REAP: UNCOVERING MONTE CARLO PROOF SEARCH WITH A LIGHTWEIGHT MODEL*

Anonymous authors
Paper under double-blind review

ABSTRACT

We present **Reap**, an open framework for learning and analyzing formal proof search in Lean 4. Reap implements policy–value-guided Monte Carlo Tree Search directly in Lean, sharing proof-state transitions, AND–OR search, and replay verification between inference and training rollouts. Structured action logs expose how neural predictions and tactic execution interact during search. Using this framework, we train **Reaper-1**, a 1.7B-parameter policy–value model, through continued pre-training, supervised tactic learning, and iterative search-based reinforcement learning. Using a separate premise selector, the final system solves 79.9% of miniF2F-test at pass@32, establishing state-of-the-art performance among open provers of comparable size while remaining competitive with substantially larger models. Beyond improved solve rates, our results suggest that RL strengthens the prover's ability to organize proofs around useful intermediate results, favoring substantial advances over repeated local adjustments. This more directed proof construction is accompanied by increasingly elaborate tactic programs: fewer explicit search decisions are needed, while more work is performed within each decision. The resulting trade-off between search effort and tactic execution cost provides insight into how learning changes the prover's problem-solving strategy.

1 INTRODUCTION

Formal theorem proving provides a reliable testbed for studying machine reasoning: proof assistants expose explicit proof states and verify completed proofs with a trusted kernel, grounding reasoning in a precisely defined environment. Recent whole-proof systems, such as Kimina-Prover, DeepSeek-Prover-V2, and Seed-Prover (Wang et al., 2025; Ren et al., 2025; Chen et al., 2025), have achieved strong mathematical performance with increasingly large models and elaborate inference pipelines, while step-level reinforcement learning has substantially improved neural theorem provers (Lample et al., 2022; Hubert et al., 2026). These advances raise a complementary question: beyond solving more theorems, how does training change the mechanism and procedure of proof search? Earlier systems (Lample et al., 2022; Polu & Sutskever, 2020) combined relatively small models with inspectable search environments, whereas more recent high-performing provers (Hubert et al., 2026; Xin et al., 2026; Wu et al., 2025) can be difficult to study independently because of limited implementation access or substantial computational requirements. This motivates our central goal: combining strong theorem-proving performance with an accessible and observable search framework.

To this end, we introduce **Reap**, an open framework that implements policy–value-guided Monte Carlo Tree Search (MCTS) directly as a Lean 4 tactic. Reap runs the semantics-critical search loop inside Lean, supporting native proof-state transitions, AND–OR search, runtime budget control, proof reconstruction, and replay verification, while external neural services provide policy and value predictions. The same implementation serves both as an inference-time proof-search tactic and as a rollout engine for reinforcement learning, providing a consistent interface between training and evaluation. Reap also records structured search traces, making the interaction between neural

*An anonymized source snapshot of Reap is included in the supplementary material. We will open-source the model weights from each training stages - continued pre-training, supervised fine-tuning, and reinforcement learning, as well as the complete reinforcement-learning pipelines.

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

predictions, tactic execution, and search directly observable. Together with the distributed **Scythe-rl** pipeline for coordinating rollouts, training, and data management, this design provides a lightweight environment for developing and analyzing search-based formal provers.

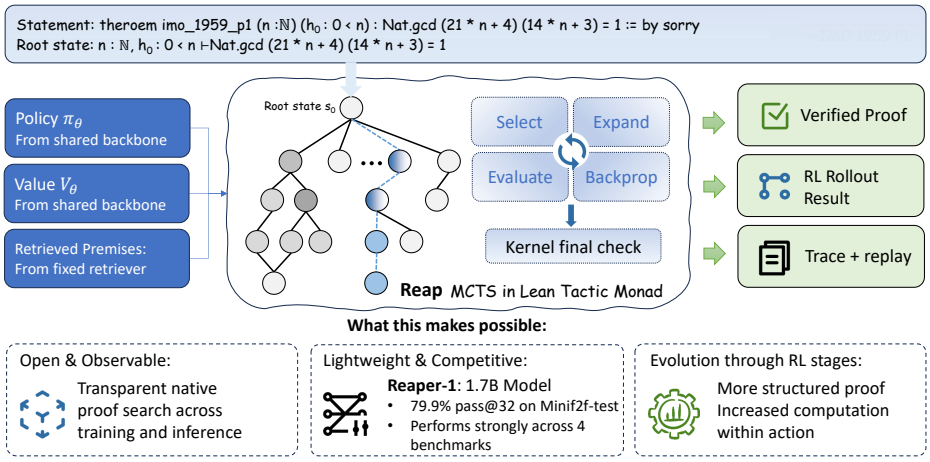

Figure 1: Overview of Reap's proof-search and learning interface. A shared policy–value backbone and a fixed premise retriever provide guidance to Monte Carlo Tree Search running inside Lean's tactic monad, where native proof-state transitions are selected, expanded, evaluated, and backed up. Successful proofs are checked by Lean's kernel and exposed through traces and replay; the same interface supports reinforcement-learning rollouts and inference, enabling transparent analysis of how training changes proof structure and computation within each action.

Using this framework, we train **Reaper-1**, a lightweight 1.7B-parameter policy–value model through continued pre-training, supervised tactic learning, and iterative MCTS-guided reinforcement learning. With a separate premise retriever following the LeanSearch-PS recipe (Shen et al., 2025), Reaper-1 achieves **79.9%** pass@32 on miniF2F-test, establishing state-of-the-art performance among open provers of comparable size. This system altogether lowers the hardware barrier and compute budget to inference and experimentation.

Beyond benchmark performance, Reap allows us to study how reinforcement learning reshapes proof search. Our analysis reveals that reinforcement learning changes both the organization of proofs and the granularity of interaction with Lean. Later checkpoints produce shorter top-level proof trajectories while increasingly using structural tactics to introduce intermediate claims and organize local reasoning. Together with our qualitative case studies, this pattern suggests that the model becomes better at identifying useful intermediate structural lemmas and concentrating proof attempts on steps that make substantive progress, reducing reliance on repeated local manipulations. At the same time, individual actions become more compound, incorporating multiple proof operations and solver calls. Shorter trajectories therefore reflect a combination of more focused proof construction and greater computation within each action. These findings highlight the need to examine proof structure and execution cost together when interpreting the benefits of reinforcement learning for proof search.

Our main contributions are as follows:

- We introduce **Reap**, an open and observable MCTS proof-search framework implemented directly as a Lean 4 tactic, supporting native proof-state transitions, AND–OR search, proof reconstruction, and replay verification.
- We develop the **Scythe-rl** training pipeline and use it to train **Reaper-1**, a lightweight 1.7B-parameter prover backbone. Integrated with Reap and a separate premise retriever, it achieves 79.9% pass@32 on miniF2F-test and strongly transfer across 4 additional benchmarks, reaching state-of-the-art performance among models of comparable size.

- Enabled by the high observability of our framework, we provide statistical analyses and detailed case studies to characterize how proof-search behavior evolves during MCTS-guided learning, revealing a transition toward search-interaction compression.

2 Related Work

LLM-based Formal Theorem Proving. Large language models have substantially advanced formal theorem proving in Lean. Recent systems generally follow either whole-proof generation or step-level interaction with the proof assistant. Representative whole-proof provers include DeepSeek-Prover-V2 (Ren et al., 2025), Kimina-Prover (Wang et al., 2025), Goedel-Prover-V2 (Lin et al., 2026), and Pythagoras-Prover (Leang et al., 2026). Step-level provers instead predict tactics conditioned on the current Lean proof state, as in HunyuanProver (Li et al., 2024), InternLM2.5-StepProver (Wu et al., 2025), REAL-Prover (Shen et al., 2025), BFS-Prover-V2 (Xin et al., 2026), and OptProver (Li et al., 2026). Retrieval-augmented systems such as LeanDojo/ReProver (Yang et al., 2023) and REAL-Prover retrieve relevant library declarations to guide proof generation. Reap follows the step-level paradigm and performs proof search over Lean states.

Search for Theorem Proving. Search is a key component of step-level theorem proving. Best-first search is widely used in recent neural provers (Xin et al., 2026). Several works have also explored MCTS-style search. HyperTree Proof Search (Lample et al., 2022) introduces an AlphaZero-inspired search procedure over the AND–OR structure of proof obligations. DeepSeek-Prover-V1.5 (Xin et al., 2025) proposes RMaxTS, an MCTS variant that encourages exploration over intermediate tactic states. AlphaProof (Hubert et al., 2026) further combines policy–value-guided tree search with reinforcement learning at large scale. Reap builds on this line of work, but implements the MCTS rollout process directly inside Lean so that search operates on native proof states and can be invoked as a Lean tactic.

Reinforcement Learning for Formal Proving. Reinforcement learning has increasingly been used to improve formal provers through feedback from proof assistants. HTPS (Lample et al., 2022) combines proof search with online learning from discovered proofs, while DeepSeek-Prover-V1.5 (Xin et al., 2025) introduces reinforcement learning from proof-assistant feedback. AlphaProof (Hubert et al., 2026) couples policy–value-guided tree search with iterative reinforcement learning, which provides the primary methodological inspiration for Reap. Recent systems further explore interleaved thinking and proving, self-play, verifier-integrated reinforcement learning, and large-scale test-time search (Lin et al., 2025; Dong & Ma, 2025; Ji et al., 2025; Chen et al., 2025).

