
ImProver 2: Iteratively Self-Improving LMs for Neurosymbolic Proof Optimization

Riyaz Ahuja
Carnegie Mellon University
riyaza@andrew.cmu.edu

Tate Rowney
Carnegie Mellon University
trowney@andrew.cmu.edu

Jeremy Avigad
Carnegie Mellon University
avigad@andrew.cmu.edu

Sean Welleck
Carnegie Mellon University
swelleck@andrew.cmu.edu

Abstract

Formal mathematics libraries are rapidly expanding, creating a growing need to refactor verified proofs for maintainability and to improve training data quality for neural provers. However, scalable proof optimization is hindered by heterogeneous and heuristically specified objectives, scarce data, and high training and inference costs. To overcome these challenges, we introduce ImProver 2, a neurosymbolic framework for automated proof optimization in Lean 4. ImProver 2 combines a data-efficient expert-iteration pipeline with a scaffold that exposes formal structure alongside lightweight informal abstractions. We further introduce a suite of metrics capturing structural proof properties. Using ImProver 2, we train a 7B-parameter model that outperforms orders-of-magnitude larger models within the same model family, and is competitive with mid-tier frontier models across metrics. We additionally demonstrate that our neurosymbolic scaffold significantly improves performance across both small and frontier models. We show that with proper scaffolding and training, small models can effectively restructure research-level proofs over complex and varied metrics, matching substantially larger systems and establishing proof optimization as a scalable, learnable task.

1 Introduction

Formal proof assistants such as Lean [Moura and Ullrich, 2021], Rocq [The Coq Development Team, 2024], and Isabelle [Wenzel et al., 2008] have transformed mathematical practice by making the correctness of proofs explicit and mechanized. Community libraries such as Lean’s Mathlib [The Mathlib Community, 2020] now grow at a pace driven both by increasing human contributions and by recent advances in neural theorem proving and autoformalization [Achim et al., 2025, Hubert et al., 2025].

This rapid expansion raises multiple concerns about issues of data quality. First, this expansion stresses the maintainability, coherence, and long-term usability of libraries due to proofs that are often heterogeneous in style and clarity [The Mathlib Community, 2020]. Second, low library quality decreases its utility as training data: modern theorem provers and autoformalizers increasingly train on these very corpora, so the structure and readability of proofs directly shape downstream prover performance [Gu et al., 2025]. Unfortunately, the growth rate of formal libraries already exceeds what human reviewers and maintainers can reliably curate; and moreover, this discrepancy is only expected to widen due to increasing volumes of machine-generated proofs which, even when guaranteed to be correct, do not carry similar guarantees as to the proof’s quality, modularity, or understandability [Chen et al., 2025].

This motivates automated proof optimization: given a verified proof, produce a formally correct rewrite that scores better under a user-defined objective, such as shorter length, higher modularity, or fewer explicit dependencies [Ahuja et al., 2025]. Because objectives vary by use case and formal context, a practical method must scale across arbitrary metrics and research-level theorems. Small specialized models are especially attractive for this setting because relevant data is scarce in general corpora, library-scale deployment can require millions of samples, and local open-weight models are more accessible to formalization projects.

We address these problems with **ImProver 2**, a self-improving pipeline that trains small language models (SLMs) to optimize Lean proofs under a wide-ranging class of metrics. Our core idea is to leverage *iterative preference optimization*: iterating between generating proof candidates, scoring them according to correctness and the desired optimization criterion, and learning from the resulting pairs of higher and lower scoring proofs. In particular, we extend the Iterative Reasoning Preference Optimization (IRPO) [Pang et al., 2024] algorithm with a new replay buffer that balances old and newly generated data for use in the next round of training, preventing model collapse and allowing for monotonic improvement over many rounds. We additionally give the model access to rich information from the Lean theorem proving environment, including goal states, informalized summaries, lemma context, and examples, which we term *neurosymbolic augmentation*. We use **ImProver 2** to train models for three different metrics: the *length* of the proof, *modularity* (the ability to divide the proof into a series of smaller lemmas), and the explicit *dependencies* used by the theorem, and show that our trained models can substantially improve over their base model and remain competitive with much larger unscaffolded systems on research-level theorems. In summary, our contributions are:

1. *Self-improving SLMs for proof refactoring*. We show that iterative preference optimization can bootstrap small language models on proof optimization in specialized research libraries, making them competitive with much larger models in several unscaffolded comparisons.
2. *Structural optimization metrics*. Beyond proof length [Ahuja et al., 2025, Gu et al., 2025], we study *modularity* and *dependency* metrics that leverage proof structure and the surrounding library. These metrics were chosen with formal mathematics experts and target distinct proof-refactoring objectives.
3. *Neurosymbolic augmentation for research mathematics*. We extract relevant lemmas or definitions, goal-state traces, and automatic informalizations of target proofs. This augmentation boosts both small and large models on proof optimization.

