GammaZero: Scaffold-Aware Hierarchical Search for Lean Theorem Proving

Anonymous ACL submission

Abstract

Recent LLM-based Lean provers have advanced through two directions: scaling complete-proof generation and searching over proof steps or sketches. Nevertheless, Lean feedback remains sparse: a candidate proof is accepted or rejected by the verifier. Whole-proof sampling is therefore brittle, while fine-grained search can be expensive and difficult to guide. We propose GammaZero, a scaffold-aware hierarchical search framework for Lean 4 theorem proving. GammaZero searches at an intermediate level by combining complete proof generation for the current goal with partial proof sketches that introduce intermediate claims using placeholders. This allows the prover to exploit high-level structure without expanding every tactic step. Lean is integrated into the search process as both verifier and feedback provider, enabling candidate proofs to be validated and refined. On miniF2F-test, GammaZero solves 190/244 theorems (77.9%) with a 512-node search budget. Under comparable generated-token budgets, GammaZero substantially outperforms flat sampling (66.4% versus 46.3%), suggesting that the gain is not simply due to generating longer proofs. Ablations confirm the contribution of scaffold-aware hierarchical decomposition. The search also produces structured proof trajectories with dense scoring signals; an aggregate score derived from these signals correlates strongly with proof-state closure, rising from 9–16% in low-score bins to 97% in the highest-score bin.

1 Introduction

Formal theorem proving provides a setting where language-model reasoning can be verified precisely. Although large language models can generate plausible Lean proof scripts (Polu and Sutskever, 2020; Yang et al., 2023; Wu et al., 2022), Lean accepts only proofs that pass parsing, elaboration, type checking, and final verification, yielding a sparse correctness signal: a proof either verifies or fails.

Existing approaches typically operate at two extremes (Xin et al., 2025b; Lample et al., 2022; Wang et al., 2025).Whole-proof generation samples complete proofs (Wang et al., 2025; Xin et al., 2025a) from the original theorem statement, but the feedback signal is binary: a single local error invalidates the entire proof, and the model receives no information about which subgoals were plausibly reachable. In contrast, tactic-level search exposes richer intermediate feedback (Xin et al., 2025b; Lample et al., 2022) by operating on individual proof states, but requires navigating a large, fine-grained action space through many sequential decisions, making search expensive and difficult to guide toward coherent high-level proof structure.

We introduce GammaZero, a scaffold-aware hierarchical search framework for Lean theorem proving. The framework combines direct proof generation with skeleton actions that decompose goals into intermediate claims and their compositional structure. Lean acts as both verifier and feedback provider, enabling candidate proofs to be validated and refined iteratively. A theorem is solved only when the reconstructed proof is accepted by Lean without placeholders.

The same search process also produces structured proof trajectories with intermediate feedback. Even failed generations may contain locally valid structure, allowing GammaZero to recover supervision from partial proofs rather than discarding them after verification failure.

Contributions.

- We introduce a scaffold-aware formulation of Lean proof search in which generated subgoals are solved inside the partial proof that created them rather than as standalone theorem statements.

- We propose a hierarchical search framework with two complementary action types: direct proof attempts and skeleton actions that decompose goals into intermediate claims and their compositional structure.
- We extract intermediate supervision from Lean feedback, including syntactic validity from AST-guided patching and dependency information from elaborated proof expressions.
- We evaluate GammaZero on miniF2F-test through ablation studies on scaffold-aware hierarchical search.

2 Related Work

LLM-based formal theorem proving. Large language models (LLMs) have shown strong potential in mathematical reasoning and formal proof generation, but generating an entire proof in one pass remains brittle, since even a small local error can invalidate the whole attempt. To mitigate this, systems such as GPT-f, ReProver via LeanDojo, DeepSeek-Prover-V1.5, and Kimina-Prover adopt a proposal–verification paradigm (Polu and Sutskever, 2020; Yang et al., 2023; Xin et al., 2025a; Wang et al., 2025) in which the model proposes proof steps and an interactive theorem prover such as Lean 4 checks correctness. This setup has improved proof search performance, and has been further advanced by recent strong open-weight and frontier models, including InternLM2.5-StepProver, Goedel-Prover, and Leanabell (Wu et al., 2024; Lin et al., 2025; Zhang et al., 2025), but most existing systems still treat failed generations as disposable inference outputs rather than reusable training signals.

Search as supervision. Because proof assistants provide only sparse binary feedback, search is essential in formal theorem proving. Prior work has explored beam search, best-first search (Xin et al., 2025b), Monte Carlo tree search (MCTS) (Xin et al., 2025a), and HyperTree proof search (Lample et al., 2022) to expand promising proof states. Beyond inference, these search procedures also produce intermediate trajectories, branching decisions, and verifier feedback that can serve as supervision for future policy learning. However, there is a structural tradeoff: tactic-level traces are often too fine-grained and difficult to scale, while whole-proof supervision is too sparse to distinguish globally flawed strategies from locally repairable mistakes.

Sketches, lemmas, and decomposition. To reduce the complexity of flat tactic-level search, systems such as ProD (Dong et al., 2024), Seed-Prover (Chen et al., 2025), and Draft, Sketch, and Prove (Jiang et al., 2022) introduce intermediate abstractions such as proof sketches, auxiliary lemmas, and decomposed reasoning steps. These methods separate high-level planning from low-level subgoal solving. GammaZero follows a similar decomposition-oriented direction through executable proof skeletons: unlike informal sketches, a skeleton is represented as a Lean scaffold with explicit placeholders for unresolved subgoals and compositional structure for reconstructing the parent theorem. Subgoals are then solved in their original proof context, which preserves local declarations and dependencies and reduces mismatch when subproofs are reassembled.

Repair and verifier feedback. Since formal verification is binary, partially correct generations are often rejected even when only a small region is invalid. To recover useful signal from such failures, recent systems incorporate repair mechanisms that localize and patch erroneous proof fragments. Extending placeholder-based recovery strategies such as APOLLO (Ospanov et al., 2026a), GammaZero uses an AST-guided "Sorrifier" (Qiu et al., 2026) to replace failing regions with `sorry` placeholders, making the script locally checkable despite unresolved obligations. These patched scripts are not accepted as final proofs, but they provide dense feedback on which parts of a generation are structurally valid and which parts require repair.