3 Methodology

3.1 Reap: Lean-Native Monte Carlo Tree Search

Reap implements a policy–value-guided Monte Carlo Tree Search inspired by AlphaProof (Hubert et al., 2026) and adapted to tactic-level theorem proving in Lean. Its key design is to run the search loop directly inside Lean's tactic monad. Proof-state transitions, tree construction, and verification remain inside Lean, allowing the same implementation to serve both as a training rollout engine and as an invocable Lean tactic. Neural policy and value evaluations are served externally: the policy proposes tactics, while the value estimates negative remaining proof-search cost and guides MCTS toward shorter successful continuations. Furthermore, Reap optionally records structured action-level logs covering model calls, tactic evaluation, and timing, enabling the analyses in Section 4.6. The Reap source snapshot included in the supplementary material provides implementation details of the search procedure.

Native proof states and tactic evaluation. The proof search is implemented natively in Lean through its metaprogramming API. State transitions manipulate Lean's proof-state objects directly and use the built-in save/restore primitives. Each node stores a `Tactic.SavedState`, and each tactic edge represents a successful transition induced by a Lean tactic. Before elaboration, Reap filters tactic syntax that is unsuitable for a proof-state transition such as `sorry` and `simp?`. Tactic execution is bounded by both Lean heartbeats and a wall-clock timeout. Only tactics that elaborate

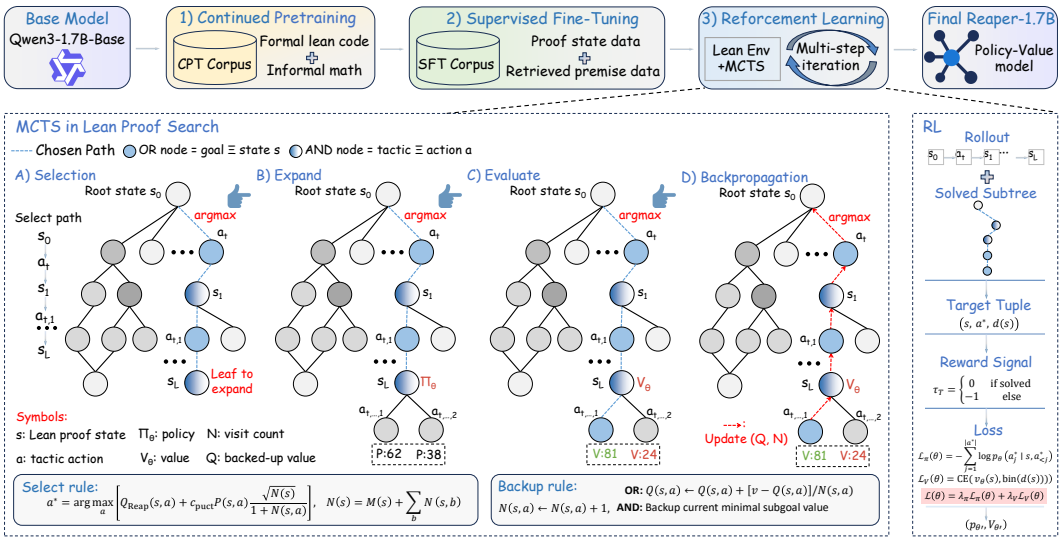

Figure 2: Overview of Reap's training pipeline and Lean-native MCTS. In the backup rule, OR/AND labels refer to the successor node's type. Both $Q(s, a)$ and the return v account for the edge cost: one for tactic edges and zero for synthetic focus edges. Here, $M(s)$ counts direct policy–value evaluations.

and execute successfully are incorporated into the search tree, together with their resulting proof states.

AND–OR proof search. Reap represents tactic choices with OR nodes and independent proof obligations with AND nodes. An OR node is considered solved once any child is solved, whereas an AND node requires that all its children be solved. Multiple goals are decomposed into an AND node only when they contain no unresolved expression metavariables; otherwise, they remain coupled in a single OR-state.

Node values estimate negative remaining proof-search cost. Each tactic edge incurs unit cost, while the synthetic focus edges connecting an AND node to its subgoals incur zero cost. OR nodes maintain visit-averaged value estimates. At an AND node, both the stored value and the value propagated to its parent are the minimum of the current values of its evaluated, unresolved children, with zero as the default for solved or unevaluated children. Thus, a fully solved AND node has value zero, and the estimated remaining cost of an AND node is determined by its hardest unresolved subgoal rather than the sum of its subgoal costs.

Progressive sampling and PUCT. Reap uses progressive sampling (Hubert et al., 2026) to balance generating new tactics and revisiting existing children. Let $N(s)$ be the visit count and $M(s)$ the number of policy–value evaluations at state s. New tactics are sampled according to

$$M(s) \leq C\, N(s)^{\alpha}. \tag{1}$$

Meanwhile, Reap selects a child using the PUCT score (Silver et al., 2018; Hubert et al., 2026)

$$\mathrm{PUCT}(s,a) = \gamma^{-V(s,a)-1} + c(N(s))P(s,a)\frac{\sqrt{N(s)}}{1+N(s,a)}, \tag{2}$$

where the exploration coefficient is

$$c(N) = c_{\mathrm{init}} + \log\frac{N + c_{\mathrm{base}} + 1}{c_{\mathrm{base}}}, \tag{3}$$

Here, $P(s, a)$ is the normalized policy prior, and $V(s, a) = \bar{v}(s') - d(a)$ is the child node's current value estimate after subtracting the edge cost, where s' is the child state, $d(a) = 1$ for tactic edges, and $d(a) = 0$ for synthetic focus edges. For visited children of OR nodes, Eq. (2) favors lower estimated remaining costs. The exploitation term is initialized to zero for unvisited children. AND nodes instead use $1 - \gamma^{-V(s,a)-1}$ for visited, unresolved children, retaining the same exploration term and excluding solved children, to prioritize harder subgoals; their unvisited children have exploitation term one.

State deduplication. When expanding an OR node, Reap executes each generated tactic and computes a serialized pretty-printed proof-state. If this key matches that of a state already on the current MCTS path, the candidate is discarded. If it matches an existing child of the current node, the candidate's prior weight is added to the existing edge and no new node is created. Every generated sample, including repeated tactic text within a sampling batch, contributes its prior weight when accepted. These weights are obtained by exponentiating temperature-scaled tactic log-probabilities and are normalized over the node's outgoing edges to obtain $P(s, a)$.

Proof reconstruction and final verification. A root node marked as solved by MCTS does not necessarily imply that the original theorem has been fully proved, as tactic execution may introduce auxiliary declarations or defer proof obligations. Reap therefore restores the original proof state, replays the selected tactics from the root, and independently validates the reconstructed proof using Lean's ordinary declaration-checking pipeline to ensure the soundness of the generated proof.

3.2 Reaper-1: Model and Training

Reaper-1 is a 1.7B-parameter policy–value model initialized from Qwen3-1.7B-Base. Training proceeds in three main stages: continued pre-training (CPT) adapts the base model to Lean code and formal mathematics, supervised fine-tuning (SFT) learns tactic prediction from proof-state transitions, and iterative reinforcement learning improves both the policy and value estimates using trajectories generated by Reap. Training details are provided in the main text and Appendix A. The model weights as well as the complete training pipelines implemented in Scythe-rl will be open-sourced.

Architecture. Reaper-1 uses a shared Transformer backbone for both policy and value prediction. The policy head is the standard language-modeling head and autoregressively generates Lean tactics conditioned on the current proof state and retrieved premises. An additional value head estimates the remaining proof-search cost of the current state using a discretized value representation. In the current implementation, policy and value predictions are requested in parallel.

Continued pre-training. Starting from Qwen3-1.7B-Base, we continue pre-training on Lean and informal mathematics data, combining FineLeanCorpus (Peng et al., 2026), LeanWorkbook (Ying et al., 2024), NuminaMath-Lean (Wang et al., 2025), DeepSeek-Prover-V1 data (Xin et al., 2024), and Mathlib v4.28.0-rc1. For Mathlib, we include source code together with term and declaration data extracted by jixia (FrenzyMath, 2026), and additionally use internally annotated abstract-algebra data.

Supervised fine-tuning. We construct SFT data primarily from existing Lean proofs and Mathlib. jixia (FrenzyMath, 2026) extracts elaboration traces and proof-state transitions, which scythe converts into step-level examples containing tactics, premises, and value-related information. For each state, PS-2 retrieves relevant library declarations, and the model is trained to predict the next tactic conditioned on the state and retrieved premises. We further normalize structural tactics such as `have` and `suffices` into the Reap transition format (Appendix E). These state–premise–tactic examples initialize the policy before reinforcement learning.

