FIPO-Prover: Formalization-Oriented Informal Proof Optimization for Efficient Formal Theorem Proving

Jingkun Ma [*1] **Yuchao Wang** [*1] **Yujia Huo** [2] **Derek F. Wong** [1]

Abstract

Informal proofs provide a natural interface for neural theorem proving, allowing large language models to reason in natural language before producing formal proof scripts. However, a persistent informal-to-formal gap remains: mathematically plausible reasoning may contain steps, dependencies, or intermediate claims that do not align with what a formalizer such as Isabelle can verify. We introduce **FIPO-Prover**, a framework for Formalization-Oriented Informal Proof Optimization that represents informal proofs as structured *Formalization-oriented Informal Proof* (FIP) states organized by step-level causal pairs. FIPO-Prover performs verification-guided graph search over FIP states, combining local revision, causal-pair expansion, branch pruning, upstream causal backtracking, and memory-augmented search to make informal reasoning more projection-friendly. On MiniF2F-Test, FIPO-Prover achieves 73.77% pass@1 and 80.33% pass@3, outperforming direct LLM formalization baselines and prior search-based frameworks. Our analysis shows that these gains arise not from additional sampling alone, but from optimizing intermediate proof states so that model-generated reasoning becomes more compatible with formal verification.

1. Introduction

Formal theorem proving such as Isabelle (Paulson, 1994) and Lean (De Moura et al., 2015) offers machine-checkable mathematical correctness, but constructing formal proofs remains a major bottleneck. Modern large language models (LLMs) can produce plausible informal arguments and increasingly fluent proof-assistant code (Polu & Sutskever, 2020; Yang et al., 2023; Xin et al., 2024a; Ren et al., 2025). Prior work has shown the value of sketches, retrieval, and decomposition for bridging informal reasoning and formal proof construction (Jiang et al., 2022b; Wang et al., 2023; Zhao et al., 2024; Varambally et al., 2025), yet informal-to-formal theorem proving is brittle because formal verification still exposes a representation mismatch.

Informal proofs often compress intermediate steps and omit dependencies, whereas formal proofs require explicit local context, traceable dependencies, and executable proof structure (Jiang et al., 2022b; Gao et al., 2025; Varambally et al., 2025; Liu et al., 2026a). Many observed Isabelle errors appear in the projected proof script, but their causes lie upstream in the informal proof structure.

This paper introduces *Formalization-oriented Informal Proof* (FIP), a structured informal proof state designed for formalization. A FIP-state makes proof steps explicit, exposes local causal relations, tracks dependencies, and aligns verifier feedback with the proof state. In this work, FIP-states are organized around step-level causal pairs and serve as the core state of a diagnosable, repairable, and traceable proof search graph.

Building on this representation, we propose **FIPO-Prover**, a verification-guided FIP search graph framework for informal-to-formal theorem proving. FIPO-Prover constructs the root FIP-state, uses a FIP-to-FP Projection Map to induce a formal proof candidate, and uses verifier feedback to update the FIP search graph. Nodes are FIP-states and edges are recovery transitions, including Root FIP-state Initialization, Local FIP-state Revision, Projected-Candidate Repair, Causal-Pair Expansion, Upstream Causal Backtracking, Branch Pruning, and a Memory-Augmented FIP Search Policy. Local FIP-state Revision updates causal pairs and dependencies inside the proof state, while Projected-Candidate Repair updates local artifacts in the projected formal candidate. Together, these transitions turn Isabelle feedback into a control signal for structured optimization over FIP-states.

We evaluate FIPO-Prover on MiniF2F-Test (Zheng et al., 2021) under strict Isabelle verification. The system achieves **73.77% pass@1** and **80.33% pass@3**, substantially outper-

[*]Equal contribution [1]NLP2CT Lab, University of Macau, Macau, China [2]School of Data Science and Information Engineering, Guizhou Minzu University, Guizhou, China. Correspondence to: Derek F. Wong <derekfw@um.edu.mo>.

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).

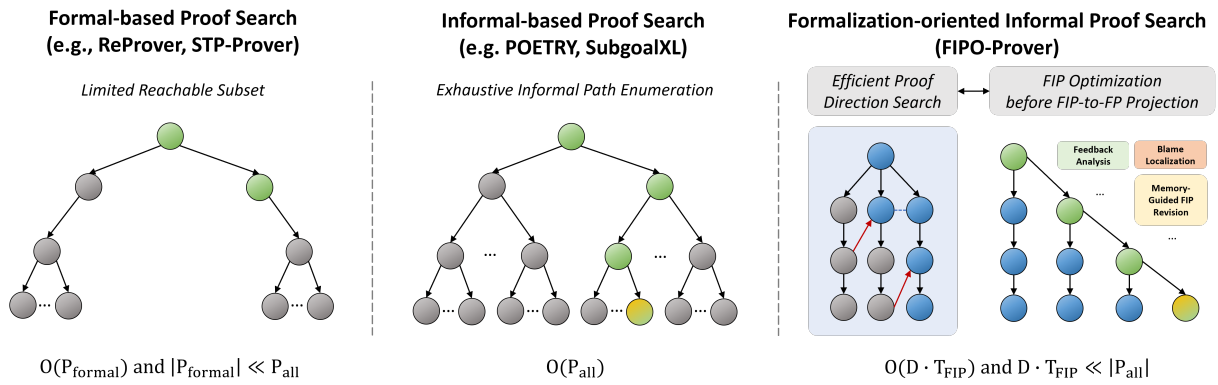


Figure 1. Conceptual comparison of proof-search paradigms. Direct formal search explores the constrained formal-proof space with cost $O(|P_{\text{formal}}|)$, where $|P_{\text{formal}}| \ll |P_{\text{all}}|$ but direct generation is brittle. Informal Proof search explores the broader proof space P_{all} with exhaustive cost $O(|P_{\text{all}}|)$. FIPO-Prover instead searches over proof directions in an evolving FIP graph and optimizes FIP-states before FIP-to-FP projection, yielding an effective cost $O(D \cdot T_{\text{FIP}})$, where D is the number of explored proof directions and T_{FIP} is the per-direction FIP optimization cost. The target regime is $D \cdot T_{\text{FIP}} \ll |P_{\text{all}}|$, supported by feedback analysis, blame localization, and memory-guided FIP revision.

forming direct LLM formalization baselines and previously reported search- or decomposition-based frameworks, including baselines with larger pass@N budgets. Beyond final success rate, our analysis shows that final FIP-states have higher downstream formalization utility, revisions make proof states more explicit and projection-friendly, multiple recovery transitions provide complementary recoveries, and the full system remains efficient under matched FIP-state budgets.

Our contributions are as follows:

- We introduce **Formalization-oriented Informal Proof** (FIP), a structured informal proof state represented through step-level causal pairs for projection, diagnosis, repair, and optimization.
- We formulate informal-to-formal theorem proving as **verification-guided FIP search graph optimization**, where nodes are FIP-states and edges are verifier-feedback-driven recovery transitions.
- We develop FIPO-Prover as a graph-centric search system that combines FIP-state initialization, projection, localized revision, candidate repair, expansion, backtracking, pruning, and memory-augmented transition control.
- We demonstrate strong MiniF2F-Test performance and provide analyses showing that the gains arise from FIP-state optimization and coordinated search control.

2. Methodology

We formulate informal-to-formal theorem proving as verification-guided search over structured informal proof states. Given a theorem statement, **FIPO-Prover** constructs a *Formalization-oriented Informal Proof* (FIP) state and iteratively optimizes an evolving FIP search graph under Isabelle feedback.

2.1. Problem Formulation

Let T denote an input theorem and $\mathcal{V}_{\text{Isa}}$ the Isabelle verifier. The goal is to produce a proof script $P^\star$ such that $\mathcal{V}_{\text{Isa}}(P^\star) = 1$. Direct generation attempts a one-step mapping $T \mapsto P^\star$. **FIPO-Prover** instead introduces an intermediate FIP-state r and searches a dynamic proof graph

$$\mathcal{G} = (\mathcal{N}, \mathcal{E}) \tag{1}$$

where each node $n \in \mathcal{N}$ stores a FIP-state and each edge $e \in \mathcal{E}$ is a recovery transition. Search succeeds when some node induces a proof candidate accepted by Isabelle. Thus verification failure becomes a structured signal for revising, expanding, pruning, or redirecting the current FIP graph.

2.2. Formalization-oriented Informal Proof States

A FIP-state is a structured informal proof designed for reliable projection, diagnosis, repair, and optimization. Given a theorem T, we represent a FIP state as

$$r = (\mathcal{C}, G) \tag{2}$$

where $\mathcal{C} = \{c_i\}_{i=1}^{n}$ is a set of step-level causal pairs and $G = (\mathcal{C}, E)$ is the dependency graph connecting them. The edge $(c_j, c_i) \in E$ indicates that step c_i depends on step c_j.

Each causal pair is written as

$$c_i = (x_i, y_i, d_i, s_i, h_i) \tag{3}$$

where x_i is the *cause side*, y_i is the *effect side*, d_i records dependencies on earlier causal pairs, s_i stores verifier-aligned state, and h_i stores revision history. The cause side contains

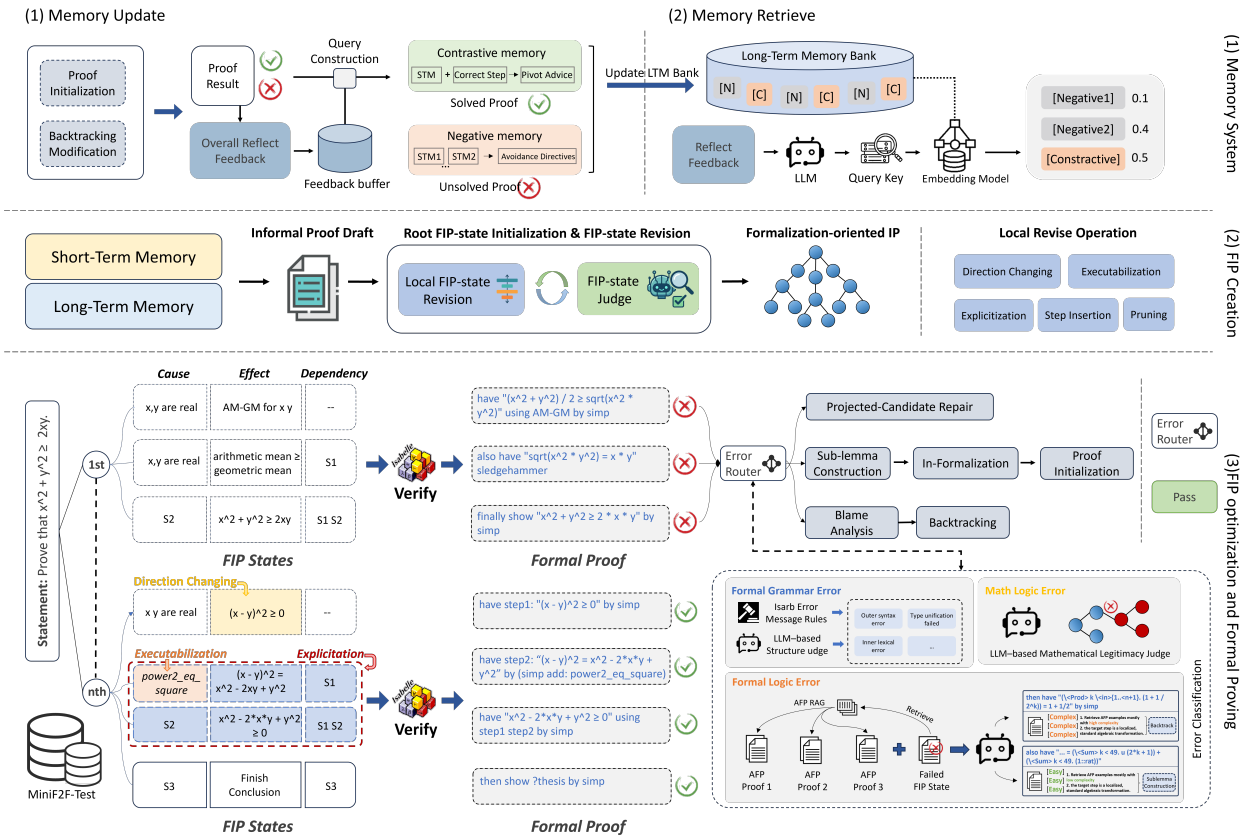


Figure 2. Overview of FIPO-Prover. FIPO-Prover performs verification-guided optimization over an evolving Formalization-oriented Informal Proof (FIP) graph, narrowing the gap between informal proof states and formal Isabelle proofs before projection. Verifier feedback is analyzed and routed to the FIP graph, enabling the policy to select effective recovery transitions through feedback analysis and blame localization. The resulting FIP-states become more projection-ready, reducing the difficulty of FIP-to-FP formalization.

local premises, contextual facts, or previously established claims, whereas the effect side contains the claim, subgoal, or intermediate result derived by the step. The graph G is induced by the dependency fields $\{d_i\}_{i=1}^{n}$.

This representation makes an informal proof locally diagnosable and repairable. When projection to Isabelle fails, verifier feedback can be localized to the responsible causal pair and the problematic component. Isabelle-oriented choices are generated only during projection or Projected-Candidate Repair.

The initial node is produced by *Root FIP-state Initialization*. The system generates an initial informal proof i_0 and initializes $r_0 = f_{\text{struct}}(T, i_0)$, where f_{struct} converts the informal proof into the FIP representation. An LLM judge iterates this initialization until the representation is explicit enough for projection and diagnosis or the budget is exhausted. Criteria and limits are detailed in Appendix A.4. In subsequent search, FIP-state optimization continually narrows the gap between causal pairs and the final formal proof, making the informal structure increasingly projection-friendly and verifier-aligned.

2.3. Projection and Normalized Verifier Feedback

At each search step, the current FIP-state is projected into an Isabelle proof candidate $p_t = f_{\text{proj}}(T, r_t)$. The FIP-to-FP Projection Map maps causal pairs and dependency structure into Isabelle proof constructs such as local facts, Isar blocks, have/show statements, tactic calls, or auxiliary lemmas. Isabelle returns verification feedback $(y_t, \epsilon_t) = \mathcal{V}_{\text{Isa}}(p_t)$.

We normalize heterogeneous Isabelle errors into internal error types. Errors are mapped back to affected causal pairs and routed to recovery transitions. Appendix Table 5 lists the error categories. A lightweight router further separates proof-script malformations from mathematical gaps before transition selection. The detailed routing logic is described in Appendix D.1.

2.4. Feedback-Driven FIP Graph Transitions

Given a FIP-state r_t and normalized feedback ϵ_t, the FIP Graph Transition Policy selects a recovery transition.

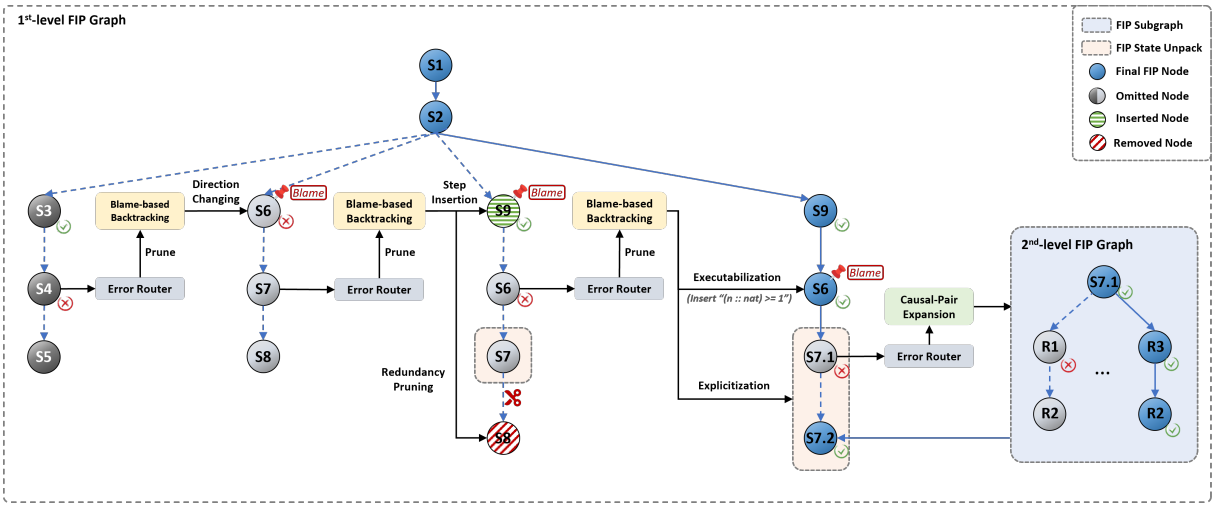


Figure 3. Search-control strategies for efficient FIP optimization based on feedback-driven FIP graph transitions.

Local FIP-state Revision Starting from the localized blame FIP-state, the policy revises affected causal pairs by adjusting their cause/effect sides, dependency links, or structural granularity. The blame state can be the currently failing node, or an upstream state identified by Upstream Causal Backtracking through the dependency graph. This revision mainly instantiates the following FIP transformation types:

- **Explicitization**: making compressed or implicit local reasoning in the current proof state more explicit.
- **Direction Changing**: changing the overall proof direction when the current route is difficult to recover.
- **Step Insertion**: adding intermediate steps or bridge claims to complete the FIP-state.
- **Redundancy Pruning**: removing redundant, ineffective, or repeatedly failing fragments of the FIP-state.
- **Executabilization**: adding executable formal expression in the FIP-state, directly rewriting FIP-state closer to the projected formal proof format.

Detailed criteria and statistics of these transformations are provided in Appendix D.

Projected-Candidate Repair Repairs are performed on the Isabelle candidate induced by current FIP-state. It mainly handles shallow errors such as syntax errors, type annotation errors, naming errors, or local reference errors, while keeping the upstream causal structure fixed. This transition does not repair tactic choices. Tactic-related failures are routed to other recovery transitions. When Error Routing classifies the failure as a non-tactic shallow error, the system performs at most $k = 3$ Projected-Candidate Repair attempts to quickly correct shallow errors produced by the LLM. The implementation details of Error Routing are described in Appendix D.1.

Causal-Pair Expansion Expansion Transition is triggered when the current step fails to project into a verifiable local proof. It expands an overly coarse causal pair

$$x_i \Rightarrow y_i$$

into a finer local causal subgraph, such as

$$x_i \Rightarrow z_1 \Rightarrow \cdots \Rightarrow y_i$$

Concretely, the system uses the cause side, effect side, and dependency links from the complete current FIP chain to construct new claims and definitions. It also re-indexes established facts, and regularizes the result into Isabelle-compatible sub-lemmas. In addition, it deduplicates equivalent proof goals to avoid repeatedly attempting the same target, thereby improving proof search efficiency.

Branch Pruning Branch Pruning marks low-promise FIP branches as inactive. It is triggered when retry, backtracking, or depth limits are reached, or when Error Router rules out shallow grammar errors and determines that the current FIP-state is too complex to be directly decomposed and solved. In particular, **FIPO-Prover** retrieves related proof fragments from the Archive of Formal Proofs (AFP) (Eberl et al., 2024) via RAG and estimates the expandability of the current FIP-state by analyzing the difficulty of fragments that use similar proof syntax or strategies. The branch labeled with low expandability under the current budget is marked as inactive. The Evidence-based Branch Expandability Analysis is detailed in Appendix D.3.

Category	Method	Metric	MiniF2F-Test	MiniF2F-Valid
Classical	Sledgehammer + heuristics	pass@1	20.9%	18.0%
Classical	Thor(Jiang et al., 2022a)	pass@1	29.9%	28.3%
Classical	Thor + expert iteration(Wu et al., 2022)	pass@1	35.2%	37.3%
Classical	Thor + Magnushammer(Mikuła et al., 2023)	pass@1	37.3%	36.9%
Direct LLM	Gemini 3.1 Pro	pass@1	54.51%	–
Direct LLM	Gemini 3 Flash	pass@1	43.85%	–
Direct LLM	GPT-5-Codex	pass@1	32.79%	–
Direct LLM	DeepSeek-V3.2-Exp-Reasoner	pass@1	38.52%	–
Direct LLM	DeepSeek-V3.2-Exp-Chat	pass@1	34.02%	–
Framework	DSP(Jiang et al., 2022b)	pass@100	39.3%	42.6%
Framework	POETRY(Wang et al., 2024)	pass@1	42.2%	42.2%
Framework	LEGO-Prover(Wang et al., 2023)	pass@100	50.0%	57.0%
Framework	SubgoalXL(Zhao et al., 2024)	pass@16384	56.1%	61.9%
Framework	HybridProver(Hu et al., 2025)	pass@128	59.4%	–
Framework	**Ours**	**pass@1**	**73.77%**	**81.55%**
Framework	**Ours**	**pass@3**	**80.33%**	–

Table 1. Results are reported on MiniF2F Isabelle test and validation splits, with each baseline kept under its originally reported sampling budget. Validation results are included when available.