3 Scaffold-Aware Hierarchical Proof Search

3.1 Context-Preserving State Representation

A proof state in GammaZero is represented as

$$s = (C, i, \Gamma, G), \tag{1}$$

where C is the surrounding partial proof, i identifies the target hole, Γ is the local context at that hole, and G is the target goal. After this representation is fixed, we use the term proof state for simplicity. The important distinction is that the same visible goal may require different proof bodies in different surrounding proofs. A subproof that verifies in isolation may fail when placed back into the partial proof that generated the goal.

3.2 Action Types: Local Attempts and Skeletons

At each proof state, the model can either try to prove the current goal directly or introduce intermediate claims. We call the first case a local proof attempt: a complete proof generated for the current goal. We call the second case a skeleton: a partial proof that introduces intermediate claims using placeholders.

A skeleton introduces intermediate claims and shows how solving those claims would complete the current goal, with missing proofs temporarily filled by placeholders. As shown in Listing 1, this makes the generated object more informative than a detached subgoal statement: the model specifies both the intermediate claims and how the pieces fit together, rather than solving every piece in the same generation.

Listing 1: A skeleton introduces an intermediate claim and shows how it will be used.

```
have h_aux : P := by
  sorry
exact combine h_aux
```

Appendix B gives a complete toy example of local attempts, skeletons, in-context hole filling, and final stitching.

3.3 AND/OR Dependencies

These two candidate types induce an AND/OR search structure. A proof state is an OR node: any successful candidate can solve it. A skeleton candidate induces an AND dependency: it can close its parent proof state only if all of its child goals are solved and the completed block verifies after its placeholders are filled. A local proof attempt is a terminal candidate that either closes the current goal or fails. Figure 1 illustrates this dependency structure and the associated scaffold context for a child proof state.

4 Feedback from Lean Verification

4.1 Final Verification Signal

Final Lean verification is the hard correctness signal. A theorem is counted as solved only if the fully assembled proof is accepted by Lean and the submitted proof contains no `sorry`, `admit`, or temporary placeholders. All other signals are auxiliary: they may guide search or label trajectories, but they do not replace final verification.

4.2 AST-Guided Patching for Syntactic Validity

Lean failures are often local: a generated proof may contain valid declarations and tactics around one invalid region. GammaZero uses AST-guided patching, called Sorrifier, to replace the smallest recoverable failing region with a placeholder and then rechecks the patched script. The resulting script is not a proof. It is used to compute a syntactic validity signal: how much of the original generated code remains checkable after localized patching. Figure 2 illustrates this process, and Appendix C provides concrete patching examples.

Let L_{orig} be the clean lines of the original generation and L_{patch} the clean lines that survive unchanged after patching. The syntactic score measures the fraction of original lines that survive unchanged and in order, optionally discounting lines reported as unused or ineffective. We denote this syntactic-validity score by r_{env}.

4.3 Expression-Dependency Signal

Syntactic validity does not imply usefulness. A generated declaration may compile but never be used in the final proof. GammaZero therefore extracts an expression-dependency signal from Lean's elaborated proof expression. Generated declarations are classified by whether they are used by the final expression and whether their bodies are solved. Let u_s, $\bar{u}_s$, and $\bar{u}_{\bar{s}}$ denote the numbers of used solved, unused solved, and unused unresolved declarations, respectively.

$$r_{\text{dep}} = \frac{u_s}{u_s + \lambda_1 \bar{u}_s + \lambda_2 \bar{u}_{\bar{s}}}. \tag{2}$$

where the coefficients λ_1 and λ_2 penalize redundant or failed steps. We typically set $\lambda_2 > \lambda_1$ to reflect that an unresolved, unused declaration is more detrimental to search progress than a solved but redundant one. This signal is used as a ranking label for trajectories, encouraging candidates whose generated claims are both solved and used by the checked proof.

4.4 AND/OR Value Backup

For trajectory labels, GammaZero also backs up delayed value through the explored AND/OR graph. Let $\mathcal{A}(s)$ be the candidates expanded from proof state s, and let $\mathcal{C}(a)$ be the child proof states introduced by candidate a, with $\mathcal{C}(a) = \emptyset$ for local proof attempts. Given the candidate score

$$r(a) = r_{\text{env}}(a) + r_{\text{dep}}(a),$$

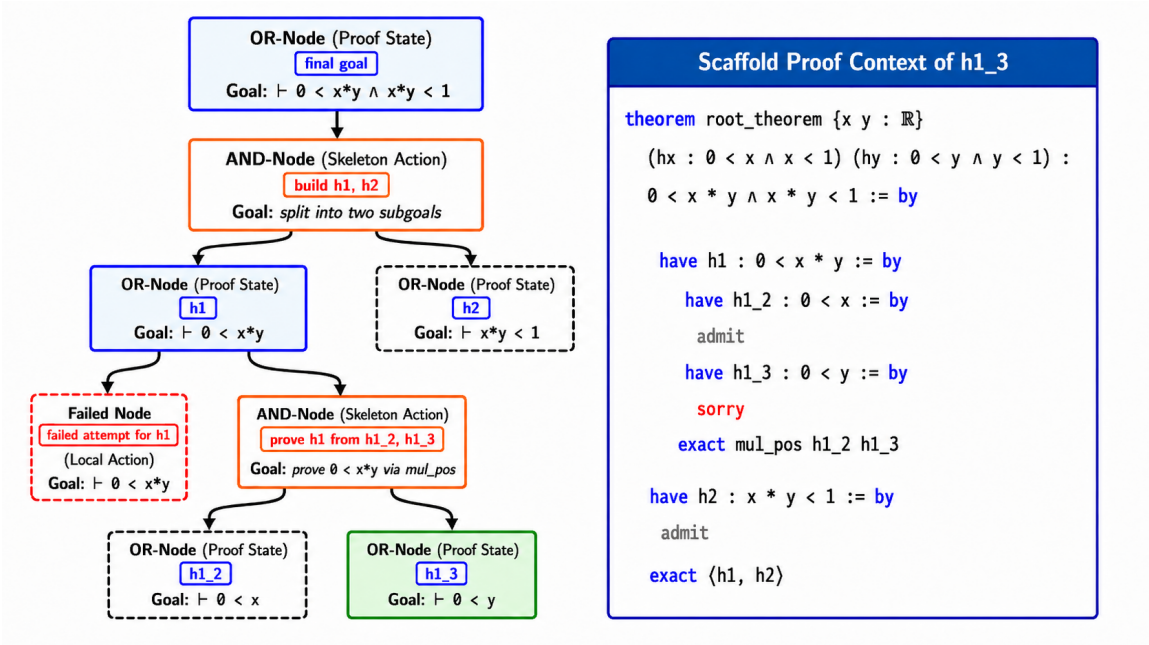

Figure 1: A skeleton candidate induces an AND dependency over child proof states, while each proof state remains an OR node over possible candidates. Child goals are solved inside the surrounding scaffold rather than as detached theorem statements.

