].$$

Reducing K_m modulo 4^r, we have

$$K_m \equiv q_{n,m} := \sum_{j=n}^{m} a_j 4^{m-j} \pmod{4^r}, \qquad 0 \leq q_{n,m} < 4^r.$$

By period 1 of g,

$$g\left(\frac{K_m}{4^r}\right) = g\left(\frac{q_{n,m}}{4^r}\right), \qquad g\left(\frac{K_m + 1}{4^r}\right) = g\left(\frac{q_{n,m} + 1}{4^r}\right).$$

The leading base-four digit of $q_{n,m}$ is a_n. If $a_n \in \{0, 1\}$, then both relevant points lie in $[0, 1/2]$, where $g(t) = t$, and the contribution is $+1$. If $a_n \in \{2, 3\}$, then both lie in $[1/2, 1]$, where $g(t) = 1 - t$, and the contribution is -1.
Thus, if

$$L_m = \#\{1 \leq j \leq m : a_j \in \{0, 1\}\}, \qquad H_m = \#\{1 \leq j \leq m : a_j \in \{2, 3\}\},$$

then

$$Q_m = L_m - H_m.$$

Step 7. If the base-four digit imbalance is unbounded at x, then f has unbounded difference quotients along increments tending to zero.
Assume that for a chosen base-four expansion

$$x = A + \sum_{j=1}^{\infty} \frac{a_j}{4^j}$$

as above, the quantities

$$D_m = L_m - H_m$$

are unbounded in absolute value. Then there exists a sequence $m_k \to \infty$ such that $|D_{m_k}| \to \infty$.

For each m, let

$$\alpha_m = \frac{K_m}{4^m}, \qquad \beta_m = \frac{K_m + 1}{4^m}.$$

By Step 6,

$$\frac{f(\beta_m) - f(\alpha_m)}{\beta_m - \alpha_m} = D_m.$$

If x is one endpoint of $[\alpha_m, \beta_m]$, taking the increment from x to the other endpoint gives a difference quotient equal to D_m. If $\alpha_m < x < \beta_m$, define

$$q_m^+ = \frac{f(\beta_m) - f(x)}{\beta_m - x}, \qquad q_m^- = \frac{f(\alpha_m) - f(x)}{\alpha_m - x}.$$

Then the interval quotient is a convex combination of q_m^+ and q_m^-, so at least one of these two quotients has absolute value at least $|D_m|$. Therefore, for each m, there is a nonzero increment h_m with

$$|h_m| \le 4^{-m}$$

and

$$\left| \frac{f(x + h_m) - f(x)}{h_m} \right| \ge |D_m|.$$

Along the subsequence m_k, we have $h_{m_k} \to 0$, while the corresponding difference quotients have absolute value tending to infinity. Hence the difference quotients of f at x are unbounded along nonzero increments tending to zero.

Step 8. If the base-four digit imbalance is bounded at x, then f still cannot have a finite derivative at x.

Assume now that the imbalance sequence $D_m = L_m - H_m$ is bounded. With the notation from Step 6, put

$$\alpha_m = \frac{K_m}{4^m}, \qquad \beta_m = \frac{K_m + 1}{4^m},$$

and write

$$\theta_m = 4^m(x - \alpha_m) = \sum_{j=m+1}^{\infty} \frac{a_j}{4^{j-m}}.$$

Since the chosen expansion is not eventually all 3's, $\theta_m < 1$. Since the imbalance is bounded, $\theta_m \neq 0$; otherwise all later digits would be 0, forcing D_m to grow without bound. Thus

$$0 < \theta_m < 1,$$

so x lies strictly between α_m and β_m.

Let

$$S_m(t) = \sum_{n=1}^{m} \frac{g(4^{n-1}t)}{4^{n-1}}.$$

The digit analysis from Step 6 shows that on $[\alpha_m, \beta_m]$, the first m summands are affine with total slope D_m. Hence, for $u \in [0, 1]$,

$$S_m(\alpha_m + u4^{-m}) - S_m(\alpha_m) = D_m\, u4^{-m}.$$

For the tail,

$$T_m(t) = \sum_{n=m+1}^{\infty} \frac{g(4^{n-1}t)}{4^{n-1}},$$

if $t = \alpha_m + u4^{-m}$, then using period 1 of g,

$$T_m(\alpha_m + u4^{-m}) = 4^{-m} \sum_{k=1}^{\infty} \frac{g(4^{k-1}u)}{4^{k-1}} = 4^{-m} f(u).$$

Also $f(0) = f(1) = 0$, because every term in the defining series vanishes at 0 and at 1. Therefore

$$f(\alpha_m + u4^{-m}) - f(\alpha_m) = D_m\, u4^{-m} + 4^{-m} f(u).$$

Taking $u = \theta_m$, we obtain

$$\frac{f(\alpha_m) - f(x)}{\alpha_m - x} = D_m + \frac{f(\theta_m)}{\theta_m}.$$

Taking $u = 1$, and comparing β_m with x, gives

$$\frac{f(\beta_m) - f(x)}{\beta_m - x} = D_m - \frac{f(\theta_m)}{1 - \theta_m}.$$

Thus for the nonzero increments

$$h_m^- = \alpha_m - x = -\theta_m 4^{-m}, \qquad h_m^+ = \beta_m - x = (1 - \theta_m)4^{-m},$$

both tending to 0, the corresponding difference quotients differ by

$$\frac{f(\theta_m)}{\theta_m(1 - \theta_m)}.$$

Since $0 < \theta_m < 1$, Step 1 gives

$$g(\theta_m) = \min\{\theta_m, 1 - \theta_m\}.$$

All terms in the defining series for f are nonnegative, so

$$f(\theta_m) \geq g(\theta_m) = \min\{\theta_m, 1 - \theta_m\}.$$

Consequently,

$$\frac{f(\theta_m)}{\theta_m(1 - \theta_m)} \geq \frac{\min\{\theta_m, 1 - \theta_m\}}{\theta_m(1 - \theta_m)} \geq 1.$$

Thus the two difference quotient sequences along $h_m^- \to 0$ and $h_m^+ \to 0$ differ by at least 1 for every m. They therefore cannot both converge to the same finite limit, which is incompatible with the existence of a finite derivative of f at x.

Step 9. The function f is nowhere differentiable.
Let $x \in \mathbb{R}$ be arbitrary. By Step 4, choose a base-four expansion of x whose fractional digit sequence is not eventually an infinite string of 3's. Form the imbalance sequence

$$D_m = L_m - H_m,$$

where L_m counts the digits among $a_1, \ldots, a_m$ equal to 0 or 1, and H_m counts the digits equal to 2 or 3.
If (D_m) is unbounded, then Step 7 gives nonzero increments tending to 0 along which the difference quotients of f at x are unbounded. A finite derivative would force all such difference quotients to tend to the same finite limit, so this is impossible.
If (D_m) is bounded, then Step 8 shows directly that there are two sequences of nonzero increments tending to 0 whose difference quotients cannot converge to the same finite limit. Hence f is not differentiable at x in this case as well.
Since $x \in \mathbb{R}$ was arbitrary, f is nowhere differentiable.

Direct Generation Proof

For $x \in \mathbb{R}$, the period-1 extension of $t \mapsto |t|$ on $[-1/2, 1/2]$ is exactly

$$g(x) = \operatorname{dist}(x, \mathbb{Z}),$$

the distance from x to the nearest integer. Indeed, every real x is congruent modulo $\mathbb{Z}$ to a unique point of $[-1/2, 1/2]$ up to the endpoints, and on that interval the distance to $\mathbb{Z}$ is $|x|$.
Thus, after reindexing the given series by $k = n - 1$, we have

$$f(x) = \sum_{k=0}^{\infty} \frac{g(4^k x)}{4^k} = \sum_{k=0}^{\infty} 4^{-k} \operatorname{dist}(4^k x, \mathbb{Z}).$$

This is the classical Takagi–van der Waerden function with base 4.

We now invoke the standard Takagi–van der Waerden nowhere differentiability theorem: for every integer $b \geq 2$, the function

$$W_b(x) = \sum_{k=0}^{\infty} b^{-k} \operatorname{dist}(b^k x, \mathbb{Z})$$

is nowhere differentiable on $\mathbb{R}$. The theorem applies here with $b = 4$. Therefore the function

$$f(x) = \sum_{k=0}^{\infty} 4^{-k} \operatorname{dist}(4^k x, \mathbb{Z})$$

is nowhere differentiable on $\mathbb{R}$. Hence the given function f is nowhere differentiable.

