MechMath: Sorrifier-Driven Formal Decomposition Workflow for Automated Theorem Proving

Ruichen Qiu [*1 2] **Yichuan Cao** [*2 3] **Junqi Liu** [*2 3] **Dakai Guo** [2] **Xiao-Shan Gao** [2 3] **Lihong Zhi** [2 3] **Ruyong Feng** [2 3]

Abstract

Recent advances in large language models (LLMs) and LLM-based agents have substantially improved the capabilities of automated theorem proving. However, for problems that require complex mathematical reasoning, current systems rarely succeed on the first try and must repeatedly modify their proof strategies. Existing approaches for handling failed attempts typically either iteratively fix errors within the proof or discard the entire proof and regenerate it from scratch. The former leads to progressively longer contexts, which progressively degrade the model's ability to attend to the remaining unresolved subproblems, while the latter is inefficient, as it may abandon mostly correct reasoning due to localized errors. To address this dilemma, we propose *MechMath*, a novel agent system that employs a sorrifier-driven formal decomposition strategy. By leveraging the `sorry` placeholder in Lean to precisely isolate unresolved subgoals while preserving the surrounding verified proof structure, MechMath extracts each failed subproblem into a clean, self-contained context and resolves it independently. This avoids both the waste of full regeneration and the excessive context length induced by repeated repairs. Experimental results on challenging mathematical competition benchmarks, including IMO 2025, Putnam 2025, miniF2F, and a subset of ProverBench, demonstrate that our agent achieves significant advantages in proving efficiency.

[1]School of Advanced Interdisciplinary Sciences, UCAS, Beijing, China [2]State Key Laboratory of Mathematical Sciences, Academy of Mathematics and Systems Science, CAS, Beijing, China [3]University of Chinese Academy of Sciences, Beijing, China. Correspondence to: Ruichen Qiu <qiuruichen20@mails.ucas.ac.cn>, Ruyong Feng <ryfeng@amss.ac.cn>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

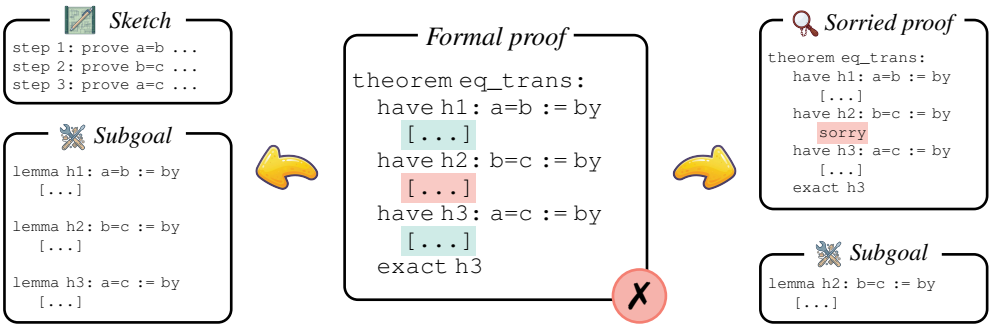
1. Introduction

Mathematical reasoning has long been one of the core research objectives in artificial intelligence. In recent years, with the rapid advancement of large language models (LLMs), their ability to handle complex mathematical reasoning tasks at the natural language level has improved significantly. State-of-the-art systems have achieved gold-medal performance in high school mathematics competitions (e.g., IMO 2025), while systems such as DeepMind's Aletheia (Feng et al., 2026) are progressively approaching the frontier of tackling research-level problems.

Despite these advances, theorem proving in natural languages is still plagued by verification difficulties, leading to potential logical hallucinations and erroneous conclusions, and lacking scalable automated validation (Chen et al., 2025b;a). In contrast, formal languages such as Lean, Rocq, and Isabelle offer superior logical rigor and verifiability. In formal math, previous neural theorem proving research primarily focused on training a single model for theorem proving tasks. Although such approaches achieved remarkable progress (Lin et al., 2025b), they still show limitations in long-horizon reasoning tasks (e.g., the failure in Putnam 2025 in Table 1). As foundation models continue to improve and agent frameworks gain increasing attention, research focus is gradually shifting toward agent systems for formal theorem proving that integrate strategic planning, tool use, and formal verification.

In this emerging agent paradigm, formal proof systems (Chen et al., 2025a; Varambally et al., 2026) rely on iterative strategies to resolve complex problems. The most direct approach involves error correction, where the agent continuously modifies the formal proof script. However, this frequently leads to contextual explosion, which severely compromises the model's attentional focus on unresolved errors. To mitigate this, subsequent research has shifted towards decomposition-based strategies. Upon failure, they discard the incorrect proof and regenerate an informal sketch to partition the problem into multiple sub-tasks.

However, this informal decomposition paradigm introduces a significant inefficiency. Whenever a proof attempt fails,



(a) Decompose Informally (b) Decompose Formally

Figure 1. Comparison between (a) informal and (b) formal decomposition. (a) Conventional approaches decompose the proof via an informal sketch, resulting in three subgoals. (b) Our method identifies the erroneous proof block and generates a sorried proof, from which we extract only one subgoal, minimizing decomposition overhead.

the entire formal proof is discarded, and the agent restarts by re-analyzing the problem in natural language and generating new sub-problems (left part of Figure 1). On the one hand, the model can often generate an informal sketch whose high-level reasoning structure is largely sound, indicating a potential capacity to generate insightful proofs. On the other hand, Lean enforces strict logical and syntactic correctness, so even minor omissions or missteps can cause proof compilation to fail. In such cases, discarding a nearly complete proof and restarting the decomposition is computationally wasteful, both in time and in resources.

This limitation motivates our strategy: a formally decomposing approach that preserves formal proof structure and avoids redundant re-analysis. Using `sorry` [1] placeholder, our approach isolates and resolves localized errors without discarding surrounding correct code (right part of Figure 1). Based on this principle, we propose a novel agent system, **MechMath**, which adopts an informal-to-formal proving pipeline and employs a formal decomposition module, **Sorrifier**, as illustrated in Figure 2. Experimental results on challenging mathematical competition benchmarks, including IMO 2025, Putnam 2025, miniF2F, and a subset of ProverBench, demonstrate that our agent achieves significant gains in proving efficiency.

2. Related Works

Formal Mathematics. Formal mathematics is the representation of mathematical definitions, theorems, and proofs in a rigorous formal language that can be processed by a computer (Harrison et al., 1996). The key benefit is that correctness is guaranteed, avoiding the risk of human error. A diverse ecosystem of formal languages and proof assistants has been developed to support this endeavor, such as Rocq and Isabelle (Kaliszyk & Rabe, 2020). Among these, Lean (Moura & Ullrich, 2021) has emerged as one of the most actively developed and community-driven systems. The hallmark of the Lean ecosystem is Mathlib (The Mathlib Community, 2020), a unified community-maintained library of formalized mathematics, which contains a vast and growing collection of definitions and theorems. Refer to Appendix A for more details.

Neural Theorem Proving. Previous research on neural theorem proving has largely focused on training a single model, which can be broadly categorized into two main paradigms: step-wise interaction (Xin et al., 2025b; 2024; 2025a; Hubert et al., 2026) and whole-proof generation (Wang et al., 2025; Ren et al., 2025; Lin et al., 2025a;b). Acknowledging the constraints of single-model approaches in long-horizon reasoning and complex proof structures, recent research has shifted toward agent-based paradigms, which augment foundation models with tool-use capabilities for the autonomous resolution of complex tasks (Luo et al., 2025). In the domain of formal mathematics, a growing body of work explores agentic workflows that augment prover models with additional components to guide proof construction, such as informal reasoning modules (Varambally et al., 2026) and interactive Lean toolchains (Dressler, 2025; Breen et al., 2025). Beyond these enhancements, more advanced paradigms have been proposed, including the training of formal provers through large-scale agentic reinforcement learning (Chen et al., 2025a), adopting autonomous multi-agent ensemble architectures (Axiom Math Team, 2025), and repurposing general coding agents as formal reasoners (Liu et al., 2026).

3. Methods

This section presents the architecture and workflow of **MechMath**. To ensure structural clarity, we first introduce core components of our system: LLMs, Lean toolkit, and Sorrifier. Then, we detail the iterative proof synthesis algorithm that orchestrates these components.

[1] In this paper, words in typewriter font are Lean commands.

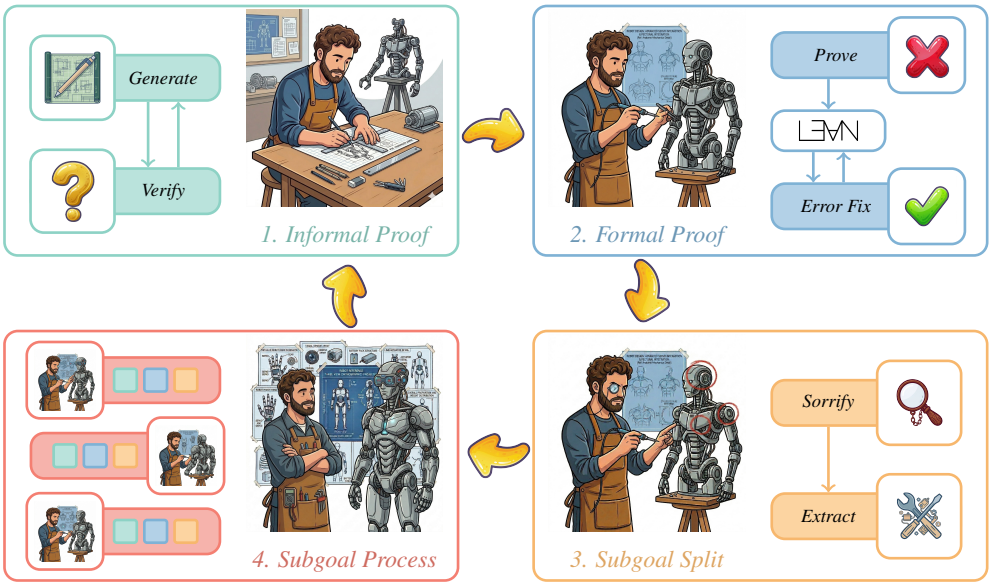


Figure 2. Workflow of MechMath. (1) An informal solution is generated, verified and iteratively improved through a feedback loop. (2) The final informal solution is then translated into a formal proof, which is further revised based on Lean's feedback. (3) If the proof continues to fail after several revisions, it is transformed into a sorried proof to extract subgoals. (4) Each resulting subgoal is subsequently handled by the same pipeline.

3.1. Components

LLMs. The agent employs three specialized LLMs, each of which has a different functional role. The **Reasoner** is an LLM with a strong mathematical reasoning ability. Its responsibilities include: (1) generating informal proof sketches for target statements; and (2) during subgoal extraction, synthesizing necessary premises from the local goal at a given `sorry` location to derive the corresponding subgoal. The **Verifier** is a general-purpose LLM, tasked with verification and evaluation. Its functions encompass: (1) evaluating the quality of informal proofs; and (2) verifying the accuracy and logical soundness of the extracted subgoals. The **Prover** is specialized in completing Lean proofs. It translates informal proof sketches into formal Lean proofs and revises incorrect proofs based on feedback.

Lean Toolkit. Beyond LLMs, we integrate the essential Lean-based utilities required by the agent, including (1) the **Lean compiler**, which checks the correctness of a Lean proof, returns detailed diagnostic feedback, and proof states; and (2) the **Lean data search tools**, which retrieve relevant theorems from Mathlib to assist in constructing proofs.

Sorrifier. In Lean, the `sorry` placeholder allows the compiler to accept the validity of a local lemma without requiring its proof to be fully constructed. Replacing erroneous components of a Lean proof with `sorry`, which is a process referred to as *sorrifying* a faulty proof (Ospanov et al., 2025), can transform an invalid proof into a syntactically valid yet incomplete one. This enables the user to focus exclusively on completing the missing parts.

Motivated by this idea, we present the **Sorrifier**, the central innovation of our pipeline. This tool repeatedly substitutes incorrect proof blocks with `sorry` placeholders, continuing this process until the full proof is successfully compiles in Lean. Building upon this principle, we move beyond the naive approach of discarding an entire failed Lean proof. Instead, we isolate and remove the erroneous portions of the proof and substitute them with `sorry` tactics, thereby producing a formally accepted but incomplete proof.