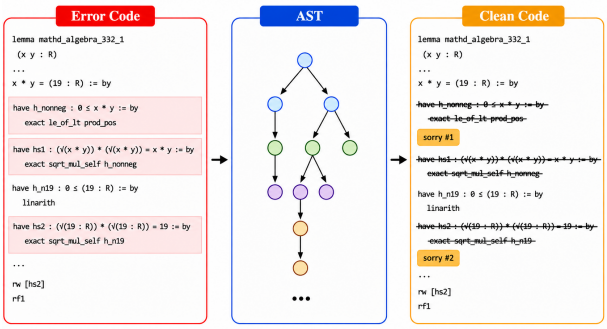

Figure 2: AST-guided patching localizes failing regions and replaces them with placeholders to recover a checkable partial script. The patched script is used only for trajectory labels and is not counted as a proof.

we define the backed-up action value as

$$Q(s,a) = r(a) + \gamma\, \mathbf{1}_{\text{closed}}(s,a) \min_{s' \in \mathcal{C}(a)} V(s'), \tag{3}$$

where $\min_{\emptyset} V = 0$. The proof-state value is then

$$V(s) = \max_{a \in \mathcal{A}(s)} Q(s,a). \tag{4}$$

The max corresponds to the OR choice among alternative candidates, while the min corresponds to the AND dependency over the children of a skeleton candidate. The indicator $\mathbf{1}_{\text{closed}}(s,a)$ ensures that delayed child value is backed up only when candidate a closes its dependency under the current search graph. This backup is used only as a trajectory-scoring label; final proof success is still determined solely by placeholder-free Lean verification.

5 GammaZero Search Procedure

5.1 Overview

GammaZero maintains a priority queue of open proof states. At each step, the model proposes local attempts and skeletons for a selected state, and Lean checks each candidate inside its surrounding partial proof. A successful local attempt closes the current goal. A valid skeleton creates child goals that are inserted into the queue; once all children close, their placeholders are filled and the completed block is rechecked by Lean. Failed candidates may still contribute trajectory labels through patching. Figure 3 summarizes the full loop.

5.2 In-Context Hole Filling

When a skeleton introduces child goals $\{G_1, \ldots, G_n\}$, GammaZero does not solve them as standalone theorem statements. Instead, each child is solved inside the surrounding partial proof C while sibling holes remain as temporary placeholders. This design preserves local declarations, coercions, and unresolved sibling context throughout verification, avoiding the type mismatches that arise when goals are

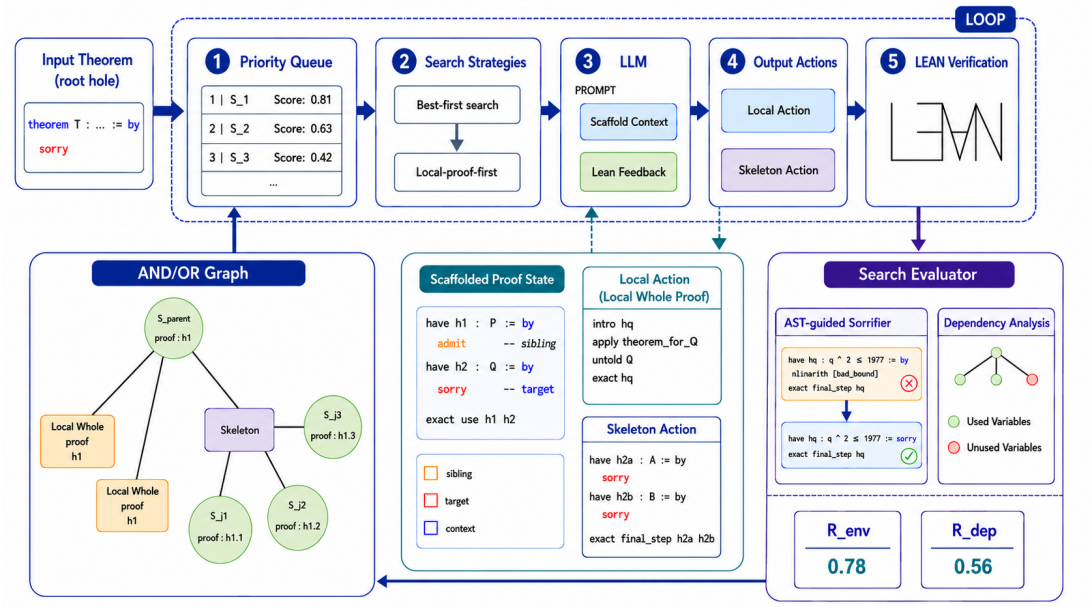

Figure 3: Overview of GammaZero. The search loop selects proof states, prompts the proposal model with scaffold context and Lean feedback, checks generated candidates with Lean, and records verification, patching, and dependency signals for the search graph.

elaborated in isolation.

5.3 Asymmetric Expansion

GammaZero uses an asymmetric expansion: local attempts discharge the current goal in a single generation without creating new nodes, while skeletons expand the graph by decomposing a goal into intermediate claims.