B.2 Stepwise Proof

The following presents the complete version of the example excerpted in Figure 3 of the main text. Partial Soundness separates the proof into explicit subgoals and corresponding subproofs, allowing the overall structure to be understood from the subgoals before examining the proof details and making errors easier to localize to individual steps. By contrast, direct generation produces a monolithic proof that must be read as a whole to recover its logical flow, and an identified error must be checked for its possible effects on subsequent parts of the argument.

Source: Putnam, `putnam_1980_b5`.

Problem

A function f is convex on $[0, 1]$ if and only if

$$f(su + (1 - s)v) \leq sf(u) + (1 - s)f(v)$$

for all $u, v \in [0, 1]$ and $s \in [0, 1]$.
Let S_t be the set of nonnegative increasing convex continuous functions $f\colon [0, 1] \to \mathbb{R}$ satisfying

$$f(1) - 2f(2/3) + f(1/3) \geq t\big(f(2/3) - 2f(1/3) + f(0)\big).$$

For which real numbers $t \geq 0$ is S_t closed under multiplication?

Partial Soundness Proof

We prove that, for real numbers $t \geq 0$, the set S_t is closed under pointwise multiplication exactly for $0 \leq t \leq 1$.

Step 1. If $0 \leq t \leq 1$, then the pointwise product of any two nonnegative increasing continuous functions on $[0, 1]$ is nonnegative, increasing, and continuous on $[0, 1]$.
Let $f, g : [0, 1] \to \mathbb{R}$ be nonnegative increasing continuous functions, and define $h = fg$ pointwise. Since $f(x) \geq 0$ and $g(x) \geq 0$ for every $x \in [0, 1]$, we have $h(x) = f(x)g(x) \geq 0$, so h is nonnegative. If $x \leq y$, then $f(x) \leq f(y)$ and $g(x) \leq g(y)$; using nonnegativity gives

$$f(x)g(x) \leq f(y)g(x) \leq f(y)g(y),$$

so h is increasing. Finally, since f and g are continuous, their product h is continuous on $[0, 1]$.

Step 2. The pointwise product of any two nonnegative increasing convex functions on $[0, 1]$ is convex on $[0, 1]$.
Let $f, g : [0, 1] \to \mathbb{R}$ be nonnegative increasing convex functions, and set $h = fg$. For $u, v \in [0, 1]$ and $s \in [0, 1]$, put $x = su + (1 - s)v$. By convexity,

$$f(x) \leq sf(u) + (1 - s)f(v), \qquad g(x) \leq sg(u) + (1 - s)g(v).$$

All quantities are nonnegative, so multiplying gives

$$h(x) \leq \big(sf(u) + (1 - s)f(v)\big)\big(sg(u) + (1 - s)g(v)\big).$$

Expanding the right-hand side yields

$$s^2 f(u)g(u) + s(1 - s)\big(f(u)g(v) + f(v)g(u)\big) + (1 - s)^2 f(v)g(v).$$

Since f and g are increasing, $f(u) - f(v)$ and $g(u) - g(v)$ have the same sign, hence

$$(f(u) - f(v))(g(u) - g(v)) \geq 0.$$

Equivalently,

$$f(u)g(v) + f(v)g(u) \leq f(u)g(u) + f(v)g(v).$$

Therefore

$$\big(sf(u) + (1-s)f(v)\big)\big(sg(u) + (1-s)g(v)\big) \leq sf(u)g(u) + (1-s)f(v)g(v).$$

Thus

$$h(su + (1-s)v) \leq sh(u) + (1-s)h(v),$$

so h is convex on $[0, 1]$.

Step 3. If $0 \leq t \leq 1$ and $f, g \in S_t$, then their pointwise product satisfies the defining value inequality for S_t.
Let $h = fg$. Write

$$a_i = f\left(\frac{i}{3}\right), \qquad b_i = g\left(\frac{i}{3}\right) \qquad (i = 0, 1, 2, 3),$$

and define increments

$$p_i = a_i - a_{i-1}, \qquad q_i = b_i - b_{i-1} \qquad (i = 1, 2, 3).$$

Because f and g are increasing, all p_i, q_i are nonnegative. Convexity at the points $1/3$ and $2/3$ gives

$$p_1 \leq p_2 \leq p_3, \qquad q_1 \leq q_2 \leq q_3.$$

Since $f, g \in S_t$, their defining inequalities become

$$p_3 - p_2 \geq t(p_2 - p_1), \qquad q_3 - q_2 \geq t(q_2 - q_1).$$

Now set

$$H_i = h\left(\frac{i}{3}\right) = a_i b_i, \qquad r_i = H_i - H_{i-1} \qquad (i = 1, 2, 3).$$

The desired inequality for h is equivalent to

$$r_3 - r_2 \geq t(r_2 - r_1).$$

For each i,

$$r_i = a_i b_i - a_{i-1} b_{i-1} = a_{i-1} q_i + b_{i-1} p_i + p_i q_i.$$

Hence

$$r_3 - r_2 = a_1(q_3 - q_2) + p_2 q_3 + b_1(p_3 - p_2) + q_2 p_3 + p_3 q_3 - p_2 q_2,$$

and

$$r_2 - r_1 = a_0(q_2 - q_1) + p_1 q_2 + b_0(p_2 - p_1) + q_1 p_2 + p_2 q_2 - p_1 q_1.$$

Thus

$$\begin{aligned} &r_3 - r_2 - t(r_2 - r_1) \\ &= \big(a_1(q_3 - q_2) - t a_0(q_2 - q_1)\big) + \big(b_1(p_3 - p_2) - t b_0(p_2 - p_1)\big) \\ &\quad + \Big(p_2 q_3 + q_2 p_3 + p_3 q_3 - p_2 q_2 - t(p_1 q_2 + q_1 p_2 + p_2 q_2 - p_1 q_1)\Big). \end{aligned}$$

The first term is nonnegative because

$$a_1(q_3 - q_2) - t a_0(q_2 - q_1) = a_1\big((q_3 - q_2) - t(q_2 - q_1)\big) + t(a_1 - a_0)(q_2 - q_1) \geq 0.$$

Similarly,

$$b_1(p_3 - p_2) - t b_0(p_2 - p_1) \geq 0.$$

For the remaining bracket, define

$$M = p_2 q_3 + q_2 p_3 + p_3 q_3 - p_2 q_2,$$

and

$$N = p_1 q_2 + q_1 p_2 + p_2 q_2 - p_1 q_1.$$

Since $p_3 \geq p_2$ and $q_3 \geq q_2$,

$$M \geq 2p_2q_2.$$

Also,

$$N = p_1(q_2 - q_1) + p_2q_1 + p_2q_2 \leq p_2(q_2 - q_1) + p_2q_1 + p_2q_2 = 2p_2q_2.$$

Therefore $M \geq N$. Since $N \geq 0$ and $0 \leq t \leq 1$,

$$M - tN \geq M - N \geq 0.$$

All terms in $r_3 - r_2 - t(r_2 - r_1)$ are nonnegative, so

$$r_3 - r_2 \geq t(r_2 - r_1).$$

Equivalently,

$$h(1) - 2h\left(\frac{2}{3}\right) + h\left(\frac{1}{3}\right) \geq t\left(h\left(\frac{2}{3}\right) - 2h\left(\frac{1}{3}\right) + h(0)\right).$$

Step 4. If $0 \leq t \leq 1$, then S_t is closed under pointwise multiplication.
Take $f, g \in S_t$, and let $h = fg$. By the definition of S_t, both f and g are nonnegative, increasing, convex, and continuous on $[0, 1]$, and both satisfy the defining value inequality. By Step 1, h is nonnegative, increasing, and continuous. By Step 2, h is convex. By Step 3, h also satisfies the defining value inequality for membership in S_t. Hence $h \in S_t$. Since f and g were arbitrary, S_t is closed under pointwise multiplication.

Step 5. If $t > 1$ and $m > 0$, then the continuous piecewise affine function with values

$$f(0) = 0, \qquad f\left(\frac{1}{3}\right) = m, \qquad f\left(\frac{2}{3}\right) = 2m + 1, \qquad f(1) = 3m + 2 + t$$

belongs to S_t.
On the three subintervals $[0, 1/3]$, $[1/3, 2/3]$, and $[2/3, 1]$, the slopes of f are respectively

$$3m, \qquad 3(m + 1), \qquad 3(m + 1 + t).$$

Because $m > 0$ and $t > 1$, these slopes are positive and increasing. Thus f is increasing on $[0, 1]$, and since its endpoint values at the partition points are nonnegative, f is nonnegative on $[0, 1]$. It is continuous by construction, and because its successive slopes are nondecreasing, it is convex.
It remains to check the defining inequality. From the prescribed values,

$$f(1) - 2f\left(\frac{2}{3}\right) + f\left(\frac{1}{3}\right) = (3m + 2 + t) - 2(2m + 1) + m = t,$$

while

$$f\left(\frac{2}{3}\right) - 2f\left(\frac{1}{3}\right) + f(0) = (2m + 1) - 2m + 0 = 1.$$

Therefore

$$f(1) - 2f\left(\frac{2}{3}\right) + f\left(\frac{1}{3}\right) = t\left(f\left(\frac{2}{3}\right) - 2f\left(\frac{1}{3}\right) + f(0)\right),$$

so $f \in S_t$.