Upstream Causal Backtracking When verifier feedback indicates that the failure cannot be resolved by local repair around the current step, **FIPO-Prover** propagates the feedback through the dependency graph of the FIP-state to locate a blame ancestor. Upstream Causal Backtracking returns to the blamed upstream causal pair or graph node, revises the corresponding FIP-state, and regenerates the downstream branch. This transition prevents the system from spending the budget on local repairs under unreliable premises or an unsuitable proof direction.

2.5. Memory-Augmented FIP Search Policy

To reduce the effective cost of FIP graph search, **FIPO-Prover** introduces a dynamically updated Memory System. Memory supports the FIP Graph Transition Policy in choosing proof directions more efficiently and avoiding repeated or Isabelle-unfriendly FIP-states. The Memory System consists of Long-term Memory (LTM) and Short-term Memory (STM). LTM stores contrastive memory across theorems, including successful proof experiences and their corresponding trial-and-error paths. These memories provide contrastive signals over successful proof directions, failed transitions, and effective recovery paths under similar verifier feedback. STM is a theorem-local stack that records all errors, failed transitions, visited FIP-states, pruned branches, and repeated error tags observed during the current proof attempt. LTM and STM jointly condition upstream causal backtracking, local FIP-state revision, and branch selection, reducing redundant exploration and improving pre-projection FIP optimization. Database construction, deduplication, top-k retrieval, and query design are detailed in Appendix E.

3. Experiments

3.1. Experimental Setup

Benchmark and Evaluation We evaluate on the Isabelle test split of MiniF2F (Zheng et al., 2021) using Isabelle2022 for formal verification through PISA (Jiang et al., 2021). MiniF2F consists of 488 high-school competition-level problems, split into validation and test sets, each with 244 problems. The standard proof answers are never used for prompting, retrieval, memory construction, or system tuning. The final evaluation set contains 244 Isabelle theorem statements. Candidate proofs are executed step by step, and a theorem is counted as solved only when PISA reports a finished proof state. We use no weak matching, string similarity, or human judgment as correctness criteria. We utilize pass@N to report theorems accepted by Isabelle verifier within the first N complete runs.

System configuration Unless otherwise specified, the full system includes Root FIP-state Initialization, formal projection, Local FIP-state Revision, Projected-Candidate Repair, Causal-Pair Expansion, Branch Pruning, Upstream Causal Backtracking, and a Memory-Augmented FIP Search Policy. Search terminates when a candidate is verified, the budget is exhausted, or all active branches are pruned. Transition-

Informal Source	DeepSeek-V3.2-Exp-Reasoner	GPT-5-Codex	Gemini 3.1 Pro	+ Lightweight Postprocess
DeepSeek-V3.2-Exp-Reasoner	24.59%	25.82%	44.26%	–
GPT-5-Codex	25.82%	23.77%	44.26%	–
Gemini 3.1 Pro	26.64%	24.59%	47.95%	54.10%
DSP(Jiang et al., 2022b)	25.82%	26.64%	43.03%	–
LEGO-Prover(Wang et al., 2023)	26.23%	29.51%	40.16%	–
Ours (Initial / Flash)	18.44%	20.90%	45.49%	55.33%
Ours (Final)	**24.59%**	**22.54%**	**53.69%**	**61.48%**

Table 2. Downstream formalization success from different informal proof sources under a fixed formalization protocol. Final FIP states are most beneficial for Gemini 3.1 Pro, the target formalizer used in FIPO-Prover for FIP optimization.

Ablation	MiniF2F-Test	Absolute Drop
Full	73.77%	–
w/o CPE	69.20%	4.57%
w/o CPE+UCB/Mem	63.11%	10.66%
w/o CPE+UCB/Mem+PCR	57.38%	16.39%
w/o CPE+UCB/Mem+PCR+LFR	45.49%	28.28%

Table 3. Cumulative ablation of FIP graph transitions. Abbreviations denote transitions removed from the FIP search graph. CPE: Causal-Pair Expansion, UCB/Mem: memory-guided Upstream Causal Backtracking, PCR: Projected-Candidate Repair. LFR: Local FIP-state Revision.

policy limits, imported libraries, model routing, and backend timeouts are detailed in Appendix A.

3.2. Baselines

We compare **FIPO-Prover** with three groups of baselines. Classical automated proving systems and heuristic methods: Sledgehammer + heuristics, Thor (Jiang et al., 2022a), Thor + expert iteration (Wu et al., 2022), and Thor + Magnushammer (Mikuła et al., 2023). Direct LLM formalization baselines: Gemini 3.1 Pro (DeepMind, 2026a), Gemini 3 Flash (DeepMind, 2026b), GPT-5-Codex (OpenAI, 2025), DeepSeek-V3.2-Exp-Reasoner, and DeepSeek-V3.2-Exp-Chat (DeepSeek-AI, 2025). Search- or decomposition-based frameworks: DSP (Jiang et al., 2022b), POETRY (Wang et al., 2024), LEGO-Prover (Wang et al., 2023), SubgoalXL (Zhao et al., 2024), and HybridProver (Hu et al., 2025). These baselines cover prover-side automation, direct proof-script generation, and high-budget search or decomposition pipelines.

3.3. Main Results

Table 1 summarizes the main results on the MiniF2F Isabelle split. **FIPO-Prover** achieves **73.77% pass@1** on MiniF2F-Test and **81.55% pass@1** on MiniF2F-Valid, establishing a new state of the art on this benchmark split. It substantially outperforms classical Isabelle systems, direct LLM formalization baselines, and prior search/decomposition frameworks.

Against the strongest direct LLM baseline, Gemini 3.1 Pro at 54.51% pass@1, **FIPO-Prover** improves by **19.26 percentage points** on MiniF2F-Test. This large gap highlights the benefit of optimizing intermediate FIP-states before formal projection, rather than directly sampling formal scripts from the base model.

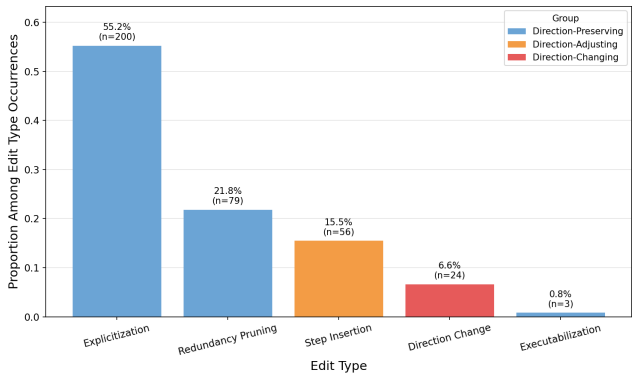
The comparison with high-budget frameworks is also informative. SubgoalXL reports 56.10% pass@16384 and HybridProver reports 59.4% pass@128, while **FIPO-Prover** obtains 73.77% in the pass@1 setting. Because these systems use different search budgets, we treat this as a contextual comparison. Nevertheless, it suggests that the improvement is not simply explained by increasing the number of independent proof attempts. For **FIPO-Prover**, pass@N improves from 73.77% at pass@1 to 80.33% at pass@3. Section 4.3 further analyzes this behavior under different post-hoc budgets.

4. Analysis

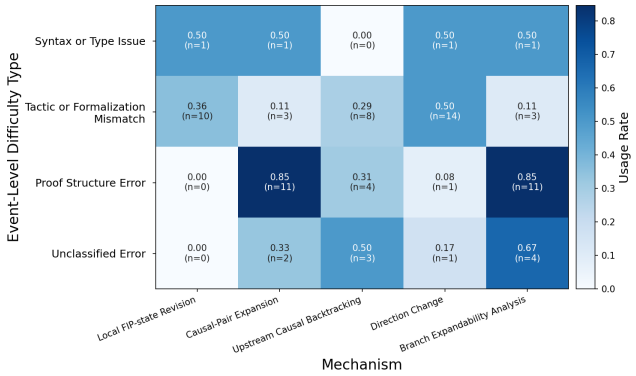
4.1. Formalization Utility of FIP-states

Direct Formalization from FIP-states We evaluate whether the optimized FIP-states remain useful outside the full **FIPO-Prover** search loop. To isolate the role of the proof representation, we fix the downstream formalization protocol and vary only the informal source. A higher success rate from final FIP-states indicates that search improves the projection readiness of the intermediate representation itself. We also report a lightweight postprocessing setting that allows bounded Projected-Candidate Repair and Causal-Pair Expansion, while disabling the full FIP search graph and unrestricted Upstream Causal Backtracking.

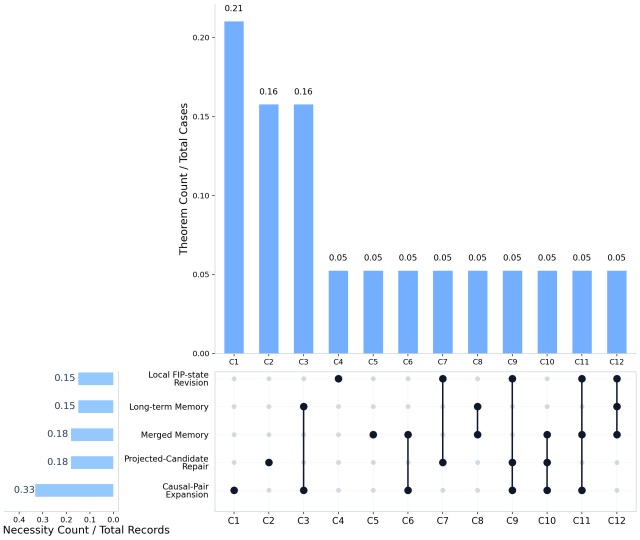
Table 2 shows that final FIP-states are most effective with Gemini 3.1 Pro as the downstream formalizer. Under Gemini 3.1 Pro, **Ours (Final)** reaches **53.69%**, outperforming all other informal sources, and further improves to **61.48%** after lightweight postprocessing. In contrast, DeepSeek and GPT-5-Codex do not uniformly benefit from the final FIP-states. This behavior is expected: **FIPO-Prover** optimizes FIP-states through a Gemini-Pro-centered formalization loop, thus iterative recovery and pruning favor representations that are easier for this target formalizer to project



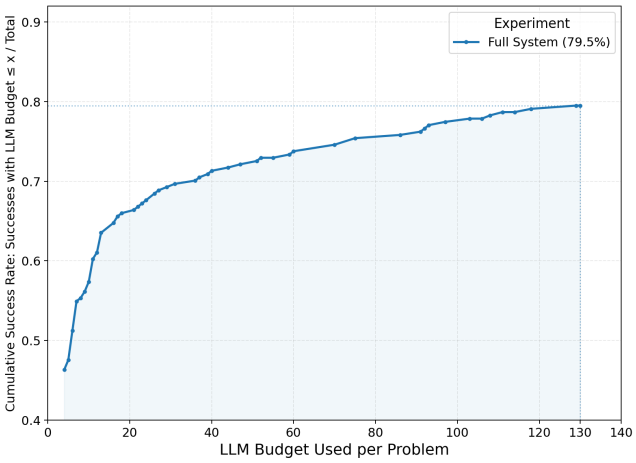
(a) Statistics of FIP transformation grouped by proof-strategy revision categories.



(b) Recovery-transition usage conditioned on verifier error types.



(c) Theorem-level necessity of FIP graph transitions in recovered successes.



(d) Cumulative verified success of FIPO-Prover under matched LLM budgets.

Figure 4. Diagnostics of FIPO-Prover's search and recovery behavior.

into Isabelle. FIP variants that might be more suitable for other formalizers can be discarded if they remain difficult for the target projector. These results suggest that FIP optimization is best viewed as target-formalizer-conditioned representation optimization.

Revision Behavior and Structural Diagnostics To understand how FIP-states improve, we analyze successful trajectories at the revision level in Figure 4a. We classify FIP revisions by transition type and by proof-strategy effect, distinguishing direction-preserving, direction-adjusting, and direction-changing revisions. **FIPO-Prover** primarily improves local proof representations rather than repeatedly changing the top-level proof direction. Explicitization dominates the revision distribution at 55.20%, and direction changes account for only 6.60%. This suggests that many failures can be recovered by making FIP-states more explicit, local, and projection-friendly. Structural analysis demonstrates optimization mainly explicitizes coarse single-step reasoning, reducing the local projection burden from FIP-states to formal Isabelle proofs.

Structural Diagnostics for Non-Direction-Changing Revisions We analyze how FIP optimization alters proof structure when the top-level mathematical direction is preserved. Specifically, we measure per-edit structural changes for non-direction-changing revisions, aiming to determine whether optimization makes FIP states more explicit, locally checkable, and projection-friendly for downstream Isabelle formalization. Detailed metric definitions are provided in Appendix F.2. Figure 5 reveals a consistent explicitization effect: explicit-step gain increases and total proof length grows, whereas the number of intermediate assertions remains nearly stable or slightly decreases. This suggests that FIP optimization does not mainly alter the global proof skeleton or introduce additional high-level assertions. In-

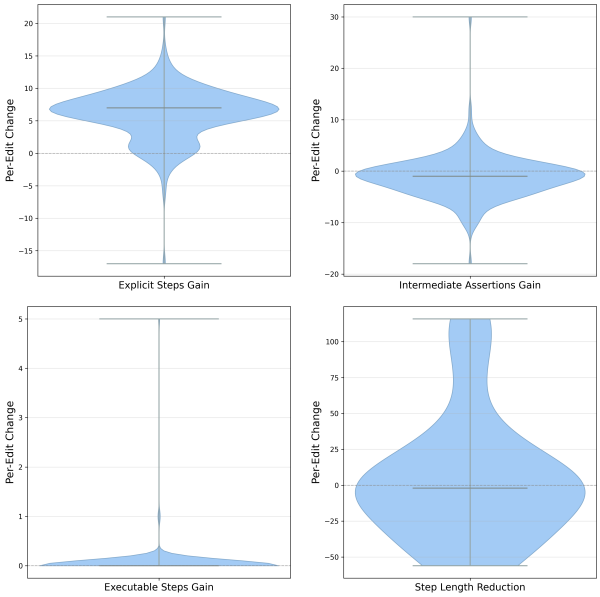


Figure 5. Structural changes in non-direction-changing FIP revisions. These diagnostics measure whether local revisions make proof states more explicit and FIP-to-FP projection-friendly.

stead, it decomposes coarse single-step reasoning into finer local transitions, reducing the burden on the formalizer when projecting FIP states into Isabelle proofs.

4.2. Contribution of Recovery Transitions

Ablation Analysis We quantify the contribution of recovery transitions through cumulative ablations over the FIP search graph. As shown in Table 3, removing Causal-Pair Expansion already degrades performance, indicating the importance of introducing missing intermediate reasoning during projection. Further disabling memory-guided Upstream Causal Backtracking, Projected-Candidate Repair, and Local FIP-state Revision leads to increasingly larger drops. These results suggest that **FIPO-Prover** benefits from the interaction of complementary recovery transitions, rather than from any single repair mechanism alone.

Theorem-level Necessity Analysis Beyond aggregate ablations, we perform theorem-level necessity analysis to identify which transitions are required for each newly solved theorem in Figure 4c. A transition is necessary for a theorem T if the full system solves T, the reference system does not, and removing that transition makes T fail. Formal metric definitions are provided in Appendix H.2.

The analysis also captures the multi-step nature of recovery. A theorem may require Causal-Pair Expansion to split a coarse reasoning pair, Upstream Causal Backtracking to revise an upstream claim, and Projected-Candidate Repair to complete the projected Isabelle proof.

Verifier Feedback and Recovery Transitions We analyze how verifier feedback shapes recovery behavior by measuring which transitions appear in successful recovery paths for each normalized error type. This diagnostic links formal verification failures to subsequent search decisions. The event-level metric is defined in Appendix H.2. Figure 4b shows that different verifier failures induce distinct recovery patterns. Proof-structure errors are primarily handled by graph-level operations such as Causal-Pair Expansion and Branch Expandability Analysis, while tactic or formalization mismatches trigger broader transitions. Ambiguous or unclassified failures rely more heavily on branch reassessment and backtracking. The overall trend indicates that **FIPO-Prover** uses verifier feedback to select targeted recovery transitions rather than treating failures as undifferentiated retries.

4.3. Budget-Efficiency Profile

We evaluate the budget-efficiency profile of the full **FIPO-Prover** system under a matched post-hoc LLM budget. For each problem, we record the number of LLM Transition-Policy invocations before the first verified proof. For a threshold x, Success@Budget is the fraction of benchmark problems solved within at most x LLM calls, measuring how quickly the system reaches verifiable proofs under fixed resources.

As shown in Figure 4d, **FIPO-Prover** solves a large fraction of instances with a modest budget: the cumulative success rate reaches approximately 65% within about 20 calls and gradually approaches the final success rate of 79.5%. Since this early regime accounts for over 80% of the eventually solved instances, the curve suggests that **FIPO-Prover** does not rely mainly on long-tail retries, but benefits from effective search control over structured FIP-states.

5. Conclusion

We presented **FIPO-Prover**, a verification-guided framework that reformulates informal-to-formal theorem proving as optimization over an evolving Formalization-oriented Informal Proof (FIP) search graph. **FIPO-Prover** addresses the gap between informal reasoning and formal verification by converting informal proofs into diagnosable, repairable, and projection-friendly intermediate states. Through step-level causal pairs and verifier feedback from Isabelle, the system enables targeted revision, repair, expansion, pruning, upstream backtracking, and memory-guided search. Experiments demonstrate improved formalization reliability and search efficiency over direct generation and prior decomposition-style methods. These results highlight intermediate proof optimization as a promising direction for scalable formal proof search.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning and Formal Theorem Proving. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

Acknowledgements

This work was supported in part by the Science and Technology Development Fund of Macau SAR (Grant Nos. FDCT/0007/2024/AKP, EF2024-00185-FST), the UM and UMDF (Grant Nos. MYRG-GRG2024-00165-FST-UMDF, MYRG-GRG2025-00236-FST), the Tencent AI Lab Rhino-Bird Research Program (Grant No. EF2023-00151-FST), the Dr. Stanley Ho Medical Development Foundation (Grant No. SHMDF-AI/2026/001), and the National Natural Science Foundation of China (Grant No. 62266013).

References

Baba, K., Liu, C., Kurita, S., and Sannai, A. Prover agent: An agent-based framework for formal mathematical proofs. *arXiv preprint arXiv:2506.19923*, 2025.

Cao, C., Song, L., Li, Z., Le, X., Zhang, X., Xue, H., and Yang, F. Reviving dsp for advanced theorem proving in the era of reasoning models. *arXiv preprint arXiv:2506.11487*, 2025.

De Moura, L., Kong, S., Avigad, J., Van Doorn, F., and von Raumer, J. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pp. 378–388. Springer, 2015.

DeepMind. gemini-3.1-pro-preview. In *DeepMind*, 2026a. URL https://docs.cloud.google.com/gemini-enterprise-agent-platform/models/gemini/3-1-pro.

DeepMind. gemini-3-flash-preview. In *DeepMind*, 2026b. URL https://docs.cloud.google.com/gemini-enterprise-agent-platform/models/gemini/3-flash.

DeepSeek-AI. Deepseek-v3.2-exp: Boosting long-context efficiency with deepseek sparse attention, 2025.

Dong, K. and Ma, T. Stp: Self-play llm theorem provers with iterative conjecturing and proving. *arXiv preprint arXiv:2502.00212*, 2025.

Dong, K., Mahankali, A., and Ma, T. Formal theorem proving by rewarding llms to decompose proofs hierarchically. *arXiv preprint arXiv:2411.01829*, 2024.

Eberl, M., Klein, G., Nipkow, T., Paulson, L., Thiemann, R., et al. Archive of formal proofs. *Online Journal. URL: https://www. isa-afp. org*, 2024.

Gao, G., Wang, Y., Jiang, J., Gao, Q., Qin, Z., Xu, T., and Dong, B. Herald: A natural language annotated lean 4 dataset. In *International Conference on Learning Representations*, volume 2025, pp. 55720–55762, 2025.

Hu, J., Zhang, J., Zhao, Y., and Ringer, T. Hybridprover: Augmenting theorem proving with llm-driven proof synthesis and refinement. *arXiv preprint arXiv:2505.15740*, 2025.

Hubert, T., Mehta, R., Sartran, L., Horváth, M. Z., Žužić, G., Wieser, E., Huang, A., Schrittwieser, J., Schroecker, Y., Masoom, H., et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, pp. 1–3, 2025.

Jiang, A. Q., Li, W., Han, J. M., and Wu, Y. Lisa: Language models of isabelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving*, pp. 378–392, 2021.

Jiang, A. Q., Li, W., Tworkowski, S., Czechowski, K., Odrzygóźdź, T., Miłoś, P., Wu, Y., and Jamnik, M. Thor: Wielding hammers to integrate language models and automated theorem provers. *Advances in Neural Information Processing Systems*, 35:8360–8373, 2022a.

Jiang, A. Q., Welleck, S., Zhou, J. P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., and Lample, G. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022b.

Lin, Y., Tang, S., Lyu, B., Wu, J., Lin, H., Yang, K., Li, J., Xia, M., Chen, D., Arora, S., et al. Goedel-prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*, 2025a.

Lin, Y., Tang, S., Lyu, B., Yang, Z., Chung, J.-H., Zhao, H., Jiang, L., Geng, Y., Ge, J., Sun, J., et al. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*, 2025b.