We additionally open-source our code and data¹.

2 Related Work

Interest in neurosymbolic theorem proving—the use of deep learning to create or manipulate verified mathematical proofs in languages such as Lean 4 [Moura and Ullrich, 2021]—has seen significant advancements in recent years [Lu et al., 2023, Li et al., 2024]. In particular, much research has focused on generating formal proofs given their statements [Polu and Sutskever, 2020], with recent systems achieving high performance on nontrivial benchmarks and internationally renowned mathematics competitions [Hubert et al., 2025, Achim et al., 2025, Chen et al., 2025]. Many systems additionally utilize neurosymbolic augmentation, providing a generative prover model with information gathered from within the proof environment [Yang et al., 2023, Ahuja et al., 2025, Lin et al., 2025]. However, both formal and informal proofs generated by current LLM-based systems often suffer from stylistic irregularities even when they are sound, including redundant steps or a structure which does not clearly represent the broader logical argument [Frieder et al., 2025].

Previous work by [Ahuja et al., 2025, Gu et al., 2025] has attempted to rectify these issues by creating LLM-based agents to refactor formal proofs. Ahuja et al. [2025] created a system capable of optimizing towards multiple metrics of improvement; however, it relied on general-purpose closed-source models, leading to substantial deployment costs and limited ability to improve performance beyond these models’ baseline. Gu et al. [2025] focused solely on optimizing the token count of proofs according to a complex tokenizer intended to reduce the time required to compile them; they do not examine other metrics, leaving out important use cases and limiting its utility to research

¹Github

mathematicians. Furthermore, the works above do not fully utilize the information available through working in an interactive theorem proving environment via goal-state extraction [Polu and Sutskever, 2020], premise retrieval [Yang et al., 2023], or auto-informalization [Hattori et al., 2025]; both of the above overlook at least one of these aspects.

3 Proof Optimization

Given a verified proof, a proof optimization agent synthesizes a semantically equivalent proof that is “better” under a user-specified objective while remaining correct according to the Lean kernel. We follow the setup of Ahuja et al. [2025], Gu et al. [2025] and introduce two additional structural metrics: *modularity* and *dependencies*.

3.1 Setup and notation

Let \mathcal{C} denote proof contexts (imports, local declarations, module metadata, etc.), \mathcal{X} theorem statements, and \mathcal{Y} proofs. We consider $(c, x, y) \in \mathcal{C} \times \mathcal{X} \times \mathcal{Y}$, where y is a purported proof of x in c which may or may not be sound.

We define a *verifier* as a computable function

$$v(c, x, y) : \mathcal{C} \times \mathcal{X} \times \mathcal{Y} \rightarrow \{0, 1\},$$

This function outputs 1 if y is a syntactically correct and sound proof of x in c . We also define

$$\mathcal{F}(c, x) = \{y \in \mathcal{Y} : v(c, x, y) = 1\}.$$

Two proofs $y, y' \in \mathcal{F}(c, x)$ are said to be *semantically equivalent*.

We use the Lean 4 language [Moura and Ullrich, 2021] as our verifier; the language’s kernel/type-checker provides a strong guarantee of correspondence with a type-theoretic model of modern mathematics.

3.2 Optimization Objective

Two semantically equivalent proofs may have significant syntactic differences, and moreover certain characteristics may make them more or less desirable for use in practice. To quantify this, we define an *optimization metric* as a computable function

$$\mu : \mathcal{C} \times \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R},$$

Given an initial theorem and proof (c, x, y_0) , we aim to find a verifiably correct proof that maximizes metric score:

$$\arg \max_{\substack{y \in \mathcal{Y} \\ v(c, x, y) = 1}} \mu(c, x, y) \quad (1)$$

In practice, we approximate this via language model-based Lean 4 code generation: by independently generating multiple variants, we may pick the one with greatest improvement, if it surpasses the original.

3.2.1 Metrics of Interest

In this work, we focus and evaluate on a collection of three metrics, which are designed to be practical and interpretable structural objectives for formal proof optimization. These metrics are proxies rather than reviewer-validated measures of subjective proof quality: they support automatic evaluation at scale, but they do not by themselves establish human maintainer preference or downstream theorem-prover utility.

- **Length:** We aim to minimize the length of proofs, measured by the number of tactics used. In practice, proof shortening (or “golfing”) is a common activity in formal mathematics [The Mathlib Community, 2020], as shorter proofs are often easier to read and maintain, and reduce overhead during compilation. As such, we define the length metric μ_{len} as the negative of the number of tactics.