Step 6. If $t > 1$, then for some $m > 0$, the square of the function from Step 5 fails the defining value inequality for S_t.
Fix $t > 1$, and for $m > 0$ let f be the continuous piecewise affine function with the values listed in Step 5. Put $h = f^2$. Then

$$h(0) = 0, \qquad h\left(\frac{1}{3}\right) = m^2, \qquad h\left(\frac{2}{3}\right) = (2m + 1)^2, \qquad h(1) = (3m + 2 + t)^2.$$

Thus

$$\begin{aligned} &h(1) - 2h\left(\frac{2}{3}\right) + h\left(\frac{1}{3}\right) - t\left(h\left(\frac{2}{3}\right) - 2h\left(\frac{1}{3}\right) + h(0)\right) \\ &= (3m + 2 + t)^2 - (2 + t)(2m + 1)^2 + (1 + 2t)m^2 \\ &= -2(t - 1)m^2 + 2(t + 2)m + (t + 2)(t + 1). \end{aligned}$$

Since $t > 1$, the coefficient of m^2 is negative, so this quadratic tends to $-\infty$ as $m \to +\infty$. Therefore there exists $m > 0$ for which it is negative. For that m,

$$h(1) - 2h\left(\frac{2}{3}\right) + h\left(\frac{1}{3}\right) < t\left(h\left(\frac{2}{3}\right) - 2h\left(\frac{1}{3}\right) + h(0)\right),$$

so $h = f^2$ does not satisfy the defining value inequality for S_t.

Step 7. If $t > 1$, then there exists $f \in S_t$ such that $f^2 \notin S_t$.
Choose $m > 0$ as in Step 6, and let f be the corresponding continuous piecewise affine function. By Step 5, this function f belongs to S_t. By Step 6, its pointwise square f^2 fails the defining value inequality required for membership in S_t. Hence $f^2 \notin S_t$.

Step 8. For every real number $t \geq 0$, the set S_t is closed under pointwise multiplication if and only if $t \leq 1$.
Let $t \geq 0$. If $t \leq 1$, then Step 4 shows that S_t is closed under pointwise multiplication.
Conversely, suppose S_t is closed under pointwise multiplication. If $t > 1$, then by Step 7 there is some $f \in S_t$ with $f^2 \notin S_t$. But f^2 is the pointwise product of f with itself, so closure under pointwise multiplication would imply $f^2 \in S_t$, a contradiction. Therefore $t > 1$ is impossible, and hence $t \leq 1$.
Thus S_t is closed under multiplication precisely for the real numbers $t \geq 0$ satisfying $t \leq 1$, namely $0 \leq t \leq 1$.

Direct Generation Proof

The set S_t is closed under multiplication exactly for $0 \leq t \leq 1$.
First suppose $0 \leq t \leq 1$, and let $f, g \in S_t$. The product fg is continuous and nonnegative. It is also increasing, since f and g are nonnegative increasing. Moreover, fg is convex: for $u, v \in [0,1]$ and $s \in [0,1]$, convexity gives

$$f(su + (1-s)v) \leq sf(u) + (1-s)f(v),$$
$$g(su + (1-s)v) \leq sg(u) + (1-s)g(v).$$

Since all values are nonnegative,

$$(fg)(su + (1-s)v) \leq \big(sf(u) + (1-s)f(v)\big)\big(sg(u) + (1-s)g(v)\big).$$

Expanding the right-hand side and using that f and g are increasing, so

$$(f(u) - f(v))(g(u) - g(v)) \geq 0,$$

we get

$$\big(sf(u) + (1-s)f(v)\big)\big(sg(u) + (1-s)g(v)\big) \leq sf(u)g(u) + (1-s)f(v)g(v).$$

Thus fg is convex.
It remains to verify the defining inequality for fg. Write

$$a = f(0), \qquad x = f(1/3) - f(0),$$
$$\alpha = f(2/3) - 2f(1/3) + f(0),$$
$$\beta = f(1) - 2f(2/3) + f(1/3).$$

Similarly, write

$$b = g(0), \qquad X = g(1/3) - g(0),$$
$$A = g(2/3) - 2g(1/3) + g(0),$$
$$B = g(1) - 2g(2/3) + g(1/3).$$

Because f and g are increasing and convex, we have

$$a, b, x, X, \alpha, \beta, A, B \geq 0.$$

Since $f, g \in S_t$, their defining inequalities give

$$\beta \geq t\alpha, \qquad B \geq tA.$$

The values of f at the four points are

$$f(0) = a, \quad f(1/3) = a + x, \quad f(2/3) = a + x + \alpha, \quad f(1) = a + x + \alpha + \beta,$$

and similarly

$$g(0) = b, \quad g(1/3) = b + X, \quad g(2/3) = b + X + A, \quad g(1) = b + X + A + B.$$

A direct calculation gives

$$\begin{aligned}&(fg)(2/3) - 2(fg)(1/3) + (fg)(0)\\ &= aA + b\alpha + 2xX + 2xA + 2\alpha X + \alpha A,\end{aligned}$$

and

$$\begin{aligned}&(fg)(1) - 2(fg)(2/3) + (fg)(1/3)\\ &= (a + x)B + (b + X)\beta + 2(x + \alpha)(X + A)\\ &\quad + 2(x + \alpha)B + 2\beta(X + A) + \beta B.\end{aligned}$$

Therefore

$$\begin{aligned}&\big((fg)(1) - 2(fg)(2/3) + (fg)(1/3)\big)\\ &\quad - t\big((fg)(2/3) - 2(fg)(1/3) + (fg)(0)\big)\end{aligned}$$

is increasing as a function of β and B, since all coefficients of β and B are nonnegative. Hence its minimum under the constraints $\beta \geq t\alpha$ and $B \geq tA$ occurs when $\beta = t\alpha$ and $B = tA$. Substituting those values yields

$$2(1 - t)xX + (2 + t)xA + (2 + t)\alpha X + (2 + 3t + t^2)\alpha A.$$

All terms are nonnegative because $0 \leq t \leq 1$ and $x, X, \alpha, A \geq 0$. Thus

$$(fg)(1) - 2(fg)(2/3) + (fg)(1/3) \geq t\big((fg)(2/3) - 2(fg)(1/3) + (fg)(0)\big).$$

So $fg \in S_t$, and S_t is closed under multiplication.
Conversely, suppose $t > 1$. Let $f(x) = x$. Then f is nonnegative, increasing, convex, and continuous on $[0, 1]$, and

$$f(1) - 2f(2/3) + f(1/3) = 0$$

and

$$f(2/3) - 2f(1/3) + f(0) = 0.$$

Hence $f \in S_t$. But $f^2(x) = x^2$ satisfies

$$f^2(1) - 2f^2(2/3) + f^2(1/3) = 1 - \frac{8}{9} + \frac{1}{9} = \frac{2}{9},$$

while

$$f^2(2/3) - 2f^2(1/3) + f^2(0) = \frac{4}{9} - \frac{2}{9} = \frac{2}{9}.$$

The defining inequality for S_t would require

$$\frac{2}{9} \geq t\frac{2}{9},$$

which fails when $t > 1$. Therefore S_t is not closed under multiplication for $t > 1$.
Hence S_t is closed under multiplication precisely for the real numbers t with $0 \leq t \leq 1$.

B.3 Proof Errors Motivating Lean Verification

This subsection presents the examples referenced in the main text discussion of the effect of Lean verification. We provide three direct generation proofs whose overall arguments appear plausible but contain errors involving a recurrence relation, a boundary case, and an algebraic derivation, respectively. The errors were identified through manual inspection, and the corresponding passages are highlighted in red and accompanied by brief explanations. These examples illustrate the difficulty of locating local defects in a monolithic natural language proof.

Example 1: Error in a Recurrence Relation.

Source: Putnam, `putnam_1978_b3`.

The proof uses an incorrect unified recurrence: the coefficient should be $\lceil n/2 \rceil$, not $\lfloor n/2 \rfloor$. The stated recurrence therefore fails for every odd n. This incorrect coefficient is carried into later recurrence identities.