Liu, G., Ma, J., and Wong, D. F. Lean2isabelle: Factorized cross-assistant proof translation. In *3rd AI for Math Workshop: Toward Self-Evolving Scientific Agents*, 2026a. URL https://openreview.net/forum?id=t9iBTo1zBG.

Liu, J., Zhou, Z., Zhu, Z., Santos, M. D., He, W., Liu, J., Wang, R., Xie, Y., Zhao, J., Wang, Q., et al. Numina-lean-agent: An open and general agentic reasoning system for formal mathematics. *arXiv preprint arXiv:2601.14027*, 2026b.

Ma, J., Zhan, R., Li, Y., Sun, D., Chan, H. P., Chao, L. S., and Wong, D. F. Visaidmath: Benchmarking visual-aided mathematical reasoning. In *The 64th Annual Meeting of the Association for Computational Linguistics*, 2026.

URL https://openreview.net/forum?id=QY3djsyQOw.

Mikuła, M., Tworkowski, S., Antoniak, S., Piotrowski, B., Jiang, A. Q., Zhou, J. P., Szegedy, C., Kuciński, Ł., Miłoś, P., and Wu, Y. Magnushammer: A transformer-based approach to premise selection. *arXiv preprint arXiv:2303.04488*, 2023.

OpenAI. Gpt-5-codex. In *OpenAI*, 2025. URL https://cdn.openai.com/pdf/97cc5669-7a25-4e63-b15f-5fd5bdc4d149/gpt-5-codex-system-card.pdf.

Paulson, L. C. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, Cham, 1994. doi: 10.1007/BFb0030541.

Polu, S. and Sutskever, I. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Ren, Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., et al. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

Shen, Z., Huang, N., Yang, F., Wang, Y., Gao, G., Xu, T., Jiang, J., He, W., Yang, P., Sun, M., et al. Real-prover: Retrieval augmented lean prover for mathematical reasoning. *arXiv preprint arXiv:2505.20613*, 2025.

Thakur, A., Tsoukalas, G., Wen, Y., Xin, J., and Chaudhuri, S. An in-context learning agent for formal theorem-proving. *arXiv preprint arXiv:2310.04353*, 2023.

Tsoukalas, G., Lee, J., Jennings, J., Xin, J., Ding, M., Jennings, M., Thakur, A., and Chaudhuri, S. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. *Advances in Neural Information Processing Systems*, 37:11545–11569, 2024.

Varambally, S., Voice, T., Sun, Y., Chen, Z., Yu, R., and Ye, K. Hilbert: Recursively building formal proofs with informal reasoning. *arXiv preprint arXiv:2509.22819*, 2025.

Wang, H., Xin, H., Zheng, C., Li, L., Liu, Z., Cao, Q., Huang, Y., Xiong, J., Shi, H., Xie, E., et al. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*, 2023.

Wang, H., Xin, H., Liu, Z., Li, W., Huang, Y., Lu, J., Yang, Z., Tang, J., Yin, J., Li, Z., et al. Proving theorems recursively. *Advances in Neural Information Processing Systems*, 37:86720–86748, 2024.

Wang, H., Unsal, M., Lin, X., Baksys, M., Liu, J., Santos, M. D., Sung, F., Vinyes, M., Ying, Z., Zhu, Z., et al. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.

Wischermann, N., Verdun, C. M., Poesia, G., and Noseda, F. Proofcompass: Enhancing specialized provers with llm guidance. *arXiv preprint arXiv:2507.14335*, 2025.

Wu, Y., Jiang, A. Q., Li, W., Rabe, M., Staats, C., Jamnik, M., and Szegedy, C. Autoformalization with large language models. *Advances in neural information processing systems*, 35:32353–32368, 2022.

Xin, H., Guo, D., Shao, Z., Ren, Z., Zhu, Q., Liu, B., Ruan, C., Li, W., and Liang, X. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024a.

Xin, H., Ren, Z., Song, J., Shao, Z., Zhao, W., Wang, H., Liu, B., Zhang, L., Lu, X., Du, Q., et al. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. In *The Thirteenth International Conference on Learning Representations*, 2024b.

Xin, R., Xi, C., Yang, J., Chen, F., Wu, H., Xiao, X., Sun, Y., Zheng, S., and Ding, M. Bfs-prover: Scalable best-first tree search for llm-based automatic theorem proving. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32588–32599, 2025a.

Xin, R., Zheng, Z., Nie, Y., Yuan, K., and Xiao, X. Scaling up multi-turn off-policy rl and multi-agent tree search for llm step-provers. *arXiv preprint arXiv:2509.06493*, 2025b.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., Zheng, C., Liu, D., Zhou, F., Huang, F., Hu, F., Ge, H., Wei, H., Lin, H., Tang, J., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Zhou, J., Lin, J., Dang, K., Bao, K., Yang, K., Yu, L., Deng, L., Li, M., Xue, M., Li, M., Zhang, P., Wang, P., Zhu, Q., Men, R., Gao, R., Liu, S., Luo, S., Li, T., Tang, T., Yin, W., Ren, X., Wang, X., Zhang, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Zhang, Y., Wan, Y., Liu, Y., Wang, Z., Cui, Z., Zhang, Z., Zhou, Z., and Qiu, Z. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R. J., and Anandkumar, A. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

Zhao, X., Zheng, L., Bo, H., Hu, C., Thakker, U., and Kong, L. Subgoalxl: Subgoal-based expert learning for theorem proving. *arXiv preprint arXiv:2408.11172*, 2024.

Zheng, K., Han, J. M., and Polu, S. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

Zhou, Y., Zhao, J., Zhang, Y., Wang, B., Wang, S., Chen, L., Wang, J., Chen, H., Jie, A., Zhang, X., et al. Solving formal math problems by decomposition and iterative reflection. *arXiv preprint arXiv:2507.15225*, 2025.

A. Dataset and Experimental Setup

A.1. MiniF2F-Test Evaluation Set

We evaluate on the Isabelle split of MiniF2F-Test. The primary source is the official `openai/miniF2F` repository. During preprocessing, we additionally cross-check against `Tonic/MiniF2F` to correct a small number of local ID/index inconsistencies and anomalous entries. The standard proof answers provided with MiniF2F-Test are never used for prompting, retrieval, memory construction, or system tuning. The final evaluation set contains 244 Isabelle theorem statements.

A.2. Isabelle Verification Backend

FIPO-Prover uses Isabelle2022 as the verification backend and PISA (Portal-to-ISAbelle) as the bridge between the FIP Graph Transition Policy and Isabelle. Candidate proofs are executed step by step through PISA. A proof is accepted only when PISA reports a finished proof state.

PISA uses an interactive theory file to establish the Isabelle proving context required for miniF2F evaluation. We configure this file to import: `HOL.HOL`, `Complex_Main`, `HOL-Library.Code_Target_Numeral`, `HOL-Library.Sum_of_Squares`, `HOL-Computational_Algebra.Computational_Algebra`, and `HOL-Number_Theory.Number_Theory`. During the evaluation, PISA per-step checking uses a 10s timeout. Sledgehammer uses a 30s internal timeout and a 35s wrapper timeout. Sledgehammer is enabled in the main configuration, with external provers available through its configuration, including `cvc5`, `vampire`, `verit`, `e`, `spass`, `z3`, and `zipperposition`. Nitpick is not explicitly invoked.

A.3. Verifier Feedback Normalization

FIPO-Prover normalizes heterogeneous Isabelle verifier feedback into the error tags in Table 5. These tags provide a compact interface between verifier traces and FIP graph transitions. The two-stage local router that maps normalized feedback to recovery transitions is detailed in Appendix D.1.

Category	Test	Test %	Valid	Valid %	Total
Analysis Calculus	69	28.28	71	29.10	140
Geometry Linear Alg.	30	12.30	29	11.89	59
Logic Sets Induction	19	7.79	20	8.20	39
Number Theory	126	51.64	124	50.82	250

Table 4. LLM-assigned MiniF2F category distribution.

error_type_tag	error_type
`syntax_or_type_issue`	syntax or type issue
`undefined_term_or_fact`	undefined term or fact
`goal_mismatch`	goal mismatch
`tactic_failure`	tactic failure
`proof_structure_error`	proof structure error
`unclassified_error`	unclassified error

Table 5. Internal error tags and their natural-language error types.

A.4. Search Budgets

Search is bounded by preset transition-policy limits. The main limits are: at most 3 formal-proof regenerations in exception recovery, at most 2 Root FIP-state Initialization judge (Root-state judge) rounds, at most 3 Upstream Causal Backtracking transitions per theorem, and global Causal-Pair Expansion depth `rec_max`=3. We set `sublemma_integration_max`=20 to prevent repeated Causal-Pair Expansion cycles at the same failure location. Projected-Candidate Repair also has a same-location retry limit; repeated ineffective local repairs transfer control to branch-level recovery.

A.5. Evaluation Metrics

For theorem j with output proof candidate P_j, define

$$s_j = \mathbf{1}\{\mathcal{V}_{\text{Isa}}(P_j) = 1\}.$$

For a test set of size N, pass@1 is

$$\text{pass@1} = \frac{1}{N}\sum_{j=1}^{N} s_j.$$

Task	Model	Temperature
Raw IP Generation	Gemini 3 Flash	0.70
Root FIP-state Initialization	Gemini 3.1 Pro	0.40
Root-state Judge	Gemini 3 Flash	0.90
Generate Formal Proof	Gemini 3.1 Pro	0.20
Verify Direction	Gemini 3 Flash	0.00
Direction Rewrite	Gemini 3 Flash	0.70
Structure Judge	Gemini 3.1 Pro	0.00
Branch Expandability Analysis	Gemini 3 Flash	0.00
Isabelle Code Repair	Gemini 3.1 Pro	0.90
Convert Formal Lemma To Informal	Gemini 3 Flash	0.70
Dependency Analysis	Gemini 3 Flash	0.20
Experience Abstraction Tags	Gemini 3 Flash	0.00
Successful Step Abstraction Tags	Gemini 3 Flash	0.00

Table 6. Task-level model routing and decoding temperature.

For pass@N, a theorem is solved if at least one of the first N independent candidates or complete runs is accepted:

$$s_j^{(N)} = \mathbf{1}\{\exists k \leqslant N,\ \mathcal{V}_{\mathrm{Isa}}(P_{j,k}) = 1\},$$

$$\text{pass@N} = \frac{1}{N_{\text{test}}} \sum_{j=1}^{N_{\text{test}}} s_j^{(N)}.$$

B. FIP Graph Transition Interfaces

Table 7 summarizes the functional interfaces used by the main FIP Graph Transition Policy. We report system-level transition roles rather than low-level scripts or state mutations.

C. FIP-state Collection and Structuring

C.1. FIP-state Representation

Runtime records are integrated into a structured FIP-state node by the FIP Graph Transition Policy. A FIP-state is a node-level representation that connects an optimized informal proof, extracted causal fields, verifier-side outcomes, and revision trajectory. Root FIP-state Initialization extracts cause side, effect side, and dependency links from the informal proof. Subsequent FIP-to-FP projection and verification attach verifier state, while Local FIP-state Revision, Projected-Candidate Repair, Causal-Pair Expansion, and Upstream Causal Backtracking add revision history. This integrated record is the object diagnosed, revised, expanded, or pruned during FIP search graph optimization.

C.2. Root FIP-state Initialization

Each Root FIP-state Initialization iteration optimizes the informal proof and extracts the causal fields of the FIP-state: cause side, effect side, and dependency links. To guide the revised informal proof towards an Isar-like formal format, the initialization prompt dynamically incorporates three category-matched few-shot examples sourced from the AFP. The root-state judge checks whether the revised informal proof is sufficiently explicit and formalization-oriented. Accepted output becomes the informal component of the next FIP node; verifier state and revision history are attached after subsequent FIP-to-FP projection and verification events.

Root FIP-state Initialization can be invoked after initial informal generation, after a direction rewrite, or after Upstream Causal Backtracking. This transition iteratively refines the informal proof so that each reasoning step can be translated into formal proof code more directly.

D. Recovery Transitions and Search Control

D.1. Grammar/Structure-Oriented Error Routing

The local recovery router first applies a heuristic split. If the verifier reason contains tactic-failure signals such as `Failed to finish proof`, `Failed to apply initial proof method`, or if the extracted failing command contains tactic keywords such as `by`, `using`, or `apply`, the failure is initially classified as `tactics`. Otherwise, it is initially classified as `grammar`.

The second stage updates the initial tactic class through an LLM structure judge. For example, initially classified as `tactics`, `structure_judge` determines whether the error reflects a mathematical gap or a proof-structure problem. `STRUCTURAL_ERROR` is mapped back to `grammar`; `MATHEMATICAL_GAP` remains `tactics`. This two-stage design separates shallow proof-script malformation from semantic gaps that require Local FIP-state Revision or Causal-Pair Expansion.

D.2. Projected-Candidate Repair

Projected-Candidate Repair is a local proof-script patching transition. It receives a failed formal candidate, condensed verifier diagnostics, and the local failed-code window, and returns either a patched candidate or no effective repair. This transition provides low-cost local recovery before more expensive Local FIP-state Revision. A preset same-location retry limit bounds repeated ineffective repairs; after the limit is reached, the FIP Graph Transition Policy moves to branch-level recovery.

D.2.1. Case Study: Projected-Candidate Repair

This case demonstrates a **Projected-Candidate Repair**, which performs bounded, in-place repair of the Isabelle candidate induced by the current FIP-state. This module is designed to resolve syntax, type, tactic, or proof-block errors while preserving the upstream causal structure.

Unlike search-level transitions (such as upstream backtracking or branch switching), Projected-Candidate Repair does not revise the global proof plan or expand the local causal structure. Instead, it repairs the projected Isabelle candidate exactly at the verifier-failing positions. In this instance, unresolvable `sledgehammer` attempts are systematically replaced with executable local proof tactics (e.g., `by auto`) and explicitly constructed sub-proof blocks. This demonstrates a genuine candidate-level repair that perfectly preserves the theorem goal, the surrounding proof skeleton, and the upstream informal direction.

D.3. Branch Pruning and AFP-RAG Evidence

Before opening a Causal-Pair Expansion branch, FIPO-Prover runs an evidence-based branch expandability gate. The gate decides whether the current failure is promising for lemma extraction or should be marked inactive through Branch Pruning. Causal-Pair Expansion is attempted only when the branch remains active and the provability/feasibility judgment passes.

Evidence-based Branch Expandability Analysis By retrieving relevant historical proofs from the AFP as empirical evidence, this analysis determines the logical potential of a proof branch to decide whether it merits further recursive expansion. AFP-RAG retrieves evidence from local failure information rather than from the full proof text. The query is constructed from the failed step and its local context: the system concatenates `failed_step + local_context`, appends a concept-fingerprint line of the form `concepts: token1 token2 ...`, normalizes the text, and sends the result to vector retrieval. Concept tokens such as `gcd`, `prime`, `pow`, or `subseteq` are extracted from the raw query to improve robustness to variable renaming and instance-level surface variation.

Before retrieval, the system normalizes mathematical surface forms into Isabelle-style textual tokens (e.g., $\leqslant$ to `\<le>`, $\in$ to `\<in>`, $\forall$ to `\<forall>`, and $\mathbb{Z}$ to `\<int>`). Retrieval first filters by the LLM-assigned mathematical category of the target statement. If category-filtered evidence is insufficient, the system falls back to full-library retrieval and returns top-k ($k = 3$) evidence snippets. The AFP vector database is built by rule-based extraction of eligible AFP lemmas/theorems followed by embedding.

D.3.1. Case Study: Branch Pruning

This case demonstrates **Branch Pruning**, where the system explicitly marks a low-promise FIP branch as inactive, stopping continued effort on a fragile direction and redirecting the search to a mathematically equivalent but verifier-friendly path.

The error originates because the automated solver struggles to implicitly handle the numerical comparison of irrational roots (e.g., verifying $-3 \leqslant \mathrm{Re}(b)$ given $(\mathrm{Re}(b))^2 = 8$). The intermediate decision logic explicitly labels this multi-step root-bounding direction as `PRUNE`. Rather than attempting bounded local repairs on an inherently brittle structure, the system backtracks upstream. The regenerated branch abandons the fragile numerical approximation route entirely, instead inserting a low-cost, verifier-friendly bounding step ($\sqrt{8} < 3$) early in the proof. This single redirection changes the downstream proof shape, allowing the verifier to succeed smoothly.

D.4. Causal-Pair Expansion

Causal-Pair Expansion is triggered only after Branch Pruning allows extraction and the provability judgment passes. It addresses tactic failures, indicating that the current causal pair is too coarse for direct closure. Expanded causal-subgraph statements are synthesized by a rule-based procedure: the system extracts available assumptions, locally established intermediate conclusions, and the unresolved target relation around the failure point, then composes a finer-grained lemma statement for the next-level proof task.

The global maximum Causal-Pair Expansion depth is `rec_max`=3. The preset limit `sublemma_integration_max`=20 prevents non-terminating cycles at the same error position, e.g., tactic failure → Causal-Pair Expansion → lemma integration → tactic failure again at the same location → repeated Causal-Pair Expansion.

D.4.1. Case Study: Causal-Pair Expansion

This case illustrates how **Causal-Pair Expansion** refines a proof that is mathematically sound but too coarse in local granularity. The key move is that one dense contradiction block in the $a < b$ branch is not treated as a single jump anymore; it is unpacked into a sequence of verifier-friendly intermediate steps while the global proof direction remains unchanged.

D.5. Upstream Causal Backtracking

When Local FIP-state Revision and Causal-Pair Expansion branches do not make progress, the system performs Upstream Causal Backtracking. The Error Analyzer attributes the current verifier failure to an upstream proof anchor and produces error feedback. If the blamed anchor is the first step, the FIP Graph Transition Policy performs global regeneration; otherwise, it rewrites the suffix from that anchor and re-runs Root FIP-state Initialization and FIP-to-FP projection. Validated prefixes are typically preserved.

D.5.1. Case Study: Upstream Causal Backtracking

This case illustrates **Upstream Causal Backtracking**, where a terminal verification failure is explicitly attributed to a representational flaw at the root of the proof graph. The case logs are presented in a chronological layout to highlight the system's decision-making process: diagnosing the root cause of an upstream failure, abandoning the invalid branch, extracting guidance from feedback and memory, and finally regenerating a formally sound downstream trajectory.

E. Memory System

The memory system contains short-term memory (STM) and long-term memory (LTM). Memory can be updated online during an evaluation run, but each complete MiniF2F-Test experiment uses an independent memory database. STM accumulates feedback within a single theorem attempt and is cleared before the next theorem. STM is consolidated to LTM only when a theorem is solved or the attempt terminates after reaching system limits.

Error Feedback in STM is constructed by the Error Analyzer. Before negative feedback is uploaded to long-term NEGATIVE memory, the Experience Abstractor abstracts it into a negative experience. The abstracted experience is used as the retrievable representation, while the failure context and original Error Feedback are retained for auditability.

LTM contains two memory types. NEGATIVE memory stores abstracted failure-to-advice experiences. CONTRASTIVE memory is written only when the theorem is eventually solved: previous Upstream Causal Backtracking events are traversed, each blamed step is compared with the corresponding final successful step, and the difference is abstracted as contrastive experience. When inserting a new LTM record, similar entries are first retrieved using the new vector. If a candidate has similarity $\mathrm{sim} \geqslant 0.9$, the new memory is treated as a duplicate or near-duplicate and merged into the existing record instead of creating a new entry.

During Upstream Causal Backtracking retrieval, the Error Analyzer first processes the proof prefix up to the current error to obtain Error Feedback. An abstracted query key is then constructed, top-k ($k = 3$) LTM records are retrieved, and current feedback is combined with the retrieved experience summaries into injection text for the next rewrite or Root FIP-state Initialization attempt.

At step t, the system constructs a retrieval query

$$q_t = \phi(T, r_t, \epsilon_t, \tau_t),$$

retrieves relevant memories

$$m_t = \mathrm{Retrieve}(q_t, \mathcal{M}_{\mathrm{long}}),$$

and conditions the FIP Graph Transition Policy on the current state, feedback, trajectory, and retrieved experience:

$$a_t \sim \pi(a \mid T, r_t, \epsilon_t, \tau_t, m_t).$$

Memory guides FIP graph transitions away from historically low-value regions.

E.1. Case Study: Memory-Augmented FIP Search Policy

This case illustrates the **Memory-Augmented FIP Search Policy**, demonstrating how FIP search leverages both short-term diagnostics and long-term historical context to navigate away from fragile informal reasoning. The case logs are presented chronologically to highlight the cognitive loop: diagnosing a local structural failure, abandoning the overly compressed branch, retrieving specific multi-scale memory guidance, and mapping those memory signals directly into a regenerated, verifier-friendly proof trajectory.

In this example, long-term memory guides the search toward a concrete-expansion strategy for small summation targets. The regenerated proof therefore replaces an abstract substitution-only step with an explicit expansion of S_5 into $a + (a + d) + (a + 2d) + (a + 3d) + (a + 4d)$, which is then simplified to $5a + 10d$.