Search algorithm. Algorithm 1 summarizes the procedure, using $\mathcal{G}$ for the AND/OR graph, $\mathcal{P}$ for the open-state queue, Π for the stitched proof, S for a batch of selected states, V for Lean checking and patching results, and $K^\star$ for the committed skeleton; a proof state s follows the representation defined earlier. PROBELOCAL samples complete proofs and records Lean feedback. When local probing stalls, NEEDSKELETON triggers decomposition based on r_{env} scores and a per-state attempt limit. RANKRESERVE commits the best valid skeleton and reserves up to $R_{\max}$ alternatives, ranked by validity, state score, depth, and complexity. PROPAGATE updates AND/OR status bottom-up: a proof state closes when any candidate succeeds; a skeleton closes its parent only after all children close and the completed block verifies. REFRESH swaps in a reserved skeleton when the current one becomes stale, BACKUPVALUES propagates trajectory values via max over candidates and min over skeleton children, and STITCHPRUNE assembles and prunes the final proof via Lean dependency analysis.

Algorithm 1: GammaZero Search

Require: theorem T, node budget B, batch size M, skeleton reserve limit $R_{\max}$
Ensure: proof Π if found, search graph $\mathcal{G}$

```
G, P ← INIT(T)
while P ≠ ∅ and budget remains and root not closed do
    S ← POPBATCH(P, M)
    for s ∈ S do
        V ← PROBELOCAL(s)
        UPDATE(G, s, V)
        if NEEDSKELETON(s) then
            V ← PROBESKELETON(s)
            K* ← RANKRESERVE(V, R_max)
            COMMIT(G, s, K*, P)
    REFRESH(G, P)
    PROPAGATE(G)
BACKUPVALUES(G)
if root is closed then
    Π ← STITCHPRUNE(G)
    return Π, G
return FAILED, G
```

6 Experiments

6.1 Experimental Scope

The experiments evaluate whether GammaZero runs end-to-end as a proof-search and data-synthesis framework: it generates actions, verifies them with Lean, expands skeleton-created subgoals, preserves partial signals, and reconstructs placeholder-free proofs when search succeeds. We focus on two questions:

- **Proof success:** whether the final reconstructed proof is accepted by Lean without placeholders.
- **Search scaling:** how much increasing the search budget beyond direct root solving contributes to the final solved theorem rate.

In this work, we report proof success, ablations, final-verification checks, and an analysis of how the recorded scores correlate with eventual proof-state closure.

6.2 Setup

All experiments use Lean 4 `v4.29.0-rc1` with a fixed Mathlib (The mathlib Community, 2020) commit, reported in the supplementary material.

Local proof attempts and skeletons are generated by `Gemini 3 Flash Preview`, used as a fixed proposal model without fine-tuning. We use Gemini because GammaZero requires structured Lean generation, including partial proof sketches with explicit placeholders for intermediate claims; many theorem-proving models are optimized to avoid incomplete proofs containing `sorry`. We acknowledge that this introduces a closed-source reproducibility dependency and provide our codebase and prompt templates in the supplementary material.

Unless otherwise stated, search uses maximum depth $D = 15$, node budget $N = 512$, 4 Lean workers, a 120-second Lean timeout, heartbeat limit 100000, and at most 50 Sorrifier patching cycles per failed generation. Generation uses up to 8192 new tokens, temperature 1.0, and the backend's default nucleus sampling setting. Baselines under Lean 4.29 are re-evaluated in the same environment; other prior results are taken from Apollo (Ospanov et al., 2026a) and original papers.

6.3 Dataset

We evaluate on the validation and test splits of `miniF2F-v2s`, the simplified aligned version of miniF2F-v2 (Ospanov et al., 2026b), which revises the original miniF2F benchmark (Zheng et al., 2021). The problems cover competition-style algebra, number theory, and inequalities, making them suitable for both local proof generation and decomposition.

6.4 Metrics

We report problem solved rate as the main metric. A theorem is solved only when the final reconstructed proof is accepted by Lean and contains no real `sorry`, `admit`, or temporary placeholders. Patched scripts and partially solved skeleton-created branches are recorded as trajectory data but do not count as solved problems. External baselines are included as published reference points rather than strict same-budget comparisons, since they differ in models, Lean versions, and search setups.

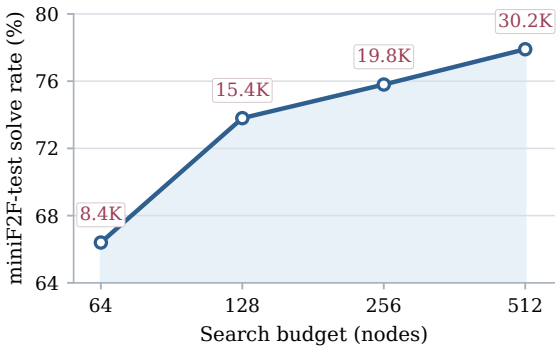

Figure 4: Search-budget scaling on miniF2F-test. Point labels show the average cumulative number of generated tokens consumed per solved theorem under early stopping.

6.5 Main Results

Table 1 reports representative miniF2F-test results across proof-generation paradigms. For GammaZero, the budget is a cumulative search-node budget rather than a count of independent whole-proof samples.

The 512-node setting solves 190/244 theorems, or 77.9%, and every reported success is rechecked as a placeholder-free Lean proof after stitching.

Scaling with search budget. Figure 4 shows how solve rate changes as the cumulative search-node budget increases. Performance rises from 66.4% at 64 nodes to 77.9% at 512 nodes, with the largest gain occurring between 64 and 128 nodes. Token labels report the cumulative average token budget per solved theorem under early stopping.

Because prior systems use different Lean versions, models, and search setups, Table 1 is mainly a reference comparison. The next ablation gives the controlled comparison, using the same Lean 4.29 environment and proposal model to isolate feedback and skeleton search.

Table 1: Representative comparison on miniF2F-test. Prior results are adapted from Apollo (Ospanov et al., 2026a) and original papers unless marked with †, which denotes our Lean 4.29 evaluations.