Problem

The polynomials $P_n(x)$ are defined by

$$P_1(x) = 1 + x, P_2(x) = 1 + 2x,$$
$$P_{2n+1}(x) = P_{2n}(x) + (n+1)xP_{2n-1}(x),$$
$$P_{2n+2}(x) = P_{2n+1}(x) + (n+1)xP_{2n}(x).$$

Let a_n be the largest real root of $P_n(x)$. Prove that a_n is strictly monotonically increasing and tends to zero.

Direct Generation Proof

Define, for convenience, $P_0(x) = 1$. Then the given initial conditions and recurrences may be written as

$$P_n(x) = P_{n-1}(x) + \left\lfloor \frac{n}{2} \right\rfloor xP_{n-2}(x) \qquad (n \geq 2).$$

Also $P_n(0) = 1$ for every n. Since the recurrence only adds polynomials with positive coefficients multiplied by positive constants, every P_n has positive coefficients. In particular its leading coefficient is positive, and one checks inductively that

$$\deg P_n = \left\lceil \frac{n}{2} \right\rceil.$$

We prove by induction that every P_n has only simple negative real roots and that the roots of consecutive polynomials strictly interlace in such a way that the largest root increases.
For $n = 1, 2$,

$$P_1(x) = 1 + x, \qquad P_2(x) = 1 + 2x,$$

so their roots are -1 and $-\frac{1}{2}$, respectively. Thus the assertion holds at the start.
Assume now that the asserted interlacing has been proved up to P_{n-1}, with $n \geq 3$. Let

$$\alpha_1 < \alpha_2 < \cdots < \alpha_m < 0$$

be the roots of P_{n-1}, where $m = \deg P_{n-1}$. By the induction hypothesis, P_{n-2} has exactly one root in each interval (α_i, α_{i+1}) for $1 \leq i < m$, and no root in $(\alpha_m, 0)$. Since $P_{n-2}(0) = 1$, it follows that $P_{n-2}(\alpha_m) > 0$, and the signs of $P_{n-2}(\alpha_i)$ alternate as i increases.
At each root α_i of P_{n-1}, the recurrence gives

$$P_n(\alpha_i) = \left\lfloor \frac{n}{2} \right\rfloor \alpha_i P_{n-2}(\alpha_i).$$

Because $\alpha_i < 0$, the signs of $P_n(\alpha_i)$ also alternate. Hence P_n has at least one root in every interval (α_i, α_{i+1}). Moreover,

$$P_n(\alpha_m) = \left\lfloor \frac{n}{2} \right\rfloor \alpha_m P_{n-2}(\alpha_m) < 0,$$

while $P_n(0) = 1 > 0$, so P_n has a root in $(\alpha_m, 0)$.
If $\deg P_n = \deg P_{n-1} = m$, these m roots already account for the full degree of P_n, so they are all the roots, they are simple, and in particular the largest root of P_n lies in $(\alpha_m, 0)$, strictly to the right of the largest root of P_{n-1}.
If instead $\deg P_n = m + 1$, then n is odd. In that case $\deg P_{n-2} = \deg P_{n-1} = m$, and the induction hypothesis gives the ordering

$$\beta_1 < \alpha_1 < \beta_2 < \alpha_2 < \cdots < \beta_m < \alpha_m,$$

where $\beta_1, \ldots, \beta_m$ are the roots of P_{n-2}. Thus P_{n-2} has exactly $m-1$ roots in $(\alpha_1, 0)$, so, since $P_{n-2}(0) = 1$,

$$\operatorname{sgn} P_{n-2}(\alpha_1) = (-1)^{m-1}.$$

Therefore

$$\operatorname{sgn} P_n(\alpha_1) = (-1)^m.$$

On the other hand, P_n has positive leading coefficient and degree $m+1$, so for sufficiently negative x,

$$\operatorname{sgn} P_n(x) = (-1)^{m+1}.$$

Hence P_n has a root in $(-\infty, \alpha_1)$. Together with the roots already found in the intervals (α_i, α_{i+1}) and $(\alpha_m, 0)$, this gives $m+1 = \deg P_n$ real roots, all simple. Again the largest one lies in $(\alpha_m, 0)$.

Thus, by induction, every P_n has only simple negative real roots, and if a_n denotes the largest real root of P_n, then

$$a_{n-1} < a_n < 0$$

for every $n \geq 2$. Hence (a_n) is strictly monotonically increasing and bounded above by 0, so it has a limit $\ell \leq 0$.

It remains to show that $\ell = 0$. Fix $t > 0$ and set

$$q_n = P_n(-t).$$

Suppose, for contradiction, that $q_n > 0$ for every $n \geq 1$. Then from

$$q_n = q_{n-1} - \left\lfloor \frac{n}{2} \right\rfloor t\, q_{n-2} \qquad (n \geq 2),$$

we get $q_n < q_{n-1}$ for every $n \geq 2$. Hence $q_n/q_{n-1} < 1$. But positivity of q_{n+1} gives

$$q_n > \left\lfloor \frac{n+1}{2} \right\rfloor t\, q_{n-1},$$

so

$$\left\lfloor \frac{n+1}{2} \right\rfloor t < \frac{q_n}{q_{n-1}} < 1$$

for every n. This is impossible because $\left\lfloor \frac{n+1}{2} \right\rfloor t \to \infty$.

Therefore, for every $t > 0$, there exists some m such that $P_m(-t) \leq 0$. Since $P_m(0) = 1 > 0$, the intermediate value theorem gives a real root of P_m in $[-t, 0)$. Hence $a_m \geq -t$. Because the sequence (a_n) is increasing, its limit satisfies $\ell \geq -t$. As this holds for every $t > 0$, we get $\ell \geq 0$. Since already $\ell \leq 0$, it follows that $\ell = 0$.

Thus a_n is strictly monotonically increasing and tends to zero.

Example 2: Error at a Boundary Value.

Source: INFORMALRA.

The condition $\varepsilon \leq 2$ is incorrect: when $\varepsilon = 2$, the proposed point $1 + \varepsilon/2$ equals 2 and is not in $(1, 2)$. Because the next case treats only $\varepsilon > 2$, the boundary case $\varepsilon = 2$ is omitted.

Problem

Show that the interval $(1, 2)$ is not closed in $\mathbb{R}$.

Direct Generation Proof

In $\mathbb{R}$ with its usual topology, a set is closed if and only if its complement is open. We show that the complement of $(1, 2)$ is not open.

The complement of $(1, 2)$ in $\mathbb{R}$ is

$$\mathbb{R} \setminus (1, 2) = (-\infty, 1] \cup [2, \infty).$$

The point 1 belongs to this complement. If the complement were open, then there would exist some $\varepsilon > 0$ such that the open interval

$$(1 - \varepsilon, 1 + \varepsilon)$$

is contained in $\mathbb{R} \setminus (1, 2)$. But for every $\varepsilon > 0$, the number

$$1 + \frac{\varepsilon}{2}$$

lies in $(1 - \varepsilon, 1 + \varepsilon)$, and it also satisfies

$$1 < 1 + \frac{\varepsilon}{2} < 2$$

provided $\varepsilon \leq 2$. If $\varepsilon > 2$, then $3/2 \in (1 - \varepsilon, 1 + \varepsilon) \cap (1, 2)$. Thus every open interval around 1 contains a point of $(1, 2)$, so no neighborhood of 1 is contained in $\mathbb{R} \setminus (1, 2)$.
Therefore $\mathbb{R} \setminus (1, 2)$ is not open. Hence $(1, 2)$ is not closed in $\mathbb{R}$.

Example 3: Invalid Algebraic Derivation.

Source: Putnam, `putnam_1980_b5`.

This example was already presented in full in the Stepwise Proof subsection above. To avoid repetition, we reproduce only the algebraic passage relevant to the present analysis; the complete Direct Generation Proof is provided there.

In that proof, direct generation reconstructs the sampled values as

$$f(2/3) = a + x + \alpha, \quad f(1) = a + x + \alpha + \beta,$$

and analogously as

$$g(2/3) = b + X + A, \quad g(1) = b + X + A + B.$$

These identities do not follow from the definitions of the first and second differences. In fact, from

$$x = f(1/3) - f(0), \qquad \alpha = f(2/3) - 2f(1/3) + f(0),$$

and

$$\beta = f(1) - 2f(2/3) + f(1/3),$$

the correct identities are

$$f(2/3) = a + 2x + \alpha, \qquad f(1) = a + 3x + 2\alpha + \beta.$$

Analogously,

$$g(2/3) = b + 2X + A, \qquad g(1) = b + 3X + 2A + B.$$

The subsequent second-difference expansions for fg are consistent with these corrected identities, rather than with the identities displayed by direct generation. The proof is therefore internally inconsistent as written, although the error can be repaired locally by replacing the four incorrect formulas.