Policy–value training objective. After each rollout stage, we collect successful trajectories that pass Lean replay and final verification and convert them into a replay dataset $\mathcal{D}_{\mathrm{RL}} = (s, a, y)$, where a is the successful tactic at proof state s and y is its value target. The policy is trained autoregressively on the complete tactic sequence. For the value target, each valid tactic transition incurs unit search cost; the remaining proof cost is backed up through the corresponding AND–OR

tree and discretized into $K = 32$ bins. We jointly optimize

$$\mathcal{L} = \lambda_P \big(\underbrace{-\mathbb{E}_{(s,a,y)\sim\mathcal{D}_{\mathrm{RL}}} [\log \pi_\theta(a \mid s)]}_{\mathcal{L}_{\mathrm{policy}}} \big) + \lambda_V \big(\underbrace{-\mathbb{E}_{(s,a,y)\sim\mathcal{D}_{\mathrm{RL}}} [\log p_\theta(y \mid s)]}_{\mathcal{L}_{\mathrm{value}}} \big). \tag{4}$$

where $\pi_\theta(a \mid s)$ is the autoregressive probability of tactic a and $p_\theta(y \mid s)$ is the predicted probability of its discretized value target.

Iterative reinforcement learning. Our RL stage follows an AlphaProof-style expert-iteration formulation: MCTS acts as a policy-improvement operator, and the network is trained on search-generated verified trajectories and backed-up value targets. Starting from the SFT checkpoint, we perform five stages of MCTS-guided reinforcement learning on approximately 25K curated training problems extracted from AoPS. During RL1–RL4, each stage performs a full pass over the training corpus while progressively increasing the MCTS simulation budget, starting from 512 simulations in RL1. After performance on training corpus begins to saturate, RL5 switches to a batched multi-rollout regime with 8,192 simulations and pass@8. Beginning with B13, we additionally introduce group-wise value-based trajectory selection. For each theorem, we retain at most two successful trajectories with the smallest absolute backed-up root values. The complete RL procedure uses 60 NVIDIA A100 GPU days.

Premise selector. Following retrieval-augmented proving practice, we condition tactic generation on relevant Mathlib declarations. Our default selector, PS-2, is trained separately following the LeanSearch-PS recipe (Shen et al., 2025), with an updated Mathlib v4.28.0-rc1 snapshot and revised data filters. It remains fixed during prover RL and supplies library declarations to Reaper-1, which performs tactic generation and value prediction. Data construction, the selector-training objective, and hyperparameters are detailed in Appendix B.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate Reaper-1 on five Lean theorem-proving benchmarks: **miniF2F-test** (Zheng et al., 2022), our primary benchmark for competition-style mathematics; **ProofNet-test** (Azerbayev et al., 2023); **FATE-M** (Shen et al., 2025) (150-problem snapshot, commit 4eb33c8); **PutnamBench** (Tsoukalas et al., 2024) (660-problem snapshot, commit 6df1d4c); and **CombiBench** (Liu et al., 2026). We decontaminate all training corpora against every evaluation benchmark using 10-gram filtering, and use these benchmarks only for retrospective evaluation, never for training or model selection.

Baselines and evaluation. We compare with representative whole-proof and step-level Lean provers. Since published systems use different inference budgets, Table 1 reports each result with its original evaluation budget rather than treating the comparisons as compute-matched. Reaper-1 is evaluated using the final RL policy–value checkpoint and the default premise-selection pipeline. Unless otherwise specified, controlled miniF2F experiments use 8,192 MCTS simulations per search and a maximum policy-generation length of 1,024 tokens. Under this configuration, evaluating all 244 miniF2F-test problems at pass@32 only takes approximately 3 hours of wall-clock time using 32 NVIDIA A100 GPUs and 800 CPU cores. The total inference time of the premise selector is approximately 8.6% of the total inference time in the experiment. All reported successes are replayed and verified by Lean.

Premise selection. Premise retrieval is provided by a separate, modular component rather than the Reaper-1 network. Our default setup uses PS-2, an independently trained Qwen3-Embedding-8B selector, which is neither jointly optimized with Reaper-1 nor adapted during RL. We evaluate inference-time replacement with Lean's rule-based selectors, without retraining the policy–value model, in Table 3.

4.2 Main Results

Despite using a 1.7B-parameter generative policy–value model, Reaper-1 is highly competitive with much larger formal provers. On miniF2F-test, it reaches 79.9% pass@32, improving over the 1.7B Kimina-Prover-RL baseline by 3.3 percentage points and outperforming several 7B step-level and whole-proof systems. The advantage is also evident beyond the primary benchmark: Reaper-1 obtains 33.9% on ProofNet-test, compared with 23.7% for REAL-Prover, and reaches 81.3% on FATE-M. On the harder suites, it solves 21/660 PutnamBench problems (pass@32) and 7/100 CombiBench problems (pass@16). Across checkpoints accumulated during our iterative training procedure, the union of solved miniF2F-test problems reaches 81.1%.

Table 1: Main benchmark results. "–" indicates that the corresponding result was not reported by the original work.

System	Budget	miniF2F	ProofNet	FATE-M	PutnamBench	CombiBench
Whole-proof systems						
DeepSeek-Prover-V2-7B (Ren et al., 2025)	pass@32	75.6%	–	–	9/660	6 (pass@16)
Goedel-Prover-V2-8B (Lin et al., 2026)	pass@32	84.6%	–	–	25/660	–
Kimina-Prover-7B (Wang et al., 2025)	pass@32	78.3%	–	–	–	7 (pass@16)
Kimina-Prover-RL-1.7B (Wang et al., 2025)	pass@32	76.6%	–	–	–	–
Pythagoras-Prover-4B (Leang et al., 2026)	pass@32	86.1%	–	–	–	–
Step-level systems						
HunyuanProver (Li et al., 2024)	pass@8	68.4%	–	–	–	–
ABEL	pass@32	41.3%	–	–	8/660	–
REAL-Prover (Shen et al., 2025)	pass@64	54.1%	23.7%	56.7%	–	–
InternLM2.5-StepProver-7B (Wu et al., 2025)	pass@32	63.8%	23.6%	–	–	–
BFS-Prover-V2-7B (Xin et al., 2026)	Accumulative	82.4%	–	–	–	–
Reaper-1 (1.7B)	pass@32	**79.9%**	**33.9%**	**81.3%**	**21/660**	**7/100 (pass@16)**

4.3 Reinforcement-Learning Progression

Figure 3 retrospectively tracks the effect of iterative reinforcement learning from the SFT checkpoint. On miniF2F-test, the number of problems solved under pass@8 increases from 134 for SFT to 191 after RL5, with the largest early improvement occurring during the first RL stage; a similar trend is observed on the full RL training corpus, where pass@1 rises from 19.16% for SFT to 45.27% after RL4. As performance begins to plateau during RL3–RL4, RL5 switches from full-corpus iterations to sequential small-batch updates with a revised rollout and trajectory-selection regime. Despite substantial batch-to-batch variation, performance improves again in the later RL5 batches, indicating that the sequential training regime is associated with extending prover capability after the earlier full-corpus iterations begin to saturate.

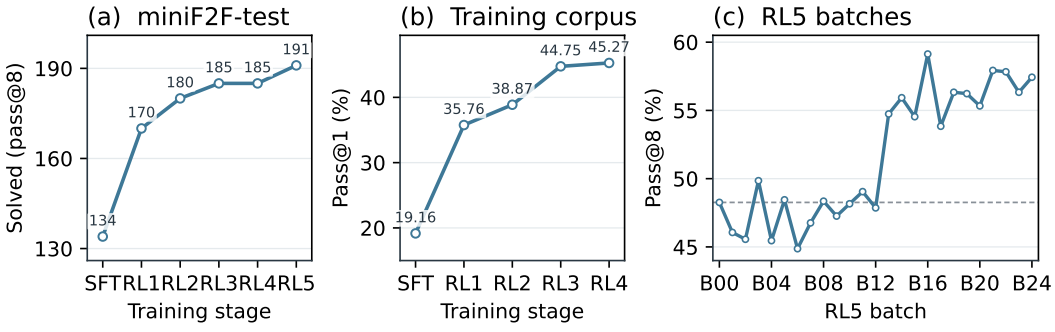

Figure 3: Reinforcement-learning progression. (a) Retrospective evaluation on miniF2F-test at 8,192 MCTS simulations and pass@8. (b) Pass@1 on the full RL training corpus. (c) Sequential RL5 small-batch evaluation; the dashed horizontal line marks the B00 value (48.26%).

4.4 Ablation Studies

Value head. We isolate the effect of the value head guidance during proof search. In the ablated configuration, the neural value prediction is disabled by fixing the predicted value of every state to

zero, while the rest of the search pipeline remains unchanged. As shown in Table 2, on the 193 problems solved by both RL5 and RL5 without the value head, RL5 produces shorter successful proofs in the sense of both average tactic steps and an average root value closer to zero. This suggests that enabling value guidance biases MCTS toward shorter successful trajectories.

Table 2: Value-head ablation on miniF2F-test with 8,192 simulations and pass@32. Tactic steps are computed by first averaging proved passes within each theorem and then averaging equally across theorems. Average tactic steps and root value are computed on the both-solved 193 problems.

Model	**Solved**	**Avg. tactic steps**	**Avg. root value**
RL5	195	16.4553	-9.9174
RL5 w/o value head	193	17.4435	-11.0221

Table 3: Inference-time retriever replacement on miniF2F-test with the same RL5 checkpoint and 8,192 MCTS simulations per search. PS-2 is neural; MePo and SineQuoNon are rule-based. Parentheses give solved/total problems.