Original

```
theorem isCoatom_iff [OrderTop A] {K :
  A} :
  IsCoatom K ↔ K ≠ T ∧ ∀ H g, K ≤
  H → g ∉ K → g ∈ H → H = T :=
  by
  simp_rw [IsCoatom, lt_iff_le_not_le,
    SetLike.not_le_iff_exists,
    and_comm (a := _ ≤ _), and_imp,
    exists_imp, ← and_imp, and_comm]
```

Optimized (Dependency)

```
theorem isCoatom_iff [OrderTop A] {K :
  A} :
  IsCoatom K ↔ K ≠ T ∧ ∀ H g, K ≤
  H → g ∉ K → g ∈ H → H = T :=
  by
  constructor <=> intro h
  <=> simp_all [IsCoatom,
    lt_iff_le_not_le,
    SetLike.not_le_iff_exists]
  <=> tauto
```

Original

```
theorem mem_cross_iff (x y : TSet γ) :
  ∀ a, a ∈ ' cross hβ hγ hδ x y ↔ ∃
  b c, a = (b, c)' ∧ b ∈ ' x ∧ c ∈
  ' y := by
  intro a
  rw [cross, mem_inter_iff, vCross_spec]
  constructor
  · rintro ⟨h₁, b, c, rfl, h₂⟩
    simp only [op_mem_converse_iff,
      vCross_spec, op_inj] at h₁
    obtain ⟨b', c', ⟨rfl, rfl⟩, h₁⟩ :=
      h₁
    exact ⟨b, c, rfl, h₁, h₂⟩
  · rintro ⟨b, c, rfl, h₁, h₂⟩
    simp only [op_mem_converse_iff,
      vCross_spec, op_inj]
    exact ⟨⟨c, b, ⟨rfl, rfl⟩, h₁⟩, ⟨b,
      c, ⟨rfl, rfl⟩, h₂⟩⟩
```

Optimized (Length)

```
theorem mem_cross_iff (x y : TSet γ) :
  ∀ a, a ∈ ' cross hβ hγ hδ x y ↔ ∃
  b c, a = (b, c)' ∧ b ∈ ' x ∧ c ∈
  ' y := by
  simp_all [cross, mem_inter_iff,
    vCross_spec,
    op_mem_converse_iff, op_inj]
  <=> aesop
```

Original

```
lemma KD_weakerThan_KDB : (Hilbert.KD α
  ) ≤s (Hilbert.KDB α) :=
  normal_weakerThan_of_subset $ by
  intro; aesop;
```

Optimized (Modularity)

```
lemma KD_weakerThan_KDB : (Hilbert.KD α
  ) ≤s (Hilbert.KDB α) := by
  have h₁ : (L0.Modal.Hilbert.KD α
  ).axioms ⊆ (L0.Modal.Hilbert.KDB
  α).axioms → (Hilbert.KD α) ≤s
  (Hilbert.KDB α) := by
    intro h
    apply normal_weakerThan_of_subset
    apply h
  have h₂ : (L0.Modal.Hilbert.KD α
  ).axioms ⊆ (L0.Modal.Hilbert.KDB
  α).axioms := by
    intro φ hφ
    cases' hφ with hφ hφ
    · simp_all [L0.Modal.Hilbert.KD]
    · simp_all [L0.Modal.Hilbert.KDB]
  exact h₁ h₂
```

Figure 1: ImProver 2 automatically optimizes human-written proofs to reduce explicit dependencies, minimize length, or maximize proof modularity, while maintaining formal correctness.

- **Dependencies:** We aim to minimize the explicit dependency footprint of proofs, measured by the number of unique external lemmas explicitly named in the proof. This metric does not measure semantic independence from the library: a proof using `simp` or `omega` may still rely on many facts internally. This encourages self-contained proofs that do not require the use or memorization of large numbers of dependency names, improving maintainability.² More specifically, given a theorem and proof (c, x, y) , we compute $\text{Deps}_{c,x,y}$, the set of all theorems explicitly named in the proof y (see Section B). The metric is then defined as $\mu_{\text{dep}}(c, x, y) := -|\text{Deps}_{c,x,y}|$.

- **Modularity:** We aim to maximize the *modularity* of proofs, which is intuitively understood to be the number of independent subproofs in our proof. This is a structural objective motivated by proof decomposition, lemma-generation workflows, and maintainable proof organization.

To quantify modularity, we postprocess Lean proofs to deconstruct them into a tree of tactics, with edges representing the logical dependencies and flow between tactics in the proof. In this tree, we mark a subset of edges as “spawned” if they correspond to goals that are introduced by the proof but are not direct subgoals of the current tactic, which naturally arise from tactics like `have`, `calc`, etc., and correspond with the informal notion of “modular” independent subproofs.