Method	Type	Lean Env.	Budget	miniF2F-test
HPV16+BFS+DC (Li et al., 2024)	Tree search	–	$600 \times 8 \times 400$	68.4%
BFS-Prover (Xin et al., 2025b)	Tree search	–	$2048 \times 2 \times 600$	70.8%
STP (Dong and Ma, 2025)	Whole-proof	–	25600	67.6%
Kimina-Prover-Preview-Distill (Wang et al., 2025)	Whole-proof	–	1024	70.8%
Kimina-Preview + Apollo (Wang et al., 2025; Ospanov et al., 2026a)	Hybrid	–	307	75.0%
Kimina-Prover-Preview-Distill-7B (Wang et al., 2025)†	Whole-proof	Lean 4.29	32	53.28%
Apollo + Kimina-Prover-Preview-Distill-7B (Ospanov et al., 2026a; Wang et al., 2025)†	Hybrid	Lean 4.29	32 depth 2	53.69%
GammaZero + Gemini 3 Flash Preview	**Ours**	Lean 4.29	max nodes = 512	**77.9%**

Method	Budget	Output Tokens	Solve Rate
Flat pass@32 (w/o feedback, w/o decomposition)	pass@32	7,696,998	46.3%
Root-only feedback search (w/o decomposition)	max nodes = 32	4,714,873	59.8%
GammaZero	max nodes = 64	7,944,163	66.4%
	max nodes = 512	17,174,849	**77.9%**

Table 2: Ablation on feedback, skeleton search, and search budget. Total Tokens are cumulative generated tokens over all 244 test problems, including unsolved ones.

6.5.1 Ablation on Feedback and Skeleton Search

This ablation removes GammaZero components in stages while keeping the same Gemini 3 Flash Preview proposal model. Removing both feedback and decomposition reduces the system to flat Gemini sampling from the original theorem prompt. Removing only decomposition keeps the GammaZero feedback loop but restricts search to the root proof state. The full system then adds skeleton-based decomposition.

We compare settings on the same test split. The setting without feedback and decomposition directly samples 32 complete proofs from the original theorem prompt, consuming 7.7M tokens across 244 problems. The setting without decomposition still samples only at the initial theorem state, but uses Lean checks, patching, and repeated verified attempts under the same protocol. This feedback loop improves over flat generation from 46.3% to 59.8% while consuming fewer total tokens (4.7M vs. 7.7M), as early stopping terminates search once a proof is found.

The full GammaZero setting adds skeleton actions that introduce intermediate child goals, solve them in context, and stitch the completed subproofs back into the parent proof. At 64 nodes, GammaZero consumes 7.9M generated tokens across 244 problems, close to the pass@32 token budget of 7.7M, yet achieves 66.4% vs. 46.3%, a gain of +20.1 points. Under comparable generated-token budgets, GammaZero substantially outperforms flat sampling, suggesting that the gain is not simply due to generating longer proofs. Scaling to 512 nodes increases the generated-token budget to 17.2M and yields a further gain to 77.9%, showing that GammaZero benefits from both framework design and search scaling.

Both search settings use the same final correctness criterion: a theorem counts as solved only if Lean accepts the reconstructed proof without `sorry` or `admit`. The gain from root-only feedback search to the 512-node setting is 18.1 points, showing that structured decomposition contributes solved roots beyond feedback-guided retrying at the initial proof state.

Note that the reported token counts reflect output tokens to measure the model's proof-generation effort. Due to the iterative in-context hole filling, the search procedure intrinsically consumes a larger volume of input tokens (e.g., 206M for the 512-node budget).

6.5.2 Ablation on Scaffold State Representation

We use `aime_1983_p1` to illustrate a false positive caused by isolated subgoal verification. In this example, a child goal created inside a skeleton is elaborated in the parent proof as a real-valued statement:

$$(w : \mathbb{R}) \neq 1.$$

An ablated verifier extracts the printed goal and reconstructs it as a standalone theorem. This loses part of the surrounding elaboration context, so the standalone theorem can instead be elaborated as the natural-number statement

$$(w : \mathbb{N}) \neq 1.$$

The isolated proof then compiles because tactics such as `rw [Nat.cast_one]` and `norm_cast` are valid for the natural-number equality $w = 1$. After reinsertion into the original skeleton, however, Lean checks the same proof against the real-valued equality $(w : \mathbb{R}) = 1$, where those tactics no longer apply. The final stitched proof fails.

This shows why child goals should not be verified as detached standalone theorems. Scaffold-aware verification checks each child proof directly inside the parent skeleton, preserving coercions, expected types, local assumptions, and unresolved sibling placeholders.

6.6 Patching and Scoring Signals

Invalid generations are common, but many contain locally checkable structure. GammaZero therefore records both r_{env}, which measures how much Lean text survives patching, and the sparser r_{dep}, which gives credit only to solved declarations used by the checked proof. These are trajectory labels rather than separate proof-success metrics.

To test whether these labels are useful as search signals, we compute an aggregate score for the first generated local attempt of each OR proof state:

$$r_{\text{mean}} = \text{mean}(r_{\text{env}}, r_{\text{dep}}).$$

We then bin local attempts by r_{mean} and measure the eventual closure rate of their parent OR state under the full search procedure. Figure 5 shows that this aggregate score is strongly correlated with downstream closure: low-score bins ($r_{\text{mean}} < 0.2$) close only ∼9–16% of parent states, while the highest bin ($r_{\text{mean}} > 0.9$) reaches ∼97%. This suggests that the recorded scores are informative for ranking partial generations, even though final correctness is still determined only by placeholder-free Lean verification.

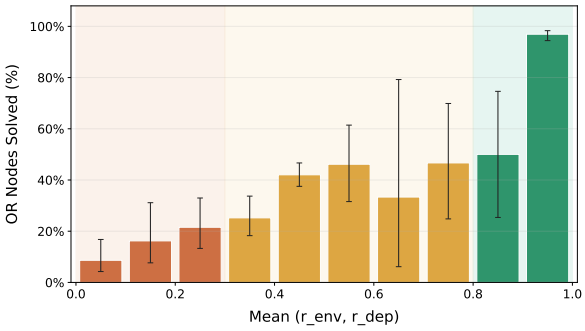

Figure 5: Predictiveness of aggregate scores. Local attempts are binned by r_{mean}, and the y-axis reports the eventual closure rate of their parent OR proof states under the full search procedure.