To fully realize this goal of maximum retention, the Sorrifier's algorithm must navigate the challenge of cascading errors. In dependent type systems like Lean, a single logical flaw often corrupts the local context, triggering a massive cascade of false errors in subsequent, otherwise correct code. If all reported errors were processed simultaneously, it would lead to unnecessary deletion of valid proof segments. Therefore, the Sorrifier is designed to operate through a highly targeted, iterative pipeline.

Specifically, upon encountering compilation errors, the Sorrifier surgically targets the *innermost* reported error. If the error lies within a `have` block, it replaces only the innermost failing proof with `sorry`. If the error occurs elsewhere, the proof is truncated exactly at the erroneous line, and `sorry` is appended to close the state. Crucially, after a single modification is made, the Lean environment is immediately recompiled, instantly removing any downstream cascading errors tied to the previously broken context. This step-by-step process repeats until the code is compiled successfully. The final output is a structurally valid Lean proof

containing a few `sorry` placeholders. Consequently, subsequent efforts can focus solely on resolving these localized subgoals, significantly reducing the overall complexity of the remaining theorem-proving tasks. A detailed algorithm is provided in the Appendix C.2.

3.2. Algorithm

By imitating the way human experts write Lean proofs, the algorithm can be broadly divided into four steps, as illustrated in Figure 2. First, an informal solution is generated and then iteratively refined through a feedback loop in which the Reasoner produces and revises the solution, while the Verifier evaluates it. Based on the final informal solution, the Prover generates a formal proof and revises it in response to feedback from the Lean. If the proof cannot be passed after several revisions, the Sorrifier processes the formal proof to extract the unresolved subgoals. Each subgoal is handled using the same three-step procedure, and this process repeats until all subgoals have been successfully fulfilled.

3.2.1. Informal Proof

When constructing Lean proofs, human experts typically begin with an informal proof sketch in natural language before translating it into a formal proof, as natural language is often better suited to exploratory reasoning. Inspired by this observation, our agent follows a similar strategy: it first generates an informal reasoning sketch, which then guides the construction of the formal proof. To obtain a coherent and logically structured informal solution, we employ a generate-and-verify loop between the Reasoner and Verifier, following the approach of Huang & Yang (2025). The Reasoner first generates an informal solution, and the Verifier evaluates its strategic soundness and logical correctness. Based on this assessment, the Reasoner refines the solution, after which a new evaluation begins. This iterative refinement process continues until the Verifier accepts the solution as correct or a predefined maximum number of iterations has been reached. To improve the reliability of verification, each candidate solution is independently evaluated three times by the Verifier and is accepted only if all three evaluations judge it correct.

3.2.2. Formal Proof

Based on the informal proof generated in the previous step, the Prover attempts to write the corresponding Lean proof step-by-step. For complex problems, it is challenging to produce a fully correct proof in a single attempt. In such cases, error messages returned by the Lean compiler are collected and provided to the Prover to fix errors. The error fixing process proceeds over multiple rounds. Although each round typically resolves some errors, for longer proofs, the Lean compiler may generate hundreds of error messages in a single pass, making it difficult for the prover to address all issues simultaneously. Moreover, fixing an error may introduce new ones. Therefore, it is unrealistic to expect that a continuously running fix–verify cycle will eventually correct all errors in the proof. After a predefined number of rounds, the proof is passed to the subgoals splitting part.

3.2.3. Subgoal Split

For the proof processed by the Sorrifier, each place filled by `sorry` will be converted into a new subgoal, and the aforementioned process will be repeated for the subgoal. The specific process is divided into the following three steps, which are illustrated in Figure 3.

Sorrify. Given a faulty proof that remains unresolved after several rounds of error correction, the Sorrifier iteratively isolates and replaces erroneous components with `sorry`. This is achieved by surgically targeting the innermost reported error and repeating this process until a successful compilation is achieved. The resulting output is a structurally valid Lean proof that passes the compiler but remains incomplete, containing only a limited number of `sorry` placeholders. Consider, for example, the proof shown in Figure 3. A type mismatch error occurs in the `have step2` of the formal proof. In response, the Sorrifier replaces this block with `sorry`, thus producing a sorried proof.

Extract. Building upon this sorried proof, our pipeline further facilitates modular reasoning by extracting localized subgoals from each `sorry` placeholder. At each occurrence of a `sorry` within the Lean proof script, the corresponding tactic state and proof goal are extracted, as illustrated in the orange block of Figure 3. The Reasoner is then tasked with identifying which hypotheses and contextual conditions in the current tactic state are relevant to the proof goal. Irrelevant hypotheses are discarded, and only those that are essential are retained. Based on this analysis, the model formulates a self-contained subgoal. In this manner, complex proofs can be decomposed into smaller, independently verifiable components, thereby streamlining subsequent theorem-proving efforts.

To ensure the reliability and effectiveness of the subgoal extraction process, the subgoals extracted by the Reasoner must undergo a two-stage evaluation by the Verifier before being integrated into the proof pipeline. The first stage involves syntactic verification using Lean, which checks whether the extracted subgoals are well-formed and adhere to the syntactic conventions of the formal language, ensuring that the subgoals can be processed in subsequent steps without compilation errors.

Following syntactic validation, the extracted subgoals are passed to the Verifier for semantic evaluation. This stage assesses the extraction correctness and the logical soundness

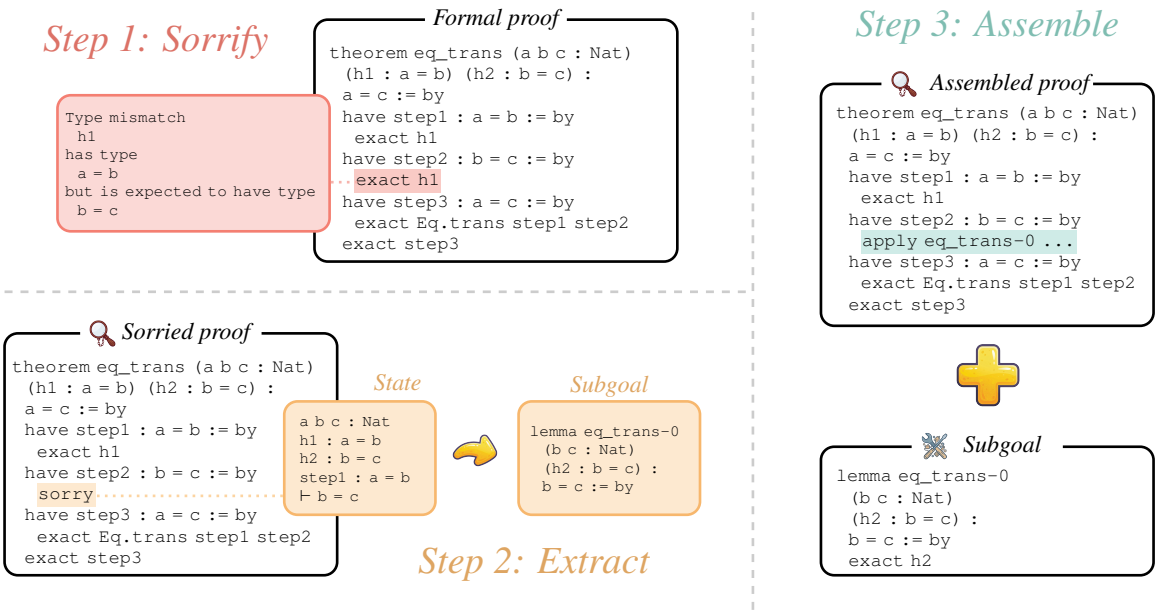


Figure 3. Example of the subgoal splitting workflow. The proof is first verified by Lean (red block, step 1), and the Sorrifier replaces erroneous parts with `sorry` placeholders. The goal state of each `sorry` position is extracted, then converted into new subtheorems (orange blocks, step 2). After a subtheorem is proved, it is integrated back into the original proof at the corresponding location using `apply` tactic (green text, step 3).

of the subgoals. Extracting correctness evaluates whether they accurately reflect the intended reasoning steps. Subgoals that are incorrectly extracted deviate from the original proof context and are likely to fail to reassemble into the complete proof. Logical soundness measures whether the subgoal can be proven, usually checked by whether there are obvious counterexamples or essential hypotheses missing. Subgoals that contain logical errors may result from a flawed proof approach, as illustrated in the Appendix B.1. They will disrupt the subsequent proof loop, potentially leading to infinite cycles or failed proof attempts.

Assemble. Once a subgoal is successfully proven, it is reassembled into the final proof structure, thereby contributing to the completion of the original theorem. As illustrated in Step 3 of Figure 3, this reintegration is achieved by replacing each `sorry` placeholder with a corresponding `apply` tactic. It allows the Lean compiler to index the relevant subgoal at each proof position, enabling the proof to be constructed incrementally. In this way, the mechanism ensures that each verified subgoal is incorporated at the correct position, maintaining a modular and incremental proof development process that complements the extraction mechanism described above. Refer to Appendix B.2 for a real example.

3.2.4. Subgoal Process

Having established the mechanisms for extracting and reintegrating subgoals, we now describe how each subgoal is processed. These subgoals undergo the aforementioned proof pipeline recursively. If a subgoal cannot be proven directly, it is passed through the Sorrifier, which generates further nested subgoals. This iterative decomposition continues until all subgoals are resolved, thereby ensuring that the original theorem is ultimately proven through a structured, hierarchical refinement process. Since any faulty proof can be sorrified and converted into subgoals, the pipeline may otherwise iterate indefinitely. To prevent infinite recursion, we introduce a termination mechanism. When a subgoal yields a nested subgoal identical to itself, we deem the proof to have made no progress and revert to a formal proving attempt. If the number of such retries reaches a predefined threshold, the subgoal is considered unprovable, and the overall proving process terminates.

4. Experiments

To comprehensively evaluate **MechMath**, we conduct experiments on 12 problems from the 2025 Putnam Competition, 4 from the 2025 IMO, the widely-used miniF2F (Zheng et al., 2022) test set, and a subset of ProverBench (Ren et al., 2025). In addition, we perform ablation studies to assess the necessity of each component and the compatibility of different underlying models.

Experiment settings. We employ Gemini 3.1 Pro (Google DeepMind, 2026) as the unified reasoner, verifier, and prover. The knowledge cutoff point for this model is January

Table 1. Putnam 2025 benchmark. (a) A single prover model or a general LLM cannot solve any problem within 32 attempts. (b) MechMath achieves high efficiency on 11 solved problems: shorter time expenditure (6/11), lower cost (10/11), and fewer theorem counts (8/11). The averages of Axiom and Numina are computed exclusive of A5.