We therefore define the modularity metric as $\mu_{\text{mod}}(c, x, y) = |\{\text{effective spawned goals in } y\}|$. The designation of which spawned goals are non-trivial and “effective” requires the use of several heuristics; a complete definition is provided in Appendix A. These safeguards reduce trivial goal-spawning and duplicate-wrapper artifacts, but they do not eliminate all stylistically debatable decompositions.

²We would like to thank Dr. Heather Macbeth of Imperial College London for reaching out to suggest the idea behind this metric.

Example Proof Tree

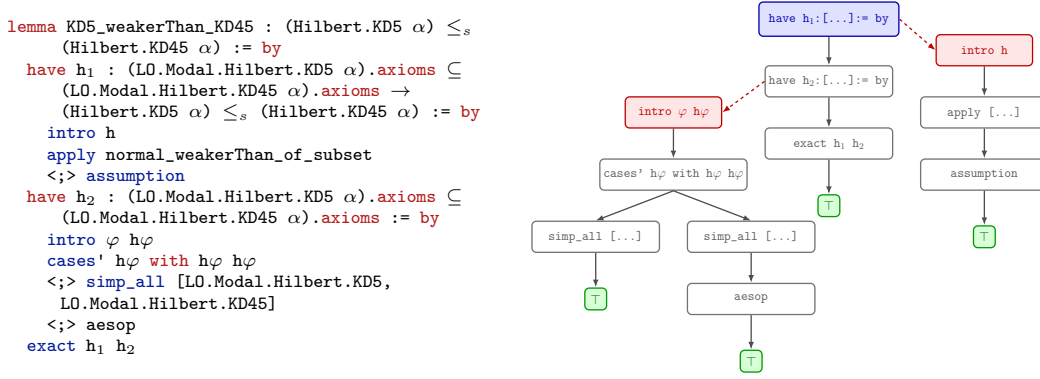


Figure 2: Example Lean proof (top) and its generated proof tree (bottom). Note the root node (in blue), and the two effective spawned goals coming from h_1 and h_2 (in red).

4 ImProver 2

We present ImProver 2, a training pipeline that bootstraps small language models (SLMs) for a target proof-optimization metric. It combines IRPO training (4.3.2), a replay buffer balancing old and newly generated data (4.3.1), and neurosymbolic augmentation for generation (4.2).

4.1 Overview

Given an un-modified base language model G_0 , at the t -th iteration ImProver 2’s core loop aims to train the model G_{t+1} from G_t as follows (also depicted in Figure 3):

1. For some budget hyperparameter $n \in \mathbb{N}$, generate n candidate proofs per problem in the training set using the current model G_t (4.2), providing the model with neurosymbolic augmentation (4.2.1) and a description of the metric to assist in generation. These new potential proofs form a new dataset (denoted $\mathcal{D}_{\text{nr}}^{(t)}$).
2. The previous iteration’s dataset, $\mathcal{D}_{\text{re}}^{(t-1)}$ (our *replay buffer*), is interleaved with $\mathcal{D}_{\text{nr}}^{(t)}$ to get $\mathcal{D}_{\text{re}}^{(t)}$ (see 4.3.1).
3. The model G_t is trained to obtain G_{t+1} (4.3.2). Our reinforcement learning policy of choice, known as Iterative Reasoning Preference Optimization or IRPO [Pang et al., 2024], relies on preference pairs of desirable/undesirable proofs, so $\mathcal{D}_{\text{re}}^{(t)}$ is filtered to remove low-quality solutions and pairs are created to form $\mathcal{D}_{\text{IRPO}}^{(t)}$ (again described in 4.3.1).
4. G_{t+1} is evaluated with n samples on the test set, again similarly to 4.2. The improvement in metric score of the new proofs over the originals is calculated.

The loop is repeated until convergence of the average improvement score, or exhaustion of the compute budget.

4.2 Generation

A central feature of ImProver 2 is self-generation of training data: at each round, the current model $G^{(t)}$ receives the theorem statement, original proof, target metric, and proof-environment context, then samples n candidate rewrites per theorem.