7 Conclusion

We presented GammaZero, a scaffold-aware hierarchical search framework for Lean theorem proving. GammaZero searches between whole-proof generation and tactic-level search: local proof attempts generate complete proofs for the current goal, while skeletons introduce intermediate claims using placeholders and allow difficult goals to be decomposed in context. By checking candidates inside their surrounding partial proofs, GammaZero avoids treating generated subgoals as detached theorem statements. Lean feedback is used throughout the search loop for verification, syntactic validity signals from patching, and dependency analysis. Experiments on miniF2F-test show that scaffold-aware hierarchical search improves proof success, and the recorded scoring signals are predictive of downstream state closure.

Limitations

GammaZero records dense scoring signals and structured proof trajectories, but it does not train a new policy or value model from these data. The scoring analysis shows that the aggregate score is predictive of downstream closure, but this should not be interpreted as evidence that the score will directly improve a learned prover without further training experiments. The evaluation focuses mainly on miniF2F-test, and comparisons to prior systems are affected by differences in Lean versions, Mathlib revisions, proposal models, budgets, and search implementations. The expression-

dependency signal is also proof-specific: it measures usage in the proof found by the search, not mathematical necessity across all possible proofs. Finally, the current implementation relies on a fixed proposal model and a finite search budget, so unsolved theorems are only unsolved under this particular model, environment, and search configuration.

References

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, and 1 others. 2025. Seed-prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*.

Kefan Dong and Tengyu Ma. 2025. Stp: self-play llm theorem provers with iterative conjecturing and proving. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org.

Kefan Dong, Arvind Mahankali, and Tengyu Ma. 2024. Formal theorem proving by rewarding llms to decompose proofs hierarchically. *arXiv preprint arXiv:2411.01829*.

Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. 2022. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*.

Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. 2022. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349.

Yang Li, Dong Du, Linfeng Song, Chen Li, Weikang Wang, Tao Yang, and Haitao Mi. 2024. Hunyuan-prover: A scalable data synthesis framework and guided tree search for automated theorem proving. *arXiv preprint arXiv:2412.20735*.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and 1 others. 2025. Goedel-prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*.

Azim Ospanov, Farzan Farnia, and Roozbeh Mohit. 2026a. Apollo: Automated llm and lean collaboration for advanced formal reasoning. *Advances in Neural Information Processing Systems*, 38:41599–41633.

Azim Ospanov, Farzan Farnia, and Roozbeh Mohit. 2026b. minif2f-lean revisited: Reviewing limitations and charting a path forward. *Advances in Neural Information Processing Systems*, 38:79964–80002.

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

Ruichen Qiu, Yichuan Cao, Junqi Liu, Dakai Guo, Xiao-Shan Gao, Lihong Zhi, and Ruyong Feng. 2026. Mechanic: Sorrifier-driven formal decomposition workflow for automated theorem proving. *arXiv preprint arXiv:2603.24465*.

The mathlib Community. 2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA. Association for Computing Machinery.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, and 1 others. 2025. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. *Advances in neural information processing systems*, 35:32353–32368.

Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Zheng Yuan, Wenwei Zhang, Dahua Lin, and Kai Chen. 2024. Internlm2. 5-stepprover: Advancing automated theorem proving via critic-guided search. *arXiv preprint arXiv:2410.15700*.

Huajian Xin, ZZ Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, and 1 others. 2025a. Deepseek-prover-v1. 5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. In *International Conference on Learning Representations*, volume 2025, pages 72274–72303.

Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Ming Ding. 2025b. 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)*, pages 32588–32599.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. Leandojo: theorem proving with retrieval-augmented language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Jingyuan Zhang, Qi Wang, Xingguang Ji, Yahui Liu, Yang Yue, Fuzheng Zhang, Di Zhang, Guorui Zhou, and Kun Gai. 2025. Leanabell-prover: Posttraining scaling in formal reasoning. *arXiv preprint arXiv:2504.06122*.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2021. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*.

A High-Performance Lean AST and Expression Servers

GammaZero performs many Lean parsing, elaboration, and checking queries during search. Repeatedly starting a fresh Lean process would make these queries dominated by startup and Mathlib-loading overhead. To reduce this cost, we use persistent Lean server processes for AST extraction and elaborated-expression analysis. This appendix describes the daemon architecture, the custom Lean executables `dump_ast_server` and `dump_expr_server`, and the Python IPC layer used to communicate with them.

A.1 Lean Startup and Import-Loading Overhead

In Lean 4, checking a file that imports Mathlib requires initializing Lean and loading the imported `olean` files before local declarations can be elaborated. For a fresh process, the latency can be decomposed as

$$T_{\text{cold}} = T_{\text{startup}} + T_{\text{load_imports}} + T_{\text{elab_local}}. \quad (5)$$

In our environment, loading the fixed Mathlib environment takes roughly 5–12 seconds. Since GammaZero may evaluate hundreds of local proof attempts and skeletons for a single theorem, spawning a new Lean process for each query would make search prohibitively slow.

A.2 Persistent Lean Daemons

To avoid repeated cold starts, GammaZero uses persistent Lean server processes. During initialization, the Python driver starts two custom Lean executables and communicates with them through standard input/output:

- **`dump_ast_server`** parses Lean syntax trees and returns byte-level source ranges.
- **`dump_expr_server`** elaborates declarations and extracts Lean expression trees for dependency analysis.

Each daemon performs a warmup pass by loading the fixed Mathlib environment once. Subsequent requests reuse this cached environment and only elaborate the local query code when the import header has not changed. This changes the dominant cost from cold startup and import loading to local parsing or elaboration.

A.3 Optimization Techniques

After warmup, local AST and expression queries are typically handled in tens of milliseconds. The speedup comes from three implementation choices.

Import-header caching. The daemon first elaborates a synthetic Lean file containing the standard import header used by the benchmark. For later queries, it compares the incoming import header against the cached header. If the header is unchanged, the daemon reuses the already-loaded Mathlib environment and skips import elaboration.