Retriever	**PS-2**	**MePo**	**SineQuoNon**
Pass@32	79.9% (195/244)	67.6% (165/244)	75.8% (185/244)

Replacing the premise retriever. We replace PS-2 at inference with Lean's built-in rule-based selectors, MePo (Meng & Paulson, 2009) and SineQuoNon (Hoder & Voronkov, 2011), keeping the final RL5 policy–value checkpoint and search configuration fixed. Despite being trained with PS-2 premises, Reaper-1 achieves 75.8% pass@32 with SineQuoNon without additional training (Table 3). This is 10 problems (4.1 percentage points) below the 79.9% obtained with PS-2; MePo achieves 67.6%. These results demonstrate that the learned prover can operate effectively with a rule-based retriever on this benchmark, supporting inference-time replacement of the neural retrieval component.

4.5 Search Scaling

We next examine how Reaper-1 scales with the MCTS simulation budget. Table 4 reports miniF2F-test performance under pass@32 while varying the maximum number of MCTS simulations allowed for each independent search. Reaper-1 already solves 188 of 244 problems (77.05%) with only 512 simulations. Coverage then begins to saturate: both 2,048 and 4,096 simulations solve the same 194 problems, while the full 8,192-simulation budget adds only one additional theorem. The scaling curve therefore exhibits clear diminishing returns in theorem coverage. The strong performance at relatively small budgets suggests that Reaper-1's final result is not solely a consequence of the largest test-time search budget.

Table 4: Search scaling of Reaper-1 on miniF2F-test under pass@32.

MCTS budget	512	1024	2048	4096	8192
Pass@32	77.05%	78.28%	79.51%	79.51%	79.92%

4.6 How Reinforcement Learning Reshapes Proof Search

Beyond improving theorem coverage, reinforcement learning also changes how the prover interacts with search. We compare the SFT, RL4, and RL5 checkpoints under the same miniF2F setting. To control for theorem difficulty, proof-level statistics are computed on the 151 problems solved by all three checkpoints. Full measurement protocols, statistical analyses, and qualitative examples are deferred to Appendices C and D.

Table 5: How reinforcement learning reshapes proof search. Coverage uses all 244 miniF2F-test problems; proof and action statistics use the 151-problem common solved set.

Metric	**SFT**	**RL4**	**RL5**
Pass@32	62.3%	78.3%	79.9%
Mean proof steps	21.7	11.2	5.2
Mean search-tree nodes	2,368	162	119
Median action characters	19	23	66
Tactic identifiers/action	1.38	1.38	2.80
Actions containing `<;>`	2.9%	4.5%	23.9%
Median evaluation latency (ms)	12.0	15.2	43.2

As shown in Table 5, RL substantially improves coverage while compressing successful search into shorter trajectories and much smaller trees. The later checkpoints also exhibit a clear change in proof organization: actions become longer and structurally denser, with greater use of compound tactic programs. Together with the qualitative cases in Appendix D, this suggests that the policy increasingly organizes proofs around useful intermediate structure and directs search toward steps that make substantive progress, rather than relying on repeated local manipulations.

This compression does not imply less underlying computation. As actions become more compound, their Lean evaluation cost also increases. We therefore describe the observed transition as *search-interaction compression*: RL reduces the number of top-level search decisions while shifting more proof construction and solver work into each individual action.

5 Limitations and Future Work

Reaper-1 uses the number of tactic-level search actions as a proxy for proof cost, which does not always reflect the actual complexity of a Lean proof step. During later RL stages, we observe that the model increasingly exploits Lean constructs that package multiple proof operations into a single search action. Such constructions can beneficially compress local reasoning and reduce search depth, but may also lower the apparent trajectory cost without reducing the underlying computation; sufficiently complex actions can increase Lean execution latency and reduce MCTS exploration efficiency. This limitation suggests that future objectives could incorporate tactic execution time, generated-token length, or finer-grained measures of action complexity. Finally, our experiments focus on Lean 4 and a single 1.7B policy–value model, so the extent to which these search and training dynamics generalize to substantially larger models or other proof assistants remains to be studied.

6 Conclusion

We introduced **Reap**, an open Monte Carlo Tree Search framework for formal theorem proving whose semantics-critical search loop runs directly inside Lean, together with **Reaper-1**, a 1.7B-parameter policy–value model trained through continued pre-training, supervised tactic learning, and iterative MCTS-guided reinforcement learning. Despite its modest scale, Reaper-1 with a separate premise retriever achieves strong performance across multiple Lean benchmarks, including 79.9% pass@32 on miniF2F-test, while remaining competitive with substantially larger provers. Beyond benchmark coverage, Reap provides a lightweight and observable platform for studying how neural policies, value estimation, formal verification, and tree search interact during theorem proving. Our analysis suggests that search-based RL can refine proof strategy, enabling the prover to make effective use of auxiliary lemmas and pursue more direct routes to a complete argument. As individual tactics take on a larger role in carrying out local reasoning, these advances also call for closer attention to their computational demands. By making such learning dynamics accessible to systematic study, Reap supports further research on training objectives that encourage meaningful proof progress.

AI USE STATEMENT

In this work, generative AI tools were used to assist with manuscript writing and organization based on detailed content provided by the authors, literature search, implementation of algorithmic components, and statistical analysis for the RL study. All AI-assisted outputs were reviewed and validated by the authors against the relevant source materials, code, and experimental data. The authors take full responsibility for the methodology, reported results, analyses, and final content of the paper.

REFERENCES

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

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

Kefan Dong and Tengyu Ma. STP: Self-play LLM theorem provers with iterative conjecturing and proving. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 14114–14136. PMLR, 2025. URL https://proceedings.mlr.press/v267/dong25h.html.

FrenzyMath. Jixia. GitHub repository, 2026. URL https://github.com/frenzymath/jixia. Version v4.28.0-rc1.

Kryštof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In *Automated Deduction – CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pp. 299–314. Springer, 2011. doi: 10.1007/978-3-642-22438-6_23.

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

Xingguang Ji, Yahui Liu, Qi Wang, Jingyuan Zhang, Yang Yue, Rui Shi, Chenxi Sun, Fuzheng Zhang, Guorui Zhou, and Kun Gai. Leanabell-Prover-V2: Verifier-integrated reasoning for formal theorem proving via reinforcement learning. *arXiv preprint arXiv:2507.08649*, 2025.

Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 26337–26349, 2022. doi: 10.52202/068431-1910. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/a8901c5e85fb8e1823bbf0f755053672-Abstract-Conference.html.

Joshua Ong Jun Leang, Zheng Zhao, Mihaela Cătălina Stoian, Qiyuan Xu, Haonan Li, Wenda Li, Shay B. Cohen, and Eleonora Giunchiglia. Pythagoras-Prover: Advancing efficient formal proving via augmented lean formalisation. *arXiv preprint arXiv:2606.12594*, 2026.

Chenyi Li, Yanchen Nie, Zhenyu Ming, Gong Zhang, Kun Yuan, and Zaiwen Wen. Opt-Prover: Bridging olympiad and optimization through continual training in formal theorem proving. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=mkHp4ZW01l.

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

Haohan Lin, Zhiqing Sun, Sean Welleck, and Yiming Yang. Lean-STaR: Learning to interleave thinking and proving. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=SOWZ59UyNc.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=j4C0nALrgK.

Junqi Liu, Xiaohan Lin, Bolton Bailey, Jonas Bayer, Yaël Dillies, Weijie Jiang, Xiaodan Liang, Roman Soletskyi, Haiming Wang, Yunzhou Xie, Beibei Xiong, Zhengfeng Yang, Jujian Zhang, Lihong Zhi, Zekai Zhu, Jia Li, and Zhengying Liu. CombiBench: Benchmarking LLM capability for combinatorial mathematics. In *3rd AI for Math Workshop: Toward Self-Evolving Scientific Agents*, 2026. URL https://openreview.net/forum?id=qeZStJVYnc.

Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009. doi: 10.1016/j.jal.2007.07.004.

Zhongyuan Peng, Yifan Yao, Kaijing Ma, Shuyue Guo, Yizhe Li, Yichi Zhang, Chenchen Zhang, Yifan Zhang, Zhouliang Yu, Luming Li, Minghao Liu, Yihang Xia, Jiawei Shen, Yuchen Wu, Yixin Cao, Zhaoxiang Zhang, Wenhao Huang, Jiaheng Liu, and Ge Zhang. CriticLean: Critic-guided reinforcement learning for mathematical formalization. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3049–3088. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.139. URL https://aclanthology.org/2026.acl-long.139/.

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

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

Ziju Shen, Naohao Huang, Fanyi Yang, Yutong Wang, Guoxiong Gao, Tianyi Xu, Jiedong Jiang, Wanyi He, Pu Yang, Mengzhou Sun, Haocheng Ju, Peihao Wu, Bryan Dai, and Bin Dong. REAL-Prover: Retrieval augmented lean prover for mathematical reasoning. *arXiv preprint arXiv:2505.20613*, 2025.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018. doi: 10.1126/science.aar6404.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. PutnamBench: Evaluating neural theorem-provers on the putnam mathematical competition. In *Advances in Neural Information Processing Systems*, volume 37, pp. 11545–11569, 2024. doi: 10.52202/079017-0368.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

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

Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Zheng Yuan, Wenwei Zhang, Dahua Lin, and Kai Chen. InternLM2.5-StepProver: Advancing automated theorem proving via critic-guided search. In *2nd AI for Math Workshop @ ICML 2025*, 2025. URL https://openreview.net/forum?id=qwCqeIg5iI.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-Prover: Advancing theorem proving in LLMs through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Haowei Zhang, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=I4YAIwrsXa.