(a) Results of single models (Pass@32)

	A1	A2	A3	A4	A5	A6	B1	B2	B3	B4	B5	B6
Goedel Prover V2 32B	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Gemini 3.1 Pro	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Claude 4.5 Opus	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

(b) Quantitative results of agents

	A1	A2	A3	A4	A5	A6	B1	B2	B3	B4	B5	B6	Avg
Time Expenditure (min)													
Hilbert	-	-	-	-	-	-	-	-	228	-	-	-	228
Seed	60	30	120	240	-	240	540	360	30	120	240	180	196
Axiom	110	180	165	107	518	259	270	65	43	112	254	494	187
Numina	27	81	30	169	2040	89	55	136	30	308	88	797	165
MechMath-Easy	**18**	45	106	114	-	**79**	-	**41**	57	**41**	104	-	67.2
MechMath-Hard	21	179	108	**64**	-	82	132	**41**	148	126	258	**93**	**114**
Cost ($)													
Hilbert	-	-	-	-	-	-	-	-	37.9	-	-	-	37.9
Numina	50	50	50	50	1000	50	50	50	50	50	50	300	72.7
MechMath-Easy	1.53	4.89	11.6	19.9	-	5.48	-	5.78	7.40	7.69	25.6	-	9.99
MechMath-Hard	4.21	29.0	19.5	10.5	-	4.91	29.8	3.72	15.4	20.1	57.0	9.01	**18.5**
Proof Tree Complexity (#Lemmas)													
Hilbert	-	-	-	-	-	-	-	-	48	-	-	-	48
Seed	27	18	23	28	-	38	21	48	26	18	103	185	49
Axiom	23	26	78	32	52	28	49	28	11	23	66	30	36
Numina	1	5	21	11	77	37	6	14	9	10	33	41	17
MechMath-Easy	**1**	**4**	**7**	15	-	4	-	8	**8**	11	39	-	11
MechMath-Hard	**1**	20	9	**10**	-	**1**	30	**2**	9	17	45	**10**	**14**

2025 [2], ensuring that it has no exposure to Putnam 2025 and IMO 2025. We employ the Kimina Lean server (Santos et al., 2025) for verification of the Lean code, and use LeanDex and Loogle [3] for the retrieval of the theorem. All experiments are conducted under Lean v4.15.0 (only for miniF2F) and v4.26.0 (for others). We compare our method against several state-of-the-art baselines: (1) single models within 32 attempts, including Goedel Prover V2 32B (Lin et al., 2025b), Gemini 3.1 Pro, Claude 4.5 Opus (Claude, 2025); (2) formal math agents, including Hilbert (Varambally et al., 2026), Aristotle (Achim et al., 2025), Axiom (Axiom Math Team, 2025), Seed-Prover 1.5 (Seed) (Chen et al., 2025a), and Numina-Lean-Agent (Numina) (Liu et al., 2026). For the open-source Hilbert baseline, we adopt Gemini-3.1-Pro-Preview as the reasoning model to ensure a fair experimental setup. For the remaining baselines, we report the performance as documented in their original documents. More details are given in Appendix C. The code for reproduction is in https://github.com/MechMath/MechMath-v1.

Easy/Hard mode. We categorize our framework into "easy" and "hard" modes. The former is the basic pipeline we described in Section 3. The latter incorporates two specialized modules to handle Lean-specific failures. First, a retriever-based tool addresses syntax-level errors, such as unknown identifier resulting from misspellings or version-induced library shifts, by providing the model with semantically relevant Mathlib documentation. Second, the Reasoner performs high-level structural and strategic analysis to assist the Prover in resolving tactic-related failures (e.g., "linarith failed to find a contradiction"). The "hard" mode presents a clear performance-resource trade-off: while it incurs higher latency and increased context overhead, it yields substantial gains in error-correction efficiency and proof reliability. We further analyze this tradeoff in the ablation study.

[2]Report on https://ai.google.dev/gemini-api/docs/models/gemini-3.1-pro-preview.

[3]LeanDex: https://leandex.projectnumina.ai/, Loogle: https://loogle.lean-lang.org/

Table 2. Quantitative results on the IMO 2025 benchmark. MechMath spends the lowest time on average.

(a) Time (min)

	P1	P3	P4	P5	Avg
Numina	277	132	360	324	273
Seed	990	300	480	60	458
MechMath-Hard	363	137	178	110	197

(b) Cost ($)

	P1	P3	P4	P5	Avg
MechMath-Hard	141	43	95	47	82

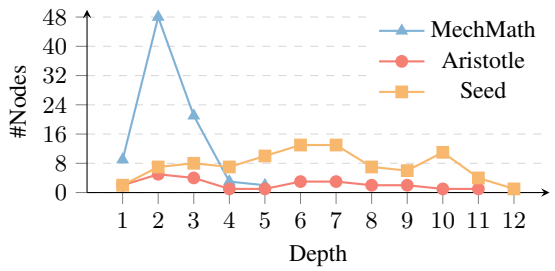


Figure 4. Statistics of proof trees of IMO 2025 P4. The tree of MechMath is wide but shallow, while others' are relatively deep.

4.1. Main Results

Putnam 2025. The Putnam is a math competition that assesses advanced problem-solving across algebra, analysis, combinatorics, and number theory. In Putnam 2025, Our agent exhibits strong effectiveness and exceptional efficiency in automated formal theorem proving. As shown in Table 1, under a fixed budget, MechMath-Easy successfully solves 9 of the 12 problems, and MechMath-Hard solves 11, substantially outperforming other baselines on average in terms of solved problems across all four metrics. MechMath-Easy merely requires less than $10 and 70 minutes per question on average. Intuitively, this efficiency stems from our sorrifier-driven formal decomposition workflow: the agent identifies errors precisely where they arise, avoiding repeated analysis and unnecessary auxiliary lemmas. Although MechMath does not solve A5 within the $100 limit, the performance profiles of Axiom and Numina confirm the high computational demands of this problem. Due to experimental constraints and a tight budget, no further attempts for A5 are performed.

IMO 2025. We further test four problems from IMO 2025. From Tables 2, MechMath-Hard only uses less than 200 minutes on average, which is roughly 2/3 time of Numina. This gap widens further when compared with Seed, with our running time less than half of Seed's. Further analysis of the proof structures shows that our proof trees are broader but shallower, while those generated by other methods contain multiple deeply nested nodes, as shown in Figure 4 (refer to Appendix B.3 for detailed tree structures). Since proofs at the same level can be parallelized, whereas parent nodes must wait for their child nodes to be completed, our shallower structure contributes to greater efficiency.

Table 3. Pass rates on miniF2F and ProverBench-NT. MechMath achieves highly competitive performance with extremely low cost.

(a) Evaluation results on the miniF2F benchmark.

Model	Acc	Note
STP	67.6	Pass@25600
Kimina-Prover-8B	78.3	Pass@32
Kimina-Prover-72B	92.2	TTRL
DeltaProver	95.9	Pass@16384
SeedProver	99.6	-
BFS-Prover-V2-32B	95.1	w/ Planner
Goedel-Prover-V2-8B	90.2	Pass@8192
Goedel-Prover-V2-32B	92.6	Self-Correction
Hilbert (GD)	99.2	11.3K LLM Calls
DeepSeek-Prover-V2-7B	82.0	Pass@8192
DeepSeek-Prover-V2-671B	88.9	Pass@8192
Hilbert (DS)	98.4	Pass@8192
MechMath-Hard (Ours)	99.2	max: 512 LLM Calls avg: 57.6 LLM Calls

(b) Evaluation results on the number theory subset (40 problems) in ProverBench. Notably, MechMath successfully identifies 4 erroneous problems within the original dataset.

Model	Acc	Note
DeepSeek-Prover-V2-7B	25.0	Pass@32
Goedel-Prover-V2-8B	25.0	Pass@32
Goedel-Prover-V2-32B	30.0	Pass@32
Hilbert (GD)	40.0	Gemini 3.1 Pro
MechMath-Hard (Ours)	60.0	max: 216 LLM Calls avg: 45.2 LLM Calls

Beyond time and monetary cost, our generated proofs are distinguished by their minimal average length and small number of auxiliary lemmas (Table 1 and Table 9), reflecting a highly disciplined and focused proof style. Refer to Appendix B.2 for the formal proof of the main theorem of IMO 2025 P3 as an example and D.1 for more results.

MiniF2F. To further validate the scalability and robustness of our approach, we conduct comprehensive evaluations on the miniF2F benchmark. As summarized in Table 3a, MechMath-Hard achieves an impressive 99.2% (242/244) pass rate on miniF2F. While this accuracy parallels the state-of-the-art baseline, Hilbert and SeedProver, our primary contribution lies in the exceptional efficiency and elegance of the solution process. To achieve the same top-tier performance, MechMath requires merely 512 LLM calls, which is a stark contrast to the 11.3K calls demanded by Hilbert. This drastically reduced reliance on LLM invocations trans-

lates directly to significant savings in both inference time and computational cost. Meanwhile, our systematic workflow consistently yields proofs that are shorter and more disciplined. For a comprehensive visualization of these advantages, Figure 6 illustrates the cumulative pass rate across varying inference budgets (LLM calls and token usage), with further statistical details in Appendix D.2.

ProverBench. We further extend our evaluation to the ProverBench. Following the evaluation protocol adopted in the recent work DreamProver (Zhang et al., 2026), we focus on a representative subset of 40 number theory problems. As shown in Table 3b, our method achieves a significant performance leap, with MechMath successfully solving 24 out of the 40 problems. In contrast, the strong baseline Hilbert (powered by Gemini 3.1 Pro Preview) only resolves 16 problems, while DeepSeek Prover and Godoel Prover solve fewer than 12 problems even with Pass@32 sampling. Notably, beyond the superior success rate, MechMath astutely identifies 4 erroneous problems (p6, p13, p20, and p28) within the original dataset. These results not only corroborate our core conclusions but also demonstrate that MechMath maintains exceptional accuracy and efficiency across benchmarks of varying scales and complexities. A breakdown of the computational costs and case studies of these erroneous problems are discussed in Appendix D.3.

4.2. Ablation Study

The ablation study encompasses three aspects: (1) comparison of the easy and the hard mode; (2) the necessity of the informal proving process; and (3) the scalability of our pipeline across different models. Due to budget constraints, the ablation experiments are conducted only on the first four Putnam problems.

Easy/Hard modes. To evaluate the performance-resource trade-off of incorporating Lean-specific feedback modules, we conducted an ablation study across three configurations: the addition of a search module, a comment module, and both of them (the "hard mode"). For relatively straightforward problems (A1 and A2), the inclusion of auxiliary tools proved counterproductive, increasing costs and occasionally misleading the model's reasoning (Table 4(a)). However, as problem complexity increased (A3 and A4), these tools became instrumental, effectively reducing both monetary cost and the volume of auxiliary lemmas. Furthermore, for problems B1 and B6, which remained intractable in the "easy" mode, the synergy of these modules enabled successful resolution, as detailed in Table 1.

Informal prove. As shown in Table 4(b), our agent benefits substantially from the integration of the informal proof. Despite the fact that incorporating an informal proving step introduces additional overhead for the exceptionally simple problem A1, its benefits become evident as the difficulty of the problem increases. Specifically, a well-structured informal proof significantly reduces errors in formal proofs, thereby decreasing the size of the proof tree (e.g., the number of lemmas) and ultimately lowering the overall overhead. Moreover, for problem A2, the absence of an informal proof renders the prover incapable of completing the proof. Motivated by the observation that an informal proof may be unnecessary for simple theorems, we introduce an additional fast proving step prior to the informal proving process to improve efficiency.

Different Models. To test the scalability of our pipeline

Table 4. Ablation study. (b), (c) and (d) is conducted on MechMath-Hard, since the easy mode fails in most of these settings.

(a) Comparison of easy and hard mode

	A1	A2	A3	A4
	Cost ($)			
Easy	1.53	4.89	11.6	19.9
+ search	3.28	13.3	11.1	13.7
+ comment	6.33	5.70	6.19	6.42
+ both (Hard)	4.21	29.0	19.5	10.5
	#Lemmas			
Easy	1	4	7	15
+ search	4	12	9	18
+ comment	8	9	9	7
+ both (Hard)	1	20	9	10

(b) Influence of the informal proof

Cost (#Nodes)	A1	A2	A3	A4
w/ informal	$4.21 (1)	$29.0 (20)	$19.5 (9)	$10.5 (10)
w/o informal	$3.90 (2)	✗	$23.3 (11)	$20.5 (44)

(c) Influence of the model

	A1	A2	A3	A4
Gemini 3.1 Pro	✓	✓	✓	✓
Claude 4.5 Opus	✓	✓	✓	✓
GPT-5.5	✓	✓	✓	✓
Deepseek 3.2 Reasoner	✓	✗	✗	✗
Seed 2 Pro	✓	✗	✗	✗
Kimi K 2.5	✓	✗	✗	✗

(d) Effectiveness of the Sorrifier mechanism

Method	Metric	A1	A2	A3	A4
Discard & Retry	LLM Calls	✗	✗	✗	✗
	Time (min)	✗	✗	✗	✗
	Cost ($)	✗	✗	✗	✗
Iterative Repair	LLM Calls	4	✗	42	126
	Time (min)	20	✗	141	518
	Cost ($)	1.59	✗	16.30	55.70
MechMath	LLM Calls	3	31	69	88
	Time (min)	18	45	106	114
	Cost ($)	1.53	4.89	11.60	19.90

across different models, we use various models as the reasoner and prover, including Gemini 3.1 Pro (Google DeepMind, 2026), Claude 4.5 Opus (Claude, 2025), GPT-5.5 (OpenAI, 2026), Deepseek 3.2 Reasoner (DeepSeek-AI et al., 2025), Seed 2.0 Pro (Bytedance Seed, 2026), and Kimi K2.5 (Kimi, 2026). Due to the fact that these models can generate dozens of combinations when paired together, and limited by experimental conditions, we always maintain consistency between Reasoner and Prover, while replacing these two models simultaneously. As reported in Table 4(c), with our formal decomposition pipeline, Gemini 3.1 Pro, Claude 4.5 Sonnet and GPT-5.5 can solve all four questions. Constrained by the Lean capabilities, Deepseek 3.2 Reasoner, Seed 2.0 Pro and Kimi K2.5 manage to solve only one problem. Yet even this modest result reflects a substantial improvement over the original model, where no problems could be solved.