Listing 2: Import-header caching inside the Lean server.

```
-- Reload the environment only when the import
    header changes.
if headerStr != lastHeader && headerStr != ""
    then
  let (env, _) <- Lean.Elab.processHeader header
      {} messages inputCtx
  currentEnv := env
  lastHeader := headerStr
else
  -- Reuse the cached Mathlib environment.
  pure ()
```

This restricts most requests to local parsing or elaboration:

$$T_{\text{warm}} \approx T_{\text{elab_local}} \ll T_{\text{cold}}. \quad (6)$$

Environment isolation. In a persistent Lean session, elaborating independent queries can add declarations to the environment. If reused directly, the modified environment may cause namespace collisions or cross-branch contamination. To keep search branches isolated, `dump_expr_server` caches a base Mathlib environment and restores it after each query, returning only the extracted declaration and expression information to Python.

Listing 3: Restoring the base Mathlib environment after local elaboration.

```
-- Elaborate the local query and inspect its
    declarations.
let finalCmdState <- elabLoop inputCtx pmctx
    parserState cmdState
let envWithFileDecls := finalCmdState.env

-- Extract local declarations and expression
    trees.
envWithFileDecls.constants.map₂.forM fun name
    cinfo => do
  dumpExprTree name cinfo.type (cinfo.value?)
```

```
-- Discard local declarations and return to the
    base environment.
return (currentHeader, baseEnv)
```

Scheduled recycling. Long-running Lean processes gradually accumulate temporary metadata and heap allocations over many requests. To limit memory growth, the Python driver tracks the number of requests served by each daemon and periodically restarts it:

$$R_{\text{limit}} = \begin{cases} 2000, & \text{for AST extraction,} \\ 500, & \text{for expression-tree extraction.} \end{cases} \tag{7}$$

When this limit is reached, the driver terminates the subprocess, starts a fresh daemon, and reruns the Mathlib warmup step.

A.4 Python IPC Driver Layer

The Python driver wraps the Lean daemons with thread-safe request methods. It also normalizes incoming Lean snippets before they are sent to the daemon. In particular, if a code snippet does not explicitly contain an import, the driver prepends `import Mathlib` and then corrects the returned byte offsets so that they still refer to the original snippet.

Listing 4: Dynamic import wrapping and byte-offset correction.

```python
def get_lean_ast(code: str) -> list:
    prefix = "import Mathlib\n"
    has_import = "import " in code

    if not has_import:
        full_code = prefix + code
        offset_bytes = len(prefix.encode("utf-8"))
    else:
        full_code = code
        offset_bytes = 0

    with tempfile.NamedTemporaryFile(
        mode="w", suffix=".lean", dir=REPL_DIR, delete=False
    ) as f:
        f.write(full_code)
        path = f.name

    try:
        blocks = _SHARED_DAEMON.get_ast(path)
        if offset_bytes > 0:
            for b in blocks:
                b["start_byte"] -= offset_bytes
                b["end_byte"] -= offset_bytes
            return [b for b in blocks if b["start_byte"] >= 0]
        return blocks
    finally:
        os.remove(path)
```

This offset correction keeps AST byte ranges aligned with the code seen by the search procedure, which is important for localized patching and prompt construction.

B Running Example

Consider the following simple theorem:

```
theorem split_imp
(P Q R : Prop)
(hpq : P -> Q)
(hpr : P -> R) :
P -> Q /\ R := by
```

The initial goal is to prove $P \to Q \wedge R$. A local proof attempt may close the current goal directly:

```
intro hp
exact And.intro (hpq hp) (hpr hp)
```

If Lean accepts this proof body and no goal remains, the current proof state is closed. This candidate does not create child proof states.

A skeleton may instead introduce intermediate claims:

```
intro hp
have hq : Q := by
  sorry
have hr : R := by
  sorry
exact And.intro hq hr
```

This skeleton does more than list the subgoals Q and R. It also specifies how the solved claims will be used to complete the parent goal, namely through the final line:

```
exact And.intro hq hr
```

The placeholders create two child proof states. When the search focuses on the first child, the surrounding scaffold is preserved, while only the proof of hq is treated as the target:

```
intro hp
have hq : Q := by
  sorry
have hr : R := by
  admit
exact And.intro hq hr
```

Here `admit` is a temporary placeholder for the sibling child. It is not part of the final proof. In this scaffold, the target `sorry` can be replaced by:

```
exact hpq hp
```

For the second child, the roles are reversed:

```
intro hp
have hq : Q := by
  admit
have hr : R := by
  sorry
exact And.intro hq hr
```

and the target proof body is:

```
exact hpr hp
```

After both children are solved, their proof bodies are inserted back into the original skeleton:

```
intro hp
have hq : Q := by
  exact hpq hp
have hr : R := by
  exact hpr hp
exact And.intro hq hr
```

The completed block is then checked again by Lean. The parent proof state is closed only if this stitched proof contains no remaining `sorry` or `admit` and verifies in the original context.

This example illustrates the boundary between the two candidate types. A local proof attempt tries to close the current target directly. A skeleton introduces proof structure and creates child proof states. Each child is solved inside the scaffold where it was created, and the parent state is closed only after all required children are solved and the completed proof verifies.

C Sorrifier Examples

This section illustrates how the Sorrifier patches different Lean failures into placeholder-compilable scripts and how the line-based syntactic score r_{env} is computed. In these examples, blank lines and comments are excluded from the line count, and we omit optional penalties for unused or ineffective lines.

Case 1: Parser error. Consider a generated proof body containing a malformed tactic:

```
have h := 1
apply h (
```

Lean reports a parser error because the opening parenthesis is not closed. The Sorrifier localizes the failing tactic line and replaces it with a placeholder:

```
have h := 1
sorry
```

The original body contains two clean lines. One line survives unchanged, while the malformed tactic is replaced. Therefore,

$$r_{\text{env}} = \frac{1}{2} = 0.5.$$

Case 2: Unresolved name and cascade cleanup. Consider a generated proof body that calls a nonexistent lemma:

```
apply nonexistent_lemma
rfl
```

Lean reports an unknown identifier error for `nonexistent_lemma`. Since the following line depends on the proof state that would have been produced by the failed command, the recovered region is replaced by a single placeholder:

```
sorry
```

The original body has two clean lines, but neither survives unchanged in the patched body. Thus,

$$r_{\text{env}} = \frac{0}{2} = 0.0.$$

This example shows that recovery is not purely local at the textual level: an early failure may invalidate later commands that depend on the missing proof state.