4.2.1 Neurosymbolic Augmentation

Formal proof environments provide substantial opportunities to obtain relevant information about a proof. We prompt our language model with additional neurosymbolic context that exposes the structure and dependencies of a problem at both a formal and informal level. This augmentation

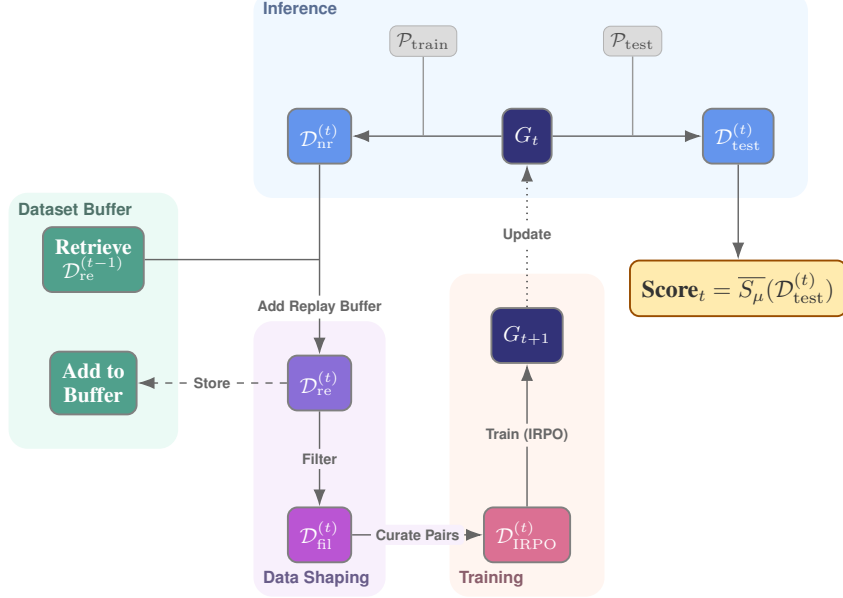


Figure 3: **ImProver² training loop.** The diagram illustrates the iterative process of generation, retrieval, filtering, and training. Node colors represent the evolution of data from initial sampling (Blue) through processing (Purple) to training (Magenta).

comes from three sources: a context slice to find relevant lemmas or definitions, goal-state traces to highlight the exact effect of each tactic on the progress of the proof, and auto-informalization to provide a higher-level natural language description of the proof in question. Each of these sources is described in detail below.

Context When working with proofs in high-dependency environments, it is likely that a given proof y_0 relies on many lemmas, definitions, and other formal objects defined in the context c . We aim to extract and serialize a minimal set of these objects to better inform our generation process: the signatures of all definitions and theorems that are directly referenced by name in the theorem statement x or the original proof y_0 are collected and provided to the model, along with any associated documentation comments. The process of collecting and filtering these is outlined in Appendix B.

Chain-of-States (CoS) Chain-of-states prompting [Ahuja et al., 2025] provides a language model with the explicit state and remaining goals of a proof following the application of each tactic/step, providing richer information about the proof’s structure than is normally available. We utilize Lean’s InfoTree structures to obtain and serialize these states, and interleave them into the original proof as comments. This process is described formally in Appendix D.

Auto-informalization Utilizing natural language to guide the generation of formal proofs has been shown to significantly increase the capabilities of LLM-based systems on formal mathematical tasks [Jiang et al., 2023]. We therefore expose natural-language sketches of each target proof to the model, providing a fuzzy layer of abstraction that captures the “meaning” of formal items while being robust to syntactic variation and surface-level noise.

More concretely, we prompt a language model using the proof’s chain-of-states information as described above to translate a Lean proof into natural language by explaining the effect of each tactic on the proof state and providing this to the proof optimizer model. We emphasize that this informalization is a secondary channel for the generator and serves simply as an additional representation of the target theorem; outputs are still prompted to be generated formally, and correctness is still judged formally.

4.3 Training

We adapt IRPO [Pang et al., 2024] for proof optimization by ranking candidates using both correctness and metric improvement, yielding denser preference signals than binary success alone. We also mix new and old samples through replay, and train only on the final proof output rather than the reasoning trace.

4.3.1 Datasets and Replay Buffer

Indiscriminate self-training can collapse model output distributions [Shumailov et al., 2024]. Our replay buffer filters new samples, combines them with existing data, and sorts them by improvement before training.

After generating $\mathcal{D}_{\text{nr}}^{(t)}$, we combine candidates for each eligible problem with prior candidates, remove trivially easy problems, and partition proofs into “winners” (compiling, high-improvement proofs) and “losers” (all others). Appendix C gives details.

IRPO Dataset We form two preference-pair types: winner–winner pairs ordered by metric improvement, and winner–loser pairs preferring valid improving proofs over failures. The construction and hyperparameters W, L appear in Algorithm 1.