Ran Xin, Zeyu Zheng, Yanchen Nie, Kun Yuan, and Xia Xiao. BFS-Prover-V2: Scaling up multi-turn off-policy RL and multi-agent tree search for LLM step-provers. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=GPrlMiGg2w.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 21573–21612, 2023. doi: 10.52202/075280-0944. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets_and_Benchmarks.html.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Jiayu Wang, Dahua Lin, and Kai Chen. Lean Workbook: A large-scale lean problem set formalized from natural language math problems. In *Advances in Neural Information Processing Systems*, volume 37, pp. 105848–105863, 2024. doi: 10.52202/079017-3357.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. MiniF2F: A cross-system benchmark for formal olympiad-level mathematics. In *The Tenth International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

A Experimental Details

This section provides additional hyperparameter details for the training and evaluation experiments in Section 4. Unless otherwise specified, proof search uses the MCTS configuration described in Section 3.1 with both policy and value prediction enabled and neural premise selection. All experiments use Lean v4.28.0-rc1 with the corresponding Mathlib snapshot, and a proof is counted as successful only after final replay verification in Lean.

Table 6 summarizes the search and inference configuration used for the controlled miniF2F experiments with 8,192 MCTS simulations. At each policy evaluation, Reaper-1 samples six candidate

Table 6: MCTS and inference hyperparameters used in the controlled miniF2F experiments.

Parameter	Value
Search algorithm	MCTS
Maximum MCTS simulations	8,192
Maximum tree nodes	65,536
Independent attempts	32
Policy samples/state	6
Retrieved premises/state	8
Maximum generation length	1,024
Generation temperature	0.99
Value model	enabled
Tactic heartbeats	10^8
Tactic timeout	10,000 ms
c_{init}	0.001
c_{base}	3,200
γ	0.99
Prior temperature	150
Progressive-sampling C	0.05
Progressive-sampling α	0.6

tactics with temperature 0.99 and a maximum generation length of 1,024 tokens, while the premise selector retrieves eight declarations for the current proof state.

The MCTS-specific hyperparameters in Table 6 instantiate the progressive-sampling rule in Eq. (1) and the PUCT rule in Eq. (2), with the exploration coefficient defined in Eq. (3). The 65,536-node cap is set above the 8,192-step simulation budget and primarily serves as a safety bound on tree growth.

Table 7 reports the main training hyperparameters not detailed in the main text. The MCTS simulation budget doubles across successive RL stages, increasing from 512 simulations per search in RL1 to 8,192 in RL5.

Table 7: Training hyperparameters for Reaper-1.

Setting	Configuration
CPT global batch size	1,024
CPT learning rate	$3 \times 10^{-4} \rightarrow 3 \times 10^{-5}$
CPT schedule	Cosine decay
RL learning rate	$2 \times 10^{-5} \rightarrow 5 \times 10^{-7}$
RL policy/value loss-weight ratio $\lambda_P : \lambda_V$	$1 : 0.3 \rightarrow 1 : 0.05$
RL1 MCTS budget	512 simulations/search
RL2 MCTS budget	1,024 simulations/search
RL3 MCTS budget	2,048 simulations/search
RL4 MCTS budget	4,096 simulations/search
RL5 MCTS budget	8,192 simulations/search (B00–B24)
RL5 passes per problem	8 (B00–B24)

B Premise Selector Data and Training

Our premise-selector dataset follows the state–positive-premise construction used by LeanSearch-PS (Shen et al., 2025), with the Mathlib database updated to the v4.28.0-rc1 snapshot. jixia and Scythe first extracts proof-state transitions and their global references. For each transition, we retain a pair (s, t^+) only when a referenced declaration is present in the filtered Mathlib corpus and its last identifier component occurs as a standalone token in the original tactic. This keeps the label tied to an explicit user-facing theorem reference and allows one query to have multiple positives.

The filtering is deliberately conservative. We keep ordinary nonempty proof states containing a goal marker and remove wrapper/parser nodes, punctuation-only nodes, placeholders, and `calc` aggregation nodes. The corpus filter keeps declarations in Mathlib with a formal statement, while excluding internal or implementation-oriented modules and names (including tactic, parser, elaborator, private, generated, and unsafe-recursion entries) and selected constructor, recursor, and opaque declarations. Exact query–positive pairs are then deduplicated, and the train/validation split is made by query so that all positives for one state remain in the same split.

For reference, the recorded construction starts from 1,024,367 extracted records, removes 13,749 empty states, 7,650 nonempty states without a goal marker, and 50 `calc` aggregation records, and leaves 1,002,918 records before pair deduplication. Deduplication removes 105,504 repeated query–positive pairs, leaving 897,414 unique pairs. With seed 42, the query-grouped split contains 896,902 training pairs and 512 validation pairs from 226 validation queries.

For a query embedding q_i, let d_{i,y_i} be its annotated positive and let $s_{ij} = \langle q_i, d_j \rangle$ be the in-batch query–document similarity. The selector is trained with the masked InfoNCE objective (van den Oord et al., 2018)

$$\mathcal{L}_{\mathrm{PS}} = -\frac{1}{B}\sum_{i=1}^{B} \log \frac{\exp(\widetilde{s}_{i,y_i}/\tau)}{\sum_j \exp(\widetilde{s}_{ij}/\tau)}, \qquad \widetilde{s}_{ij} = \begin{cases} -\infty, & s_{ij} > s_{i,y_i} + \delta, \\ s_{ij}, & \text{otherwise}, \end{cases} \tag{5}$$

where τ and δ control the score-based fake-negative mask. Candidates that are substantially more similar than the recorded positive are removed from the denominator in Eq. (5). The objective remains one-directional (query to premise); the mask is a conservative way to reduce the penalty from one-to-many labels and semantically related in-batch candidates.

The selector is trained with full-parameter ms-swift embedding SFT using InfoNCE, batch negatives, a temperature of 0.1, and the fake-negative mask in Eq. (5). The reported selector uses Qwen3-Embedding-8B, three epochs, learning rate 5×10^{-5}, maximum sequence length 512, bf16 precision, per-device batch size 8, eight data-parallel workers, and dropped incomplete training batches. No hard-negative mining or auxiliary symmetric query–query objective is used in this run; the selector is trained only to rank premises for the current proof-state query.

C Detailed Analysis of Reinforcement-Learning Dynamics

We study not only whether RL solves more theorems, but also how it changes the reasoning during MCTS search. The analysis has two complementary parts. First, we compare SFT, RL4, and RL5 on the same 244 miniF2F-test problems under 32 independent searches and 8,192 MCTS simulations per search. Second, we follow 25 consecutive online-RL rollout batches to characterize when the observed style changes emerge. The paired comparison controls the theorem set and is therefore our primary evidence for checkpoint-level changes; the batch sequence is observational because its theorem sets evolve over training.

C.1 Measurement Protocol

A theorem is counted as solved only when at least one saved trajectory contains a nonempty proof that passes final replay in Lean. For each solved theorem, its *canonical proof* is the verified proof from the earliest successful pass, breaking ties by the smaller number of top-level actions.

Proof-style comparisons use the 151-problem *paired common core* solved by all three checkpoints. This prevents changes in theorem difficulty from being mistaken for changes in style. A proof step is one action submitted by MCTS to Lean. Such an action can contain the <;> combinator, multiple lines, or a nested `by` block, and is therefore not necessarily an atomic tactic. We consequently report action length and internal tactic density alongside top-level step count.

For search-wide cost, we additionally stream all completed `tactic_eval` records with valid start and stop timestamps. Here, "accepted" means that the Lean evaluator returned no error. Exact counts, means, acceptance rates, and structural shares are computed over all calls. Reported quantiles use deterministic uniform reservoir samples. Unless stated otherwise, confidence intervals are

problem-clustered bootstrap intervals, solved-set transitions use the exact McNemar test, and paired continuous outcomes use a two-sided Wilcoxon signed-rank test on $\log(1 + x)$.

C.2 Capability Expansion and Successful-Search Efficiency

Table 8 separates coverage on all 244 problems from proof and search statistics on the 151-problem common core. The dominant capability gain occurs by RL4: pass@32 increases from 62.3% to 78.3%, with 40 newly solved problems and one loss relative to SFT (exact McNemar $p = 3.82 \times 10^{-11}$). RL5 reaches 79.9% and is a strict superset of the RL4 solved set, adding four problems without a loss.

Table 8: Coverage and successful-search efficiency across checkpoints. Coverage uses all 244 miniF2F-test problems; the remaining rows use one canonical proof for each of the 151 problems solved by all three checkpoints.