F. Annotation Protocols and Analysis Metrics

F.1. FIP Transformation Types and Proof-Strategy Revision Categories

FIP transformation annotation labels topological changes to informal proofs at the event level. Each sample is a `(before_ip, after_ip)` pair from the proof records. Transformation types include `Direction_Change`, `Redundancy_Pruning`, `Step_Insertion`, `Explicitization`, and `Executabilization`. The first four are assigned by an LLM judge; `Executabilization` is a rule-based overlay triggered when Isabelle/Isar-like formal cues increase after the edit.

F.1.1. Direction Change

This transformation uses upstream backtracking to replace the original mathematical strategy, rewriting the proof from polynomial coefficient matching to point evaluation.

Model: `Gemini 3 Flash`

Type: `Direction_change`

Problem_ID: `mathd_algebra_276`

Math_problem: The expression $10x^2 - x - 24$ can be written as $(Ax - 8)(Bx + 3)$, where A and B are integers. What is $AB + B$? Show that it is 12.

Informal Proof:
Step 1: (Type Declaration and Assumptions): Let A and B be of type **Integer** (`int`). Assume the polynomial equivalence holds for all x: $(A * x - 8) * (B * x + 3) = 10 * x^2 - x - 24$.

Step 2: (Expand the polynomial): By expanding the left-hand side using the distributive property over integers, we obtain: $(A * x - 8) * (B * x + 3) = (A * B) * x^2 + (3 * A - 8 * B) * x - 24$.

Truncated Sub-path: Coefficient Matching Strategy

~~### Step 3: (Extract coefficient equations): Since the polynomials are equivalent for all x, their corresponding coefficients must be equal. Matching the x^2 coefficients yields $A * B = 10$. Matching the x coefficients yields $3 * A - 8 * B = -1$.~~

~~### Step 4: (Derive a quadratic equation for B without division): Take the linear coefficient equation $3 * A - 8 * B = -1$ and multiply both sides by B. This yields $3 * (A * B) - 8 * B^2 = -B$. Substitute the known value $A * B = 10$ into this equation to get $3 * 10 - 8 * B^2 = -B$, which simplifies to $30 - 8 * B^2 = -B$.~~

...

Regenerated Sub-path: Point Evaluation Strategy

Step 3: (Evaluate at specific points): To find the relationship between A and B, we instantiate the identity $(A * B) * x^2 + (3 * A - 8 * B) * x - 24 = 10 * x^2 - x - 24$ at specific values of x.
- By letting $x = 1$, we have:
$(A * B) * 1^2 + (3 * A - 8 * B) * 1 - 24 = 10 * 1^2 - 1 - 24$
$A * B + 3 * A - 8 * B = 9$.
- By letting $x = -1$, we have:
$(A * B) * (-1)^2 + (3 * A - 8 * B) * (-1) - 24 = 10 * (-1)^2 - (-1) - 24$
$A * B - (3 * A - 8 * B) = 11$.

Step 4: (Solve for $A * B$ and $3 * A - 8 * B$): Using the system of equations derived in Step 3:
1. $(A * B) + (3 * A - 8 * B) = 9$
2. $(A * B) - (3 * A - 8 * B) = 11$
Adding the two equations:
$2 * (A * B) = 20 \Rightarrow A * B = 10$.
Subtracting the second equation from the first:
$2 * (3 * A - 8 * B) = -2 \Rightarrow 3 * A - 8 * B = -1$.

Step 5: (Determine integer values for A and B): We know $A * B = 10$ and A, B are integers. Substituting $B = 10/A$ into the second equation:
$3 * A - 8 * (10/A) = -1$
$3 * A^2 - 80 = -A$
$3 * A^2 + A - 80 = 0$
$(3 * A + 16)(A - 5) = 0$
Since A must be an integer, $A = 5$.
Substituting $A = 5$ back into $A * B = 10$, we find $B = 2$.

Step 6: (Calculate $AB + B$): Now we compute the value of the expression $AB + B$ using the determined values:
$AB + B = (5 * 2) + 2 = 10 + 2 = 12$.

Step 7: (Final Conclusion): Thus, we have shown that:
$AB + B = 12$.

F.1.2. EXECUTABILIZATION

This transformation improves formal executability without changing the mathematical direction of the proof. The search truncates abstract algebraic steps and regenerates the sub-path by injecting explicit Isabelle/HOL type annotations (highlighted in green), ensuring strict type matching for the proof engine's executable semantics.

Model: `Gemini 3 Flash`

Type: `Executabilization`

Problem_ID: `mathd_numbertheory_175`

Math_problem: What is the units digit of 2^{2010}? Show that it is 4.

Informal Proof:
Step 1: (Type Declaration): Let all numbers be of type **Natural Number** (`nat`). The units digit of a number x is defined algebraically as $x \bmod 10$. We aim to evaluate $2^{2010} \bmod 10$.

Step 2: (Algebraic Expansion): For any natural number $n \geqslant 1$, expand 2^{n+4} using exponent rules: $2^{n+4} = 2^n * 16$. This can be rewritten as $2^n * (1 + 15) = 2^n + 2^n * 15$.

Truncated Sub-path: Abstract Algebraic Factorization

~~### Step 3: (Factorization for Modulo): Since $n \geqslant 1$, we can factor a 2 from 2^n to reveal a multiple of 10: $2^n * 15 = (2^{n-1} * 2) * 15 = 2^{n-1} * 30 = (2^{n-1} * 3) * 10$.~~

~~### Step 4: (Periodicity Modulo 10): Substituting this back gives $2^{n+4} = 2^n + (2^{n-1} * 3) * 10$. Taking modulo 10 on both sides, the multiple of 10 vanishes. This yields the periodicity property: $2^{n+4} \bmod 10 = 2^n \bmod 10$.~~

...

Regenerated Sub-path: Executable Type Annotations

Step 3: (Successor Substitution): For any natural number $(n :: nat) \geqslant 1$, we can express n as the successor of some natural number $(m :: nat)$, such that $n = m + 1$. We substitute this into the term $(2^n * 15 :: nat)$ from the expansion in Step 2:
$(2^n * 15 :: nat) = (2^{m+1} * 15 :: nat) = ((2 * 2^m) * 15 :: nat)$.

Step 4: (Simplification of Terms): Using the associative and commutative properties of multiplication for the type `nat`, we can group the constants to simplify the expression:
$((2 * 2^m) * 15 :: nat) = (2^m * (2 * 15) :: nat) = (2^m * 30 :: nat)$.

Step 5: (Modular Periodicity): We now evaluate the expression modulo 10. We observe that $(30 \bmod 10 :: nat) = 0$. Therefore, the product $(2^m * 30 \bmod 10 :: nat)$ is 0. Applying this to the result from Step 2, we find that for any $n \geqslant 1$:
$(2^{n+4} \bmod 10 :: nat) = ((2^n + 2^n * 15) \bmod 10 :: nat) = (2^n \bmod 10 :: nat)$.
This result establishes that the units digit of 2^n follows a cycle of length 4 for all natural numbers $n \geqslant 1$.

Step 6: (Decomposition of the Exponent): We decompose the exponent 2010 using the period of 4:
$(2010 :: nat) = (4 * 502 + 2 :: nat)$.
Since $(2 :: nat) \geqslant 1$, we apply the periodicity property repeatedly to reduce the exponent:
$(2^{2010} \bmod 10 :: nat) = (2^{4*502+2} \bmod 10 :: nat) = (2^2 \bmod 10 :: nat)$.

Step 7: (Final Evaluation): We compute the value of the resulting power:
$(2^2 :: nat) = 4$.
Thus, $(4 \bmod 10 :: nat) = 4$.

Step 8: (Conclusion): We have shown that $2^{2010} \bmod 10 = 4$. Therefore, the units digit of 2^{2010} is 4.

F.1.3. EXPLICITIZATION

Rather than changing the proof strategy, this transformation expands compressed arithmetic reasoning into explicit intermediate checks. While preserving the overall mathematical direction, the search truncates the overly compressed steps and regenerates the sub-path with detailed, step-by-step algebraic verifications to satisfy the formal engine's granularity requirements.

Model: `Gemini 3 Flash`

Type: `Explicitization`

Problem_ID: `mathd_algebra_756`

Math_problem: Given $2^a = 32$ and $a^b = 125$ find b^a. Show that it is 243.

Informal Proof:
Step 1: (Type Declaration and Assumptions): Let a and b be of type **Natural Number** (`nat`). Assume the given equations: $2^a = 32$ and $a^b = 125$.

Truncated Sub-path: Implicit Arithmetic Computations

~~### Step 2: (Rewrite the first equation): By arithmetic computation, we know $32 = 2^5$. Substitute this into the first assumption to obtain the equation $2^a = 2^5$.~~

...

~~### Step 5: (Rewrite the second equation): By arithmetic computation, we know $125 = 5^3$. Substitute this into the equation to obtain $5^b = 5^3$.~~

...

~~### Step 8: (Conclusion): By arithmetic computation, $3^5 = 243$. Thus, we conclude that $b^a = 243$.~~

Regenerated Sub-path: Explicit Arithmetic Verifications

Step 2: (Normalize the first equation):
We recognize that 32 can be expressed as a power of 2 **by calculating its factors:** $2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 16, 2^5 = 32$. Substituting this into the first equation, we obtain $2^a = 2^5$. We assert that since $32 > 1$, a must be greater than 0.

...

Step 5: (Normalize the second equation):
We recognize that 125 can be expressed as a power of 5 **by calculating its factors:** $5^1 = 5, 5^2 = 25, 5^3 = 125$. Substituting this into the equation $5^b = 125$, we obtain $5^b = 5^3$.

...

Step 8: (Apply Power Law and Calculate):
By the power of a power rule for real numbers, $(x^y)^z = x^{yz}$. We can apply this logic to break the calculation of 3^5 into smaller, verifiable pieces:
- First, we calculate $3^4 = 3^{2\times 2} = (3^2)^2 = 9^2 = 81$.
- Then, we calculate $3^5 = 3^4 \times 3^1 = 81 \times 3 = 243$.

Step 9: (Final Conclusion):
In view of the above steps, we have shown that $b^a = 243$. Therefore, the proof is complete.

F.1.4. Redundancy Pruning

The resulting revision removes low-value detours and produces a shorter, cleaner proof trajectory. The search actively truncates pedantic constraint verifications and fragmented substitutions, regenerating a streamlined and mathematically equivalent sub-path that improves search efficiency.

Model: `Gemini 3 Flash`

Type: `Redundancy_Pruning`

Problem_ID: `amc12b_2021_p4`

Math_problem: Ms. Blackwell gives an exam to two classes. The mean of the scores of the students in the morning class is 84, and the afternoon class's mean score is 70. The ratio of the number of students in the morning class to the number of students in the afternoon class is $\frac{3}{4}$. What is the mean of the scores of all the students?
(A) 74 **(B)** 75 **(C)** 76 **(D)** 77
Show that it is **(C)** 76.

Informal proof:
Step 1: (Type Declaration & Ratio Assumption): Let n_M, n_A, S_M, S_A, and k be of type **Real** (`real`). To represent the $3:4$ ratio, assume there exists a constant $k > 0$ such that $n_M = 3 * k$ and $n_A = 4 * k$.

Truncated Sub-path: Redundant Constraint Verification & Fragmented Algebra

---

~~### Step 2: (Constraint Verification): Because $k > 0$, it strictly follows that the number of students $n_M > 0$ and $n_A > 0$.~~

~~### Step 3: (Morning Class Sum): The mean of the morning class is given by the equation $S_M/n_M = 84$. Since $n_M > 0$ (non-zero), we can safely multiply both sides by n_M to obtain $S_M = 84 * n_M$.~~

~~### Step 4: (Afternoon Class Sum): The mean of the afternoon class is given by $S_A/n_A = 70$. Since $n_A > 0$ (non-zero), we can safely multiply both sides by n_A to obtain $S_A = 70 * n_A$.~~

~~### Step 5: (Substitution into Sums): Substitute the definitions $n_M = 3 * k$ and $n_A = 4 * k$ into the sum equations. This yields $S_M = 84 * (3 * k) = 252 * k$ and $S_A = 70 * (4 * k) = 280 * k$.~~

...

Regenerated Sub-path: Consolidated Algebraic Formulation

Step 2: Express Sum of Scores for Each Class
Using the definition of the mean (Sum / Count = Mean), we calculate the total score for each class:
- For the morning class, the sum of scores is $S_M = 84 \cdot n_M = 84 \cdot (3k) = 252k$.
- For the afternoon class, the sum of scores is $S_A = 70 \cdot n_A = 70 \cdot (4k) = 280k$.

Step 3: Determine Total Students and Total Sum
We now find the total number of students and the total sum of their scores:
- The total number of students is $n_{total} = n_M + n_A = 3k + 4k = 7k$.
- The total sum of all scores is $S_{total} = S_M + S_A = 252k + 280k = 532k$.

Step 4: Calculate the Overall Mean
The mean score for the combined classes is the ratio of the total sum of scores to the total number of students:
Mean $= \frac{S_{total}}{n_{total}} = \frac{532k}{7k}$.

Step 5: (Final Calculation):
Since $k > 0$, it follows that $k \neq 0$. We can therefore simplify the expression by dividing both the numerator and the denominator by k:
Mean $= \frac{532}{7} = 76$.

Step 6: Conclusion
The mean of the scores of all the students is 76. This confirms that the correct option is **(C)**.

F.1.5. Step Insertion

This transformation inserts a missing intermediate derivation when the original proof leaves a logical bridge implicit. Instead of abruptly guessing the inverse and verifying it, the search truncates the premature verification and regenerates the branch by inserting constructive steps using the Extended Euclidean Algorithm.

Model: `Gemini 3.1 Pro`

Type: `Step_Insertion`

Problem_ID: `mathd_numbertheory_34`

Math_problem: Find $9^{-1} \pmod{100}$, as a residue modulo 100. (Give an answer between 0 and 99, inclusive.) Show that it is 89.

Informal Proof:
Step 1: (Type Declaration and Goal Formulation): Let all numbers $(9, 89, 100, 801)$ be of type Integer (`int`). We aim to find and verify the multiplicative inverse of 9 modulo 100.

Truncated Sub-path: Direct Verification without Derivation

~~### Step 2: (Verify Canonical Range): By direct arithmetic evaluation, observe that $0 \leqslant 89$ and $89 < 100$. Thus, 89 is a valid canonical residue modulo 100.~~

...

Regenerated Sub-path: Extended Euclidean Algorithm Insertion

Step 2: (Extended Euclidean Algorithm - Division): To find the inverse, apply the Euclidean algorithm to 100 and 9. Set up the division equation: $100 = 11 \times 9 + 1$.

Step 3: (Extended Euclidean Algorithm - Rearrangement): Rearrange the equation from Step 2 to express the remainder 1 as a linear combination of 100 and 9: $1 = 100 - 11 \times 9$.

Step 4: (Determine Initial Inverse): Take the equation from Step 3 modulo 100. Since $100 \equiv 0 \pmod{100}$, the equation becomes $1 \equiv -11 \times 9 \pmod{100}$. This shows that -11 is a multiplicative inverse of 9 modulo 100.

Step 5: (Convert to Canonical Residue): Convert the negative inverse -11 to a positive canonical residue modulo 100 by adding the modulus: $-11 + 100 = 89$. Thus, the candidate inverse is 89.

Step 6: (Verify Canonical Range): By direct arithmetic evaluation, observe that $0 \leqslant 89$ and $89 < 100$. Thus, 89 is a valid canonical residue modulo 100.

...

F.2. Structural Metrics

We define four structural metrics to characterize how a FIP-state revision changes the organization of an informal proof without changing its top-level proof direction. All metrics are computed as per-edit deltas between the revised FIP-state and its preceding FIP-state.

Explicit-step Gain An explicit step is a top-level proof segment introduced by a Markdown-style heading of the form "`Step k: Reasoning Content`". Explicit-step gain is the change in the number of such top-level proof steps. Positive values indicate that the revision expands the proof into more explicit reasoning units.

Intermediate-assertion Gain Intermediate assertions are explicit mathematical claims appearing within an explicit step. Since one explicit step may contain multiple local reasoning moves, it may also contain multiple intermediate assertions. This metric measures the change in the number of such internal claims, capturing whether the revision adds or removes local supporting assertions.

Executable-step Gain Executable steps are approximated by regex matches against Isabelle-like cues, including tactics, Isar commands, type annotations, and definition-unfolding patterns. Executable-step gain measures the change in the number of steps containing such cues, serving as a proxy for direct alignment with Isabelle-style formalization.

Step-length Reduction Step-length reduction measures the decrease in the length of explicit proof steps after revision. Positive values indicate that long or over-compressed steps are shortened or decomposed into more local reasoning units, while negative values indicate longer revised steps.

F.3. Verifier Error Type Annotation

Verifier Error Type is annotated by deterministic regular-expression matching over PISA error messages. The offline classifier reads persisted verifier feedback, applies an ordered rule list, and uses the first matching rule. Fine-grained outputs include `syntax_error`, `unresolved_term`, `type_error`, `missing_assumption`, `goal_mismatch`, `failed_tactic`, `simp_failure`, `proof_block_error`, and `unknown`. These are mapped to coarse buckets such as `syntax_or_type_issue`, `undefined_term_or_fact`, `tactic_failure`, and `proof_structure_error`.

F.3.1. CASE STUDY: GOAL_MISMATCH

Model: `Gemini 3.1 Pro`

Type: `goal_mismatch`

Problem_ID: `mathd_numbertheory_427`

Formal_proof (error highlighted):

```
theorem mathd_numbertheory_427:
  fixes a :: nat
  assumes "a = (\<Sum> k\<in> {n. n dvd 500}. k)"
  shows "(\<Sum> k \<in> {n. prime n \<and> n dvd a}. k) = 25"
proof -
  (* ### Step 1: (Define the target) *)
  (* Let A be the sum of the positive divisors of 500. We will show that the sum of the distinct prime
  ↪  divisors of A is 25. *)

  (* ### Step 2: (Factorize 500) *)
  (* We bound the divisors of 500 to the finite set {1..500} to prepare for evaluation. *)
  have "(\<Sum> k\<in> {n. n dvd 500}. k) = (\<Sum> k\<in> {n \<in> {1..500}. n dvd 500}. k)"
  proof (rule sum.cong)

  show "{n. n dvd 500} = {n \<in> {1..500::nat}. n dvd 500}"

      using dvd_imp_le[of _ 500] by auto
  qed simp
  also have "(\<Sum> k\<in> {n \<in> {1..500::nat}. n dvd 500}. k) = 1092"
    by eval
  ...
```

Verifier_error:

```
Reason: Step error: Failed to refine any pending goal
At command "'show "{n. n dvd 500} = {n \<in> {1..500::nat}. n dvd 500}"'" (line 17)
Local statement fails to refine any pending goal
Failed attempt to solve goal by exported rule:
{n. n dvd 500} = {n \<in> {1..500}. n dvd 500}
At command "'show "{n. n dvd 500} = {n \<in> {1..500::nat}. n dvd 500}"'" (line 17)
```

Analysis: This case illustrates `goal_mismatch` because the written statement is mathematically plausible but does not match the subgoal shape required by the active proof context. The verifier does not report a syntax problem, a type error, or a missing fact; instead, it indicates that the local `show` command fails to refine any pending goal. The error therefore reflects a mismatch between the written step and the proof obligation currently in scope, rather than a problem with expression well-formedness or fact availability.