Sorrifier To evaluate the efficacy of the Sorrifier mechanism, we conducted a controlled experiment on Putnam A1-A4 under a strict budget of 128 LLM calls, comparing it against two baselines: (1) *discard & retry*: The system first generates an informal proof and subsequently performs up to 128 independent formal generation attempts. (2) *iterative repair*: Upon producing the initial formal proof, the system directly executes up to 127 consecutive repair iterations on it, with the corresponding error feedback provided in each round. As summarized in Table 4(d), the *discard & retry* strategy failed entirely (0/4), while the *iterative repair* strategy solved only three problems (3/4). In contrast, although MechMath consumes more LLM calls on certain problems to extract and evaluate subgoals, it significantly reduces overall inference time and API costs. This substantiates the core advantage of the Sorrifier mechanism: by isolating erroneous segments into clean, self-contained contexts for independent resolution, it avoids the resource waste of full global regeneration and prevents the context length explosion typical of repeated repairs, ultimately enabling highly efficient formal proof derivation.

4.3. Additional Experiments and Analysis

To address the concern that localized repair mechanisms (e.g., the Sorrifier) might struggle to maintain global inductive invariants in highly structural paradigms like mathematical induction, MechMath effectively mitigates this potential limitation through rigorous verification of informal proofs and precise evaluation of formal subgoals. As shown in Table 5, the system achieves consistent and exceptionally high pass rates across various heterogeneous subsets of miniF2F, notably securing an 8/8 success rate on induction problems that require global planning. Furthermore, the system performs exceptionally well on the Putnam and IMO 2025 benchmarks, which encompass a diverse range of distinct mathematical disciplines such as continuous real analysis, complex combinatorics, structural discrete geometry, and game theory, thereby demonstrating profound cross-domain robustness.

Table 5. Pass rates across diverse categories on miniF2F

Category / Source Subset	Passed / Total
IMO Subset	19 / 20
AIME Subset	15 / 15
AMC Subset	45 / 45
Algebra	88 / 88
Number Theory	67 / 68
Induction	8 / 8
Overall	242 / 244

Besides, In complex proofs, the agent might inadvertently deduce logically flawed subgoals. To avoid pursuing unprovable structures, MechMath employs a dual-layer insurance mechanism comprising verification for informal sketches and subgoal assessments, which is further discussed in Appendix E.2. We also provide more detailed analysis of Sorrifier mechanism in Appendix E.3

5. Conclusion

In summary, we propose a novel agent workflow-**MechMath**, which adopts a strategy centered on sorrifier-driven formal decomposition and significantly improves efficiency. It novelly leverages the `sorry` placeholder to surgically isolate and resolve localized errors while preserving the structurally correct surrounding proof. This approach enables iterative subgoal decomposition without discarding valuable progress, effectively bridging the gap between informal reasoning and formal verification. Experimental results on challenging mathematical competition benchmarks, including IMO 2025 and Putnam 2025, demonstrate that MechMath achieves significant improvements in proving efficiency, highlighting the effectiveness of formal decomposition over traditional informal approaches.

Limitations and Future Work. MechMath employs a fixed pipeline that follows an iterative cycle of informal proof, formal proof, and subgoal split. However, an ideal agent may require the flexibility to adapt based on context. For example, it can omit the informal proof stage for simpler problems or perform additional error-fix iterations for formal proof on harder ones. In future work, we will modularize the sorrifier as a standalone component integrated into the harness. This design preserves the efficacy of the sorrifier-driven formal decomposition while enabling the model to autonomously invoke the component as needed.

References

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

Axiom Math Team. From seeing why to checking everything. Technical report, Axiom Math Team, 2025. URL https://axiommath.ai/territory/from-seeing-why-to-checking-everything.

Benjamin Breen, Marco Del Tredici, Jacob McCarran, Javier Aspuru Mijares, Weichen Winston Yin, Kfir Sulimany, Jacob M. Taylor, Frank H. L. Koppens, and Dirk Englund. Ax-prover: A deep reasoning agentic framework for theorem proving in mathematics and quantum physics, 2025. URL https://arxiv.org/abs/2510.12787.

Bytedance Seed. Seed2.0 model card: Towards intelligence frontier for real-world complexity. https://seed.bytedance.com/en/seed2, 2026.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, Zhihong Wang, Mingxuan Wang, Chenrui Wei, Shufa Wei, Huajian Xin, Fan Yang, Weihao Gao, Zheng Yuan, Tianyang Zhan, Zeyu Zheng, Tianxi Zhou, and Thomas Hanwen Zhu. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience, 2025a. URL https://arxiv.org/abs/2512.17260.

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

Claude. System card: Claude opus 4.5. https://www-cdn.anthropic.com/bf10f64990cfda0ba858290be7b8cc6317685f47.pdf, November 2025.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuduan Wang, Yue Gong, Yuhan Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni, Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin Shen,

Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. Deepseek-v3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

Oliver Dressler. Lean lsp mcp: Tools for agentic interaction with the lean theorem prover, 3 2025. *URL https://github.com/oOo0oOo/lean-lsp-mcp*, 2025.

Tony Feng, Trieu H. Trinh, Garrett Bingham, Dawsen Hwang, Yuri Chervonyi, Junehyuk Jung, Joonkyung Lee, Carlo Pagano, Sang hyun Kim, Federico Pasqualotto, Sergei Gukov, Jonathan N. Lee, Junsu Kim, Kaiying Hou, Golnaz Ghiasi, Yi Tay, YaGuang Li, Chenkai Kuang, Yuan Liu, Hanzhao Lin, Evan Zheran Liu, Nigamaa Nayakanti, Xiaomeng Yang, Heng-Tze Cheng, Demis Hassabis, Koray Kavukcuoglu, Quoc V. Le, and Thang Luong. Towards autonomous mathematics research, 2026. URL https://arxiv.org/abs/2602.10177.

Google DeepMind. Gemini 3.1 pro model card. https://deepmind.google/models/model-cards/gemini-3-1-pro/, February 2026. Published: February 2026.

John Harrison et al. *Formalized mathematics*. Turku Centre for Computer Science Turku, 1996.

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

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026. ISSN 1476-4687. doi: 10.1038/s41586-025-09833-y. URL https://doi.org/10.1038/s41586-025-09833-y.

Cezary Kaliszyk and Florian Rabe. A survey of languages for formalizing mathematics. In *International Conference on Intelligent Computer Mathematics*, pp. 138–156. Springer, 2020.

Kimi. Kimi k2.5: Ai that sees, codes, and works like an expert. https://www.kimi.com/ai-models/kimi-k2-5, 2026.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia LI, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover: A frontier model for open-source automated theorem proving. In *Second Conference on Language Modeling*, 2025a. URL https://openreview.net/forum?id=x2y9i2HDjD.

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

Junqi Liu, Zihao Zhou, Zekai Zhu, Marco Dos Santos, Weikun He, Jiawei Liu, Ran Wang, Yunzhou Xie, Junqiao Zhao, Qiufeng Wang, Lihong Zhi, Jia Li, and Wenda Li. Numina-lean-agent: An open and general agentic reasoning system for formal mathematics, 2026. URL https://arxiv.org/abs/2601.14027.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. Large language model agent: A survey on methodology, applications and challenges, 2025. URL https://arxiv.org/abs/2503.21460.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pp. 625–635. Springer, 2021.

OpenAI. Introducing GPT-5.5. https://openai.com/index/introducing-gpt-5-5/, June 2026.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. APOLLO: Automated LLM and lean collaboration for advanced formal reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=fxDCgOruk0.

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

Marco Dos Santos, Hugues de Saxcé, Haiming Wang, Ran Wang, Mantas Baksys, Mert Unsal, Junqi Liu, Zhengying Liu, and Jia Li. Kimina lean server: A high-performance lean server for large-scale verification, 2025. URL https://arxiv.org/abs/2504.21230.

The Mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pp. 367–381, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370974. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024. ISSN 1476-4687. doi: 10.1038/s41586-023-06747-5. URL https://doi.org/10.1038/s41586-023-06747-5.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=GN8OdkTo3B.

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

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data, 2024. URL https://arxiv.org/abs/2405.14333.

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

Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Ming Ding. BFS-prover: Scalable best-first tree search for LLM-based automatic theorem proving. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32588–32599, Vienna, Austria, July 2025b. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1565. URL https://aclanthology.org/2025.acl-long.1565/.

Youyuan Zhang, Jialiang Sun, Hangrui Bi, Chuqin Geng, Wenjie Ma, Zhaoyu Li, and Xujie Si. Dreamprover: Evolving transferable lemma libraries via a wake-sleep theorem-proving agent. *arXiv preprint arXiv:2604.26311*, 2026.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics, 2022. URL https://arxiv.org/abs/2109.00110.

A. Brief Introduction to Lean

Lean is an open-source interactive theorem prover (ITP) and a functional programming language developed primarily by Moura & Ullrich (2021). Built upon a version of dependent type theory known as the *Calculus of Inductive Constructions* (CiC), Lean provides a rigorous framework for formalizing mathematical definitions and verifying the correctness of proofs.

Unlike traditional informal mathematics, Lean requires that every logical step be verified by a small, trusted kernel, ensuring a high degree of mathematical certainty. It has gained significant traction within the mathematical community, notably through the *mathlib* project (The Mathlib Community, 2020), which is a massive collaborative library of formalized mathematics. In practice, users employ various tactics to resolve logical goals. The following table summarizes the primary commands (tactics) used within a `by` block to manipulate the proof state and construct formal arguments.

Table 6. Fundamental Tactics for Formal Proof Construction in Lean.

Tactic	Functionality	Example Context
`have`	Introduces an intermediate assertion or lemma to the local context.	`have h : n > 0 := ...`
`let`	Defines a local term or function with a specific value.	`let f := n + 1`
`apply`	Matches the current goal with the conclusion of a theorem.	`apply Nat.add_comm`
`exact`	Closes a goal that matches a given term or hypothesis precisely.	`exact h`
`intro`	Moves a universal quantifier ($\forall$) or premise ($\rightarrow$) into the context.	`intro n hn`
`cases`	Performs case analysis on a hypothesis or inductive type.	`cases h with | hl | hr`
`rw`	Substitutes terms in the goal using an equality or equivalence.	`rw [h_eq]`
`simp`	Invokes a simplification engine to automate trivial rewrites.	`simp [add_assoc]`

B. Real Examples in the MechMath Pipeline

All examples presented in the main text are toy examples. In this section, we provide some real examples collected from our experiments, which may be better understood.

B.1. Example of a Logical Error in Subgoal Extracion

Here is an example of a logical error in subgoal extraction, which indicates the necessity of evaluation. A Lean error exists in the proof of the lemma, and is replaced with the red `sorry` block by the sorrifier. However, as the verifier model points out, there is a logical flaw here, which results in an impossibility to prove. This kind of subgoal should not be extracted. Instead, the prover will try to fix the whole proof to ensure the logical soundness.