Metric	SFT	RL4	RL5
Pass@32	152 (62.3%)	191 (78.3%)	195 (79.9%)
Proof steps, mean	21.7	11.2	5.2
Proof steps, median [IQR]	7 [2, 18]	2 [1, 8.5]	2 [1, 3.5]
One-step proofs	23.2%	49.0%	49.0%
Tree nodes, mean	2,368	162	119
Tree nodes, median [IQR]	901 [10, 3,891]	3 [2, 18]	3 [2, 7]
Root visits, median	922	2	2

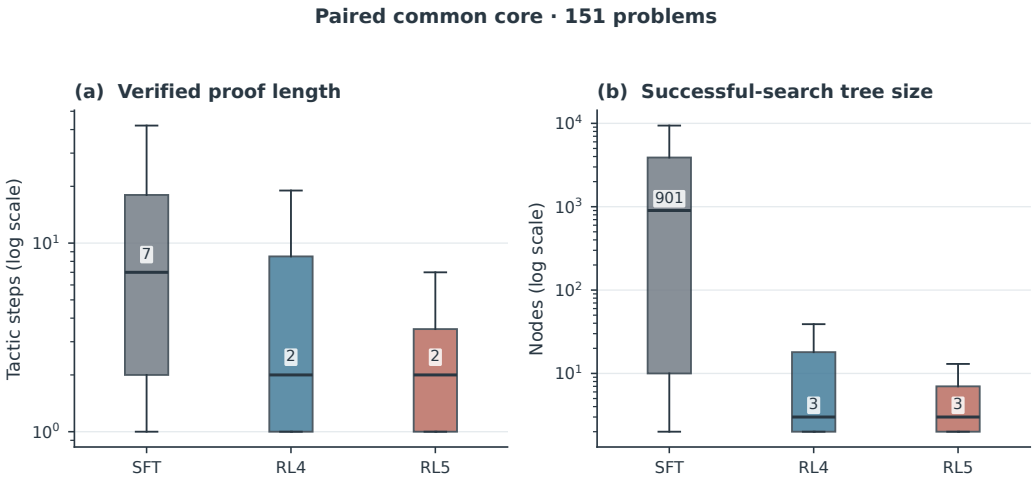

Figure 4: Verified proof length and successful-search tree size on the paired 151-problem common core. Each checkpoint contributes one earliest-success canonical proof per theorem. Boxes show the interquartile range and the annotated line is the median; both axes are logarithmic.

Figure 4 shows that the gain is not only in final coverage. From SFT to RL4, mean proof length falls by 10.44 top-level actions (95% bootstrap CI $[-20.93, -2.15]$), and the mean successful tree shrinks by 2,206 nodes ($[-2,731, -1,723]$). Both paired changes are significant ($p < 10^{-8}$ and $p < 10^{-20}$, respectively). RL5 further reduces the mean proof length and tree size, but their medians remain at two actions and three nodes. Thus, the later-stage effect is concentrated in the long tail: among the 191 theorems shared by RL4 and RL5, 81 proofs become shorter, 78 are unchanged, and 32 become longer, while the mean step difference is -11.88 ($[-18.01, -6.80]$). Selecting each theorem's shortest verified proof instead of its earliest one preserves the same ordering (11.8, 7.8, and 3.9 mean actions for SFT, RL4, and RL5).

The tree reduction is especially important for interpreting the proof-length result. It indicates that RL often causes a successful continuation to be found after expanding fewer search states, rather than merely shortening the emitted text.

C.3 From Solver Calls to Structured Macro-Actions

The successful proofs exhibit a two-stage style transition. We classify an action by its outermost tactic head. The *structural lemma* category contains `have`, `suffices`, `replace`, `let`, `haveI`, `wlog`, and `set`; solvers nested inside these actions are not counted as outer automation. Figure 5 therefore describes the organization of top-level search actions, while Table 9 measures their internal complexity.

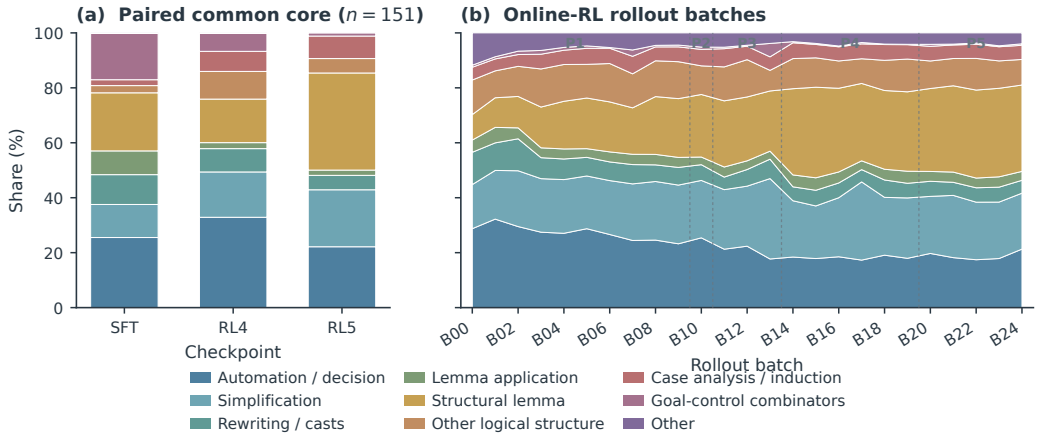

Figure 5: Outer-head tactic composition with a shared taxonomy and legend. (a) Action-weighted shares on the paired common core across checkpoints. (b) Shares across online-RL rollout batches.

Table 9: Granularity of canonical-proof actions on the paired common core. "Identifiers/action" counts tactic identifiers lexically throughout each action, including nested blocks.

Metric	SFT	RL4	RL5
Action characters, median	19	23	66
Tactic identifiers/action	1.38	1.38	2.80
Contains `<;>`	2.9%	4.5%	23.9%
Multiline action	7.0%	3.5%	24.7%
Contains nested `:= by`	1.5%	3.8%	17.5%
Explicit retrieved-premise use	6.4%	7.8%	19.5%

The transition from SFT to RL4 is primarily a shift toward direct, high-leverage search actions. Outer automation rises from 25.5% to 32.9%, case analysis and induction from 2.1% to 7.3%, and goal-control combinators fall from 16.8% to 6.5%. This is consistent with RL4 learning to invoke arithmetic solvers or split the relevant cases earlier, thereby collapsing the search tree.

The transition from RL4 to RL5 changes the abstraction boundary. Structural-lemma actions rise from 15.9% to 35.3%, while outer automation falls to 22.2%. The decline in outer automation does not mean that solvers disappear: per 100 top-level actions, lexical occurrences of `norm_num` increase from 23.4 to 48.1, `simp` from 13.8 to 30.2, and `have` from 16.1 to 41.9. Solvers are increasingly embedded inside local facts, branches, and compound tactic programs. The top-level action count therefore compresses at the same time that each action becomes longer and internally denser.

This distinction matters conceptually. Later RL learns a more programmatic proof interface for MCTS—one action can introduce a lemma, dispatch several subgoals, normalize expressions, and invoke one or more solvers. We refer to this as *search-interaction compression*, not a direct reduction in reasoning depth. The larger premise-use share further suggests that RL5 more often integrates retrieved library facts into these structured programs.

C.4 Where the Computation Moves

To determine whether fewer search interactions translate into cheaper proof search, we analyze 48.57 million completed tactic evaluations across the three runs. Table 10 and Figure 6 reveal a non-monotone trajectory. RL4 generates shorter, more frequently accepted candidates than SFT, but its use of computationally heavy solvers makes the median call slightly slower. RL5 then shifts probability mass toward long macro-actions: the median candidate grows to 89 characters, median evaluator latency reaches 43.2 ms, and acceptance falls to 35.0%.

Table 10: Search-wide Lean tactic evaluations. Counts and rates are exact over all completed calls; quantiles use a deterministic 200,000-call uniform reservoir per run. Summed duration is overlapping service time, not wall-clock runtime.

Metric	SFT	RL4	RL5
Completed evaluator calls	24.57M	18.01M	5.99M
Tactic chars, median [IQR]	47 [19, 87]	25 [15, 47]	89 [57, 159]
Latency ms, median [IQR]	12.0 [5.4, 35.0]	15.2 [4.4, 74.6]	43.2 [9.7, 229.2]
Evaluator accepted	58.1%	61.3%	35.0%
Calls exceeding 1 second	1.85%	3.84%	9.73%
Calls/problem, median	29,512	992.5	257
Summed evaluator duration	699 h	1,067 h	805 h

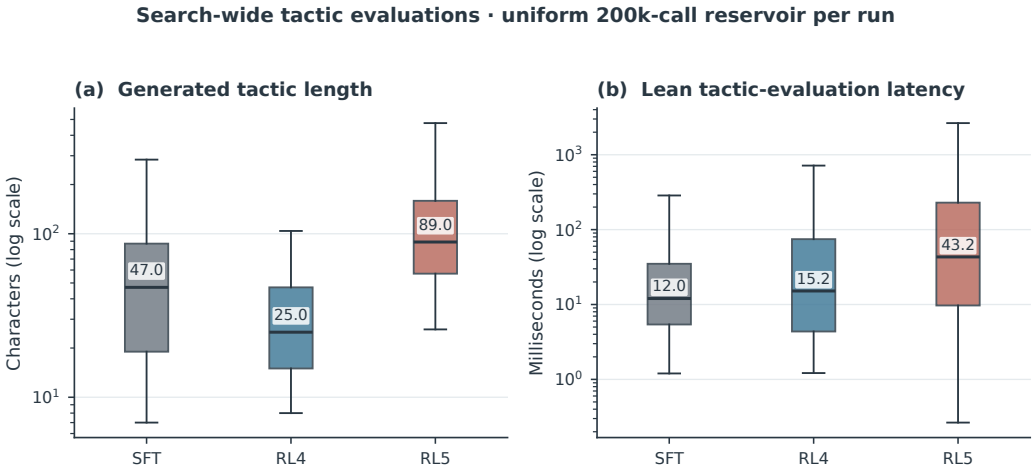

Figure 6: Search-wide distributions of generated tactic length and Lean evaluation latency. Boxes show the interquartile range, whiskers show the 5th and 95th percentiles, and annotations mark medians. Each run contributes a uniform 200,000-call reservoir; axes are logarithmic.