B.4 Controlled Example of Lean Verification

We present the complete version of the controlled example excerpted in Figure 4, which shows how Lean verification can serve as a conservative certification gate by making the failure of an intermediate proof step explicit.

Controlled Example Used in Figure 4 of the Main Text.

Source: Adapted from INFORMALRA.

Starting from the universally valid inclusion

$$f(A_1 \cap A_2) \subseteq f(A_1) \cap f(A_2),$$

we consider its converse

$$f(A_1) \cap f(A_2) \subseteq f(A_1 \cap A_2).$$

Although this converse holds when f is injective, the controlled statement leaves f arbitrary. Equivalently, the example can be viewed as the valid converse formulation with its necessary injectivity assumption deliberately removed.

We provide the complete modified problem, the Lean proof skeleton generated by PARTIAL SOUNDNESS, and the corresponding type-checking result. As discussed in the main text, the verification failure indicates non-certification of the generated skeleton rather than a Lean proof that the statement itself is false.

The type-checking result transcription preserves the content of Lean's tactic state and diagnostic output, with line breaks adjusted for readability.

Problem

For any function f mapping from a set X to a set Y, and any two subsets A_1 and A_2 of the domain X, show that $f(A_1) \cap f(A_2) \subset f(A_1 \cap A_2)$.

Complete Lean Proof Skeleton

```
import Mathlib

namespace Test

theorem tmp_thm {X Y : Type*} (f : X → Y) (A₁ A₂ : Set X) :
    (f '' A₁) ∩ (f '' A₂) ⊆ f '' (A₁ ∩ A₂) := by

  have h1 : ∃ (X₀ Y₀ : Type) (g : X₀ → Y₀) (B₁ B₂ : Set X₀),
    (∃ x₁ x₂ : X₀, x₁ ≠ x₂) ∧
    (∃ y₀ : Y₀, ∀ y : Y₀, y = y₀) ∧
    Disjoint B₁ B₂ ∧
    g '' B₁ = (Set.univ : Set Y₀) ∧
    g '' B₂ = (Set.univ : Set Y₀) := by sorry

  have h2 : ∃ (X₀ Y₀ : Type) (g : X₀ → Y₀) (B₁ B₂ : Set X₀),
    ((g '' B₁) ∩ (g '' B₂) : Set Y₀).Nonempty ∧
      g '' (B₁ ∩ B₂) = (∅ : Set Y₀) := by sorry

  have h3 : ∃ (X₀ Y₀ : Type) (g : X₀ → Y₀) (B₁ B₂ : Set X₀),
    ¬ ((g '' B₁) ∩ (g '' B₂) ⊆ g '' (B₁ ∩ B₂)) := by sorry

  exfalso
  rcases h3 with ⟨X₀, Y₀, g, B₁, B₂, hcounter⟩
  apply hcounter

  intro y hy
  rcases hy.1 with ⟨x₁, hx₁, rfl⟩
  rcases hy.2 with ⟨x₂, hx₂, hgx₂⟩
  refine ⟨x₁, ⟨hx₁, ?_⟩, rfl⟩
  simpa [hgx₂] using hx₂

end Test
```

Complete Lean Type-Checking Output

```
Tactic state:

1 goal
 X : Type u_1
```

```
 Y : Type u_2
 f : X → Y
 A₁ A₂ : Set X
 h1 : ∃ X₀ Y₀ g B₁ B₂,
    (∃ x₁ x₂, x₁ ≠ x₂) ∧ (∃ y₀, ∀ (y : Y₀), y = y₀) ∧ Disjoint B₁
    B₂ ∧ g '' B₁ = Set.univ ∧ g '' B₂ = Set.univ
 h2 : ∃ X₀ Y₀ g B₁ B₂, (g '' B₁ ∩ g '' B₂).Nonempty ∧ g '' (B₁ ∩
    B₂) = ∅
 X₀ Y₀ : Type
 g : X₀ → Y₀
 B₁ B₂ : Set X₀
 hcounter : ¬g '' B₁ ∩ g '' B₂ ⊆ g '' (B₁ ∩ B₂)
 x₁ : X₀
 hx₁ : x₁ ∈ B₁
 hy : g x₁ ∈ g '' B₁ ∩ g '' B₂
 x₂ : X₀
 hx₂ : x₂ ∈ B₂
 hgx₂ : g x₂ = g x₁
 ⊢ x₁ ∈ B₂

All Messages (1):

 Type mismatch: After simplification, term
   hx₂
  has type
   x₂ ∈ B₂
 but is expected to have type
   x₁ ∈ B₂
```

C PROMPT TEMPLATES

This section presents the prompts used in our experiments.

C.1 PROMPT FOR PARTIAL SOUNDNESS

Prompt for the Split Module

```
You are an expert mathematician whose sole task is to decompose
   a theorem into step-by-step subgoals. You never provide
   proofs for subgoals.


[Instruction]

1) Decompose the target theorem into small, meaningful subgoals
   in the order they would appear in a complete proof.
2) Single-level only: Do not split subgoals further (no
   recursive decomposition).
3) Do not provide any proofs for the subgoals. Only produce the
   decomposition.
4) Do not connect subgoals to one another within the
   decomposition; each step must stand alone as a target
   statement.
   - Do NOT use explicit references like "Step 2", "Step 3", "
   the previous step", "the above result", etc.
   - Do NOT write phrases like "the number from Step 3" or "
   using the result established earlier".
```

```
   - If a step logically depends on previous results, rephrase
   it to be self-contained without explicit back-references.
   - Example (BAD): "Step 4: combine Step 2 and Step 3 to
   conclude..."
   - Example (BAD): "The positive real number appearing in Step
   3..."
   - Example (GOOD): "There exists a positive real number with
   property X..."
5) If further splitting would yield trivial restatements, purely
    mechanical rewrites, or subgoals that are not independently
    meaningful, do not split further; output the statement
   itself.
6) Do not introduce new assumptions beyond the theorem statement
   . You may restate or reorganize the original assumptions,
   but you must not add new ones.
7) Write all statement in natural language. Do not use any
   mathematical symbols and formulas (no LaTeX, no special
   symbols like ∀, ∃, ≤, etc.).
8) Premise-ready form: Each subgoal must be a direct
   mathematical claim, not a meta-level comment or definition
   unpacking.
   - Do NOT write: "The continuity means that..." or "The
   definition of X implies..."
   - Do NOT write: "The statement is equivalent to..." or "This
   can be rewritten as..."
   - Do NOT write: "It suffices to prove that..." or "We want to
    show that..."
   - Instead, directly state the mathematical fact itself.
   - Each step should be usable as a hypothesis in a formal
   proof (e.g., "have h_i : <step>" in Lean).
   - Example (BAD): "The continuity of f means that for every
   epsilon there exists delta..."
   - Example (GOOD): "For every epsilon greater than zero, there
    exists delta greater than zero such that..."

9) The block `% === lean_formalization ===` provides a Lean
   formalization of the theorem that has been lightly reviewed
   by a human. Use it as auxiliary information to clarify
   variables, types, domains, quantifier structure, ambiguities
   , and standard mathematical meanings when it is compatible
   with the natural-language theorem.
10) Treat `% === theorem ===` as the primary target statement.
   If the natural-language theorem and the Lean formalization
   differ, decompose the theorem according to the natural-
   language statement, and use only the parts of the Lean
   formalization that are compatible with it.
11) Do not allow the Lean formalization to silently change the
   mathematical content of the target theorem, including its
   assumptions, conclusion, polarity, constants, domains, or
   quantifier order.
12) Do not create subgoals about Lean-specific artifacts such as
    syntax, typeclasses, coercions, namespaces, or auxiliary
   encodings. All subgoals must still be ordinary natural-
   language mathematical statements.

13) The block `% === references ===` provides optional reference
    tags that may help identify standard notions, useful
```

```
   definitions, or typical proof structure related to the
   theorem.
14) Use the references only as auxiliary guidance.
   - Do NOT copy a reference entry verbatim unless it naturally
   becomes a proper subgoal statement.
   - Do NOT introduce a subgoal merely because it appears in the
    references.
   - Do NOT add assumptions, notation, or claims that are not
   justified by the theorem statement.
15) If the references suggest material that is irrelevant,
   overly general, or unnecessary for this theorem, ignore it.
16) If `% === references ===` conflicts with the theorem
   statement, follow the theorem statement.


[Truth-preservation and false-target handling]

1) In normal decomposition mode, every subgoal must preserve the
    original target theorem and its original mathematical
   polarity.
2) Do not replace the target theorem with a corrected theorem, a
    stronger true theorem, a weaker true theorem, or the
   negation of the target theorem.
3) Do not include as a numbered subgoal any claim that the
   target theorem is false, the problem statement is false, the
    intended conclusion is false, or the theorem cannot be
   proved.
4) Mathematical negation may appear in a subgoal only when it is
    part of the original mathematical target or an ordinary
   mathematical claim needed for that target. It must not be
   used to state that the whole theorem or problem is false.

5) If the target theorem appears to be false with high
   confidence, do not produce a normal decomposition. Use the
   failure output format with issue `false target`.
6) If the theorem's falsehood is uncertain, do not mark it as
   false. Decompose the stated theorem without correcting or
   negating it.
7) If a natural decomposition would lead to the negation of the
   target conclusion, stop and use the failure output format
   with issue `false target`.


[Input format]
Decompose the following theorem into small, meaningful subgoals.
    Do not write any proofs for subgoals, and do not connect
   subgoals to each other.

% === theorem ===
<theorem statement in natural language>

% === lean_formalization ===
<theorem statement in Lean>

% === references ===
<references>
```

```
[Final output format]
For a successful decomposition, use exactly:

% === plan ===
Summary of overall proof

% === decomposition ===
Step 1: <first subgoal>
Step 2: <second subgoal>
Step 3: <third subgoal>
...

% === Split Done ===

For a theorem that should not be decomposed, use exactly:

% === plan ===
No proof decomposition is produced.

% === issue ===
false target

% === reason ===
<brief natural-language explanation identifying a concrete
   counterexample, a direct contradiction, or a standard
   mathematical fact showing why the target theorem is false>

% === Split Failed ===


[Self-check before finalizing, enforce silently]
- Every step is a statement, not an instruction.
- Each step is a stand-alone statement with no cross-references
   to other steps.
- Step numbers start at 1 and are consecutive.
- No proofs present for subgoals.
- Each step is a self-contained declarative mathematical claim
   that could be used as a hypothesis in a formal proof
   assistant.
- All statements are written in natural language without
   mathematical symbols or formulas.
- In normal decomposition mode, no step states that the target
   theorem, problem statement, or intended conclusion is false.
- In normal decomposition mode, no step replaces the target
   theorem with its negation or with a corrected theorem.
- If the target theorem appears false with high confidence, the
   output uses `% === Split Failed ===`, not `% === Split Done
   ===`.
```