Part of the proof for a lemma:

```
lemma putnam_2025_a2-0 (x : ℝ) (hx : x ∈ Icc (0 : ℝ) Real.pi) (h1 : sin x ^ (2 : ℕ) ≤ x ^ (2 : ℕ)) :
  (1 : ℝ) / Real.pi * x * (Real.pi - x) ≤ sin x := by
  have hpi_pos : 0 < Real.pi := Real.pi_pos
  have h_bound : ∀ y ∈ Icc (0 : ℝ) (Real.pi / 2), (1 : ℝ) / Real.pi * y * (Real.pi - y) ≤ sin y := by
    intro y hy
    have hy_nonneg : 0 ≤ y := hy.1
    have hy_half : y ≤ Real.pi / 2 := hy.2
    have h_jordan := Real.mul_le_sin hy_nonneg hy_half
    have h_quad_le : (1 : ℝ) / Real.pi * y * (Real.pi - y) ≤ (2 : ℝ) / Real.pi * y := by
      have hy_pi_sub : Real.pi - y ≤ Real.pi := by linarith [hy.1]
      calc (1 : ℝ) / Real.pi * y * (Real.pi - y)
        _ ≤ (1 : ℝ) / Real.pi * y * 2 := by
            sorry
        _ = (2 : ℝ) / Real.pi * y := by ring
    exact le_trans h_quad_le h_jordan
  rcases le_or_gt x (Real.pi / 2) with h_half | h_half
  · exact h_bound x ⟨hx.1, h_half⟩
  · have h_symm : sin (Real.pi - x) = sin x := Real.sin_pi_sub x
    have h_pi_sub : Real.pi - x ∈ Icc (0 : ℝ) (Real.pi / 2) :=
      ⟨by linarith [hx.2], by linarith [h_half]⟩
    calc (1 : ℝ) / Real.pi * x * (Real.pi - x)
      _ = (1 : ℝ) / Real.pi * (Real.pi - x) * (Real.pi - (Real.pi - x)) := by ring
      _ ≤ sin (Real.pi - x) := h_bound (Real.pi - x) h_pi_sub
      _ = sin x := h_symm
```

Analysis by the verifier: The goal is to prove: $\frac{1}{\pi}y(\pi - y) \leq \frac{2}{\pi}y$. Let's test this inequality for a valid value of y, such as $y = 0.5$, which clearly sits within $[0, \pi/2]$ (since $\pi/2 \approx 1.57$). Substituting $y = 0.5$ into the left-hand side: $\frac{1}{\pi}(0.5)(\pi - 0.5) = 0.5 - \frac{0.25}{\pi} \approx$

$0.5 - 0.079 = 0.421$. Substituting $y = 0.5$ into the right-hand side: $\frac{1}{\pi}(0.5)(2) = \frac{1}{\pi} \approx 0.318$. The inequality evaluates to $0.421 \leq 0.318$, which is mathematically strictly false.

B.2. Example of a Proof Written by MechMath

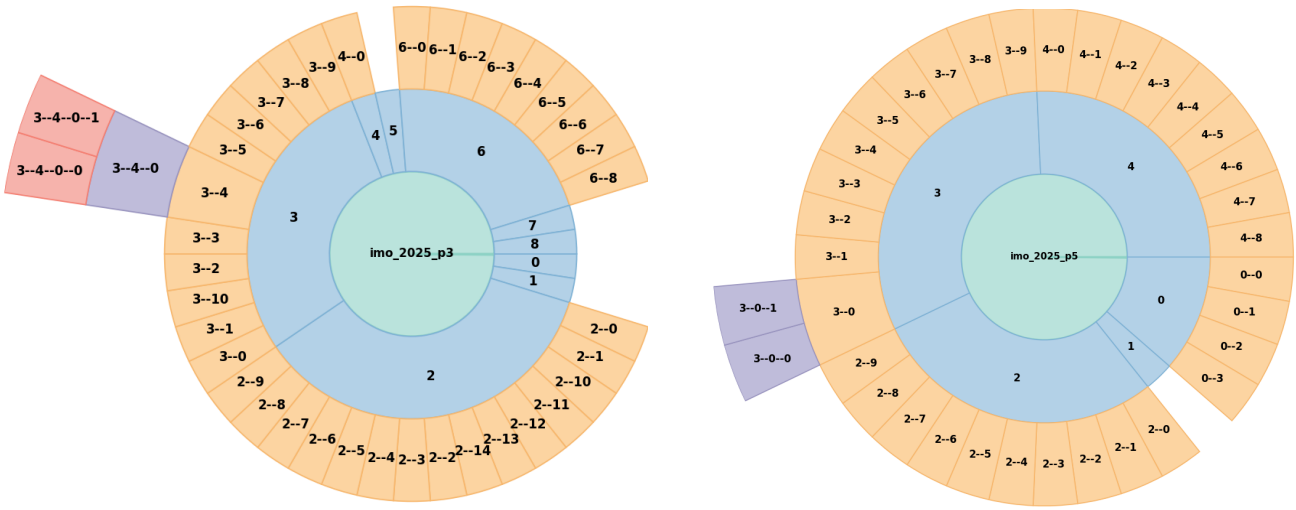
The following Lean code is part of the proof of IMO 2025 P4, which is generated by MechMath. The proof structure is highly disciplined, with each `have` block corresponding to a specific step in the informal sketch (marked blue). More importantly, instead of naively translating the informal sketch into formal proof, prover will fix the error according to the Lean feedback, Lean search results, and comments of verifier. For example, the prover adds two more `have` blocks between `step6` and `step7` to supplement necessary logic.

Within the `have` block, some Lean proofs contain errors. Through the formal decomposition workflow, these error blocks are first converted into `sorry` by the sorrifier, then extracted as subgoals for further proving. Once all subgoals have been proven, they are assembled to the original proof (marked orange). Detailed proof of subgoals is omitted. All proofs of our experiments can be accessed in the github repository.

```
theorem imo_2025_p3 (IsBonza : (ℕ+ → ℕ+) → Prop)
  (hisBonza : ∀ f, IsBonza f ↔ ∀ a b : ℕ+, (b : ℕ) ^ (a : ℕ) ≡ (f b : ℕ) ^ (f a : ℕ) [MOD f a]) :
  IsLeast {c : ℝ | ∀ f, IsBonza f → ∀ n : ℕ+, (f n : ℝ) ≤ c * (n : ℝ)} 4 := by
  constructor
  · intro f hf n
    have hBonza : ∀ a b : ℕ+, (b : ℕ) ^ (a : ℕ) ≡ (f b : ℕ) ^ (f a : ℕ) [MOD f a] := (hisBonza f).mp hf
    have step1 : f 1 = 1 := by exact imo_2025_p3_0 f hBonza
    have step2 : ∀ p a : ℕ+, Nat.Prime (p : ℕ) → (p : ℕ) ∣ (f a : ℕ) → (p : ℕ) ∣ (a : ℕ) := by exact
     imo_2025_p3_1 f hBonza step1
    have step3 : ∀ a : ℕ+, padicValNat 2 (f a : ℕ) ≤ padicValNat 2 (a : ℕ) + 2 := by
      exact fun (a : ℕ+) ↦ imo_2025_p3_2 f hBonza step1 step2 a
    have step4 : (∃ p : ℕ+, Nat.Prime (p : ℕ) ∧ (p : ℕ) % 2 = 1 ∧ (f p : ℕ) > 1) → ∀ a : ℕ+, f a = a := by
      exact
        fun (a : ∃ (p : ℕ+), Nat.Prime ↑p ∧ ↑p % (2 : ℕ) = (1 : ℕ) ∧ ↑(f p) > (1 : ℕ)) (a_1 : ℕ+) ↦
        imo_2025_p3_3 IsBonza hisBonza f hf hBonza step1 step2 step3 a a_1
    have step5 : (∀ p : ℕ+, Nat.Prime (p : ℕ) → (p : ℕ) % 2 = 1 → (f p : ℕ) = 1) → ∀ a : ℕ+, (f a : ℝ) ≤ 4 *
     (a : ℝ) := by
      exact
        fun (a : ∀ (p : ℕ+), Nat.Prime ↑p → ↑p % (2 : ℕ) = (1 : ℕ) → ↑(f p) = (1 : ℕ)) (a_1 : ℕ+) ↦
        imo_2025_p3_4 f hBonza step1 step2 step3 step4 a a_1
    have step_final : (f n : ℝ) ≤ 4 * (n : ℝ) := by
      by_cases h_exists : ∃ p : ℕ+, Nat.Prime (p : ℕ) ∧ (p : ℕ) % 2 = 1 ∧ (f p : ℕ) > 1
      . have h_fa_a : f n = n := step4 h_exists n
        have h_fa_R : (f n : ℝ) = (n : ℝ) := by exact_mod_cast h_fa_a
        have hn_pos : 0 ≤ (n : ℝ) := by positivity
        linarith
      . push_neg at h_exists
        have h_all : ∀ p : ℕ+, Nat.Prime (p : ℕ) → (p : ℕ) % 2 = 1 → (f p : ℕ) = 1 := by
          exact fun (p : ℕ+) (a : Nat.Prime ↑p) (a_1 : ↑p % (2 : ℕ) = (1 : ℕ)) ↦
            imo_2025_p3_5 f h_exists p a a_1
        exact step5 h_all n
    exact step_final
  · intro c hc
    let F_val (n : ℕ) : ℕ :=
      if n % 2 = 1 then 1
      else if n % 4 = 2 then 2
      else 4 * 2 ^ padicValNat 2 n
    have hF_pos : ∀ n : ℕ+, 0 < F_val (n : ℕ) := by
      intro n
      dsimp [F_val]
      split_ifs
      . omega
      . omega
      . positivity
    let F : ℕ+ → ℕ+ := fun n => ⟨F_val (n : ℕ), hF_pos n⟩
    have step6 : IsBonza F := by exact imo_2025_p3_6 IsBonza hisBonza F_val rfl hF_pos F rfl
    have step7_F_val : F_val 4 = 16 := by exact imo_2025_p3_7 F_val rfl
    have step7_F4 : (F (4 : ℕ+) : ℝ) = 16 := by
      have h4 : (F (4 : ℕ+) : ℕ) = 16 := step7_F_val
      exact_mod_cast h4
    have step7 : 4 ≤ c := by exact imo_2025_p3_8 IsBonza c hc F step6 step7_F4
    exact step7
```

B.3. Example of the Proof Tree

Proof tree of the IMO 2025 P3 and P5. The innermost node is the root nodes. Outer small sectors of each sector is the child nodes of that node. The proof tree generated by MechMath is wide but relatively shallow.



(a) Tree of IMO 2025 P3

(b) Tree of IMO 2025 P5

Figure 5. Proof tree of the IMO 2025 P3 and P5. The innermost node is the root nodes. Outer small sectors of each sector is the child nodes of that node. The proof tree generated by MechMath is wide but relatively shallow.

C. Implementation Details of Experiments

C.1. Experiment Settings

Datasets. To comprehensively evaluate the effectiveness of MechMath, we select Putnam 2025 and IMO 2025. William Lowell Putnam 2025 is a notoriously challenging mathematics competition for undergraduate students, containing 12 problems from different fields of mathematics. IMO 2025, the International Mathematical Olympiad, is the most prestigious high school mathematics competition, featuring 6 problems that test combinatorial, geometric, and number-theoretic insights. IMO 2025 Problem 2 is a geometric problem that is difficult to formalize in Lean and is usually solved with different approaches such as AlphaGeometry (Trinh et al., 2024); therefore, we exclude it from the test.

Evaluation. To comprehensively evaluate our method's performance, we report the results on generating valid formal proofs for each problem, alongside the corresponding line counts and the number of lemmas. Furthermore, to assess efficiency, we also record the time expenditure and API cost for full proof generation. We assign a budget of $100 for Putnam 2025 and $200 for IMO, except $300 for Putnam 2025 A5, which is widely recognized for its difficulty due to the absence of essential theorems.

Baselines. We compare our results against several state-of-the-art baselines: Seed-Prover 1.5, Axiom, and Numina-Lean-Agent, based on their reported performance. In addition, we also compare with Hilbert, using Goedel Prover 32B as the formal prover and pairing it with Gemini 3.1 Pro Preview as the informal model, which is the same base model used for our agent, to maintain a fair and consistent experimental environment. For other hyperparameters, we rigorously set them to their experimental configuration. We also allocate the same budget for Hilbert to perform a fair comparison.