Much of RL5's raw latency and error penalty is a composition effect rather than a uniform degradation. In the 64–127-character bin, for example, RL4 and RL5 have nearly identical acceptance (33.0% versus 34.2%) and accepted-call median latency (46.6 versus 47.1 ms). Nevertheless, longer programs create more opportunities for parse, elaboration, or tactic failure, and RL5 places far more mass in the long bins. Error calls are also slower than accepted calls at every checkpoint.

The number of completed calls per problem drops dramatically, but total evaluator work does not fall in proportion. From RL4 to RL5, call count falls by 66.7%, whereas summed evaluator duration falls by only 24.6%. From SFT to RL4, summed evaluator duration even rises despite fewer calls. Later RL thus reallocates computation from tree expansion to execution inside each action. Optimizing only proof success or top-level trajectory length can favor this exchange; a deployment-oriented objective may additionally need to regularize generated tokens, evaluator errors, latency, or the expanded cost of tactic combinators.

C.5 Longitudinal Dynamics over 25 Online-RL Batches

The checkpoint comparison gives a controlled view of three endpoints. To resolve the transition in more detail, we analyze the 25 consecutive rollout batches (B00–B24), comprising 25,069 problem–batch instances, 56,959 verified successful attempts, and 345.6 million completed tactic evaluations. Adjacent batches use different theorem sets, so the resulting trends describe the joint training-and-rollout process rather than a pure checkpoint treatment effect.

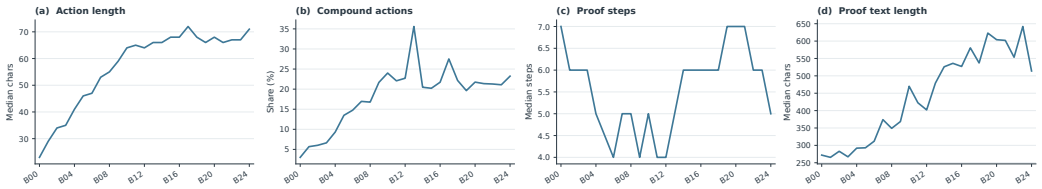

Figure 7: Canonical-proof granularity across online-RL batches: (a) median action length, (b) compound-action share, (c) median top-level proof steps, and (d) median proof-text length.

The most stable longitudinal signal is the growing size of a top-level action. From B00 to B24, median action length increases from 23 to 71 characters, the share containing <;> from 3.0% to 23.2%, and median proof text from 272 to 514 characters, while median top-level proof length changes only from seven to five actions. Across canonical proofs, structural-lemma heads rise from 9.3% in B00 to 31.4% in B24. Their batch-wise share has Spearman $\rho = 0.929$ ($p = 2.00 \times 10^{-11}$), and almost all of the increase comes from `have`. Lexical solver use rises at the same time. This reproduces the endpoint result: proof programs become more locally structured, not merely more reliant on automation.

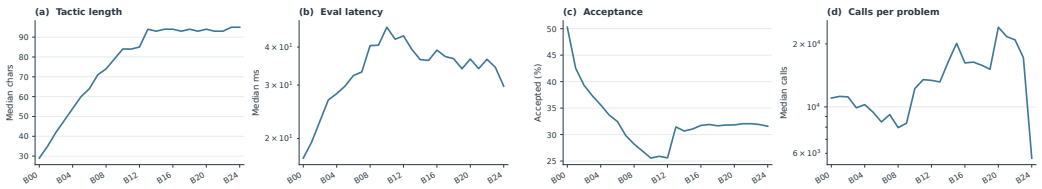

Figure 8: Search-wide tactic-evaluation trends across online-RL batches: (a) median tactic length, (b) median evaluation latency, (c) acceptance rate, and (d) median evaluation calls per problem.

Search-wide evaluation reveals an early transition followed by stabilization. During B00–B09, median evaluated tactic length grows from 29 to 79 characters, median latency from 17.2 to 40.6 ms, and acceptance falls from 50.3% to 26.9%. B10–B12 forms the lowest-acceptance regime: median actions are 84–85 characters and acceptance is only 25.6–25.9%. At B13, median action length reaches 94 characters, yet acceptance jumps to 31.4% and median latency falls to 39.3 ms. Thereafter, length remains near 93–95 characters, acceptance near 31–32%, and latency mostly between 34 and 39 ms.

A coarse adjustment using a common action-length-bin distribution gives the same conclusion. Comparing B00–B12 with B13–B24, length-standardized acceptance increases from 28.9% to 34.9% ($p = 2.49 \times 10^{-5}$ across batches), while standardized mean evaluator duration falls from 559.9 to 442.6 ms ($p = 1.01 \times 10^{-4}$). Later RL therefore appears to improve the executability of an already learned macro-action representation rather than continuing to enlarge it indefinitely. This improvement does not by itself imply lower end-to-end compute: longer individual tactic evaluations can offset reductions in the number of search interactions even under a fixed search budget.

D Qualitative Case Study: Proof-Style Evolution

The aggregate analysis in Appendix C shows that later RL compresses long successful trajectories while increasingly packaging several Lean operations into one top-level action. The value objective in Reaper-1 assigns a unit cost to each tactic-level search action and trains the value head toward

the remaining cost of successful continuations. The controlled ablation in Table 2 further shows that this objective reduces average successful-proof length while yielding similar overall coverage. We now examine a representative theorem to show what the corresponding change looks like inside a complete proof.

Listings 1 and 2 show complete Lean proofs produced by the RL4 and RL5 checkpoints for the same miniF2F theorem. Both prove the identity

$$\sum_{k<n} k^3 = \left(\sum_{k<n} k\right)^2,$$

but they reach the result through substantially different proof trajectories.

RL4 proof. The RL4 checkpoint discovers a valid proof, but its trajectory is dominated by repeated local normalization and repair. After the initial induction, the proof repeatedly invokes `simp`, `ring`, `ring_nf`, `rw`, and automation, introduces a second induction, and revisits closely related algebraic states multiple times.

```
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat

theorem induction_sumkexp3eqsumksq_rl4 (n : ℕ) :
    (∑ k ∈ Finset.range n, k ^ 3) = (∑ k ∈ Finset.range n, k) ^ 2 := by
  induction n
  ·
    simp [Finset.sum_range_succ]
  ·
    simp [Finset.range_succ, Finset.sum_insert, *]
    ring
    abel
    simp_all [pow_succ]
    ring_nf
    simp [pow_succ]
    simp [mul_assoc]
    clear * -
    induction' ⟨ℕ⟩ with n hn
    ·
      rfl
    ·
      simp [Finset.sum_range_succ]
      ring_nf
      rw [hn]
      ring_nf
      abel_nf
      rw [add_comm]
      cases n
      ·
        rfl
      ·
        ring
        simp [add_comm, add_left_comm, add_assoc,
          mul_comm, mul_left_comm, mul_assoc] at *
        ring
        simp [Finset.sum_range_succ] at *
        ring_nf at hn
        simp [add_assoc] at *
        simp [add_assoc, add_comm, add_left_comm] at *
        simp [Finset.sum_range_succ] at *
        ring
        rw [Finset.sum_range_id]
        ring_nf at hn ⊢
        simp [mul_add, mul_comm, mul_left_comm] at *
        ring
        simp [add_assoc, add_comm, add_left_comm] at *
        ring
        simp [Nat.mul_sub_left_distrib,
          Nat.mul_sub_right_distrib, Nat.mul_add,
          Nat.mul_comm, Nat.mul_left_comm] at hn ⊢
        ring_nf
        ring_nf at *
        simp [mul_assoc, mul_comm, mul_left_comm] at hn ⊢
```

```
        ring
        symm
        ring_nf at *
        have := hn
        aesop
        cases n
        ·
          norm_num [this]
        ·
          ring
          rw [add_comm]
          have this := this
          ring
          ring_nf at *
          aesop
          cases n
          ·
            grind
          ·
            ring
            ring_nf at this
            revert this
            repeat rw [Finset.sum_range_id]
            grind [mul_succ, mul_add, add_mul, add_assoc,
              add_comm, add_left_comm, Nat.pow_succ,
              Nat.mul_sub_left_distrib,
              Nat.mul_sub_right_distrib,
              Nat.div_eq_of_lt, Nat.succ_mul]
```

Listing 1: Complete proof generated by the RL4 checkpoint.