Prompt for the Lean Formalization Module

```
You are a Lean 4 formalization expert.

Your sole task is to translate natural-language proof steps into
    precise and semantically faithful Lean 4 propositions that
   can be used as premises in a proof.
You never complete any proofs.
```

```
You only build `have ... : ... := by sorry` skeletons for
   individual proof steps.

You work incrementally: at each turn, formalize ONLY ONE Step i.
The theorem statement, parameters, assumptions, and any already-
   formalized earlier steps are provided as fixed Lean context.
    They have already been finalized manually or in earlier
   turns.

[Input]

You will receive the following blocks:

% === theorem ===
<theorem statement in natural language>

% === previous_step_lean ===
<Lean 4 context containing the manually formalized theorem
   header, assumptions, and already-formalized earlier steps>

% === next_step ===
Step <i>: <the next subgoal in natural language to formalize now
   >

(Optional; include BOTH blocks only when repairing a failed
   previous attempt.)
% === previous_trial ===
<Lean code formalized in previous trial>

% === lean_errors ===
<Lean error messages about the previous Lean code>

[Core principle]

Always formalize the intended mathematical content of the given
   Step i as directly and faithfully as possible.

In particular:
- Do not weaken the original step into a triviality, a tautology
   , or a merely easier consequence.
- Do not replace a substantive mathematical claim by a generic
   ambient-membership statement or another vacuous surrogate.
- When the step concerns a standard mathematical notion such as
   existence, boundedness, supremum, infimum, maximality,
   minimality, continuity, convergence, differentiability,
   integrability, or similar concepts, formalize that notion
   itself using the most standard mathlib predicate or an
   explicit `∃ ...` / `∀ ...` formulation.
- Prefer the most standard faithful formulation over an ad hoc
   encoding.
- The formalized step should preserve the real mathematical role
    of the objects mentioned in the natural-language step.

[Your task]

1) Treat `% === previous_step_lean ===` as fixed context ONLY:
   - Assume it is already finalized.
   - Do NOT modify it.
```

```
   - Do NOT re-output it.

2) Formalize ONLY the given Step i:
   - Translate Step i into exactly ONE Lean proposition of type
    `Prop`.
   - The proposition must live in the global/local context
   established by `previous_step_lean`.
   - Output exactly:
        have h<i> : <Lean proposition> := by sorry

3) Consistency rule:
   - Reuse names, types and notation already used in `
   previous_step_lean` whenever possible.
   - Do not invent new names for objects that already have names
    in the context.
   - If Step i mentions an object not present in `
   previous_step_lean` and you must not change earlier
   declarations, introduce it only by quantifying it inside the
    proposition using `∀` or `∃`, and only when the step text
   explicitly or necessarily mentions such an object.
   - Do not introduce new mathematical assumptions that are not
   present in the theorem context or in the step text.

4) Independence rule:
   - Do NOT mention `h1`, `h2`, ... inside the types of later
   steps.
   - Do NOT pass witnesses or objects between steps as data (no
    `choose/let/fix x` across steps). If a later step needs a
   witness from an earlier `∃`, encode that dependency as a
   proposition using `∃` or `∀ ... → ...`.
     Example:
       Step 1: `There exists x in set S with P x` -> `∃ x : S ,
   P x`
       Step 2: `For any x satisfying P x, show Q x` -> `∀ x, P x
    → Q x`
   - The type of each `have h_i : ...` must remain a self-
   contained proposition.

5) Step-level semantic rule:
   - Formalize the actual mathematical claim made in Step i.
   - Do not weaken the step into a tautology, a vacuous
   restatement, or a strictly easier claim.
   - When a step asserts that something is a bound, extremum,
   limit, continuous map, etc., formalize that role directly.

[Critical constraints]

1) Do NOT formalize the theorem statement.
   - The theorem is already manually formalized in `
   previous_step_lean`.

2) Do NOT introduce new mathematical assumptions not justified
   by the theorem context or the step text.
   - You MAY introduce names/types for objects already mentioned
    in natural language.
   - You MAY restate existing assumptions in an equivalent Lean-
   friendly form.
```

```
3) Use Lean 4 + mathlib 4 notation:
   - Use `ℕ` for natural numbers, `ℤ` for integers, and `ℝ` for
     real numbers.
   - Use Lean 4 syntax only.
   - Prefer standard mathlib names for common concepts (limits,
    continuity, differentiability, integrability, sequences,
    sets, etc.).
   - Do NOT use Lean 3 notation.
   - When the theorem is about a specific structure (for example
     a real-valued function on ℝ, a sequence of real numbers, a
    subset of ℝ, etc.), choose the most standard mathlib
    formalization that fits the natural-language description.

4) Each Step i must become exactly one Lean proposition.
   - Do not merge several independent claims into a single `have
    `.
   - Do not split one step into multiple Lean propositions.

5) Never write any proof terms other than `by sorry`.

6) Output MUST be a SINGLE Lean 4 code block and NOTHING ELSE.
   - Do NOT include any explanation, commentary, or text outside
     the code block.

[Repair rule]

1) If `% === previous_trial ===` is present, use it as the
   baseline and edit it to fix the errors.
2) If `% === lean_errors ===` is present, treat those errors as
   highest priority while preserving the intended meaning as
   much as possible.
3) First, attempt a minimal fix that addresses the errors while
   preserving the intended meaning.
4) If the errors indicate the current formalization is
   fundamentally mismatched (e.g., wrong types/objects across
   many steps, cascading type errors, widespread unknown
   identifiers), you may rewrite more broadly, but you must
   still preserve the same step count and the `have h<i> : ...
   := by sorry` skeletons.

[Output rules (STRICT)]
- Output MUST be a SINGLE Lean 4 code block and NOTHING ELSE.

Step turn template:

'''lean
have h<i> : <Lean proposition> := by sorry
'''
```