Case 3: Partial recovery inside a skeleton. The following skeleton contains a valid setup step before an invalid tactic:

```
have h : 1 + 1 = 2 := by
  rfl
apply nonexistent_lemma
exact h
```

The first two lines define a valid intermediate claim. The third line fails because the lemma name cannot be resolved, and the final line becomes unusable after the failed application. The patched body is:

```
have h : 1 + 1 = 2 := by
  rfl
sorry
```

The original body has four clean lines. The first two lines survive unchanged, while the remaining two lines are replaced by a placeholder. Therefore,

$$r_{\text{env}} = \frac{2}{4} = 0.5.$$

This case illustrates why patching is useful for scoring skeletons: even when the generated proof does not close the goal, valid setup lines and intermediate claims can still be preserved and rewarded.

D Case Study: Scaffolded Recovery from a Failed Whole-Proof Attempt

Figure 6 illustrates GammaZero on `mathd_algebra_323`. The theorem involves the function $f(x) = x^3 - 8$ and requires reasoning about the inverse of f at the value 19. The key difficulty is to justify that f is injective, so that `Function.invFun` can be unfolded through a left-inverse argument.

The contrast is not the mathematical idea, but the proof representation. A representative whole-proof attempt introduces the useful intermediate claim that f is injective, but then fails at a local step by calling a nonexistent lemma:

```
have h_inj : Function.Injective f := by
  rw [h0]
  intro x y h
  exact power_left_injective ... h
```

A complete-proof sample is rejected because this local error invalidates the whole candidate. GammaZero instead preserves the useful plan as a skeleton:

```
have h_inj : Function.Injective f := by
  sorry
have h_f3 : f 3 = 19 := by
  sorry
have h_inv_f_f : ∀ x, Function.invFun f (f x) =
    x :=
  Function.leftInverse_invFun h_inj
```

The difficult child `h_inj` is then solved through monotonicity rather than a hallucinated injectivity lemma:

```
have h_mono : StrictMono (fun x : ℝ => x ^ 3) :=
    by
  exact Odd.strictMono_pow (by decide)
rw [h0]
exact h_mono.add_const (-8) |>.injective
```

This example shows why scaffolded decomposition can recover from whole-proof brittleness. The failed whole-proof attempt contains a useful high-level claim, but one invalid line prevents verification. The skeleton keeps that claim as part of a parent proof plan, exposes it as a child goal, and allows the child to be solved in the original context. After the required children are solved, the completed scaffold is checked again by Lean as a placeholder-free proof.

On this theorem, direct Gemini whole-proof sampling verifies $0/32$ complete samples under our setup, while GammaZero produces a 16-line verified proof body at search depth 2.

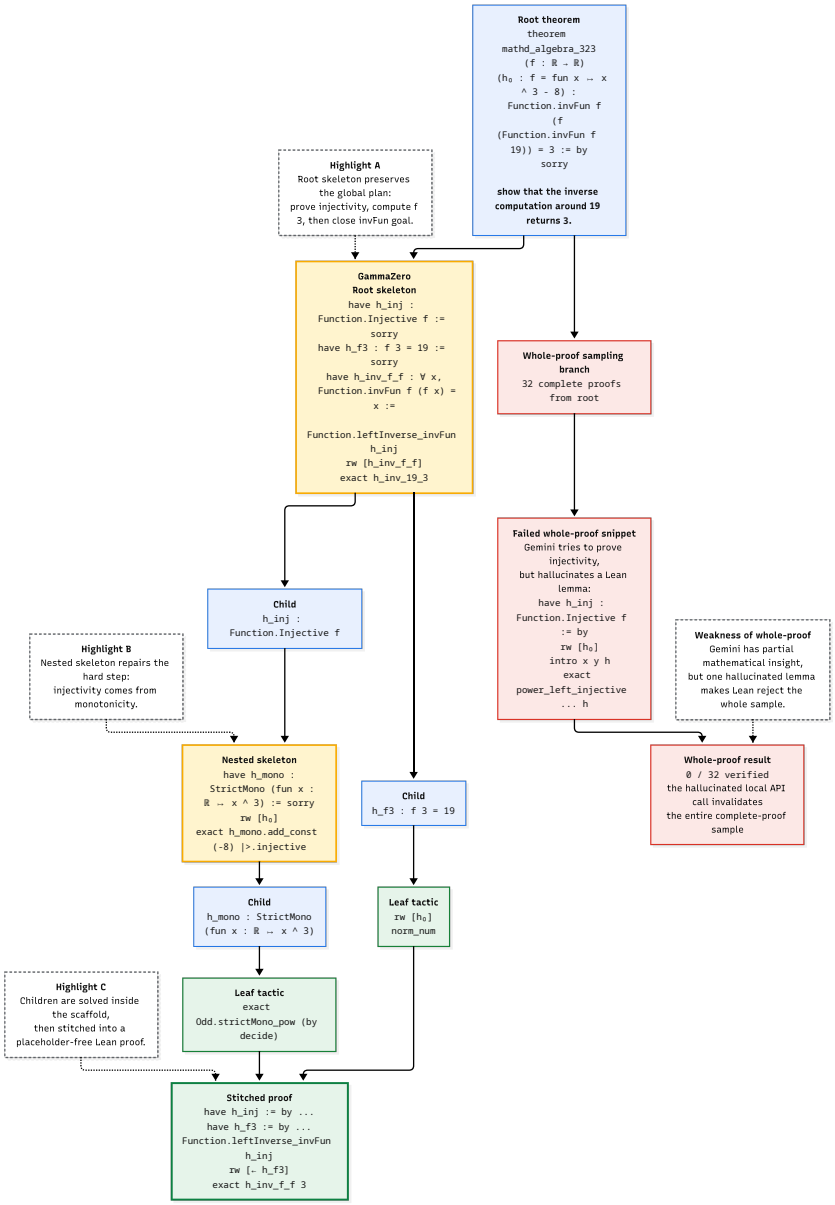

Figure 6: Case study for `mathd_algebra_323`. A whole-proof attempt identifies the need for injectivity but fails because of a hallucinated Lean lemma. GammaZero preserves the same high-level plan as a skeleton, decomposes injectivity into a monotonicity subgoal, solves the child obligations, and stitches them into a verified proof.