The proof has a strongly exploratory character: the model repeatedly changes the algebraic normal form and applies automation. In particular, several rounds of `ring`/`ring_nf` and `simp` are interleaved with nested case analysis and another induction.

RL5 proof. By RL5, the proof is considerably more concentrated. The checkpoint starts from the same induction structure, immediately incorporates the induction hypothesis, rewrites the square and the closed form of the finite sum, and reduces the remaining obligation to a targeted parity split followed by arithmetic automation.

```
theorem induction_sumkexp3eqsumksq_rl5 (n : ℕ) :
    (∑ k ∈ Finset.range n, k ^ 3) = (∑ k ∈ Finset.range n, k) ^ 2 := by
  induction n with
    | zero => ?_
    | succ n ih => ?_
  ·
    simp_all
  ·
    simp [Finset.sum_range_succ, ih]
    rw [add_sq]
    rw [Finset.sum_range_id]
    cases n
    ·
      simp at ih ⊢
    ·
      ring_nf
      simp [add_assoc, add_comm, add_left_comm] at ih ⊢
      ring_nf at *
      simp [Nat.add_assoc, Nat.add_comm,
        Nat.add_left_comm] at *
      aesop
        (add simp [Nat.pow_two, Nat.mul_div_cancel_left])
      rcases n.even_or_odd' with ⟨m, rfl | rfl⟩ <;>
        grind [Nat.mul_div_cancel']
```

Listing 2: Complete proof generated by the RL5 checkpoint for the same theorem.

The difference is not merely textual brevity. The RL5 proof follows a much clearer global progression: *induction* → *use of the induction hypothesis* → *algebraic normalization* → *targeted parity split* → *arithmetic closure*. In contrast, the RL4 proof spends many more search actions on repeated normalization, nested case analysis, and local recovery.

This behavior is consistent with the intended effect of Reaper-1's cost-sensitive value-guided search ablation in Table 2. Because each tactic-level transition incurs a unit cost, lower-cost successful

continuations receive more favorable value targets, creating pressure toward proof trajectories that reach solved states through fewer search decisions. This example further illustrates what the corresponding change can look like at the level of an individual proof.

Shorter continuations can also be advantageous to MCTS because they reduce the effective depth at which a successful leaf must be discovered, allowing a fixed simulation budget to concentrate on fewer proof decisions. We view this as one mechanism that may contribute to the continued improvement on harder problems in later RL stages. Moreover, the final RL5 proof also illustrates a limitation of our current cost definition: the use of <;> packages multiple internal tactic applications into one search-level action. Thus, the learned preference for shorter trajectories captures genuine removal of redundant search in this example, but may also encourage syntactic compression of several Lean operations into a single action, as discussed in Section 5.

E TACTIC NORMALIZATION

Lean's structural tactics expose points at which the proof state branches or changes shape. Tactics such as `have`, `suffices`, and `calc` can therefore provide valuable training signals, including the decomposition choices that precede the final proof steps. During ordinary elaboration, however, these tactics expect their proof bodies to be supplied as complete tactic blocks. A standalone action such as `have ... := by` leaves a nested goal unfinished and cannot be replayed as a valid tactic transition.

We normalize such actions by replacing each nested `by` block with a placeholder and exposing the resulting subgoals with Lean's bullet syntax ("·"). The normalized form is a valid tactic sequence: the outer structural tactic remains visible as one search action, while each generated subgoal is solved by a separate bullet. This preserves the original goal decomposition and makes the transition available to training and analysis.

The following pairs show the transformation for three representative structural tactics. In each pair, the first listing is the original nested form and the second is the bullet-normalized form.

The `have` tactic. Here we replace the nested proof of `key` with `?_` and expose the continuation and the new subgoal as two bullet goals. Listings 3 and 4 show the original and normalized forms.

```
theorem coe_eq_inv_of_sq_eq_one
    (hϕ : FixedPointFree ϕ) (h2 : ϕ^[2] = _root_.id) :
    ⇑ϕ = (·⁻¹) := by
  ext g
  have key : g * ϕ g = 1 := by
    simpa [List.range_succ] using hϕ.prod_pow_eq_one h2 g
  rwa [← inv_eq_iff_mul_eq_one, eq_comm] at key
```

Listing 3: Original nested `have` proof.

```
theorem coe_eq_inv_of_sq_eq_one
    (hϕ : FixedPointFree ϕ) (h2 : ϕ^[2] = _root_.id) :
    ⇑ϕ = (·⁻¹) := by
  ext g
  have key : g * ϕ g = 1 := ?_
  · rwa [← inv_eq_iff_mul_eq_one, eq_comm] at key
  · simpa [List.range_succ] using hϕ.prod_pow_eq_one h2 g
```

Listing 4: Bullet-normalized `have` proof.

The `suffices` tactic. The nested conclusion is replaced by `from ?_`; the two bullets then prove the conclusion and the auxiliary equivalence introduced by `suffices`, respectively. Listings 5 and 6 show the corresponding pair.

```
/-- Conjugacy classes form a partition of G, stated in terms of cardinality. -/
theorem sum_conjClasses_card_eq_card [Fintype <| ConjClasses G] [Fintype G]
    [∀ x : ConjClasses G, Fintype x.carrier] :
    ∑ x : ConjClasses G, x.carrier.toFinset.card = Fintype.card G := by
  suffices (Σ x : ConjClasses G, x.carrier) ≃ G by
    simpa using (Fintype.card_congr this)
  simpa [carrier_eq_preimage_mk] using Equiv.sigmaFiberEquiv ConjClasses.mk
```

Listing 5: Original nested `suffices` proof.

```
/-- Conjugacy classes form a partition of G, stated in terms of cardinality. -/
theorem sum_conjClasses_card_eq_card [Fintype <| ConjClasses G] [Fintype G]
    [∀ x : ConjClasses G, Fintype x.carrier] :
    ∑ x : ConjClasses G, x.carrier.toFinset.card = Fintype.card G := by
      suffices (Σ x : ConjClasses G, x.carrier) ≃ G from ?_
      · simpa using (Fintype.card_congr this)
      · simpa [carrier_eq_preimage_mk] using Equiv.sigmaFiberEquiv ConjClasses.mk
```

Listing 6: Bullet-normalized `suffices` proof.

The `calc` tactic. Each nested proof attached to a calculation step becomes a placeholder, and the four resulting equality goals are discharged by bullets after the calculation block. Listings 7 and 8 show this conversion.

```
/-- In a nontrivial monoid with zero, zero and one are different. -/
instance NeZero.one : NeZero (1 : M₀) := ⟨by
  intro h
  rcases exists_pair_ne M₀ with ⟨x, y, hx⟩
  apply hx
  calc
    x = 1 * x := by rw [one_mul]
    _ = 0 := by rw [h, zero_mul]
    _ = 1 * y := by rw [h, zero_mul]
    _ = y := by rw [one_mul]⟩
```

Listing 7: Original nested `calc` proof.

```
/-- In a nontrivial monoid with zero, zero and one are different. -/
instance NeZero.one : NeZero (1 : M₀) := ⟨by
  intro h
  rcases exists_pair_ne M₀ with ⟨x, y, hx⟩
  apply hx
  calc
    x = 1 * x := ?_
    _ = 0 := ?_
    _ = 1 * y := ?_
    _ = y := ?_
  · rw [one_mul]
  · rw [h, zero_mul]
  · rw [h, zero_mul]
  · rw [one_mul]⟩
```

Listing 8: Bullet-normalized `calc` proof.

The same transformation is applied to the other structural tactics used by the rollout pipeline, including `cases`, `induction`, `obtain`, `show`, `haveI`, `let`, and `letI`. The normalized representation keeps these state transitions explicit while remaining executable by Lean's elaborator.

F From Lean Source Code to MCTS Trees

To support SFT on existing Lean codebases, we use Jixia to generate elaboration traces for the source code, which include the proof states before and after each tactic, together with the tactic's source span and the constants referenced in its elaborated subtree. These traces describe successful proofs, but they are not yet represented in the raw MCTS-tree schema used by Scythe-rl. In particular, when a tactic replaces one goal with several goals, the resulting proof obligation must be represented by an AND node.

We traverse jixia's elaboration tree to collect executable tactic events, then reconstruct proof transitions from their recorded input and output states and source ranges. This is necessary because the elaboration tree contains other nodes such as parser wrappers, compound tactics, and nested local proof blocks, which are irrelevant in our state-action model.

In the resulting tree, each proof state is represented by an OR node with one child, where the outgoing edge is the tactic used on that state. If the tactic produces exactly one remaining goal, the edge

leads directly to the OR node for that goal. If it produces multiple goals, the edge instead leads to an AND node, which then leads to the OR nodes for this set of goals.

The result is then serialized using the same MCTS-tree schema as proof-search rollouts. Each real tactic edge records the constants found in its elaborated subtree in the `premise` field. The format is fully compatible with MCTS trees generated by proof search. The converted tree can therefore enter the same replay and training pipeline as a tree produced by proof search, allowing existing Lean source repositories, such as Mathlib, to be freely mixed alongside search-generated rollouts.