4.3.2 IRPO (Iterative Reasoning Preference Optimization)

With this dataset $\mathcal{D}_{\text{IRPO}}^{(t)}$, we calculate the IRPO loss on each item T as the (weighted) sum of the DPO loss of the preference pair and the negative log-likelihood (NLL) loss over the winner:

$$\begin{aligned} \mathcal{L}_{\text{IRPO}}(T) = & \mathcal{L}_{\text{DPO}}(y_{T,\ell}, y_{T,w} \mid \Psi(c_T, x_T, y_{T,0}), \mu) \\ & + \alpha \mathcal{L}_{\text{NLL}}(y_{T,\ell}, y_{T,w} \mid \Psi(c_T, x_T, y_{T,0}), \mu) \end{aligned}$$

In the above, we represent the relevant neurosymbolic augmentation with the function Ψ . After training with respect to this objective for one epoch, we obtain G_{t+1} .

5 Experiments

5.1 Setup

We evaluate ImProver 2 on all three metrics using public research-level mathematics repositories, benchmarking against open-source and closed-source baselines at varying parameter counts.

Dataset and split We utilize Lean proofs from several open-source projects formalizing research-level mathematics across multiple domains [The Mathlib Community, 2020, Tooby-Smith, 2025, Tao, 2026, van Doorn et al., 2026, Wilshaw, 2026, Buzzard and Taylor, 2026, Saito and Noguchi, 2026, Sergeev et al., 2026]. We hold out all theorems in miniCTX-v2 as the test set. To prevent data leakage, we additionally exclude from training every theorem that appears in the same source file as a miniCTX-v2 theorem. Training and validation sets are drawn from all remaining files in an 80%/20% split, although we pre-filter the Mathlib portion of the dataset to a uniformly sampled subset of 37 files, due to its significantly larger scale.

We use miniCTX-v2 as a proxy for deployment on human-written research-level mathematics. Although ImProver 2 could in principle be applied to other domains, such as long machine-generated proofs, these settings fall outside the scope of our main evaluation. As a qualitative test, however, we also evaluate ImProver 2 on machine-generated AlphaProof proofs from the 2024 International Mathematical Olympiad in Appendix F.2.

Evaluation protocol All evaluations use Lean v4.17.0. For all main results, we evaluate with best@16 sampling (4.2) and report the improvement score $\mu(c, x, y) - \mu(c, x, y_0)$ for all problems in the test set and $\mu \in \{\mu_{\text{len}}, \mu_{\text{dep}}, \mu_{\text{mod}}\}$. With this we compute the mean of the improvement scores, as well as ancillary metrics such as the compilation accuracy $\mathcal{A}(\mathcal{D}_{\text{test}}^{(t)})$ and improved accuracy $\mathcal{A}_{\mu}^{+}(\mathcal{D}_{\text{test}}^{(t)})$, defined as the percentage of compiling theorems that have a strictly positive improvement

Table 1: **Frontier & Intra-family Evaluations.** Comparison with frontier models, intra-family baselines, and prior proof optimization systems. Mean improvement at best@16 on MiniCTX-v2. Input/output cost per 1M tokens also listed for API models.

Model	Length	Mod.	Dep.	Cost
DS-R1 7B	0.118	0.003	0.050	Local
DS-R1 14B	0.140	0.037	0.093	Local
DS-R1 671B	0.308	0.055	0.153	\$0.70/\$2.50
GPT-4o	0.336	0.034	0.050	\$2.50/\$10
GPT-oss-120B	0.321	0.075	0.181	Local
GPT-5-nano	0.087	0.065	0.106	\$0.05/\$0.40
GPT-5-mini	0.330	0.109	0.203	\$0.25/\$2
GPT-5-chat	0.346	0.118	0.046	\$1.25/\$10
GPT-5-high	0.660	0.120	0.208	\$1.25/\$10
ImProver	0.355	0.088	0.047	\$2.50/\$10
ImProver 2	0.330	0.143	0.206	Local

Table 2: **Per-iteration improvements.** Progression of mean improvement at best@16 on MiniCTX-v2 across IRPO training iterations across all three metrics.

Iteration	Length	Mod.	Dep.
Base	0.118	0.003	0.050
+ Scaffold	0.236	0.007	0.056
1	0.265	0.062	0.137
2	0.318	0.134	0.206
3	0.330	0.143	0.165
4	0.299	0.096	N/A

score. We operate with a base model of G_0 = DeepSeek-R1-Distill-Qwen-7B [DeepSeek-AI, 2025]. All models are evaluated, prompted – and for G_0 , trained – with the hyperparameters and configuration described in Appendix E.

Systems compared Our main evaluation compares ImProver 2 against three classes of baselines on MiniCTX-v2 under best@16 sampling: (i) the DeepSeek-R1 family at multiple scales, to study parameter scaling within a fixed model family; (ii) frontier GPT-based and open-weight systems, including GPT-5-high (a full-size high-reasoning variant), GPT-5-chat, GPT-5-mini, GPT-5-nano, and GPT-oss-120B; and (iii) the prior ImProver system and its base model GPT-4o. We also report current API inference costs in Table 1.