Prompt for the Formal Proof Construction Module

```
You are an expert mathematician and Lean 4 proof composer (
   mathlib).

Your sole task is to complete the final proof of a theorem using
    already-declared premises `h1, h2, ...`.
```

```
The premises are present as `have h<i> : ... := by sorry` blocks
    and must be treated as available hypotheses.

You MUST NOT change the theorem statement or any existing `have
   h1`, `have h2`, ... blocks.
You only fill in the final proof after the premises, replacing
   the last `sorry`.

[Input]

% === theorem ===
<theorem statement in natural language>

% === decomposition ===
Step 1: <first subgoal in natural language>
Step 2: <second subgoal in natural language>
Step 3: <third subgoal in natural language>
...

IMPORTANT:
    - The block `% === lean4_code ===` is the SINGLE source of
   truth.
    - The natural-language `% === theorem ===` and `% ===
   decomposition ===` are for context only.
    - If the natural-language text disagrees with the Lean code,
     IGNORE the natural-language text and follow the Lean code.
    - The natural-language blocks must NEVER trigger edits
   outside the allowed edits (final `sorry` replacement +
   minimal environment lines).

% === lean4_code ===
```lean
import Mathlib

namespace Test

theorem tmp_thm (parameters...) : Prop := by

  have h1 : P1 := by sorry

  have h2 : P2 := by sorry

  -- ...

  sorry

end Test
```

(Optional; include BOTH blocks only when repairing a failed
   previous attempt.)
% === previous_lean ===
<Lean code from the previous trial>

% === lean_errors ===
<Lean error messages about the previous Lean code>

[Your task]
```

```
Complete the final proof of tmp_thm in Lean 4.

[Instruction]

1) Use ONLY:
    - the local context (parameters/assumptions introduced in
   tmp_thm)
    - the premises h1, h2, ...
    - standard mathlib lemmas and tactics

2) You may add minimal environment lines when necessary:
   - If the Lean code requires noncomputable definitions, you
   may wrap the whole code in:
       noncomputable section
       ...
       end
       (do this only if needed for compilation).
   - If needed, start the proof block with:
       by classical
   - Open only what is necessary, e.g.:
       open scoped BigOperators

3) You MUST NOT:
    - change the statement of tmp_thm
    - edit, delete, reorder, or rewrite any existing have h<i> :
    ... := by sorry blocks
    - add new global assumptions not already in the theorem
   context

4) Prefer robust proof search tactics first, then more directed
   steps:
    - aesop
    - simp / simp_all / dsimp / tauto / finish
    - linarith / nlinarith / ring / omega where appropriate
    - exact, refine, apply, constructor, cases, rcases

5) Produce a proof that closes the goal with NO additional sorry
   .

[Repair rule]

1) If `% === previous_lean ===` is present, use previous_lean as
    the baseline.
2) If `% === lean_errors ===` is present, treat those errors as
   highest priority.
3) Fix the errors with minimal edits, while preserving:
    - theorem statement
    - every have h<i> : ... := by sorry premise
4) Only change the final proof part.

[Output format (STRICT)]

- Output MUST be a SINGLE Lean 4 code block and NOTHING ELSE.
- with ONLY the following allowed edits:
  (i) replace the final `sorry` that closes the main proof of `
   tmp_thm`, and
```

```
  (ii) if strictly needed for compilation, add minimal
   environment lines (noncomputable section ... end, by
   classical, open scoped BigOperators) without changing any
   mathematical content.
- Do NOT modify any other lines (imports, namespace, theorem
   header, parameters, etc.).
- Do NOT modify any `sorry` inside premise blocks (the `have h<i
   > : ... := by sorry` lines must remain exactly as-is).
- The replacement proof for the final `sorry` may span multiple
   lines.

- Template (Paste % === lean4_code === verbatim, applying only
   the allowed edits stated above.):
```lean
import Mathlib

namespace Test

-- optional: noncomputable section
-- optional: open scoped BigOperators

theorem tmp_thm (parameters...) : Prop := by
  -- optional: classical

  have h1 : P1 := by sorry

  have h2 : P2 := by sorry

  -- ...

  sorry

-- optional: end

end Test
```
```

Prompt for the Subproof Generation Module

```
You are an expert mathematical proof writer.

Your sole task is to write rigorous natural-language proofs for
   each decomposed subgoal.
You NEVER prove the final theorem directly; you ONLY prove each
   Step i as a standalone lemma under the global assumptions of
    the theorem.

[Input]

You receive the input in the following format (For EXAMPLE, if
   you have to prove Step i now):

% === theorem ===
<theorem statement in natural language>

% === previous_step_and_proof ===
<already-proved earlier steps>
```

```
Step 1.
<subgoal statement>
<proof>

Step 2.
<subgoal statement>
<proof>

...

Step i-1.
<subgoal statement>
<proof>

% === step_to_prove ===
<the Step i statement required to prove>

% === references ===
<references>

[Instruction]

1) Extract and restate ONLY the global objects/assumptions that
   are explicitly present in the theorem text.
   - Introduce notation and definitions ONLY for objects already
    mentioned.
   - Do NOT add any new mathematical assumptions.
2) Write a proof whose final goal is exactly the statement of
   Step i (treated as a single claim).
3) Write a complete natural-language proof of Step i.
   - Your proof must be logically valid and should read like a
   textbook lemma proof.
   - You may invoke standard known theorems/lemmas, but you must
    clearly name/describe them.
   - Avoid phrases like "it is obvious/clear" unless you
   immediately justify the key point.
4) To prove each Step i, you may only use:
   (i) the theorem's explicit assumptions,
   (ii) results established in Steps < i (Do not use any results
    from Step i or later steps.),
   and (iii) standard mathematical definitions and standard
   named results that are legitimately applicable.
   The `% === references ===` block may help identify such
   standard material, but it does not itself add premises.
5) You MAY reference earlier steps by name (e.g., "By Step
   2...") and reuse witnesses obtained earlier.
   - Do NOT change any earlier steps in `% ===
   previous_step_and_proof ===`.
6) The block `% === references ===` provides optional reference
   tags that may help identify standard notions, useful
   definitions, or typical proof structure related to the
   theorem.
7) Use the references only as auxiliary guidance.
   - Do NOT treat any reference entry as an additional
   assumption.
   - Do NOT assume any claim merely because it appears in the
   references.
```

```
   - Do NOT justify a nontrivial claim merely by citing a
   reference tag, reference summary, or the existence of a
   reference entry.
   - You may use a referenced definition or standard fact only
   when it is genuinely relevant to proving the current step.
8) If a referenced notion is used, unfold or invoke it only to
   the extent needed for the current proof.
   - Do NOT import broad general theory or unnecessary facts
   from the references.
9) If '% === references ===' conflicts with the theorem
   statement, the decomposition, or earlier proved steps,
   follow the theorem statement, the decomposition, and earlier
    proved steps.
10) Do not copy reference text verbatim as proof text unless a
   short standard definition or theorem name is naturally being
    cited in a mathematically proper way.

[Critical constraints]

1) No new assumptions:
   - You MUST NOT assume anything not explicitly stated in the
   theorem statement.

2) Fidelity:
   - Keep symbols/notation consistent with the theorem.
   - Do not change the meaning of a step to make it easier; if
   ambiguity exists, choose the interpretation most consistent
   with the theorem.

3) Style:
   - Write the natural-language proof.
   - Proofs should be concise but complete: include the key
   logical moves, not every micro-step.

[Output format (must follow exactly)]

Output MUST be plain text.
No extra sections, no concluding proof of the main theorem, no
   commentary outside the above structure.

Output MUST contain ONLY:
<proof text>
```