F.3.2. CASE STUDY: PROOF_STRUCTURE_ERROR

Model: Gemini 3.1 Pro

Type: proof_structure_error

Problem_ID: mathd_numbertheory_552

Formal_proof (error highlighted):

```
theorem mathd_numbertheory_552:
  fixes f g h :: "nat \<Rightarrow> nat"
  assumes "\<forall> x>0. f x = 12 * x + 7"
      and "\<forall> x>0. g x = 5 * x + 2"
      and "\<forall> x>0. h x = gcd (f x) (g x)"
    shows "(\<Sum> k \<in> range h. k) = 12"
proof -
  (* ### Step 1: (Type and Function Definition) *)
  (* Let x be a variable of type nat and assume x > 0. Define the function h(x) = gcd(12*x + 7, 5*x + 2).
  ↪  *)

  (* ### Step 2: (Establish divisibility of a linear combination) *)
  (* By the fundamental properties of the greatest common divisor, h(x) divides both 12*x + 7 and 5*x + 2.
  ↪  *)
  have h_div: "\<And>x. x > 0 \<Longrightarrow> h x dvd 11"
  proof -
    fix x :: nat
    assume "x > 0"
    then have "h x = gcd (12 * x + 7) (5 * x + 2)" using assms by simp
    moreover have "gcd (12 * x + 7) (5 * x + 2) dvd (12 * x + 7)" by simp
    moreover have "gcd (12 * x + 7) (5 * x + 2) dvd (5 * x + 2)" by simp
    diminishingly have "h x dvd (12 * x + 7)" and "h x dvd (5 * x + 2)" by auto

    (* ### Step 3: (Simplify the linear combination) *)
    (* Using integer algebra, expand the expression: 5*(12*x + 7) - 12*(5*x + 2) = 11. *)
    then have "h x dvd 5 * (12 * x + 7)" and "h x dvd 12 * (5 * x + 2)"
      sledgehammer
    then have "h x dvd (5 * (12 * x + 7) - 12 * (5 * x + 2))"
      sledgehammer
    moreover have "5 * (12 * x + 7) - 12 * (5 * x + 2) = (11::nat)"
      by simp
    ultimately show "h x dvd 11" by simp
  qed

  (* ### Step 4: (Bound the possible values) *)
  (* Since 11 is a prime number, its only positive integer divisors are 1 and 11. *)
  have h_vals: "\<And>x. x > 0 \<Longrightarrow> h x \<in> {1, 11}"
  proof -
    fix x :: nat
    assume "x > 0"
    then have "h x dvd 11" using h_div by simp
    then show "h x \<in> {1, 11}"
      sledgehammer
  qed

  (* ### Step 5: (Verify the achievability of 1) *)
  (* To show 1 \<in> V, instantiate x = 1. *)
  have "h 1 = 1"
  proof -
    have "1 > (0::nat)" by simp
    then have "h 1 = gcd (12 * 1 + 7) (5 * 1 + 2)" using assms by simp
    also have "... = gcd 19 7" by simp
    also have "... = 1" sledgehammer
    finally show ?thesis by auto
  qed
```

```
  then have "1 \<in> range h" by auto

  (* ### Step 6: (Verify the achievability of 11) *)
  (* To show 11 \<in> V, instantiate x = 4. *)
  have "h 4 = 11"
  proof -
    have "4 > (0::nat)" by simp
    then have "h 4 = gcd (12 * 4 + 7) (5 * 4 + 2)" using assms by simp
    also have "... = gcd 55 22" by simp
    also have "... = 11" sledgehammer
    finally show ?thesis by auto
  qed
  then have "11 \<in> range h" by auto

  (* ### Step 7: (Determine the exact set of values) *)
  (* By mutual inclusion, the set of possible values is exactly V = {1, 11}. *)
  have "range h = {1, 11}"
    sledgehammer

  (* ### Step 8: (Compute the final sum) *)
  (* The sum of the elements in the finite set V = {1, 11} is algebraically evaluated as 1 + 11 = 12. *)

  then have "(\<Sum> k \<in> range h. k) = (\<Sum> k \<in> \{1, 11\}. k)" by simp

  also have "... = 12" sledgehammer
  finally show ?thesis by auto
  ...
```

Verifier_error:

```
Reason: Step error: Illegal application of proof command in "prove" mode
At command "'then'" (line 30)
```

Analysis: This case illustrates `proof_structure_error` because the failure arises from proof-state management rather than from mathematical content. The command `then` is used as if an immediately chainable preceding fact were available in the correct mode, but Isabelle reports that the command is illegal in the current `prove` state. The error therefore reflects a structural inconsistency in proof sequencing, rather than a missing fact or an unresolved mathematical gap.

F.3.3. Case Study: syntax_or_type_issue

Model: `Gemini 3.1 Pro`

Type: `syntax_or_type_issue`

Problem_ID: `amc12a_2002_p6`

Formal_proof (error highlighted):

```
theorem amc12a_2002_p6:
  fixes n ::nat
  assumes "n>0"
  shows "\<exists> m > n. \<exists> p. m * p \<le> m + p"
proof -
  (* ### Step 1: (Define the solution set) *)
  (* Let S be the set of all natural numbers m such that m > 0 and there exists a natural number p > 0
  ↪  satisfying m * p <= m + p. We must show that S is infinite, which means for any n, there is an m >
  ↪  n in S. *)

  (* ### Step 2: (Fix an arbitrary positive integer) *)
```

```
  (* Fix an arbitrary natural number m > n. We will show that m satisfies the condition. *)
  define m where "m = n + 1"
  have "m > n" using m_def by simp

  (* ### Step 3: (Instantiate the existential witness) *)
  (* To show the existence of a positive integer p satisfying the inequality, choose the candidate p = 1.
  ↪  Observe that p > 0 holds trivially. *)

  define p where "p = 1 :: nat"

  have "m * p \<le> m + p"
    using p_def by simp
  then have "\<exists> p. m * p \<le> m + p"
    by auto
  ...
```

Verifier_error:

```
Reason: Step error: Type unification failed: Clash of types "bool" and "nat"

Failed to meet type constraint:

Term:  p = (1::??'a) :: bool
Type:  nat

Coercion Inference:

Local coercion insertion on the operand failed:
"bool" is not a subtype of "nat"

Now trying to infer coercions globally.

Coercion inference failed:
"nat" is not a subtype of "bool"

At command "'define p where "p = 1 :: nat"'" (line 20)
```

Analysis: This case illustrates `syntax_or_type_issue` because the verifier explicitly reports a `Type unification failed` exception and identifies the offending command. The intended proof move is straightforward: introduce a witness `p = 1` for the existential goal. However, Isabelle interprets the local statement in a way that creates a mismatch between a Boolean equality term and the expected natural-number type. The error therefore reflects a type-level incompatibility in the written step, rather than a mathematical gap or an insufficient tactic.

F.3.4. Case Study: tactic_failure

Model: `Gemini 3.1 Pro`

Type: `tactic_failure`

Problem_ID: `mathd_algebra_332`

Formal_proof (error highlighted):

```
theorem mathd_algebra_332:
  fixes x y ::real
  assumes "(x + y) / 2 = 7"
    and "sqrt (x * y) = sqrt 19"
  shows "x^2 * y^2 = 158"
proof -
  (* ### Step 1: (Type Declaration): Let x and y be variables of type Real. *)
```

```
  (* This is handled by the fixes clause in the theorem statement. *)

(* ### Step 2: Identify Given Information and Variable Types *)
  (* This is handled by the assumptions in the theorem statement. *)

  (* ### Step 3: Derive the Sum and Product of x and y *)
  have c1: "x + y = 14" using assms(1) by auto
  have c2: "x * y = 19" using assms(2) by auto

  (* ### Step 4: Apply the Algebraic Identity for the Sum of Squares *)
  have c3: "x^2 + y^2 = (x + y)^2 - 2 * (x * y)" by sos

  (* ### Step 5: Substitute the Derived Values into the Identity *)
  have c4: "(x + y)^2 - 2 * (x * y) = 14^2 - 2 * 19" using c1 c2 by auto

  (* ### Step 6: Perform the Final Calculation *)
  have c5: "(14::real)^2 - 2 * 19 = 158" by auto
  have c6: "x^2 + y^2 = 158" using c3 c4 c5 by auto

  (* ### Step 7: Conclusion *)
  show ?thesis
using c6

by auto

qed
  ...
```

Verifier_error:

```
Reason: Step error: Failed to apply initial proof method (line 35):
using this:
x\<^sup>2 + y\<^sup>2 = 158
goal (1 subgoal):
 1. x\<^sup>2 * y\<^sup>2 = 158
At command "'by auto'" (line 35)
```

Analysis: This case illustrates `tactic_failure` because the proof script is structurally valid and the available facts are well-formed, yet the chosen method cannot close the remaining goal. The verifier output shows that the context provides `x^2 + y^2 = 158`, whereas the target is `x^2 * y^2 = 158`. The error therefore reflects the insufficiency of `auto` for the current obligation, rather than an issue of parsing, scoping, or proof-command legality.

F.3.5. Case Study: undefined_term_or_fact

Model: `Gemini 3.1 Pro`

Type: `undefined_term_or_fact`

Problem_ID: `amc12b_2021_p1`

Formal_proof (error highlighted):

```
theorem amc12b_2021_p1:
  shows "card {x::int. \<bar>real_of_int x\<bar> < 3 * pi} = 19"
proof -
  (* ### Step 1: Define the Set of Integers *)
  (* Let S be the set of all integers x that satisfy the given inequality. *)
  define S where "S = {x::int. \<bar>real_of_int x\<bar> < 3 * pi}"

  (* ### Step 2: Approximate the Boundary Values *)
```

```
(* We know that pi is approximately 3.14159, so 3 * pi is between 9 and 10. *)

have "pi > 3" using pi_gt_3 by simp

then have "3 * pi > 9" by simp
have "pi < 4" using pi_less_4 by simp
then have "3 * pi < 12" by simp
...
```

Verifier_error:

```
Reason: Step error: Undefined fact: "pi_gt_3" (line 14)
At command "'using pi_gt_3'" (line 14)
```

Analysis: This case illustrates `undefined_term_or_fact` because the failure is triggered by an explicit reference to a fact name that Isabelle cannot resolve in the current context. The verifier identifies both the missing object, `pi_gt_3`, and the exact command that attempts to use it. The error therefore reflects a direct lookup failure in the proof environment, rather than a mathematical gap or an insufficient tactic.

F.4. Human Evaluation Agreement

To assess the reliability of the LLM annotations used throughout the study, we conduct a human–LLM agreement evaluation. Because both logical decomposition and edit-type identification require mathematical reasoning expertise, two Master's students in Mathematics independently annotate the evaluation set using the same annotation guidelines. The evaluation follows a double-blind protocol: annotators do not see the LLM predictions or each other's annotations, and the LLM does not access any human-labeled examples. For all reported results, agreement metrics are computed separately for each annotator and then averaged.

The evaluation consists of two annotation tasks corresponding to the analyses reported in the main text. For reproducibility, we include the full human annotation instructions used in this evaluation in Tables 17 and 18, which correspond to assertion counting and proof edit-type annotation, respectively.

F.4.1. Edit Type Classification Task

Task Description This task examines structural changes between proof versions. For each edit event, annotators compare the intermediate proof representations before and after the modification (`before_ip` and `after_ip`) and assign one or more edit types from five categories: Explicitization, Redundancy_Pruning, Step_Insertion, Direction_change, and Executabilization. We select a random subset of 40 samples (`random_seed=42`) for evaluation.

Results As shown in Table 15, the LLM achieves an average sample-level Jaccard similarity of 80.8% and a label-level Micro-F1 score of 85.2%. Under a lenient coverage criterion, where all human-assigned labels are contained in the LLM prediction, the agreement rate reaches 87.5%. These results indicate that the LLM captures the major edit operations identified by human annotators, providing support for the edit-type analyses presented in Section 4.1. The remaining disagreements mainly arise from cases in which the LLM assigns additional edit labels beyond those selected by human annotators.

F.4.2. Assertion Counting Task

Task Description This task evaluates the identification of assertions in informal proofs. Annotators first decompose each proof into locally valid reasoning segments and then derive the number of assertions according to the rule `assertion = continuous_logical_chain - 1`. The evaluation is conducted on 40 samples.

Results Exact agreement is achieved on 30.0% of the samples. Given the inherent subjectivity involved in determining the granularity of reasoning segments, exact matches alone provide a limited view of annotation quality. As shown in Table 16, the LLM annotations show a strong correlation with human annotations in estimating proof complexity (Pearson $r = 0.934$), and 95.0% of predictions differ from the human counts by no more than two assertions (MAE = 1.25). These results suggest

that the LLM preserves the relative ordering and distributional patterns of assertion counts used in the analyses reported in Section 4.2, despite occasional differences in fine-grained segmentation decisions.

F.4.3. HUMAN INSTRUCTION

For transparency, we present the human annotation instructions used in the agreement study. Table 17 shows the guidelines for assertion counting, including the counting unit, exclusion rules, and the transition-to-assertion conversion procedure. Table 18 presents the guidelines for proof edit-type annotation, including the legal labels, their definitions, and the funnel-style decision process used to ensure consistent label assignment across annotators.

G. Model Settings

This section details the comprehensive parameter configurations for all models used in our experiments. For specific model versions and key hyperparameters, please refer to Table 19.

H. Additional Diagnostics and Reproducibility Notes

The main text reports aggregate recovery-transition and budget analyses. We include additional diagnostics here to make the analysis pipeline auditable without interrupting the main narrative: non-direction-changing structural changes, ordered recovery paths, Upstream Causal Backtracking distance, and theorem-level marginal gain by first-success FIP-state cost.

H.1. FIP Source Evaluation Metric

Let R denote a source of informal proof or FIP-state. The direct downstream formalization success rate is

$$\mathrm{DirectFormalizationSuccess}(R) = \frac{\#\{\text{theorems directly solved from } R\}}{N}.$$

H.2. Recovery Path Diagnostics

The main text summarizes recovery transitions by contribution and error type. The following figures preserve complementary event-level views: ordered recovery paths, the distance between failed and blamed steps during Upstream Causal Backtracking, and theorem-level marginal gain as a function of first-success FIP-state cost.

For theorem-level necessity analysis, define $\mathrm{Necessary}(T, m) = 1$ if the full system solves theorem T, the reference system does not, and the variant with transition m removed fails on T. Each newly solved theorem has a necessary transition set

$$\mathcal{S}(T) = \{m : \mathrm{Necessary}(T, m) = 1\}.$$

For error type e and transition m, define

$$R_{e,m} = \frac{\#\{x : \mathrm{ErrorType}(x) = e,\ \mathrm{Recovered}(x) = 1,\ m \in P(x)\}}{\#\{x : \mathrm{ErrorType}(x) = e,\ \mathrm{Recovered}(x) = 1\}},$$

where $P(x)$ is the set of recovery transitions used in the recovery path for error event x. Rows need not sum to one.

For budget analysis, let $C_T^\star$ be the cumulative budget consumed before the first verified proof for theorem T, and set $C_T^\star = \infty$ if no proof is found. The matched-budget success rate is

$$\mathrm{Success@Budget}(B) = \frac{\#\{T : C_T^\star \leqslant B\}}{N}.$$

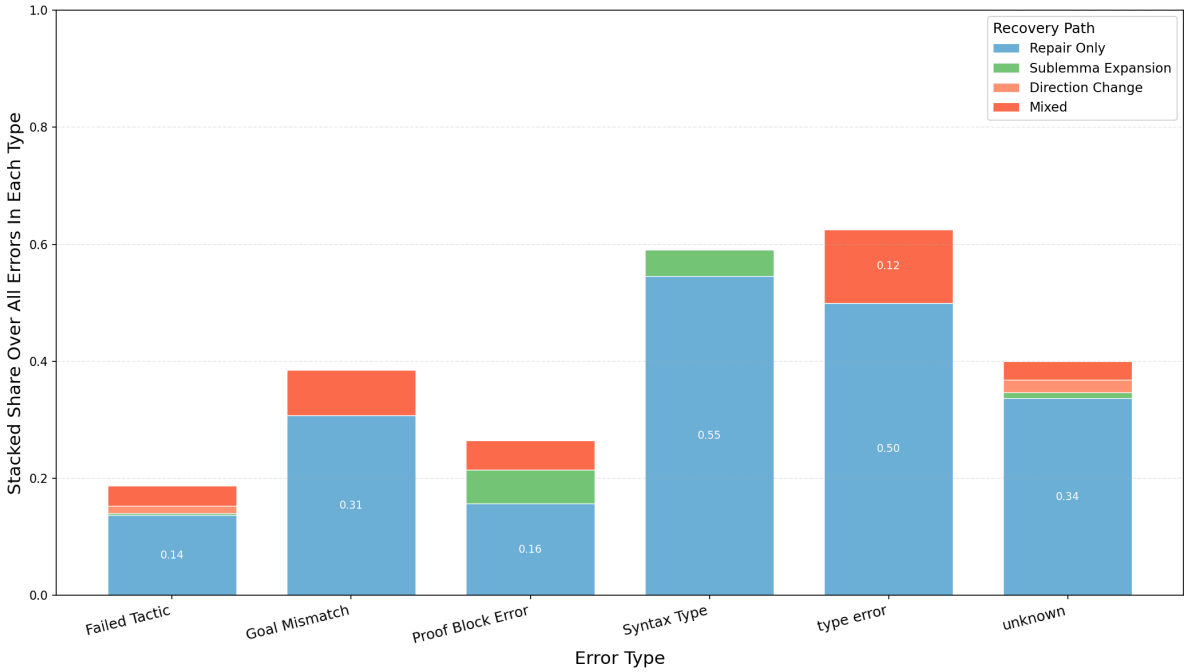


Figure 6. Frequent recovery paths observed after verifier failures. The figure complements the error-recovery heatmap by preserving recovery-transition order.

I. Related work

I.1. Formal-Space Proof Search

Neural formal theorem proving has made substantial progress in recent years. One major line of work directly searches in the formal proof space. Early work such as GPT-f (Polu & Sutskever, 2020) pioneered the use of generative language models for formal proof search. Later systems, including ReProver (Yang et al., 2023) and REAL-Prover (Shen et al., 2025), improve tactic generation through retrieval augmentation and expert iteration. To tackle the large proof-state space, BFS-Prover (Xin et al., 2025a;b) and STP-Prover (Dong & Ma, 2025) optimize tree-search algorithms, while the DeepSeek-Prover series (Xin et al., 2024a;b; Ren et al., 2025) and AlphaProof (Hubert et al., 2025) incorporate large-scale reinforcement learning to enhance formal proving capabilities. In addition, Goedel-Prover (Lin et al., 2025a;b) and Kimina-Prover (Wang et al., 2025) use explicit verifier feedback for iterative self-correction.

These methods typically take tactics, formal subgoals, proof states, or proof scripts as their core search objects, making them effective for local proof completion, formal code generation, and short-horizon proof search. However, formal proof search remains constrained by both the intrinsic difficulty of formal proof languages and the current capabilities of language models. Proof languages such as Lean 4 (De Moura et al., 2015) and Isabelle (Paulson, 1994) require every step to be syntactically valid, semantically appropriate, and precisely aligned with the current proof state, contextual assumptions, and available library facts. Compared with natural-language mathematical reasoning, formal proof languages are often more fine-grained, localized, and redundant, which makes direct search in the formal proof space inefficient for complex theorems and less effective for long-horizon planning and generalization. These limitations motivate methods that use informal reasoning, proof sketches, and subgoal decomposition as higher-level guidance for formal proof construction.

I.2. Informal-Guided Proof Search

A second line of work leverages informal reasoning, proof sketches, retrieval mechanisms, or subgoal decomposition to guide formal proof constructio, benefiting from evolving model capabilities (Ma et al., 2026). DSP (Jiang et al., 2022b) introduced the draft-sketch-prove paradigm. LEGO-Prover (Wang et al., 2023) facilitates the reuse of intermediate proof artifacts through a dynamically growing library and retrieval mechanisms. Systems such as LeanSearch aim to bridge the semantic gap between informal queries and formal corpora. More recently, DSP+ (Cao et al., 2025) re-evaluates

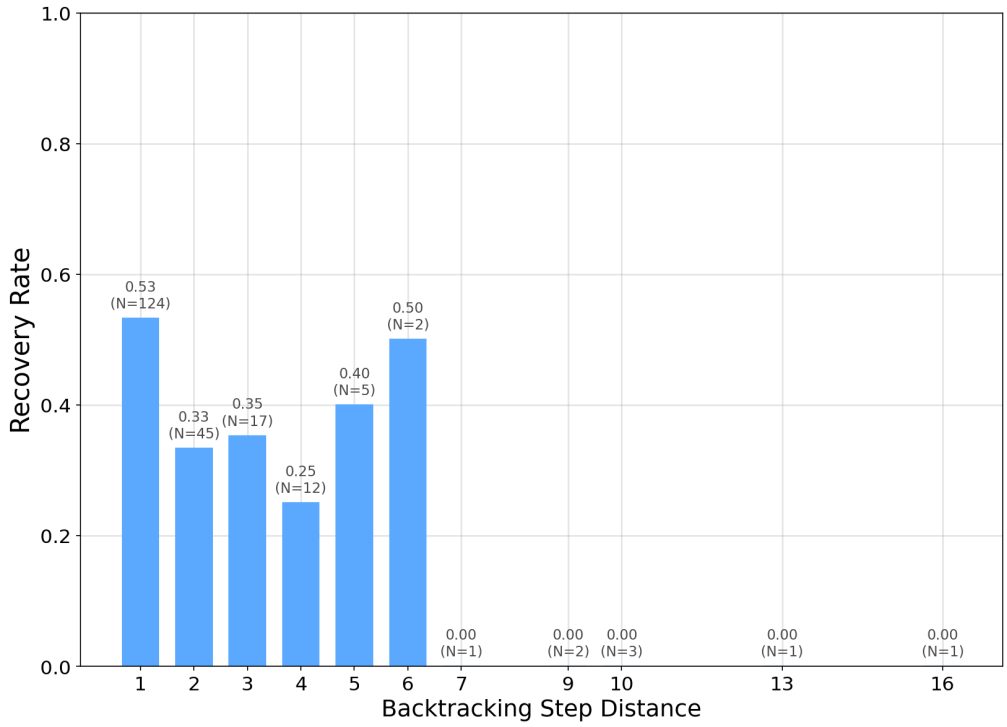


Figure 7. Upstream Causal Backtracking distance and local recovery rate. Distance is computed as the difference between the failed-step index and the blamed-step index; larger distances indicate upstream rewriting rather than local patching.

sketch-guided proving in the context of modern reasoning models and prover LLMs.

Other frameworks decompose complex theorems into more tractable proof obligations. SubgoalXL (Zhao et al., 2024), POETRY (Wang et al., 2024), ProD-RL (Dong et al., 2024), and DeepSeek-Prover-V2 (Ren et al., 2025) use subgoal learning, recursive proving, or reinforcement learning to guide proof construction. Agentic systems, including COPRA (Thakur et al., 2023), Prover-Agent (Baba et al., 2025), ProofCompass (Wischermann et al., 2025), DeltaProver (Zhou et al., 2025), and Numina-Lean-Agent (Liu et al., 2026b), further integrate verifier feedback into iterative proof construction, proof repair, or decomposition loops.