Implementation Details. For the implementation of our MechMath, we use Gemini-3.1-Pro-Preview as the base model, whose training data, as mentioned in their model card, is cut off in January 2025, thereby avoiding data leakage. We use LeanDex as our retrieval engine to retrieve the relevant Mathlib theorems for the agent. For informal solution generation, we employ 16 iterations of refinement. In each iteration, we perform 3 verification passes to ensure the correctness of the generated proof. For the formal proving, we perform an initial formalization attempt, followed by up to 4 rounds of error correction based on Lean's compiler feedback. Subsequently, each error identified by the Sorrifier will undergo a 3-stage "extract-assess" evaluation. If a logical error is detected, the resulting diagnostic comments and the corresponding `sorry` locations are passed back to perform further error fixing. The prompts used during the process are provided in the next section.

C.2. Algorithm of the Sorrifier

Algorithm 1 Sorrifier Algorithm: Iterative Proof Patching

input Initial formal code C, Lean Compiler $Compiler$, Max iterations N
output Patched code C_{final}, Boolean $is_success$

for $i = 1$ to N **do**
  ▷ *Cleanup statements located after existing 'sorry'*
  $C \leftarrow$ CleanUpRedundantCode(C)
  ▷ *Compile code and retrieve verification results*
  $result, errors \leftarrow$ Compiler.Verify(C)
  if $result$ is Valid (allowing 'sorry') **then**
    return (C, True)
  end if
  if $errors$ is empty **or** $Compiler$ timeouts **then**
    return (C, False)
  end if
  ▷ *Locate the primary error's line and column*
  $err \leftarrow$ SelectPrimaryError$(errors)$
  ▷ *Find structural blocks overlapping the error line*
  $overlapping_blocks \leftarrow$ FindBlocksContainingLine$(C, err.\text{line})$
  ▷ *KEY 1: Isolate the innermost block by finding the one with the minimum length*
  $innermost_block \leftarrow$ FindShortestBlock$(overlapping_blocks)$
  ▷ *KEY 2: Apply different strategies based on whether the error is inside or outside a block*
  if $innermost_block$ is Empty **then**
    ▷ *Strategy for OUTSIDE: The error is not wrapped in a 'have' or similar block. Truncate the statement at the error position and append 'sorry'.*
    $C_{new} \leftarrow$ LineSorry$(C, err.\text{line}, err.\text{column})$
  else
    if $innermost_block.\text{type} \in \{\text{calc}, \text{choose}\}$ **then**
      $C_{new} \leftarrow$ BlockSorry$(C, innermost_block)$
    else
      ▷ *For 'have' or 'replace', target the proof body*
      $C_{new} \leftarrow$ BodySorry$(C, innermost_block, err)$
    end if
  end if
  ▷ *Prevent infinite loops if no valid modification*
  if $C = C_{new}$ **then**
    return (C, False)
  end if
  $C \leftarrow C_{new}$
end for
return (C, False)

C.3. Code for Reproduction

We include our code in the supplement material. After the review process, it will be made public available.

C.4. Prompts Used in the Experiment

C.4.1. Generate Informal Proof

```
You are a Formal Logic Expert and Mathematical Proof Engine.  Your goal is to derive
proofs that are rigorously structured, formalization-ready, and devoid of ambiguity.

Process Instruction:  Before providing the final response, use your internal
```

monologue/thinking process to fully solve the problem, verify every algebraic substitution, and ensure the logical chain is unbroken. Once your derivation is complete, you must output **exactly and only one <informal_solution></informal_solution> block** containing the step-by-step proof (without thinking).

Core Constraints:

- Purely Algebraic/Symbolic: Do NOT use geometric intuition or visual symmetry. All geometric concepts must be translated into precise algebraic/analytic definitions.

- Atomic Steps: Decompose reasoning into the smallest possible logical units.

- No Hand-waving: Forbidden phrases include "obviously," "it is clear that," or "intuitively."

- Direct Output: Provide only the solution. Do NOT include internal monologue, "thought" blocks, or introductory/concluding conversational filler.

Formatting Requirements:

- Tags: Enclose the entire response in <informal_solution> tags.

- Structure: Present the proof as a discrete list of points.

- Naming Convention: In your final <informal_solution> block, use the exact prefix Step X: (e.g., Step 1:, Step 2:) for each point, preceded by a bullet point (dot).

Derivation Instructions:

- Definitions: Explicitly state all variable types, definitions, and assumptions at the start.

- Step-by-Step Derivation: Number every step sequentially.

- Explicit Justification: For EACH step, explicitly state the rule of inference, algebraic identity, axiom, or theorem used.

- Calculations: Show every intermediate stage of simplification or substitution. Do not skip algebraic manipulation steps.

- No Formal Code: Focus solely on the informal solution of the problem. Do not write any Lean code.

Problem Statement:
{problem}

C.4.2. Verify Informal Proof

Your task is to evaluate the quality of a solution to a problem. The problem may ask for a proof of a statement, or ask for an answer. If finding an answer is required, the solution should present the answer, and it should also be a rigorous proof of that answer being valid.

Please evaluate the solution and score it according to the following criteria:

- If the solution is completely correct, with all steps executed properly and clearly demonstrated, then the score is 1

- If the solution is generally correct, but with some details omitted or minor errors, then the score is 0.5

- If the solution does not actually address the required problem, contains fatal errors, or has severe omissions, then the score is 0

- Additionally, referencing anything from any paper does not save the need to prove the reference. It's okay IF AND ONLY IF the solution also presents a valid proof of the reference argument(s); otherwise, if the solution omits the proof or if the proof provided is not completely correct, the solution should be scored according to the criteria above, and definitely not with a score of 1

Please carefully reason out and analyze the quality of the solution below, and in your final response present a detailed evaluation of the solution's quality followed by your score.

Therefore, your response should be in the following format:

Here is my evaluation of the solution:

[Your evaluation here. You are required to present in detail the key steps of the solution or the steps for which you had doubts regarding their correctness, and explicitly analyze whether each step is accurate: for correct steps, explain why you initially doubted their correctness and why they are indeed correct; for erroneous steps, explain the reason for the error and the impact of that error on the solution.]

Based on my evaluation, the final overall score should be: \boxed...

[where ... should be the final overall score (0, 0.5, or 1, and nothing else) based on the above criteria]

Do not evaluate any semantics in informal solution related to Lean4; focus solely on the quality of the informal solution. If the informal solution mentions any specific Lean code, clearly point out that it should not include Lean code, and ignore the code in the evaluation. ---

Here is your task input:

Problem
{problem}

Solution
{student_solution}

C.4.3. Refine Informal Proof

You are given a mathematical problem, an existing solution, and feedback on that solution.

Your task is to produce a **revised solution** that is more complete, rigorous, and clearly justified.

Problem
{problem}

Previous Solution
{solution}

Feedback
{feedback}

Instructions

- Process Instruction: Before providing the final response, use your internal monologue/thinking process to fully solve the problem, verify every algebraic substitution, and ensure the logical chain is unbroken. Once your derivation is complete, you must output **exactly and only one <informal_solution></informal_solution> block** containing the step-by-step proof (without the thinking process).

- Carefully read the feedback and determine which points are **valid** and which may be due to **misunderstanding or evaluator error**.

- If you **agree** with a feedback item:

- Revise the solution to fix the issue.

- Add missing steps, clarify logical transitions, or strengthen rigor as needed.

- If you **disagree** with a feedback item:

- Keep the original reasoning if it is correct.

- Add **explicit explanations or clarifications** to prevent future misunderstandings.

- Do **not** simply restate the feedback.

- The final solution should be:

- Self-contained

- Logically coherent

- Mathematically rigorous

- Easy to follow for a careful reader

- Focus solely on the informal solution of the problem. DO NOT write any Lean code. ---

Output Format

- Use your internal monologue/thinking process to fully solve the problem, and provide **only** the revised solution in your final output.

- Provide exactly and only one <informal_solution> block.

- In your final <informal_solution> block, use the exact prefix Step X: (e.g., Step 1:, Step 2:) for each point, preceded by a bullet point (dot).

C.4.4. System Prompt of Formal Proving and Fixing

You are a Lean 4 and Mathlib expert.

Your task is to produce a complete, correct, and fully typechecking proof of a Lean 4 theorem.

Mandatory Rules (Must Follow Exactly)

1. Do NOT modify the formal statement in any way.
- Do not change names, arguments, universes, implicit arguments, or formatting.
- The theorem header must appear exactly as given in formal statement.

2. The final proof MUST be complete.
- The proof must not contain `sorry`, `admit`, `by_cases?`, `by_contra?`, or any placeholder.
- You CANNOT use the tactic `native_decide`and `aesop`.
- All goals must be fully resolved.

3. Use only Lean 4 syntax compatible with Mathlib.
- All identifiers must exist in Mathlib.
- The code should compile in a standard Mathlib environment.
- If a lemma from `search_results`is used, apply it with correct arguments.

Lean Hints:

Followings are some useful hints for Lean 4:

1. When dealing with inequalities, equalities and arithmetic operations like subtraction or division in `ℕ`(natural numbers), beware of truncation. Use `ℝ`, `ℚ`or `ℤ`when possible for arithmetic operations. Avoid using `ℕ`unless required by the theorem statement.
2. Be ESPECIALLY careful about implicit types while defining numeric literals. AVOID patterns like `0 - 1`or `1 / 2`without specifying the types.
3. ALWAYS specify types when dealing with numeric values to avoid ambiguities and unexpected behavior.
4. Use `simp only [specific_lemmas]`rather than bare `simp`to avoid unpredictable simplifications.
5. Use `rw [→ lemma]`for reverse direction. When `rw`fails, try `conv => rhs; rw [lemma]`to target specific subterms. nth_rw n [lemma] to rewrite only the nth occurrence.
6. When `ring`fails on ring expressions, try `ring_nf`first to normalize, or cast to a concrete ring like `ℝ`where the tactic works better.
7. Apply `norm_num`for concrete arithmetic calculations and `norm_cast`to simplify type coercions systematically.
8. Use `by_contra h`for proof by contradiction, which introduces the negation of the goal as hypothesis `h`.
9. If you get a `no goals to be solved`error, it means that the previous tactics already solved the goal, and you can remove the subsequent tactics.

10. When proving theorems, ALWAYS write the proof in tactic mode, starting the proof with `:= by`.
11. Do NOT use `begin`, `end`blocks in your proof. This is invalid in Lean 4.
12. Use the `\in`rather than `in`for any set (include for `\sum`)
13. When working with inequalities or `calc`blocks, pay special attention to the direction of the inequality signs.

Strategy Guidelines (Non-output)

- You may freely combine tactics (`simp`, `linarith`, `ring`, `nlinarith`, `omega`, `aesop`, etc.) and term-style proofs.
- You may introduce local `have`or `let`bindings inside the proof.
- Prefer robustness and clarity over cleverness.

Output

- You must return EXACTLY ONE ```lean4```code block.
- The code block must contain the final complete proof only.
- You must NOT include any other ```lean4```blocks anywhere in the output.

C.4.5. Generate Formal Proof

You need to formalize the provided 'Informal Solution' to prove the provided 'Formal Statement'. You MUST follow the structure of the informal solution closely, ensuring that each step in the informal solution corresponds to a clear and logically sound step in the Lean proof. Long and complete proof is expected for higher reward.

Formal Statement ```lean4 {formal_statement} ```

Informal Solution (MUST follow this structure) {informal_solution}

Critical Proof Principles

1. **No Reliance on "Ready-Made" Theorems**:
- These problems are mathematically nontrivial. Do NOT solve the goal by a single `apply`of a powerful library lemma that encapsulates the entire result.
- Use only basic definitions and elementary lemmas (e.g., rewriting definitions, arithmetic manipulation, manual inequalities).

2. **From First Principles & Step-by-Step**:
- Break the proof into small, explicit steps using `have`, `calc`, or intermediate lemmas.
- Each formal step must correspond to a clear piece of reasoning. If a conclusion requires multiple ideas, split it.