We also evaluate all generators with and without the neurosymbolic scaffold Ψ to isolate the effect of augmentation from the effect of training. For each metric, we train IRPO until validation improvement regresses and report the best checkpoint for that objective.

5.2 Main Results

Table 1 shows that ImProver 2 improves the DeepSeek-R1 7B base model on all three objectives, leads all evaluated unscaffolded systems on modularity, and is competitive with frontier models on dependency and length. Namely, after IRPO training, the model improves from 0.118 to 0.330 on length, 0.003 to 0.143 on modularity, and 0.050 to 0.206 on dependency. These gains also exceed the larger DeepSeek-R1 14B and DeepSeek-R1 671B baselines on every metric, suggesting that task-specific training can compensate for substantial generic scale within this model family.

Against frontier and prior systems, ImProver 2 is strongest on the structural metrics in the unscaffolded comparison. It leads all evaluated unscaffolded systems on modularity and is effectively tied with GPT-5-high on dependency (0.206 vs. 0.208). On length, it matches GPT-5-mini, exceeds unscaffolded GPT-oss-120B, and trails the high-reasoning GPT-5-high and prior multi-step GPT-4o-based ImProver system. When frontier models receive the same scaffold (Table 3), several outperform ImProver 2; we therefore interpret these results as evidence for effective specialization rather than dominance over the strongest scaffolded frontier systems.

We next unpack these aggregate results through three complementary analyses: intra-family parameter scaling, best@ n comparisons against frontier and prior systems, and the evolution of performance across IRPO iterations. We then isolate the contribution of the neurosymbolic scaffold across model families.

5.2.1 Performance and Parameter Scaling

A central question is whether proof optimization performance is primarily driven by scale or by task-specific specialization, when all else is equal. Figure 4 shows that within the DeepSeek-R1 model family, larger models generally achieve higher mean improvement under a fixed prompting and

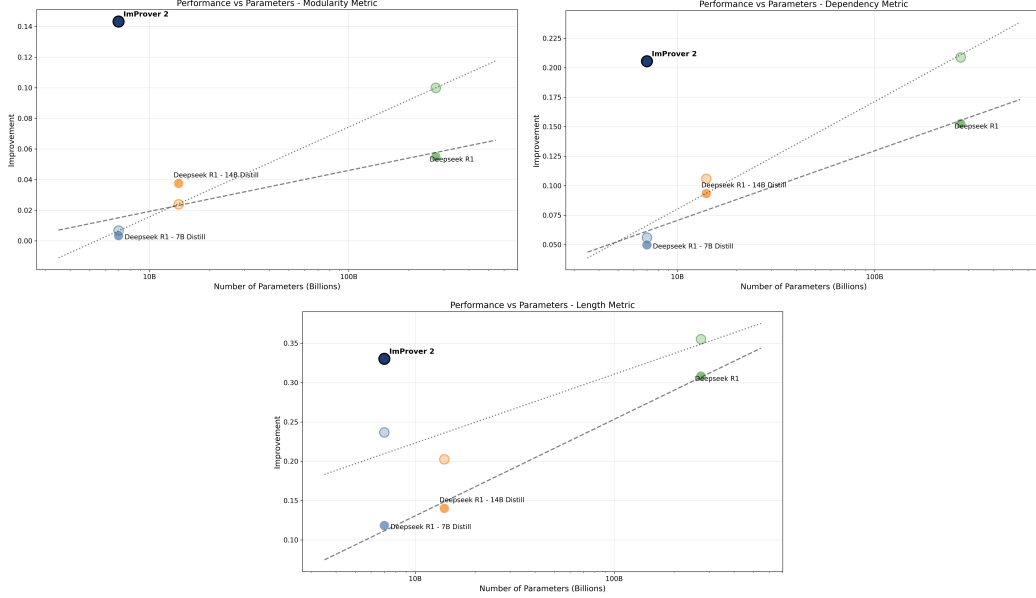


Figure 4: Effect of parameter count on model performance on mean improvement at best@16 across all three metrics, with ImProver 2 marked.

sampling protocol. This trend is clearest for length and dependency, indicating that generic reasoning capacity does help with proof refactoring.

At the same time, scale is not the full story. ImProver 2, trained from a 7B base model, outperforms the much larger 671B DeepSeek model on all three objectives. The gap is especially pronounced for modularity (0.143 vs. 0.055) and dependency (0.206 vs. 0.153), suggesting that for structural proof optimization, the relevant bottleneck is not only reasoning capacity but also whether the model has been adapted to the structure of the formal task.

This interpretation is reinforced by the accuracy metrics. Namely, within the same model family, larger general-purpose models often maintain stronger raw compilation rates, but ImProver 2 is more likely to produce compiling proofs that are also improved under the target metric.