Among these methods, HILBERT (Varambally et al., 2025) is the most closely related to our work. HILBERT performs large-scale recursive Lean proof search guided by informal proofs, expanding subproofs over a proof graph. Its strong performance on benchmarks such as miniF2F (Zheng et al., 2021) and PutnamBench (Tsoukalas et al., 2024), especially under large inference budgets such as pass@1840, demonstrates the effectiveness of organizing proof search around informal proof structures. However, HILBERT does not explicitly optimize the informal proof itself. Instead, generated informal proof steps are largely adopted as scaffolds for downstream formalization and recursive decomposition. Thus, it can still be viewed as a large-scale recursive formal proof search assisted by informal proofs.

While informal-guided methods demonstrate the value of coupling informal reasoning with formal verification, they still do not fully address the representation mismatch between informal mathematical reasoning and proof assistant execution. Informal proofs often compress intermediate derivations, omit implicit dependencies, or introduce overly strong intermediate claims, whereas proof assistants such as Lean 4 (De Moura et al., 2015) and Isabelle (Paulson, 1994) require every step to be executable with respect to the current proof state and available facts. Existing methods typically treat the informal proof as a static sketch, a decomposition guide, a source of candidate lemmas, or a scaffold for recursive search, rather than as a mathematical object that can be continuously revised and optimized. Consequently, verifier feedback is usually applied locally to formal code, tactics, or generated subgoals, and is rarely routed back systematically to the upstream informal proof structure that caused the failure.

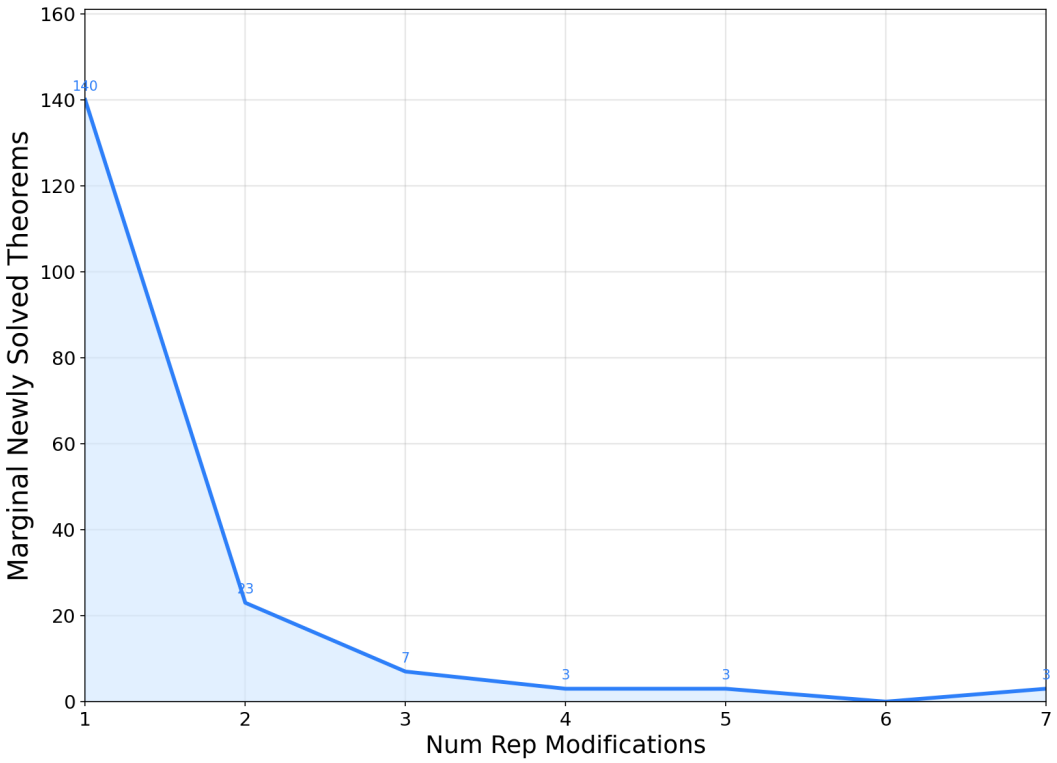


Figure 8. Theorem-level marginal gain diagnostics for first-success FIP-state cost.

J. Supplementary Research Notes

J.1. Artifact Usage

All external artifacts and model services used in this work were accessed only for research purposes within the scope of this study. We did not use derivatives of research-only artifacts outside research contexts. The system code produced by this work is intended for research use and will be released under the CC-BY-SA-4.0 license, subject to compatibility with the access conditions of the original resources used.

J.2. Models and Experiment Details

We report available model configuration details in Appendix G. For closed-source models used through provider APIs, exact parameter counts, GPU hours, and backend computing infrastructure are not disclosed by the service providers and therefore cannot be reported. All main results reported in this paper correspond to single experimental runs because full formal-prover evaluation is computationally expensive, involving repeated LLM calls, verifier interactions, and Isabelle proof checking.

J.3. Human Annotation Detail

For the human annotation agreement study, we recruited student volunteers with relevant mathematical background. Participation was voluntary, and no crowdsourcing platform was used. All annotations will be used for research-only purpose.

J.4. AI Assistants in Research

AI assistants were used only for writing support, such as grammar correction, polishing, and improving clarity.

K. Prompts

Transition role	System behavior	Input / output role
Verification	Execute candidate proofs step by step against Isabelle and record verifier outcomes.	Formal candidate → verifier state, trace, failure location
Root FIP-state Initialization	Convert raw or revised informal proofs into explicit, formalization-oriented FIP-states.	Raw/current IP → optimized IP and causal fields
Root-state Judge	Check whether the revised informal proof is explicit, local, and suitable for projection.	Before/after IP → PASS/feedback
FIP-to-FP Projection Map	Generate an Isabelle candidate from the current FIP-state.	FIP-state → Isabelle candidate
Structure judge	Distinguish proof-structure errors from mathematical gaps.	Tactic-like error → structural vs math-gap label
Projected-Candidate Repair	Patch the local formal proof text using verifier diagnostics and the failed-code window.	Failed code window → patched formal candidate
Branch Pruning / Causal-Pair Expansion Gate	Decide whether a failed branch should be marked inactive or expanded into a finer causal subgraph using local evidence.	Local failure + AFP-RAG evidence → PRUNE/EXTRACT_LEMMA
Upstream Causal Backtracking	Attribute a verifier failure to an upstream proof anchor and produce rewrite feedback.	Failure frontier → blame anchor and rewrite feedback
Direction Rewrite	Rewrite the proof suffix from the blamed anchor while preserving validated prefixes when possible.	Blamed anchor + feedback → rewritten suffix
Memory-Augmented FIP Search Policy	Abstract failure and recovery experience into retrievable search-control memories.	Feedback/experience → retrieval key and stored memory
Text Normalization	Normalize mathematical surface forms before retrieval so that equivalent concepts are matched more robustly.	Query/text → normalized retrieval tokens

Table 7. Main FIP graph transition roles.

Field	Example content
Cause side	$\mathcal{C}_1 : a \mid b$; $\mathcal{C}_2 : b \mid c$; $\mathcal{C}_3 : \exists k.\ b = ak$, derived from $\mathcal{C}_1$; $\mathcal{C}_4 : \exists l.\ c = bl$, derived from $\mathcal{C}_2$.
Effect side	$\mathcal{E}_1 : c = bl$; $\mathcal{E}_2 : c = (ak)l$, by substituting $\mathcal{C}_3$ into $\mathcal{E}_1$; $\mathcal{E}_3 : c = a(kl)$, by associativity; $\mathcal{E}_4 : a \mid c$, by the divisibility definition.
Dependency links	1) $\mathcal{E}_2 \leftarrow \{\mathcal{C}_3, \mathcal{E}_1\}$: substitution of the witness equation for b. 2) $\mathcal{E}_3 \leftarrow \{\mathcal{E}_2\}$: algebraic reassociation. 3) $\mathcal{E}_4 \leftarrow \{\mathcal{E}_3\}$: construction of the witness kl.
Verifier state	PISA fails at the corresponding `show` command; normalized error indicates tactic failure after Sledgehammer cannot close the goal.
Revision history	1) r_0: Initial FIP-to-FP projection fails at the divisibility goal. 2) r_1: Projected-Candidate Repair is attempted but returns no effective patch. 3) r_2: Branch gating permits Causal-Pair Expansion. 4) r_3: A witness-construction sublemma is inserted and re-verified.

Table 8. Illustrative FIP-state node record. The example is schematic and illustrates field semantics rather than a dataset row.

Model: `Gemini 3.1 Pro`
Type: `Projected-Candidate Repair`
Problem_ID: `amc12a_2021_p19`
Math_problem: How many solutions does $\sin\left(\frac{\pi}{2}\cos x\right) = \cos\left(\frac{\pi}{2}\sin x\right)$ have in $[0, \pi]$? Show that it is **(C)** 2.
Formal_statement: `theorem amc12a_2021_p19: shows "card {x :: real. 0 <= x & x <= pi & sin (pi / 2 * cos x) = cos (pi / 2 * sin x)} = 2"`
Triggering Verifier Error: `Sledgehammer error (line 52): fail to finish the proof`
Module Action Summary:

- **Target location:** The verifier-failing candidate lines inside the Step 8 monotonic-segment proof.
- **Affected form before:** A projected Isabelle candidate that relies on unresolved `sledgehammer` calls at the failing positions.
- **Mechanism action:** Perform bounded in-place repair of the projected candidate by replacing failed `sledgehammer` calls with executable local proof blocks.
- **Resulting form after:** The theorem goal, proof skeleton, and upstream informal direction remain fixed, while the local proof obligations are discharged by explicit derivations such as the proof of ‘b1‘.

Formal proof repair: inline diff view

```
...
(* ### Step 7: Splitting the Interval *)
have "{x :: real. 0 <= x & x <= pi & sin (pi / 2 * cos x) = cos (pi / 2 * sin x)} = {0, pi/2}"
proof (intro equalityI subsetI)
  fix x
  assume "x ∈ {x :: real. 0 <= x & x <= pi & sin (pi / 2 * cos x) = cos (pi / 2 * sin x)}"
  then have x_bounds: "0 <= x" "x <= pi" and eq: "sin (pi / 2 * cos x) = cos (pi / 2 * sin x)" by auto
  have eq2: "sin (pi / 2 * cos x) = sin (pi / 2 - pi / 2 * sin x)" using eq step1 by auto
(* ### Step 8: Analyzing the First Monotonic Segment *)
  ~~have b1: "- (pi / 2) <= pi / 2 * cos x" sledgehammer~~
  have b1: "- (pi / 2) <= pi / 2 * cos x"
  proof -
    have "- (pi / 2) = (pi / 2) * (- 1 :: real)" by simp
    also have "...<= (pi / 2) * cos x" using cos_ge_minus_one[of x] by (intro mult_left_mono) auto
    finally show ?thesis .
  qed
...
```

Table 9. Case Study: Projected-Candidate Repair

Model: `Gemini 3 Flash`
Type: `Branch Pruning`
Problem_ID: `proof_record_amc12a_2020_p15`
Math_problem: In the complex plane, let A be the set of solutions to $z^3-8=0$ and let B be the set of solutions to $z^3-8z^2-8z+64=0$. What is the greatest distance between a point of A and a point of B? Show that it is **(D)** $2\sqrt{21}$.
Module Action Summary:

- **Key location:** Step 2's root construction ($\pm\sqrt{8}$), which later manifests as an unresolvable verifier obligation at Step 4.
- **Affected span:** The specific proof branch attempting to maximize distances using the irrational root $2\sqrt{2}$ without providing explicit numerical bounding.
- **Trigger mechanism:** The Isabelle solver fails to verify the side condition $-3 \leqslant \mathrm{Re}(b)$ after substituting the irrational roots.
- **Mechanism action:** The system evaluates AFP-RAG retrieval complexity, issues a `PRUNE` directive, halts this brittle branch, and forces upstream backtracking.

Retained upstream context
Step 1: Solve for set A: $A = \{2, -1+i\sqrt{3}, -1-i\sqrt{3}\}$.
Step 2: Factor the polynomial for set B: $z^3-8z^2-8z+64=(z^2-8)(z-8)$.
Step 3: Determine the roots of B: $B = \{8, 2\sqrt{2}, -2\sqrt{2}\}$.

Pruned sub-path: fragile irrational approximations
~~### Step 4: Calculate distances between points of A and B.~~
~~Compute candidate distances directly from expressions involving $2\sqrt{2}$ and compare them without explicit bounds.~~
~~### Step 5: Argue that $\sqrt{84}=2\sqrt{21}$ is maximal via approximation-heavy comparisons.~~

Branch deactivation: complexity diagnosis and pruning decision
Verifier Failure Signal: `Failed to apply initial proof method... goal: - 3 <= Re b.`
AFP-RAG Retrieval Profile: Low-complexity generic bounds were retrieved, but the main structural match was `Miscellany:am_gm2` with `HIGH` complexity.
Analysis: This profile indicates that the current subgoal has drifted into a heavy inequality-solving route rather than a lightweight arithmetic continuation.
Iter-Check Verdict: `PRUNE`
Reason: `The subgoal is not a feasible candidate for a standalone lemma; terminate this branch and backtrack upstream.`
Outcome: The search stops in this direction immediately and later regenerates a new branch based on explicit numerical bounds.

Table 10. Case Study: Branch Pruning with AFP-RAG

Model: `Gemini 3.1 Pro`
Type: `Causal-Pair Expansion`
Problem_ID: `mathd_algebra_756`
Math_problem: If $2^a = 32$ and $a^b = 125$, show that $b^a = 243$.
Module Action Summary:

- **Triggering Failure:** A coarse causal jump at Step 3 causes `sledgehammer` and `simp` to fail.
- **Causal Gap Diagnosis:** `Structure_Judge` identifies a `MATHEMATICAL_GAP`. `ITER_CHECK` dictates `EXTRACT_LEMMA` based on low-complexity RAG evidence.
- **Sub-lemma Synthesis:** The system extracts available local assumptions and target relations, synthesizing a finer-grained lemma (`lemma_s3_l20_d1`).
- **Integration & Closure:** The successfully verified sub-lemma is integrated back into the main proof, bridging the original causal gap via `smt`.

Phase 1: Coarse Causal-Pair Failure

```
have "2 powr (5::real) = 32" by auto
then have "2 powr a = 2 powr (5::real)" using assms(1) by auto
(* ### Step 3: (Solve for $a$) *)
then have "a = 5" by (simp add: powr_inj)
```

Verifier Error: `Failed to finish proof (line 20): goal (1 subgoal): 2 powr a = 32 ⟹ a = 5`

Phase 2: Diagnosis & Expansion Decision

Structure Judge: `category=MATHEMATICAL_GAP`
"The proof structure is valid, but the simp method with powr_inj is insufficient to discharge the goal."
ITER_CHECK Verdict: `EXTRACT_LEMMA`
"The RAG evidence consistently identifies power-related lemmas as LOW complexity, which justifies extracting the arithmetic subgoals as standalone lemmas."

Phase 3: Sub-lemma Synthesis (Finer-grained Target)

```
lemma lemma_s3_l20_d1:
  fixes a b :: real
  assumes "2 powr a = 32" and "a powr b = 125"
  and h0 : "2 powr (5::real) = 32" and h1 : "2 powr a = 2 powr (5::real)"
  shows "a = 5"
proof -
  . . . (* Intermediate formal steps bridging the exponential injection *) . . .
qed
```

Phase 4: Lemma Integration (Causal-Pair Closed)

```
have "2 powr (5::real) = 32" by auto
then have "2 powr a = 2 powr (5::real)" using assms(1) by auto
(* ### Step 3: (Solve for $a$) *)
then have "a = 5" by (smt (verit) ‹2 powr 5 = 32› assms(1) assms(2) lemma_s3_l20_d1)
then have "5 powr b = 125" using assms(2) by auto
. . .
```

Table 11. Case Study: Causal-Pair Expansion (Sub-lemma Extraction & Integration)

Model: `Gemini 3 Flash`
Type: `Upstream Causal Backtracking`
Problem_ID: `numbertheory_notequiv2i2jasqbsqdiv8`
Math_problem: Let a and b be two integer numbers. Show that the statement "a and b are both even iff $8 \mid a^2 + b^2$" is false.
Module Action Summary:

- **Failed Step:** Step 7.
- **Blamed Step:** Step 1.
- **Root cause:** The original proof introduced the counterexample only narratively, rather than as a formal witness for the existential negation of the universal statement.
- **Mechanism action:** Backtrack to the upstream logical formulation, explicitly restate the goal as an existential negation, and regenerate the downstream proof from that corrected root representation.

Failure diagnosis and abandoned upstream branch
Root-Cause Analysis Log: *"The prover is attempting to prove the general biconditional rather than applying the specific counterexample. Step 1 introduced the counterexample as a narrative 'Let' instead of a formal instantiation of the existential goal."*
Failed Step: Step 7
Blamed Step: Step 1
~~### Step 1: (Define the counterexample): Let $a = 2$ and $b = 4$. We will show that this pair falsifies the biconditional.~~
...
~~### Step 7: (Conclusion): Since $(a, b) = (2, 4)$ is a counterexample, the universal statement is false.~~

Regeneration guidance
Short-term Feedback: *"**When disproving a universal statement, explicitly state the existential negation** and then instantiate it with the concrete witnesses."*
Long-term Memory Context: Retrieved negative historical memories warning against implicit logical gaps; recommended formally closing the search space before invoking automation.

Regenerated downstream branch
Step 1: Define the logical statement and the goal: Let $P(a, b)$ mean "a and b are both even" and $Q(a, b)$ mean "$8 \mid a^2 + b^2$". To disprove $\forall a, b \in \mathbb{Z}, P(a, b) \iff Q(a, b)$, it suffices to show $\exists a, b \in \mathbb{Z}$ **such that** $\neg(P(a, b) \iff Q(a, b))$.
Step 2: Select a specific counterexample: choose $a = 2$ and $b = 4$.
Step 3: Verify $P(2, 4)$: both 2 and 4 are even, so $P(2, 4)$ is true.
Step 4: Compute $a^2 + b^2$: $2^2 + 4^2 = 4 + 16 = 20$.
Step 5: Verify $Q(2, 4)$: $8 \nmid 20$, so $Q(2, 4)$ is false.
Step 6: Demonstrate the failure of the biconditional: $P(2, 4)$ is true while $Q(2, 4)$ is false, hence $P(2, 4) \iff Q(2, 4)$ is false.
Step 7: Conclude the disproof: therefore the universal statement is false.

Table 12. Case Study: Upstream Causal Backtracking

Model: `Gemini 3 Flash`
Type: `Memory-Augmented FIP Search Policy`
Problem_ID: `mathd_algebra_342`
Math_problem: The sum of the first 5 terms of an arithmetic series is 70. The sum of the first 10 terms is 210. What is the first term of the series? Show that it is $\frac{42}{5}$.
Module Action Summary:

- **Failed Step:** Step 3.
- **Blamed Step:** Step 3.
- **Short-term signal:** The verifier cannot automatically expand a fixed small summation such as the five-term case using only `algebra_simps`.
- **Long-term signal:** Historical memory recommends direct evaluation or explicit expansion for small concrete summation targets instead of compressed symbolic derivations.
- **Mechanism action:** Abandon the compressed branch and regenerate a verifier-friendly proof that explicitly writes out the five-term expansion before simplification.

Failure diagnosis and abandoned branch
Root-Cause Analysis Log: *"The error at Step 3 occurs because the Isabelle simplifier cannot automatically expand a summation over a fixed numeral like 5 into its constituent terms using only* `algebra_simps`*."*
Failed Step: Step 3
Blamed Step: Step 3
~~### Step 3: (Sum Formula Application): The sum formula is $S_n = \frac{n}{2}[2a + (n-1)d]$.~~
~~### Step 4: (Formulating Equations): For $n = 5$, $70 = \frac{5}{2}[2a + 4d]$; for $n = 10$, $210 = 5[2a + 9d]$.~~
~~### Step 5: (Solving the System): From $14 = a + 2d$, substitute into $42 = 2a + 9d$ to solve for a.~~

Dual memory guidance
Short-term Memory: *"**Explicitly show the expansion of the terms** $(a + (a + d) + \cdots + (a + 4d))$ before simplifying."*
Long-term Memory: *"For a small concrete summation target, prefer **direct evaluation or explicit expansion** over a compressed symbolic route."*

Regenerated downstream proof
Step 3: (Arithmetic Series Formula): $S_n = \frac{n}{2}[2a + (n-1)d]$.
Step 4: (Applying the Formula for $n = 5$): $70 = \frac{5}{2}[2a+4d]$. **Equivalently,** $70 = a+(a+d)+(a+2d)+(a+3d)+(a+4d) = 5a + 10d$**.** Hence $14 = a + 2d$.
Step 5: (Applying the Formula for $n = 10$): $210 = \frac{10}{2}[2a + 9d] = 5[2a + 9d]$, so $42 = 2a + 9d$.
Step 6: (Expressing d in Terms of a): From $14 = a + 2d$, we obtain $d = \frac{14-a}{2}$.
Step 7: (Solving for a): Substitute into Step 5: $42 = 2a + 9\left(\frac{14-a}{2}\right)$.
Step 8: (Final Calculation): Multiply by 2: $84 = 4a + 126 - 9a = 126 - 5a$, hence $5a = 42$ and $a = \frac{42}{5}$.
Step 9: (Conclusion): The first term of the arithmetic series is $a = \frac{42}{5}$.