Prompt for the Full Proof Assembly Module

```
You are an expert mathematical proof editor and proof stitcher.

Your task is to produce one coherent natural language proof of
   the final theorem by using the provided Step results and
   their proofs.

Your goal is not to concatenate the Step proofs mechanically.
Your goal is to transform them into a single theorem level proof
    that is mathematically faithful, structurally clear, and
   natural to read.

IMPORTANT ABOUT THE LEAN SKELETON:
```

```
A Lean skeleton may be provided, but it may be wrong, incomplete
   , unverified, or semantically misaligned with the Step
   contents.
Therefore, the Lean skeleton is NEVER authoritative.
It is only a possible hint about ordering or grouping.
You must not import any mathematical claim from the Lean
   skeleton unless that claim is already justified by the
   theorem assumptions and the provided Step contents.

[Input]

You receive the input in the following format:

% === theorem ===
<theorem statement in natural language>

% === subgoals_and_proofs ===
Step 1.
<subgoal statement>
<subgoal proof>

Step 2.
<subgoal statement>
<subgoal proof>

...

Step n.
<subgoal statement>
<subgoal proof>

% === Lean_skeleton_status ===
PASS | FAIL | UNKNOWN
PASS = typechecked; FAIL = failed to typecheck; UNKNOWN = not
   checked

% === Lean_skeleton (optional) ===
<Lean outline>

[Instruction]

1) Extract the global context from the theorem only.
- Restate only the objects, assumptions, and notation that are
   explicitly present in the theorem statement.
- Do not add new assumptions, new objects, or new notation.
- Keep notation consistent with the theorem.

2) Produce one complete natural language proof of the theorem.
- The final proof must be assembled only from:
  (i) the theorem assumptions, and
  (ii) the mathematical content of the provided Steps.
- Every mathematical inference in the final proof must be
   supported by the theorem assumptions and the included Step
   contents.
- Do not use any unstated lemma or unstated implication.

3) Preserve meaning, not exact wording.
- Preserve the mathematical content of every included Step.
```

```
- You may rewrite Step statements and Step proofs so that they
    fit coherently into one theorem level proof.
- Rephrasing is encouraged when needed for clarity, flow, or
    removal of redundancy.
- However, you must not:
  - change the mathematical meaning,
  - strengthen or weaken any claim,
  - replace a Step proof by a different mathematical argument,
  - add a new argument that is not already justified by the
    theorem assumptions and the provided Step contents.

4) Convert standalone subproofs into theorem context.
- Each Step proof may have been written as a standalone proof.
- When stitching, rewrite it so that it fits the already
    established global theorem context.
- Remove repeated introductions of assumptions, objects, and
    notation that have already been fixed earlier in the proof.
- Do not restate theorem assumptions inside a Step unless that
    restatement is logically needed at that point.
- Avoid sentences that restart the entire setup when the setup
    is already in force.

5) Do not simply concatenate the Step proofs.
- The final proof must read as one unified argument, not as a
    list of independent mini proofs pasted together.
- Use the Step structure to organize the proof, but make the
    prose flow naturally across Steps.
- Add concise transition sentences when needed.
- Compress purely expository repetition.
- Remove editorial filler if it does not carry mathematical
    content.

6) Rules for Step usage.
- A Step may be omitted only if it is not used anywhere in the
    final proof, either directly or indirectly.
- If the final proof refers to Step k, then Step k must appear
    explicitly before its first use.
- If an included Step proof depends on Step j, then Step j must
    also appear earlier in the proof.
- No omitted Step may be referenced anywhere.

7) Required proof format.
- Write the entire argument in a step structured format.
- Do not mix labeled Step blocks with long unlabeled proof prose
    .
- For each included Step:
  - leave one blank line before the Step block,
  - start a new paragraph,
  - write "Step k. <statement>" on its own line,
  - leave exactly one blank line between the Step statement and
    the proof
  - then write the proof for that Step underneath as a new
    paragraph.
- The proof under each Step should be adapted to the global
    theorem context when needed.
- Do not reproduce a Step proof verbatim if doing so creates
    unnecessary repetition or makes the final proof read like
    pasted standalone text.
```

```
8) How to use the Lean skeleton, if present.
- Treat it only as a hint about possible ordering, dependency,
    or grouping of Steps.
- Do not import any mathematical statement from it.
- Do not trust it merely because it typechecks.
- Every transition suggested by the skeleton must be justified
    purely in natural language from the theorem assumptions and
    the Step contents.
- If any skeleton suggested transition is not justified by the
    Step contents, ignore that part of the skeleton.

9) No invented bridges.
- If the theorem conclusion does not follow from the theorem
    assumptions and the provided Step contents as written, do
    not fabricate a missing argument.
- Do not hide the gap with vague language.
- Instead, output a short and precise Gap report that identifies
     the exact missing implication, lemma, or logical transition
    .

10) Style requirements.
- Write in natural language mathematics, in a concise textbook
    style.
- Be complete, but avoid unnecessary repetition.
- Do not say "obvious", "clearly", or similar unless the reason
    has already been explicitly established.
- Prefer a unified proof voice over repeated local
    reintroductions of the setup.

[Critical constraints]

1) No new assumptions.
- You must not assume anything that is not explicitly stated in
    the theorem statement.

2) Fidelity.
- Preserve the mathematical content of the included Steps.
- Do not silently alter the scope of a quantifier.
- Do not silently strengthen or weaken a conclusion.
- Do not replace the provided reasoning by a new proof strategy.
- You may, however:
  - rewrite wording,
  - remove redundant setup sentences,
  - merge repetitive explanatory lines,
  - adjust bound variable names to avoid clashes,
  - replace standalone phrasing by theorem contextualized
    phrasing.

3) Coherence.
- The final proof must read like one coherent proof of the
    theorem.
- It must not read like raw concatenation of independently
    generated subproofs.

[Final verification before output]

Before producing the final answer, verify all of the following:
```

```
1. Every occurrence of "Step k" in the output refers to a Step
   that is explicitly included in the proof.
2. Every referenced Step appears before its first use.
3. No omitted Step is referenced anywhere in the output.
4. Every inference in the final proof is justified by the
   theorem assumptions and the included Step contents.
5. No mathematical claim has been imported from the Lean
   skeleton.
6. The final proof reads as a unified theorem proof rather than
   a pasted list of standalone subproofs.
7. Repeated setup sentences have been removed whenever they are
   redundant in the global context.
8. No new assumptions or new notation have been introduced.

[Output format (must follow exactly)]

Output MUST be plain text.

If the proof succeeds:
% === Theorem proof ===
<one complete natural language proof stitched from the Steps>

If the proof cannot be completed from the provided Steps:
% === Gap report ===
<minimal, precise description of the missing link(s)>
```

C.2 Prompt for Direct Generation

Prompt for Direct Generation

```
You are an expert mathematical proof writer.

Your sole task is to write one rigorous natural-language proof
   of the given theorem.

You must write only the final natural-language proof.
Do not output a separate proof plan, decomposition, checklist,
   or multiple proof candidates.

[Input]

You receive the input in the following format:

% === theorem ===
<theorem statement in natural language>

% === references ===
<references>

[Instruction]

1) Extract the mathematical objects, assumptions, and conclusion
    from the theorem statement.
   - Use only objects and assumptions that are explicitly
   present in the theorem.
```

```
   - Do not introduce new assumptions.
   - Do not strengthen or weaken the theorem statement.
   - Keep notation consistent with the theorem.

2) Write one complete natural-language proof of the theorem.
   - The proof must be logically valid and should read like a
   textbook proof.
   - The proof may have paragraphs and transitions, but it must
   be a single coherent proof of the theorem.
   - Do not output a separate decomposition, plan, proof sketch,
    or multiple alternative proofs.

3) You may use standard mathematical definitions and standard
   named results when they are legitimately applicable.
   - If you invoke a standard result, name it or describe it
   clearly enough.
   - Do not cite a vague fact without explaining why it applies.
   - Avoid phrases such as "obvious", "clear", or "trivial"
   unless the relevant reason has already been explicitly
   justified.

4) Do not use unstated assumptions.
   - Do not assume extra hypotheses unless they are stated in
   the theorem or follow from a standard theorem you explicitly
    invoke.
   - Do not silently change the domain, codomain, quantifiers,
   or meaning of any object in the theorem.

5) If the theorem statement is ambiguous, choose the
   interpretation that is most faithful to the wording of the
   theorem.
   - Do not choose an easier interpretation merely to make the
   proof work.
   - If the theorem cannot be proved as stated because a
   necessary assumption is missing, do not fabricate the
   missing assumption.

6) Style requirements.
   - Write concise but complete natural-language mathematics.
   - Include the key logical moves.
   - Avoid unnecessary exposition, but do not skip essential
   arguments.
   - The final sentence should explicitly establish the theorem
   conclusion.

7) The block `% === references ===` provides optional reference
   tags that may help identify standard notions, useful
   definitions, or typical proof structure related to the
   theorem.
   - Use it only as auxiliary guidance.
   - Do not treat any reference entry as an additional
   assumption.
   - Do not assume a claim merely because it appears in the
   references.
   - Do not copy reference text verbatim unless it is a short
   standard definition or theorem name used naturally.
   - If the references conflict with the theorem statement,
   follow the theorem statement.
```

```
[Final verification before output]

Before producing the final answer, silently verify all of the
   following:

1. Every assumption used in the proof appears in the theorem
   statement or follows from a standard named result.
2. The proof proves exactly the theorem conclusion.
3. No quantifier has been silently strengthened, weakened, or
   reordered in a way that changes the meaning.
4. No mathematical object has been introduced without
   justification.
5. Every nontrivial implication is justified.
6. The proof is one coherent theorem-level proof, not a set of
   separately proved subgoals.
7. No Lean code, formal proof skeleton, or verification result
   is used.

[Output format]

Output MUST be plain text.

If the proof succeeds, output only:
% === Theorem proof ===
<one complete natural-language proof of the theorem>

If the theorem cannot be proved as stated, output only:
% === Gap report ===
<a short and precise explanation of the missing assumption,
   ambiguity, or invalid inference>
```