3. **Proof Format**:
- You CAN NOT write new lemma or definition in the ```lean4```blocks.
- You CAN NOT change the original formal statement.
- Only ONE theorem or lemma is allowed in the lean block.
- The proof must not contain `sorry`, `admit`, `by_cases?`, `by_contra?`, or any placeholder.

4. **Structure Alignment**:
- The Lean proof must strictly mirror the structure of the provided Informal Solution. Do not introduce a fundamentally different strategy.
- You MUST completely trust that the Informal Solution represents the most streamlined proof. Follow its proof method and process unconditionally, translating it strictly into Lean form without deviation.
- For each step in the Informal Solution, formalize it into Lean code, preserving both the logical flow and mathematical reasoning of the original proof.

C.4.6. Fix Formal Proof

The Lean code you wrote before contains errors. Your task is to modify the code so that it compiles successfully, while preserving the original mathematical intent as much as possible.

Error Message from Lean:
{error_message}

{llm_comments}

{search_results}

Critical Proof Principles

1. **No Reliance on "Ready-Made" Theorems**:
- These problems are mathematically nontrivial. Do NOT solve the goal by a single `apply`of a powerful library lemma that encapsulates the entire result.
- Use only basic definitions and elementary lemmas (e.g., rewriting definitions, arithmetic manipulation, manual inequalities).

2. **From First Principles & Step-by-Step**:
- Break the proof into small, explicit steps using `have`, `calc`, or intermediate lemmas.
- Each formal step must correspond to a clear piece of reasoning. If a conclusion requires multiple ideas, split it.

3. **Proof Format**:
- You CAN NOT write new lemma or definition in the ```lean4```blocks.
- You CAN NOT change the original formal statement.
- Only ONE theorem or lemma is allowed in the lean block.
- Make sure to follow all the rules and guidelines provided in the system prompt.

4. **Structure Alignment**:
- Follow the structure of the original proof as closely as possible. Do not introduce a fundamentally different strategy.
- Do not skip the steps included in the original proof. Instead, try to fix the errors while keeping the overall proof structure intact.

Hints for Fixing Errors:
- Pay attention to the mistakes made in history, avoid repeating them.
- When you find that a certain mistake cannot be corrected multiple times, try to change your approach.

C.4.7. COMMENT ON FORMAL PROOF

Role: You are an expert Lean 4 formal verification engineer and mathematician.

Task: Review the Lean 4 formal proof provided below (written by Claude) and provide strategic feedback.

Rules:
1. Scope: Do not comment on the problem statement or the hypothesis of the theorem itself. Focus entirely on the proof script.
2. Logic & Dead Ends: Identify if the proof has entered a ``dead end'' (e.g., goals that are logically unprovable, circular reasoning, or contradictory states).
3. Proof Strategy: Focus on the high-level logic. Suggest more efficient or elegant proof paths. Provide a brief Lean snippet to demonstrate any recommended strategy.
4. Structural Analysis: Identify structural flaws in the proof (e.g., poor sub-goal management, deeply nested tactics, or redundant branching). Suggest better structures using `have`blocks where appropriate.
5. Code Snippets: Use brief Lean 4 code blocks (max 5-10 lines) to demonstrate the most important parts of the refactoring proof.
6. Tactic Tolerance: Do not nitpick specific Mathlib tactic names or minor syntax errors unless they fundamentally break the logic.

Output Format:
1. Numbered List Only: Use a numbered list (1, 2, 3...) for your comments.
2. Conciseness: Keep your feedback brief and technical.
3. No Conversational Filler: Do NOT include any introductory text, concluding remarks, or offers for further help (e.g., do not say ``Would you like me to...'' or ``I hope this helps'').
4. Strict Termination: Your response must end immediately after the final numbered point.
5. No comments on the problem statement or the hypothesis of the theorem itself.
6. You are preferred to use the code block in your comments for the important parts of the refactoring proof.

Autual Task:

```
Comment on the following Lean 4 formal proof:
```lean4
{lean_code}
```
```

D. Full Experimental Results on multiple benchmarks

In this section, we provide extended details and a more granular perspective on the miniF2F, ProverBench, Putnam, and IMO evaluation, including the LLM calls, token usage, time consumption, cost, lemma counts and proof length.

D.1. Putnam 2025 and IMO 2025

To evaluate the capability of MechMath on more challenging mathematical problems, we subject the agent to elite-level competitions, specifically the Putnam 2025 and IMO 2025. Table 7 and Table 8 summarize the problem-by-problem resource consumption across both benchmarks, documenting the total token usage, the total number of interactive environment recompilations, and the total volume of LLM calls. Compared to miniF2F and ProverBench, these advanced undergraduate and international olympiad problems exhibit a substantially higher baseline of structural difficulty, naturally eliciting scaling behavior from our iterative feedback loops. Specifically, for the Putnam 2025 benchmark, MechMath requires an average of 91.3 LLM calls and 1 M tokens per problem. For the more formidable IMO 2025 dataset, the resource demand intensifies further, requiring an average of 643.5 LLM calls alongside approximately 8 M tokens.

Table 7. Detailed inference costs and proof metrics across the problems resolved by MechMath on Putnam 2025.

	A1	A2	A3	A4	A5	A6	B1	B2	B3	B4	B5	B6	Avg
#Tokens (K)	175	531	1,209	1,912	-	836	1,660	663	898	871	2,732	484	1,088
#Sorrify	2	8	16	16	-	8	24	5	10	9	28	13	12.6
LLM Calls	7	35	73	92	-	41	143	59	110	85	302	57	91.3

Table 8. Detailed inference costs and proof metrics across the problems resolved by MechMath on IMO 2025.

	P1	P3	P4	P5	Avg
#Tokens (M)	13.77	4.81	8.34	5.04	7.99
#Sorrify	155	60	104	73	98.0
LLM Calls	981	450	726	417	643.5

Table 9. Proof length of the Putnam 2025 benchmark. The averages of Axiom and Numina are computed exclusive of A5.

	A1	A2	A3	A4	A5	A6	B1	B2	B3	B4	B5	B6	Avg
Hilbert	-	-	-	-	-	-	-	-	1220	-	-	-	1220
Seed	631	469	927	1095	-	881	849	1613	584	628	2499	2594	1161
Axiom	556	458	1089	825	1878	468	1179	346	302	993	1310	862	763
Numina	365	401	422	605	3263	835	328	690	292	648	929	1820	667
MechMath-Easy	339	643	616	792	-	507	-	519	533	700	2022	-	741
MechMath-Hard	**112**	798	560	793	-	**280**	1425	**224**	627	**568**	1256	**578**	**656**

D.2. MiniF2F

Out of the 244 problems, this workflow successfully resolves 242, achieving an impressive 99.2% overall pass rate. Since some problems in miniF2F can be solved directly by Prover in one turn without relying on any informal solution, we exclude them from our data statistics in the following figures and tables, and focus only on the more challenging ones.

Figure 6 and Table 10 detail the performance metrics of the agent on miniF2F across four distinct inference-time dimensions. As illustrated, MechMath demonstrates a massive advantage in reducing computational overhead: to reach its peak pass rate, the agent utilizes a maximum of only 512 LLM calls and 4.73M tokens per problem—representing a 22.1× and 10.3× reduction compared to the 11.3K calls and 48.7M tokens demanded by Hilbert, respectively. The detailed statistics further reveal that our typical operational workload is vastly lighter. Among the successfully solved problems, MechMath requires

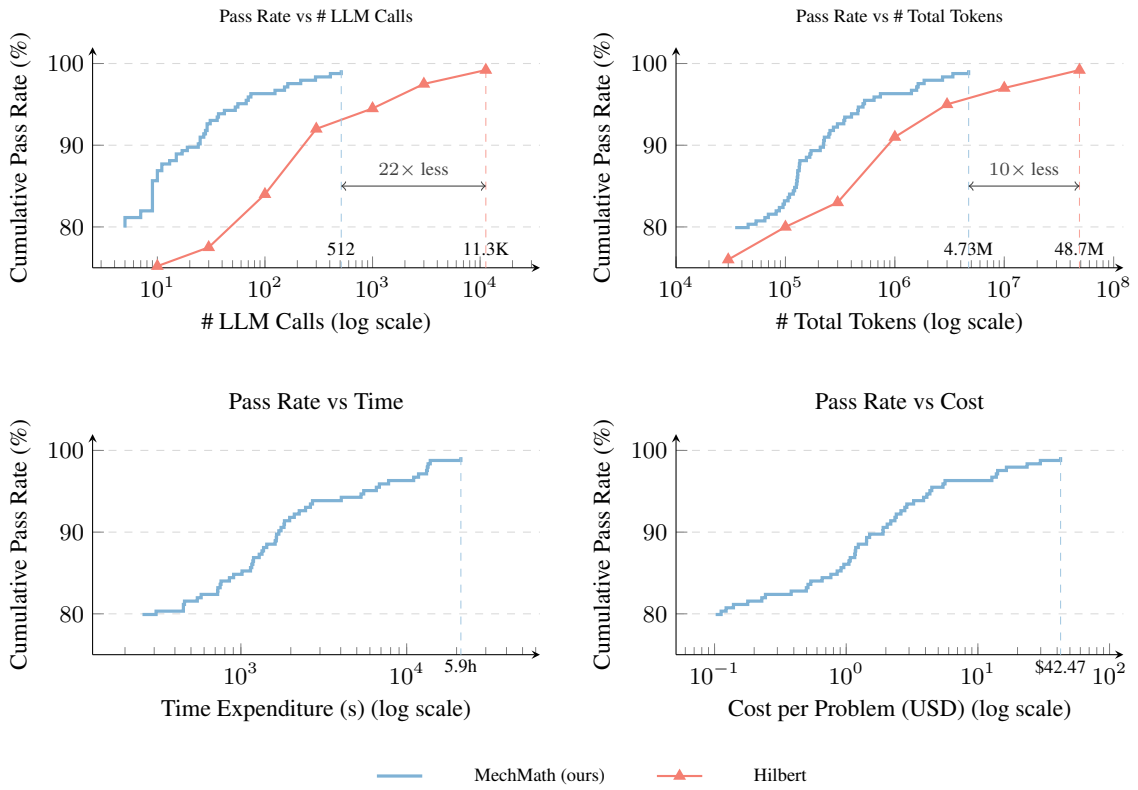


Figure 6. We compare the cumulative pass rate of MechMath against HILBERT as a function of four metrics per sample: (top left) total number of LLM calls, (top right) total token usage, (bottom left) time expenditure, and (bottom right) monetary cost. Note that since the raw data for HILBERT is not publicly available, their reference points are approximated by extracting values at key nodes directly from their original plots.

an average of merely 57.6 LLM calls and 549K tokens. This exceptional efficiency in token and API usage is directly attributed to our systematic workflow: MechMath extracts each failed subproblem into a clean, self-contained context and resolves it independently. This strategy elegantly avoids both the computational waste of full proof regeneration and the excessive context length typically induced by repeated repairs.

This drastic reduction in computational volume translates directly into highly practical monetary costs: even under our maximum budget constraints, the cost is capped at $42.47 per problem, with the actual execution average remaining as low as $4.66. More importantly, regarding absolute time expenditure, the significant acceleration of MechMath is driven not only by the reduced API volume but also by our generated proof structure. Because proofs at the same level can be parallelized—whereas parent nodes in deeply nested structures must sequentially wait for their child nodes to be completed—our shallower proof trees contribute to vastly greater time efficiency, solving a typical problem in approximately 62 minutes (capped at 5.9 hours). This structural conciseness is further evidenced by the proof metrics: successfully generated solutions require an average of only 4.9 lemmas and 232.4 Lean lines. Ultimately, by combining the low-token isolation mechanism with a highly parallelizable shallow architecture, MechMath inherently bypasses the severe time latency and prohibitive API costs associated with the heavy sampling strategies of prior state-of-the-art baselines, yielding proofs that are both exceptionally efficient and highly disciplined.

Table 10. Detailed inference costs and proof metrics across the problems resolved by MechMath on miniF2F (exclude simple prombles).