Table 13. Case Study: Memory-Augmented FIP Search Policy

Proof-Strategy Revision Category	Mapped FIP Transformation Types	Definition
Strategy-Preserving Revision	`Executabilization`, `Explicitization`, `Redundancy_Pruning`	The overall proof strategy is unchanged. The revision only makes local steps more explicit, more executable, or removes redundant detours.
Strategy-Adjusting Revision	`Step_Insertion`	The overall proof strategy is mostly preserved, but the revision introduces a new intermediate assertion, bridge step, or subgoal to connect existing reasoning gaps.
Strategy-Changing Revision	`Direction_Change`	The top-level mathematical strategy is substantially replaced, such as switching to a different main lemma, proof method, induction scheme, or contradiction-based route.

Table 14. Mapping from FIP Transformation Types to Proof-Strategy Revision Categories.

Metrics	Score
Sample Size	40
Exact Match	65.0% (26/40)
Coverage Match (Human $\subseteq$ LLM)	87.5% (35/40)
Avg. Sample-level Jaccard	80.8%
Micro Precision	80.3%
Micro Recall	90.7%
Micro F1-Score	85.2%

Table 15. Agreement Results for Edit Type Classification

Metrics	Score
Sample Size	40
Pearson Correlation (r)	**0.934**
Mean Absolute Error (MAE)	1.25
Root Mean Square Error (RMSE)	2.51
Mean Deviation (LLM - Human)	-0.25
Exact Match	30.0% (12/40)
Samples with Error $\leqslant 1$	77.5% (31/40)
Samples with Error $\leqslant 2$	95.0% (38/40)

Table 16. Agreement Results for Assertion Counting

Assertion Annotation Instructions

Task Objective and Core Definition

Task objective: Given a mathematical informal proof segmented by `###Step...`, count the total number of *continuous causal assertion chains*. This task differs from earlier explicit-intermediate-assertion extraction: annotators count *chains*, not individual equations or statements.

Core definition: Within one step, an assertion chain is a continuous local causal progression in which an earlier proposition is used to derive a later proposition.

Counting Rules

- **Unit of counting:** Count chains rather than individual formulas. A chain may contain two, three, or more consecutive propositions and is still counted as one chain.
- **Multiple chains within one step:** If one step contains two or more independent lines of reasoning, count them separately.
- **No cross-step carry-over:** A chain cannot cross step boundaries. Counting restarts at each new step.
- **Non-counted cases:** Isolated statements, single-hop derivations, purely procedural prompts, purely narrative remarks, and closing phrases are counted as zero. If the premise appears only in the previous step and the current step states only the result, the current step is also counted as zero.
- **Chained equalities/inequalities:** Consecutive equalities, inequalities, or implication-like expressions in the same local thread count as one chain. If the reasoning branches, each branch is counted separately.

How to Determine the Assertion Count Inside One Step

Step 1: Identify each valid reasoning segment. A valid reasoning segment is a locally complete causal thread centered on the same subgoal within one step. It is more than a single ordinary update. Single isolated statements, single-hop derivations, procedural prompts, closing remarks, or cases where the premise is only in the previous step do not form a valid reasoning segment.

Step 2: Count the number of reasoning transitions. For each valid reasoning segment, count how many consecutive transitions it contains.

Step 3: Convert transitions into assertions. For each valid reasoning segment, use `assertion_count=continuous_logical_chain-1`.

Step 4: Sum independent segments. If a step contains multiple independent valid reasoning segments, apply the conversion rule to each segment separately and sum the results. Do not subtract once from the whole step. Typical signs of independence include an explicit branch marker, a shift to another subproblem, or two derivations that do not share intermediate links.

Table 17. Human annotation instructions for assertion counting.

Proof-Edit-Type Annotation Instructions

Task Setting and Legal Labels
Human-filled field: `edit_types` (a string array)
Legal labels: `Explicitization`, `Step_Insertion`, `Redundancy_Pruning`, and `Direction_change`. Spelling must match exactly.

Label Definitions

- **`Explicitization`:** The proof direction stays unchanged, while a previously compressed local jump is expanded into a finer-grained reasoning sequence whose newly introduced content is consumed immediately.
- **`Step_Insertion`:** The proof direction stays unchanged, while the revision inserts a bridge statement or subgoal that is consumed later across steps rather than immediately.
- **`Redundancy_Pruning`:** The proof direction stays unchanged, while detours, intermediate anchors, or unnecessary steps are removed so that the proof becomes shorter and more direct.
- **`Direction_change`:** The core proof strategy is fundamentally replaced. This label has the highest priority.

Primary Funnel-Style Decision Process
Scope of the funnel: First judge only the four core topological labels: `Direction_change`, `Redundancy_Pruning`, `Step_Insertion`, and `Explicitization`.
Step 1 (highest priority): Ask whether the core proof strategy has been fundamentally replaced. If yes, assign `Direction_change` and stop immediately.
Step 2 (only if Step 1 is no): Ask whether intermediate anchors or detour steps were removed, making the proof chain shorter and more direct. If yes, add `Redundancy_Pruning` and stop the funnel.
Step 3 (only if Steps 1 and 2 are both no): Inspect the newly added content. If it is suspended for later cross-step use, assign `Step_Insertion`. If it is introduced and consumed immediately inside a local chain, assign `Explicitization`.

Multi-Label Output Rules
Mutual exclusivity of `Direction_change`: Once `Direction_change` is assigned, it must be the only label in `edit_types`.
Combination rule: `Step_Insertion` and `Explicitization` are usually mutually exclusive for the same newly added span, but they may co-occur when they describe different local modifications in the same proof.

Table 18. Human annotation instructions for proof edit types.

Model	Hyperparameters

Gemini 3.1 Pro (DeepMind, 2026a)	`model = gemini-3.1-pro-preview`
	`top_p = 0.95 (default)`
	`max_tokens = 60000`
	`temperature = 0.0` (Structure Judge)
	`temperature = 0.2` (FIP-to-FP Projection Map)
	`temperature = 0.4` (Root FIP-state Initialization)
	`temperature = 0.9` (Projected-Candidate Repair)

Gemini 3 Flash (DeepMind, 2026b)	`model = gemini-3-flash-preview`
	`top_p = 0.95 (default)`
	`max_tokens = 60000`
	`temperature = 0.0` (Verification, Evidence-based Branch Expandability Analysis, Text normalization)
	`temperature = 0.2` (Upstream Causal Backtracking)
	`temperature = 0.7` (Raw IP Generation, Direction Rewrite, Convert Formal Lemma to Informal)
	`temperature = 0.9` (Root FIP-state Judge)

Qwen-8B (Yang et al., 2025)	`model = qwen-8b`
	`enable_thinking = true`
	`temperature = 0.2` (Generate Formal Proof)
	`temperature = 0.7` (Generate Informal Proof)

Qwen3-30B-A3B (Yang et al., 2025)	`model = qwen3-30b-a3b`
	`enable_thinking = true`
	`temperature = 0.2` (Generate Formal Proof)
	`temperature = 0.7` (Generate Informal Proof)

DeepSeek-V3.2-Exp (DeepSeek-AI, 2025)	`model = deepseek-v3.2-exp` `temperature = 0.2` (Generate Formal Proof) `temperature = 0.7` (Generate Informal Proof)

GPT-5-Codex (OpenAI, 2025)	`model = gpt-5-codex`
	`temperature = N/A (unsupported)`
	`all_parameters = default`

Table 19. Model versions and key hyperparameters under the unified paired evaluation protocol.

Prompt for Raw Informal Proof Generation

User Prompt

TASK: Write an informal mathematical proof for the given problem.

IMPORTANT OUTPUT FORMAT (for easy extraction)

You MUST output exactly one block delimited by the following markers, and nothing else:

INFORMAL_PROOF_START >>>

(your proof text)

INFORMAL_PROOF_END >>>

INSTRUCTIONS

1. Prove the statement. You may choose ANY proof style (narrative, case analysis, contradiction, algebraic manipulation, etc.).
2. Use clear mathematical reasoning and include key intermediate arguments needed for correctness.
3. Keep the proof self-contained and avoid irrelevant commentary.

```
>>> PROBLEM STATEMENT:
{informal_statement}
```

Prompt for Root FIP-state Initialization

System Prompt
You are an expert Mathematical Logic Refiner. Your goal is to bridge the huge gap between "Human Informal Proofs" and "Isabelle/Isar Formal Proofs".
You will receive a raw Informal Proof. Your task is to rewrite it into a **Structured, Type-Explicit, Logic-First** proof script.

The 3 Laws of Refinement (CRITICAL RULES)

1. **Type & Scope Anchoring (Prevent "nat" errors):**
   – If the problem involves subtraction of variables ($a - b$) or mentions "integer roots", you MUST explicitly declare variables as **Integers** (`int`), not Natural Numbers (`nat`). Isar's `nat` subtraction truncates ($3 - 5 = 0$), which breaks proofs.
   – *Trigger:* "integer", "subtraction", "roots".
2. **Calculation Black-boxing (Simp > Set):**
   – Do NOT describe simple counting using complex Set Theory (e.g., "cardinality of set"). Isabelle's automation struggles with sets but excels at algebra.
   – Instead, convert set problems into **Algebraic Equations** (e.g., "Solve $2k - 1 = 199$").
   – *Trigger:* "cardinality", "number of terms", "set".
3. **Constraint Triggering (Safe Inequalities):**
   – When dividing an inequality or multiplying both sides, you MUST explicitly state the sign of the term (e.g., "Since $1 - c > 0$"). Isar will not divide inequalities without this proof.
   – *Trigger:* "divide", "multiply", "reciprocal", ">", "<".

Workflow

1. **Scan**: Read the input paragraph.
2. **Diagnose**: Output a `[Diagnosis Log]` listing specific risks found based on the 3 Laws.
3. **Refine**: Output the `[Refined Proof Script]` using a numbered list. Use standard logical verbs (Define, Assume, Have, Show).

Part 1: Refinement Skill Demonstrations

```
Study these examples to learn **HOW** to diagnose and fix logical issues using the 3 Laws.

[3 more Root FIP-state Initialization examples...]
```

User Prompt
Part 2: Domain Style References

```
These are valid proofs from the Archive of Formal Proofs (AFP) for problems similar to the current
↪  task. Use them ONLY as a reference for **granularity, tone, and step-size**.

### Reference A
{{afp_example_1}}

### Reference B
{{afp_example_2}}

### Reference C
{{afp_example_3}}

---

# Execution Task
**Instructions:**
1.  **Analyze**: Read the **Target Proof** below.
2.  **Diagnose**: Output a `[Diagnosis Log]` (following Part 1 methodology) identifying risks based on
↪  the 3 Laws.
3.  **Refine**: Rewrite the proof into a `[Refined Proof Script]`.
    * **Logic**: Fix all issues identified in the diagnosis.
    * **Style**: Mimic the clarity and structure of the **Part 2** references.

**Target Proof:**
{{current_proof_text}}

**Output:**
```

Prompt for Root FIP-state Iteration

System Prompt
You are an expert Mathematical Logic Repair Agent. Your goal is to **fix, split, or improve** an existing Structured Proof Script based on specific feedback.
You will receive:

1. **Current Steps**: A structured proof script (formatted with `### Step N`).
2. **Feedback/Issue**: A specific reason why the previous version was rejected (e.g., "Step 2 is too complex", "Missing variable definition").

Your task is to rewrite the steps to address the Feedback while **strictly preserving** the correctness and style of the unmodified steps.

The 3 Laws of Iteration

1. **Decomposition (Split, Don't Invent):**
   – If Feedback says a step is "too complex", split the *existing logic* into multiple atomic `### Step` entries.
   – **Constraint**: Do NOT add new mathematical derivations (like expanding odd to `2k+1`) unless the Feedback explicitly asks for it. Just split the logic structure.
2. **Preservation (Context Consistency):**
   – Keep variable names, types, and the flow of the proof identical to the input, unless asked to fix them.
3. **Explicit Injection (Fill Gaps):**
   – If Feedback says "Information Missing", insert a new `### Step` containing *exactly* the missing info described in the Feedback.

Workflow

1. **Analyze**: Read the `Current Steps` and the `Feedback`.

2. **Locate**: Identify the specific `### Step` index mentioned in the Feedback.
3. **Rewrite**: Output the corrected script.

```
# Part 1: Repair Skill Demonstrations (Real-World Density)

[3 more Root FIP-state Iteration examples...]

---

# Execution Task
**Instructions:**
1. Read the **Current Steps** and **Feedback** below.
2. Output the corrected `[Refined Proof Script]` using the `### Step N` format.
```

User Prompt

```
# Execution Task
**Current Steps:**
{{current_steps}}

**Feedback:**
{{feedback_reason}}

**Output:**
```

Prompt for Root-state Judgement

User Prompt

You are a **Proof Refinement Auditor** for an auto-formalization system. Your goal is to compare the **Original Proof Text** (Unrefined) with the **Refined Steps** (Refined) to ensure the refinement is ready for formal verification.

Input Data

1. **Original Text**: The raw informal proof segment.
2. **Refined Steps**: The structured steps generated by the Refiner.

Evaluation Criteria (The "Relative Check")

1. Information Fidelity (Crucial)

- **Rule**: Does the Refined version contain ALL key mathematical details from the Original Text?
- **Semantic Equivalence**: Semantic equivalence is allowed.
  – *Example*: Replacing "$x > 0$" with "using lemma `x_pos`" is **ACCEPTABLE**.
  – *Example*: Replacing "Divide by 2" with "Apply division" is **ACCEPTABLE**.
- **Forbidden**: However, **dropping explicit conditions** is FORBIDDEN.
  – *Fail Case*: If Original says "Since n is prime and $n > 2$..." and Refined just says "Therefore...", missing the "prime" condition, this is a FAIL.

2. Structural Decomposition (Decomposition Check)

- **Rule**: Did the Refiner actually break down complex logic into sequential steps?
  – If the Original contains connectors like "because", "since", "implies", "and", the Refined version **MUST** split these into separate steps.
- **Over-refinement Policy**: **Do NOT penalize over-refinement.**
  – It is acceptable (and safer) if the Refiner splits a simple equation like "$x = y + z + 1$" into multiple small variable assignments.
  – **Only penalize UNDER-refinement** (leaving complex logical jumps intact).

Output Format

Output **ONLY** a JSON object:

```
{
  "reason": "Brief explanation. If FAIL, specify what info is missing or where decomposition is
  ↪  needed.",
  "suggestion": "Specific instruction for the Refiner." (Optional),
  "status": "PASS" | "FAIL"
}


# Few-Shot Examples

[2 more Root-state Judgement examples...]

# Current Task
**Original Text**:
{{original_text}}

**Refined Steps**:
{{refined_steps}}

# Audit Output (JSON)
```

Prompt for FIP-to-FP Projection Map

User Prompt

INSTRUCTION

You are a mathematician expert in Isabelle formal language. Your task is to convert informal mathematical proofs into complete Isabelle formal proofs.

Output Format Requirements (CRITICAL)

You MUST generate your response in this EXACT format:

```
>>> FORMAL_PROOF_START >>>
[complete_formal_proof_here]
>>> FORMAL_PROOF_END >>>
```

CRITICAL FORMATTING RULES:

1. Use the EXACT delimiters: ">>> FORMAL_PROOF_START >>>" and ">>> FORMAL_PROOF_END >>>"
2. Your output MUST start with the EXACT theorem/lemma statement provided in the "Formal Statement to Use" section
3. Following the statement, generate the proof block (`proof - ... qed`)
4. NO markdown code block markers (“‘) around the Isabelle code
5. The formal proof should be a single continuous Isabelle proof block starting with theorem/lemma statement
6. Do NOT break the code into multiple segments with explanations
7. [Important] MUST include step comments in the format (*### Step X: description*) for EVERY major proof step
8. Start with the complete theorem/lemma statement followed by the proof block

CRITICAL WARNING:

- ALWAYS include BOTH ">>> FORMAL_PROOF_START >>>" AND ">>> FORMAL_PROOF_END >>>" delimiters

- Include the COMPLETE theorem statement (fixes, assumes, shows) AND the proof
- If you miss either delimiter, the proof extraction will FAIL
- The delimiters are case-sensitive and must be EXACTLY as shown
- NO additional text should appear after ">>> FORMAL_PROOF_END >>>"

Step Comment Requirements (MANDATORY)

- **EVERY proof must include step comments using the format: (*### Step X: description*)**
- Step comments should correspond to the steps in the informal proof
- Each major logical step in the proof should have a corresponding step comment
- Comments should be clear and descriptive
- Example format: (*### Step 1: Establish positivity conditions*)

Isabelle Syntax Guidelines

1. Define all variables with types: `x y :: real` for $x + y = 12$
2. Product: `*` (e.g., `2 * x`, `2 * x * y`)
3. Division: `/` (e.g., `1 / (2 * x)`, `1 / (2 * x * y)`)
4. Power: `^` (e.g., `x ^ 2`, `x ^ 2 * y ^ 2`)
5. Square root: `sqrt` (e.g., `sqrt 2`, `sqrt (x * y)`)
6. Absolute value: `abs` (e.g., `abs x`, `abs (x * y)`)

Proof Strategy

- Consider all terms' corresponding grammatical expressions in Isabelle instead of LaTeX expressions
- Ensure proofs are well-organized with clear logical flow
- Use `sledgehammer` for uncertain tactics
- **ALWAYS include step comments in the format (*### Step X: ...*)**

```
# EXAMPLES

[3 more FIP-to-FP Projection Map examples...]

# TASK

**Notice: The formal proof you generate should contain a description of each step, the format of
↪  description like this: (* ### Step X: ... *)**

Given the problem, informal proof, and formal statement below, generate the formal proof following the
↪  EXACT format shown in examples above:

>>> Problem: {informal_statement}

>>> Informal Proof: {informal_proof}

>>> Formal Statement to Use: {formal_statement}

>>> FORMAL_PROOF_START >>>
```

Prompt for Informal Proof Mathematical Verification

User Prompt

TASK: Verify whether given conditions are sufficient to prove a mathematical conclusion.

INSTRUCTIONS

1. You will be given mathematical "Conditions" and a "Conclusion to be verified".
2. Analyze whether the conclusion can be logically derived from the given conditions.
3. Check each step in the conclusion for logical validity and mathematical correctness.
4. Identify any gaps, errors, or unjustified assumptions in the reasoning.
5. Provide a detailed analysis explaining your verification process.
6. Give a final answer: True (if the conclusion can be proven) or False (if it cannot).
7. Present your final answer in the format: \boxedTrue or \boxedFalse.

OUTPUT FORMAT (STRICT)

- Your VERY LAST line MUST be exactly one of:
  – \boxedTrue
  – \boxedFalse
- Do NOT output any other \boxed... anywhere in the response.
- Do NOT output anything after the final \boxedTrue/\boxedFalse.
- If unsure, output \boxedFalse.

IMPORTANT CLARIFICATION

- You are NOT asked to compute the final numeric answer of the original problem.
- You are asked to judge whether "Conclusion to be verified" is logically derivable from the given "Conditions" (and previous steps).
- Even if you can compute numbers, you MUST still output only \boxedTrue or \boxedFalse as required.

```
>>> EXAMPLES:

[2 more informal proof mathematical verification examples...]

>>> CONDITIONS: {informal_statement}
{previous_steps}

>>> CONCLUSION TO BE VERIFIED: {current_step_content}

>>> VERIFICATION ANALYSIS:
```

Prompt for Structure Judgement

User Prompt

You are an **Isabelle/Isar Failed-Step Diagnostic Expert**. Your job is to classify a failed proof step into exactly one of two root-cause categories:

- **STRUCTURAL_ERROR**: the proof script has an invalid Isar structure/state transition/context management issue (Isabelle "cannot make sense of the script"), even if the mathematics would be true.
- **MATHEMATICAL_GAP**: the proof structure is valid, but the current automation and available lemmas are insufficient to

prove the goal (a genuine reasoning/lemma gap).

Your classification is used to route the next stage:

- structural issues → **code repair**
- mathematical issues → **lemma extraction / step-by-step proof**

Input

You will receive two text sections (in this fixed order):

1. **Code Context**: the failing line ("Failed Proof Step") plus several lines of preceding context (may include `proof/next/qed`, `fix/assume/have/show`, `also/finally`, etc.).
2. **Error Log**: Isabelle error/warning messages (often `Failed to apply proof method`, `Illegal application of proof command`, `Unknown variable`, `Calculation not possible`, etc.).