	# LLM Calls	# Total Tokens	Time (min)	Cost (USD)	# Lemmas	# Lean lines
Avg	57.6	549 K	62.1	4.66	4.9	232.4
Min	5	35 K	4.2	0.10	1	21
Max	512	4.73 M	353.9	42.47	36	1442

D.3. ProverBench

Across evaluations on the 40-theorem number theory subset of ProverBench and the Hilbert subset, MechMath successfully resolved 24/40 problems, significantly outperforming Hilbert's baseline of 16/40. Furthermore, MechMath demonstrates exceptional resource efficiency and operational control: resolving a target theorem requires an average of only 45.2 LLM calls (approximately 336K tokens) at a highly economical cost of $2.97, yielding concise proof structures that average just 4.25 auxiliary lemmas and 166 lines of formal code. Even under the most complex scenarios, the system successfully resolves the problem within an upper bound of 216 calls and 1.62M tokens ($14.89), thanks to our Sorrifier-based isolation mechanism that effectively prevents exponential context growth. This underscores both the effectiveness and efficiency of our proposed framework.

Notably, we also identified four problematic theorems within the benchmark, detailed as follows:

- number_theory__p6: The theorem attempts to formalize rational solutions for the Pell-like equation $x^2 - dy^2 = 1$ parameterized by $t \in \mathbb{Q}$. However, it neglects the constraint where the denominator $d \cdot t^2 - 1 = 0$. In Lean 4, division by zero is syntactically defined but mathematically breaks the intended logic for certain values of t (e.g., when $d = 1, t = 1$). MechMath correctly flagged this as a failure in the stage of informal proof.
- number_theory__p13: This theorem states that all odd divisors d of $5x^2 + 1$ are congruent to $1, 3, 7$, or $9 \pmod{20}$. The formalization uses the integer type $\mathbb{Z}$ but fails to constrain d to positive integers ($d > 0$). Consequently, negative odd divisors like $d = -1$ yield negative remainders under Lean's % operator, which falsifies the disjunction and renders the theorem unprovable without an explicit positivity constraint.
- number_theory__p20: While the natural language docstring claims that no positive integer n can be expressed as the product of six consecutive positive integers, the formal statement defines the variables using the natural number type $\mathbb{N}$ without enforcing $a > 0$ or $n > 0$. This introduces a trivial counterexample at the boundary: $a = 0$ and $n = 0$, where $0 \times 1 \times 2 \times 3 \times 4 \times 5 = 0^5 = 0$. Since a valid counterexample exists, the theorem's goal to prove False is inherently falsified.
- number_theory__p28: The theorem incorrectly asserts that for a prime $p \equiv 3 \pmod 4$, the remainder of $N = \prod_{k=0}^{p-2}(k^2 + 1)$ divided by p is 4. This is mathematically impossible; the true remainder is 2. For instance, if $p = 3$, then $N = (0^2 + 1)(1^2 + 1) = 2$, and $2 \pmod 3 = 2 \neq 4$. The informal verifier consistently rejected all translation attempts for this problem due to this structural contradiction.

Table 11. Detailed inference costs and proof metrics across the problems resolved by MechMath on number theory subsets in ProverBench.

	# LLM Calls	# Total Tokens	Time (min)	Cost (USD)	# Lemmas	# Lean lines
Avg	45.2	336 K	28.8	2.97	4.25	166
Min	5	35 K	5.7	0.02	1	9
Max	216	1.62 M	145.6	14.89	19	536

E. Detailed Analysis of Experiments

E.1. Empirical Evaluation of Proof Structure Preservation

In terms of structural preservation, the Sorrifier maximizes the utility of the already-generated correct proof scaffold during local repairs. Empirically, across 104 successful invocations on miniF2F and 135 on ProverBench, the mechanism achieved an average reuse rate of 74.5% and 71.1% of the original proof skeleton lines, respectively, with individual case reuse rates spanning from 5% to 98%. This quantitative resilience underscores the efficiency of our decomposition strategy; it establishes a critical algorithmic safeguard that prevents extensive segments of sound mathematical reasoning from being completely discarded due to isolated, low-level tactical failures.

E.2. Empirical Evaluation of Subgoal Extraction

To provide a transparent analysis of the failure modes within the Sorrifier mechanism and the extraction pipeline, this section presents a quantitative breakdown of subgoal anomalies across different benchmarks. As described in Section 3.2.3, the

Reasoner is explicitly prompted to identify which hypotheses and contextual conditions in the current tactic state are relevant to the proof goal, discarding only those that are demonstrably irrelevant. This step is designed precisely to prevent the omission of essential assumptions. More importantly, our pipeline does not treat subgoal extraction as a one-shot process. Each extracted subgoal must undergo a two-stage evaluation by the Verifier before being integrated into the proof pipeline:

1. **Syntactic Validation:** Utilizing Lean checks whether the extracted subgoals are well-formed and adhere to the syntactic conventions of the formal language, ensuring that the subgoals can be processed without compilation errors.
2. **Semantic Evaluation:** This stage assesses extraction correctness and logical soundness. Extraction correctness evaluates whether the subgoals accurately reflect the intended reasoning steps, while logical soundness checks whether the subgoal can be proven, typically by examining whether there are obvious counterexamples or essential hypotheses missing.

Subgoals that fail either stage are rejected and returned to the Reasoner for re-extraction. This two-stage evaluation loop serves as a safety net against both missing assumptions and misaligned subtheorems, preventing flawed subgoals from entering the subsequent proof stages and triggering recursive failures or inflated search costs.

The proportion of all three types of errors occurring in all subgoals is summarized in Table 12. The experimental result shows that the error rates of generated subgoals remain minimal on simpler benchmarks like `miniF2F`, but naturally rise when confronting much harder problems such as `Putnam 2025` and `IMO 2025`. Crucially, despite this increase in subgoal anomalies on harder tasks, MechMath consistently maintains its state-of-the-art (SOTA) success rates across the board. This resilient performance demonstrates that the dual-stage evaluation loop effectively intercepts potential logical errors and defects, preventing flawed reasoning from polluting the final proof stream.

Table 12. Quantitative breakdown of error types occurring across all generated subgoals across different benchmarks.

Error Type	Putnam 2025	IMO 2025	miniF2F	ProverBench
Syntax	0.61%	5.60%	0.41%	5.13%
Extraction	3.05%	4.85%	0.00%	1.92%
Logic	1.22%	1.49%	1.23%	7.05%

E.3. Scope of the Sorrifier and Global Regeneration Boundaries

To strike an optimal balance between repair efficiency and search exploration, MechMath implements a progressive "local-decomposition-first, global-regeneration-second" strategy. Upon encountering a proof failure, the system avoids immediately resorting to expensive global reconstruction; instead, it preferentially invokes the Sorrifier to isolate and salvage the localized failure region. Global regeneration is strictly treated as a tactical fallback, executed only when a defect stems from a foundational structural flaw or when local decomposition triggers a severe logical rejection during verification.

Specifically, we define clear functional boundaries based on the nature of proof anomalies, which directly correspond to different runtime decisions of the system:

- **F1 — Leaf-step failure:** In this scenario, the global scaffold and intermediate assertions are entirely sound; the failure is restricted to a single tactic or an unclosable low-level code block. The Sorrifier exhibits maximum efficiency here, precisely localizing the exact block, replacing it with a `sorry` macro, and extracting it as a self-contained independent subgoal to maximize the reuse of the existing scaffold.
- **F2 — Unsound subgoal / False intermediate claim:** This occurs when an intermediate `have` assertion is mathematically false or leads to a logical dead end. In this tier, the Sorrifier's role shifts from "local repair" to **"precise diagnosis"**—it generates a critical `LOGIC_ERROR` signal to halt downstream error propagation, forcing the pipeline to backtrack and invoke global regeneration equipped with targeted historical hints.
- **F3 — Structural / Skeleton defect:** This represents the highest tier of strategic missteps, such as selecting an incorrect induction variable, invalid induction hypotheses, or flawed case splits. Because these defects disrupt the core mathematical architecture and sit entirely beyond the scope of the Sorrifier's local processing, addressing F3 flaws relies exclusively on the upstream module executing a thorough global skeleton reconstruction.

To demonstrate the practical efficacy of the system when confronting F2-level logical flaws, we first introduce the interception mechanism against erroneous global or intermediate structures. During the subgoal split phase, the subgoal assessment serves as the core line of defense against logical deviations. If the context extracted by the Reasoner is incomplete, the Verifier flags an `extract error` to prompt re-extraction. More importantly, if the extracted subgoal is diagnosed with a `logic error` (meaning the premises contradict the goal or cannot entail it), the agent is explicitly prohibited from attempting futile local repairs; instead, the system triggers an `error fix` state on the overarching formal proof body, forcing it to correct the underlying logical flaw.

This mechanism plays a critical interception role during the empirical evaluation of the challenging **Putnam 2025 A2** problem. During the proof process, the agent proposes an intermediate bound requiring it to prove the following inequality:

$$\frac{1}{\pi}y(\pi - y) \leq \frac{2}{\pi}y, \quad \forall y \in \left[0, \frac{\pi}{2}\right]$$

The Verifier mathematically invalidates this inequality through counterexample computation (e.g., substituting $y = 0.5$ yields a left-hand side of 0.421, which is greater than the right-hand side of 0.318) and decisively flagged it as a `logic error`. This successfully intercepts a doomed local proof search and forced the pipeline to backtrack to the formal proof spine for structural reorganization.

Empirical statistics on the miniF2F benchmark further confirm the necessity of this module: across the evaluation, the Verifier successfully intercepts a `logic error` 5 times across 3 distinct problems and corrected an `extract error` once. By seamlessly integrating rigorous upstream global planning with real-time logical scrutiny of formal subgoals, MechMath effectively neutralizes theoretical and logical defects.

Furthermore, to clearly delineate the dynamic flow between local repair (Sorrifier) and global regeneration when confronting intertwined high-level F3 skeleton defects and F1 leaf-step failures, we provide an induction-heavy case study (from ProverBench, `number_theory__p31`) that relies heavily on global structure.

Stage 1 — Wrong induction hypothesis (F3-level global skeleton defect) The agent initially attempted strong induction directly on the predicate. Because the subsequent steps require applying the induction hypothesis to `n.minFac`, and the current induction hypothesis (`ih`) fails to expose the statement in this form, the inductive step cannot be closed. This represents an architecture-breaking F3-level defect where no leaf-level `sorry` replacement can recover from the directional misstep.

```
induction n using Nat.strongInductionOn with
| ind n ih =>
  intro hn
  let a := n.minFac
  have ha_prime : a.Prime := Nat.minFac_prime (by omega)
  have ha_dvd : a ∣ n := Nat.minFac_dvd n   -- Need IH on n/a, but ih cannot provide it
```

Stage 2 — Regenerated correct skeleton Since the proof cannot be locally patched, the system decisively abandons localized repair and triggers global regeneration. By restructuring the proof architecture and strengthening the induction hypothesis, the system introduces an explicit bound n and inducts on it, successfully generalizing the claim to hold for all $m \leq n$. At this point, the global induction skeleton becomes completely sound.

```
have H3_aux : ∀ n m : ℕ, m ≤ n → m % 4 = 3 → ∃ q : ℕ, q.Prime ∧ q % 4 = 3 ∧ q ∣ m := by
  intro n
  induction n with
  | zero =>
    intro m hm_le hm_mod
    have : m = 0 := by omega   -- Base case: vacuous
    omega
```

Stage 3 — Sorrifier local repair (F1-level leaf-step failure) Armed with a correct global induction skeleton, the remaining failure reduces to a single isolated leaf fact, downgrading the error to an F1 tier. Here, the Sorrifier perfectly preserves the fully generated induction architecture and replaces **only** the failing leaf step with a `sorry` macro, extracting it as a standalone subgoal lemma to be resolved independently, thereby eliminating the computational overhead of redundant global reconstruction.

```
have hm_cases : m = n + 1 ∨ m ≤ n := by omega
rcases hm_cases with rfl | m_le_n
· let a := (n + 1).minFac
  have ha_prime : a.Prime := Nat.minFac_prime (by omega)
  obtain ⟨b, hab⟩ : ∃ b, n + 1 = a * b := Nat.minFac_dvd (n + 1)
  have hb_pos : 0 < b := by sorry   -- ← Localized leaf gap; the scaffold above is perfectly reused
```