Task

Do **only** classification with minimal justification:

1. Choose `category`: `STRUCTURAL_ERROR` or `MATHEMATICAL_GAP`
2. Provide a one-sentence `reason` (root cause only; no long tutorial)
3. Provide `confidence` (0.0–1.0)
4. Provide `evidence`: quote **1–3** short, decisive snippets copied verbatim from Code Context and/or Error Log
5. Provide `next_action`: must be `"CODE_REPAIR"` or `"LEMMA_EXTRACTION"`

Hard requirement: your final output must be **ONLY** a single JSON object (no Markdown, no extra text).

Decision Procedure (stop at the first match)
Step 1: Is there a strong "structure/context/syntax" signal?

If the Error Log or the context contains any strong signal below, output **STRUCTURAL_ERROR**:

- **Invalid Isar state or illegal command application**
  – e.g. `Illegal application of proof command`, `Illegal state`, `Local statement fails` (pointing to `have/show` structure)
- **Scope / binding / context mismatch**
  – e.g. `Unknown variable`, `Variable ... not fixed`, `Undeclared...`, `Free variable`, or goal mentions variables not introduced by `fix/assume`
- **Broken calculation/proof chain**
  – e.g. `Calculation not possible`, errors around `also/finally`
  – typical: `have "... " for x` / `CURSORTOKEN1BLOCKx` / `CURSORTOKEN2BLOCKx` followed by `also`; or switching binders mid-chain
- **Induction / cases structure incomplete causing context mismatch**
  – typical: missing generalization (`arbitrary:`) or missing `fix` in induction step; misusing `moreover/ultimately` where `cases/consider` is needed
- **Type/parse failures that look like "script form" errors**
  – e.g. `Type unification failed` accompanied by wrong arity/notation/binder usage

Rule: If Isabelle indicates a command/state/scope problem, prefer **STRUCTURAL_ERROR** even if the goal is also hard.
Step 2: Otherwise, is it mainly "method cannot solve the goal"?

If the Error Log is primarily tactic/automation failure without scope/state complaints, output **MATHEMATICAL_GAP**:

- `Failed to apply proof method`
- `Auto tactic failed`
- `Sledgehammer found no proof`
- `simp/auto/blast/metis` cannot finish, while the context structure looks standard and well-scoped (`fix → assume → have → show`, or a complete induction/cases skeleton)

Step 3: Ambiguous cases (still must choose one)

- If both kinds of hints appear, choose **STRUCTURAL_ERROR** (structure errors can block any math).
- If the log is too short (only "failed"), then:
  – if the context shows structural risk patterns (calculation chains, incomplete induction step), choose STRUCTURAL_ERROR;
  – else choose MATHEMATICAL_GAP and lower `confidence`.

Output Schema (JSON)

Your output must satisfy:

- `category`: `"STRUCTURAL_ERROR"` or `"MATHEMATICAL_GAP"`
- `next_action`: `"CODE_REPAIR"` or `"LEMMA_EXTRACTION"` (one-to-one mapping with `category`)
- `confidence`: number in [0, 1]
- `reason`: one sentence
- `evidence`: array of 1–3 strings (verbatim quotes)

Schema example (illustrative only):

```
{
  "error_category": "STRUCTURAL / TYPE / SYNTAX",
  "diagnosis": "Brief explanation...",
  "original_code_to_replace": "The exact string from the input context that needs to be removed.",
  "fixed_code": "The new string to insert in its place."
}
```

```
## Examples

[2 more Structure Judgement examples...]

---

## Current Input
{code_content}
{error_log}
```

Prompt for Projected-Candidate Repair

System Prompt
You are an **Isabelle/HOL Syntax & Structure Repair Engine**. Your goal is to fix broken Isabelle code based solely on the provided Error Log and Code Context.

Core Mechanism: Find and Replace

To enable the automated script to apply your fix, you must identify:

1. **The Target**: The exact substring of code from the input that is causing the error (or needs restructuring).

2. **The Replacement**: The corrected version of that string.

Input Data

1. **Failed Code Context**: A snippet of code containing the error.
2. **Error Log**: The exact error message.

Diagnostic Handbook (Strict Rules)

Analyze the Error Log to identify the Root Cause and apply the specific fix:

Category A: Context & Structural Mismatches (High Priority)

- **Symptom**: `Calculation not possible`, `Failed to refine any pending goal`, `Illegal application of proof command`.
- **Diagnosis**: The Isar state does not support the current command. This is often due to improper scope or quantifier barriers.
- **Fix Rules**:
  – **The "also have" Trap**: If `also have` fails and the previous line ends with `... for x` (quantified), you **MUST** refactor the block.
  – *Wrong*: `have "P x" for x ... also have ...`
  – *Fix*: Change to `fix x have "P x" ... also have ...`
  – **The "moreover" Trap**: If `moreover` or `ultimately` fails, ensure you are not mixing them with `also/finally` incorrectly.
  – **The "induction" Trap**: If a proof fails at the start of induction, check if variables need to be generalized using `arbitrary: ....`

Category B: Type System Errors

- **Symptom**: `Type unification failed`, `No type arity`, `atomic type expected`.
- **Fix Rules**:
  – Look for numeric literals (e.g., `1`, `0`) used in ambiguous contexts. Force types using `(1::real)` or `(1::complex)`.
  – Check for mixed `nat` (natural number) subtraction. `a - b` in `nat` is 0 if `b > a`. If the logic implies negative numbers, cast to `int`.

Category C: Syntax & Symbol Errors

- **Symptom**: `Outer syntax error`, `Inner syntax error`, `Undefined constant`.
- **Fix Rules**:
  – **Typos**: Correct obvious misspellings (e.g., `lenth` → `length`, `Sum` → `sum`).
  – **Operators**: Ensure operators like `SUM`, `INT` use the correct Isabelle ASCII/Symbol syntax (e.g., `\sum` vs `sum`).
  – **Delimiters**: Fix missing quotes `"` or parentheses.

Constraints

1. **Output JSON ONLY**.
2. **Verbatim Match**: The `original_code_to_replace` field MUST be an **exact substring** of the provided **Failed Code Context**. Do not change indentation or spacing in the "original" field, or the replacement will fail.
3. **Minimal Context**: Include enough context in `original_code_to_replace` to ensure it is unique, but do not replace the entire file if only one line is wrong.
4. **Structural Integrity**: For structural errors (Category A), make sure to select the **entire block** (e.g., the previous `have` line AND the current `also` line) as the target to replace.

```
# Few-shot Examples (real repair patterns)

[2 more Projected-Candidate Repair examples...]

# Output Format
```

```
{
  "error_category": "STRUCTURAL / TYPE / SYNTAX",
  "diagnosis": "Brief explanation...",
  "original_code_to_replace": "The exact string from the input context that needs to be removed.",
  "fixed_code": "The new string to insert in its place."
}
```

User Prompt
Repair Task

1. Failed Code Context

The code causing the error (marked with >>>) and its context:

```
{formal code}
```

2. Isabelle Error Log

The compiler's complaint:

```
{error message}
```

Instruction

Based on the Error Log, identify the syntax or structural flaw. If this is a **Structural Error** (e.g., "Calculation not possible" due to quantifiers), rewrite the context (e.g., use `fix x`) to make the state transition valid. Generate the JSON output with the fixed code.

Prompt for Branch Expandabiliy Analysis

User Prompt

Instructions (Role: Feasibility Assessor)

You are the gatekeeper for the Lemma Extraction module. Your goal is to determine if the **Failed Subgoal** is a feasible candidate for a standalone lemma based on AFP evidence retrieved by RAG.
Important: `AFP_RAG` already contains BOTH:

- the AFP snippet(s)
- the associated Manual Metadata (e.g., `[COMPLEXITY: LOW]`, `[COMPLEXITY: HIGH]`, `[NO MATCH]`)

Decision Rules

Step 1) Evidence-Based Complexity Analysis

- Scenario A: `[NO MATCH]` appears in `AFP_RAG`
  – Decision: PRUNE (hallucination risk / library gap)
- Scenario B: `[COMPLEXITY: HIGH]` appears in `AFP_RAG`
  – Decision: PRUNE (too hard; needs splitting)
- Scenario C: `[COMPLEXITY: LOW]` appears in `AFP_RAG`
  – Decision: EXTRACT_LEMMA
- If metadata is missing/ambiguous:
  – If the AFP snippet clearly indicates a trivial/one-liner proof, treat as LOW → EXTRACT_LEMMA
  – Otherwise be conservative → PRUNE

CRITICAL RULE

- Do NOT evaluate whether the lemma is "general" or "reusable".
- If the RAG evidence suggests LOW complexity, you MUST output EXTRACT_LEMMA.

Step 2) Missing Assumption Analysis (Only when verdict is EXTRACT_LEMMA)

- Compare the AFP reference snippet(s) inside `AFP_RAG` with the Failed Subgoal.
- If the AFP snippet contains explicit assumptions that are missing (e.g., `finite A`, `x > 0`, typeclass constraints), list them in `payload.missing_assumptions`.
- If none, use an empty list.

Output (HARD REQUIREMENT)

- Output ONLY a single JSON object (no Markdown, no code fences, no extra text).
- Output schema:
  – `branch`: must be `"TACTICS"`
  – `rag_analysis`: one short sentence
  – `verdict`: must be `"PRUNE"` or `"EXTRACT_LEMMA"`
  – `payload`:
  – `reason`: required when verdict is `"PRUNE"` (otherwise can be empty string)
  – `missing_assumptions`: array of strings (use `[]` if none)
  – `confidence`: number in [0, 1]

JSON schema example (copy this structure; do NOT add any surrounding "‘ fences):

```
{
  "branch": "TACTICS",
  "rag_analysis": "Found LOW-complexity AFP match.",
  "verdict": "EXTRACT_LEMMA",
  "payload": {
    "reason": "",
    "missing_assumptions": [
      "finite S"
    ],
    "confidence": 0.82
  }
}
```

Inputs

1) Failed Subgoal (Formal): formal_proof
2) Simulated RAG Evidence (From AFP) + Manual Metadata: AFP_RAG
After reading the Inputs above, return the JSON object now.

Prompt for Convert Lemma Statement into Informal Statement

User Prompt

Please convert the following formal Isabelle lemma into a clear and understandable informal mathematical statement.

```
The formal lemma contains mathematical conditions and a goal to prove. Your task is to translate this
↪  into natural language that explains what the lemma is trying to establish.

IMPORTANT OUTPUT FORMAT:
You must output exactly one block delimited by the following markers, and nothing else.

>>> INFORMAL_STATEMENT_START >>>
(your informal statement)
>>> INFORMAL_STATEMENT_END >>>

Guidelines:
1. Extract the main mathematical concepts and relationships.
2. Explain what conditions are assumed from the `fixes` and `assumes` clauses.
3. Clearly state what needs to be proven from the `shows` clause.
4. Use clear mathematical language that a mathematician would understand.
```

```
5. Keep the mathematical precision while making it readable.
6. Do not include analysis, hidden reasoning, or any text outside the markers.

Formal lemma:
{formal_lemma}
```

Prompt for Informal Proof Generation

User Prompt

TASK: Generate a complete mathematical proof for the given problem.

INSTRUCTIONS

1. You will be given a mathematical problem statement.
2. Create a step-by-step proof that is logically sound and mathematically rigorous.
3. Use clear mathematical notation and proper formatting.
4. Each step should be numbered and include both the mathematical work and explanation.
5. Show all intermediate calculations and reasoning.
6. Conclude with a clear final answer or statement.
7. Use the format: ### Step X: [Description] followed by the mathematical work and explanation.

```
>>> EXAMPLES:

[2 more informal proving examples...]

>>> PROBLEM STATEMENT: {informal_statement}

>>> SOLUTION:
```

Prompt for Informal Proof Generation with Feedback

User Prompt

TASK: Generate a complete mathematical proof for the given problem.

INSTRUCTIONS

1. You will be given a mathematical problem statement and specific approaches to avoid.
2. Create a step-by-step proof that is logically sound and mathematically rigorous.
3. Use clear mathematical notation and proper formatting.
4. Each step should be numbered and include both the mathematical work and explanation.
5. Avoid the specified problem-solving approaches mentioned in the feedback.
6. Conclude with a clear final answer or statement.
7. Use the format: ### Step X: [Description] followed by the mathematical work and explanation.

```
>>> EXAMPLES:

[2 more informal proving with feedback examples...]

>>> PROBLEM STATEMENT: {informal_statement}
```

```
>>> APPROACHES TO AVOID: {feedback}

>>> SOLUTION:
```

Prompt for Upstream Causal Backtracking

User Prompt

You are an expert Mathematical Logic Analyzer and Isabelle/Isar Assistant. Your task is to analyze a failed proof attempt, identify the root cause, and output a structured JSON analysis.

Task Definitions & Field Rules

You must generate a JSON object. **CRITICAL: Generate the fields in the exact order below to ensure logical deduction.**

1. **raw_error_message** (string):
   – The original error message (provided in input).

2. **error_step_index** (integer):
   – The step where the Verifier actually reported the error (provided in input).

3. **root_cause_analysis** (string):
   – **Goal**: Trace the error back to its origin using Chain of Thought reasoning.
   – **Process**: Start from the Error Step and look backwards. Ask: "Why did this fail? Does it rely on a previous definition, assumption, or simplification that was too weak or risky?"
   – **Output**: A logic trace explaining *why* a specific earlier step is responsible for the later failure.

4. **bad_pattern_raw** (string) → *For Vector Database Indexing*:
   – **Goal**: Abstract feature extraction.
   – **Style**: Objective, declarative, third-person.
   – **Abstraction**: Replace specific variable names (x, y, n) with types or roles (e.g., "a variable", "the denominator").
   – **Format**: "Action [X] performed on Context [Y] leading to Risk [Z]."
   – **Constraint**: Do NOT use second-person ("You"). Do NOT include specific numbers unless they are mathematically significant constants (like 0 or 1).

5. **feedback_text** (string) → *For the Proof Generator*:
   – **Goal**: Instruction for correction that survives backtracking.
   – **Style**: Directive, second-person ("You"), constructive.
   – **CRITICAL REQUIREMENT**: **Do NOT rely solely on specific step numbers** (e.g., "In Step 4..."), because step numbers will change when the user regenerates the proof.
   – **Format**: Instead, describe the **faulty action**. Use phrases like:
   – "In the previous attempt, the step where you [Action X]..."
   – "Defining [Variable] as [Formula] caused..."
   – "The logic involving [Strategy Y] led to..."
   – Link the cause and effect clearly.

6. **blamed_step_index** (integer):
   – **The Conclusion**: Based on your `root_cause_analysis` above, identify the specific step index where the wrong path was taken.
   – This is the step that needs to be rewritten (the destination of the backtrack).

Output Format

- Output **ONLY** a valid JSON object.
- Do not include markdown formatting (like "`json ...` ") unless requested.
- Do not add conversational text like "Here is the analysis".

```
# Few-Shot Examples

[2 more Upstream Causal Backtracking examples...]
```

```
# Current Task Input
1. Proof History:
{{proof_history}}
2. Error Step: Step {{error_step_index}}
3. Error Message: {{error_message}}

# Analysis Output (JSON)
```

Prompt for Text normalization

User Prompt

You are a Mathematical Feature Extractor for a vector retrieval system. Your task is to convert raw descriptions of proof attempts and errors into high-level, abstract Semantic Tags.

Input Format

You will receive:

1. Action: What the prover tried to do (from bad_pattern_raw).
2. Resulting Error: The system error message (from raw_error_message).

Output Rules (Strict)

1. Format: Output a comma-separated list of tags in brackets:

   `[Category: Value], [Category: Value], ...`

2. Abstraction:
   – IGNORE specific variable names (x, y, n, f), numbers, or function names.
   – GENERALIZE them to roles: Variable, Expression, Polynomial, Denominator.

3. Taxonomy (Use these categories):
   – [Domain: ...] (e.g., Algebra, Number Theory, Calculus, Logic)
   – [Operation: ...] (e.g., Division, Integration, Induction, Simplification)
   – [Strategy: ...] (e.g., Substitution, Case Split, Contradiction, Term-wise)
   – [Risk: ...] (e.g., Division by Zero, Type Mismatch, Timeout, Bound too loose)

4. Constraints:
   – Do NOT output JSON.
   – Do NOT output extra text, headers, or markdown code fences.
   – Always output at least 2 tags; include [Risk: ...] if you can infer it from Resulting Error.

```
# Few-Shot Examples

[3 more Text normalization examples...]

# Current Task
Input:
Action: {{bad_pattern_raw}}
Resulting Error: {{raw_error_message}}

Output Tags:
```

Prompt for Direction Rewrite

User Prompt

TASK: Complete the following mathematical proof.

INSTRUCTIONS

1. You will be given a ”Problem” and an ”Incomplete Proof”.

2. Your response must be the FULL, completed proof.
3. Your response MUST start by repeating the "Incomplete Proof" section verbatim.
4. After repeating the incomplete proof, continue DIRECTLY with the next logical steps.
5. DO NOT write any introductory text, conversation, or explanation. Your response must begin with the first line of the proof.
6. Maintain the exact style and formatting (`### Step ...`) of the original.

```
[2 more Direction rewrite examples...]

>>> Problem: {informal_statement}
>>> Incomplete Proof: {informal_proof}
>>> Full Proof:
```

Prompt for Direction Rewrite with Feedback

User Prompt

TASK: Complete the following mathematical proof while avoiding specified approaches.

INSTRUCTIONS

1. You will be given a "Problem", an "Incomplete Proof", and "Approaches to Avoid".
2. Your response must be the FULL, completed proof.
3. Your response MUST start by repeating the "Incomplete Proof" section verbatim.
4. After repeating the incomplete proof, continue DIRECTLY with the next logical steps.
5. DO NOT write any introductory text, conversation, or explanation. Your response must begin with the first line of the proof.
6. Maintain the exact style and formatting (`### Step ...`) of the original.
7. AVOID using any of the specified approaches mentioned in the feedback.

```
>>> EXAMPLES:

[2 more Direction Rewrite with Feedback example...]

>>> PROBLEM: {informal_statement}
>>> INCOMPLETE PROOF: {informal_proof}
>>> APPROACHES TO AVOID: {feedback}
>>> FULL PROOF:
```

Prompt for Contextual FP Projection Map

User Prompt

You are an expert Isabelle/HOL theorem prover tasked with continuing formal proof development. Your goal is to complete partial formal proofs by generating the remaining proof steps in proper Isabelle syntax.

Task Description

You will be given:

1. An informal mathematical statement and proof
2. A partial formal Isabelle proof that needs continuation
3. You must generate the remaining proof steps to complete the formal proof

Important Instructions

1. **Continuation Task**: You are continuing an existing proof, not starting from scratch

2. **Maintain Consistency**: Ensure your continuation aligns with the existing proof structure and variable definitions
3. **Proper Syntax**: Use correct Isabelle/HOL syntax and tactics
4. **Logical Flow**: Follow the informal proof structure when possible
5. **Use Sledgehammer**: When uncertain about tactics, use "sledgehammer" to find appropriate proof methods

Output Format Requirements (CRITICAL)

You MUST provide your continuation in this EXACT format:

```
>>> CONTINUATION_START >>>
[Your continuation content only - do not repeat the "Proof So Far" section]
>>> CONTINUATION_END >>>
```

CRITICAL FORMATTING RULES:

1. Use the EXACT delimiters: ">>> CONTINUATION_START >>>" and ">>> CONTINUATION_END >>>"
2. Generate ONLY the continuation part between the delimiters
3. Do NOT repeat any steps from the "Proof So Far" section
4. The continuation should seamlessly connect to the existing proof
5. End with appropriate closing statements (e.g., "qed")
6. Include step comments in the format (*### Step X: description*) for major steps that correspond to a numbered step in the informal proof

```
>> Examples:
Here are successful examples of formal proof continuation:

[3 more Contextual FP Projection Map examples...]

>> Your Task:
Now continue the following formal proof:

>>> Problem: {informal_statement}
>>> Proof: {informal_proof}
>>> Proof So Far: {formal_proof}
>>> CONTINUATION_START >>>
```

Prompt for Memory-Augmented FIP Search Policy

User Prompt

You are a Mathematical Feature Extractor for a vector retrieval system. Your task is to convert a SUCCESSFUL pivot step (good example) into high-level, abstract Semantic Tags.

Input

You will receive:

1. Good Pivot Step (a single informal proof step from the successful path).

Output Rules (Strict)

1. Format: Output a comma-separated list of tags in brackets:

   `[Category: Value], [Category: Value], ...`

2. Abstraction:
   – IGNORE specific variable names (x, y, n, f), numbers, or function names.
   – GENERALIZE them to roles: Variable, Expression, Polynomial, Denominator.

3. Taxonomy (Use these categories):
   – [Domain: ...] (e.g., Algebra, Number Theory, Calculus, Logic)
   – [Operation: ...] (e.g., Division, Integration, Induction, Simplification)
   – [Strategy: ...] (e.g., Substitution, Case Split, Contradiction, Term-wise)
   – [Goal: ...] (e.g., Simplify Denominator, Establish Non-zero, Fix Type Casting, Reduce Complexity)

4. Constraints:
   – Do NOT output JSON.
   – Do NOT output extra text, headers, or markdown code fences.
   – Always output at least 2 tags, and include [Goal: ...] whenever possible.

```
# Few-Shot Examples

[3 more feature extraction examples...]

# Current Task
Input:
Good Pivot Step: {{good_informal_step}}

Output Tags:
```