We therefore interpret the scaling results as evidence that proof optimization is only partly a scale problem; it is also a specialization problem, and one for which iterative preference optimization is particularly effective.

5.2.2 Comparison to Frontier and Prior Systems

We compare ImProver 2 against frontier closed-source models, a large open-weight baseline, and the prior ImProver system. The resulting picture is mixed but informative: frontier systems are already strong proof rewriters, yet their strengths differ by objective, while ImProver 2 is most competitive on the structural metrics.

Among the frontier baselines evaluated without scaffolding, ImProver 2 achieves the strongest modularity score (0.143), showing that iterative task-specific training can produce structural rewrites that generalist frontier models do not always find under the same single-shot, unscaffolded protocol. On dependency, ImProver 2 (0.206) is competitive with the strongest frontier model evaluated, GPT-5-high (0.208), and leads GPT-5-mini (0.203); we treat all three as effectively tied at the level of mean performance. On length, ImProver 2 (0.330) matches GPT-5-mini and exceeds unscaffolded GPT-oss-120B (0.321), trailing only GPT-5-high (0.660), which represents a qualitatively different operating point in terms of inference cost.

Relative to large open-weight baselines, ImProver 2 is also strong in the unscaffolded comparison. It matches or exceeds unscaffolded GPT-oss-120B on all three metrics despite starting from a smaller base model. We note that GPT-oss-120B is a sparse mixture-of-experts model activating approximately 5B parameters per token, so the raw parameter count comparison overstates the

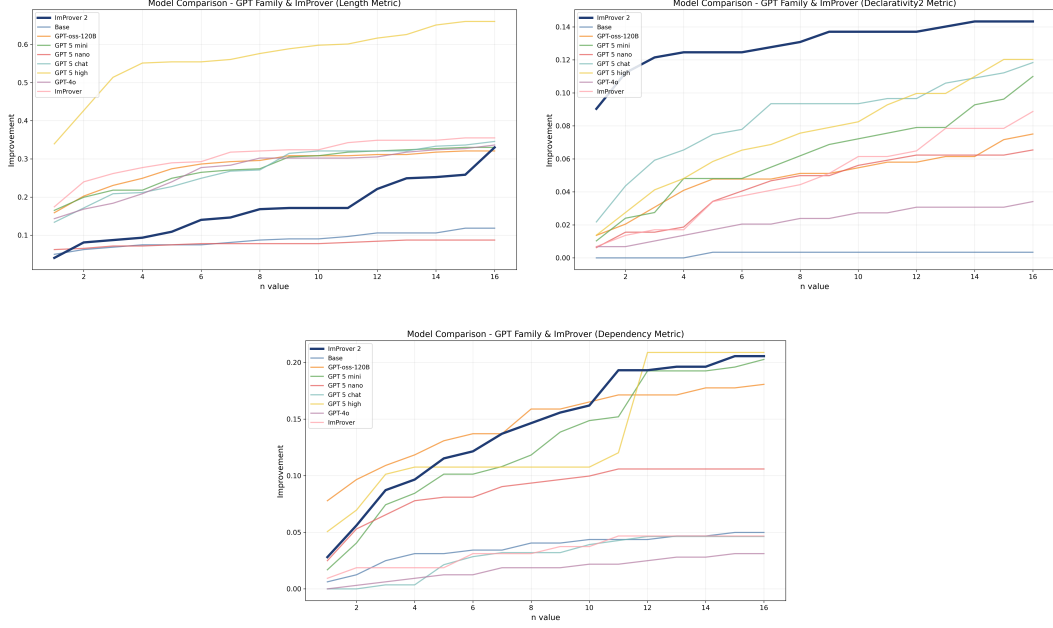


Figure 5: Comparison of ImProver 2 against frontier GPT-based models and ImProver, evaluated on mean improvement at best@ n on all metrics, from $n = 1$ to $n = 16$.

computational gap. Moreover, when GPT-oss-120B receives the same scaffold, it substantially outperforms ImProver 2 on length and dependency (Table 3). We therefore interpret the result as evidence that metric-specific iterative self-improvement can make a small dense model useful and competitive in some settings, not as evidence that it dominates the strongest scaffolded open-weight baselines.

The comparison with the prior ImProver system is also revealing. The original ImProver system leads ImProver 2 on length (0.355 vs. 0.330), consistent with its use of a multi-step prompting strategy with a strong proprietary base model. By contrast, ImProver 2 leads on modularity (0.143 vs. 0.088) and outperforms ImProver substantially on dependency (0.206 vs. 0.047), suggesting that iterative structural supervision can learn refactoring behaviors that are not reliably induced by prompting alone.

5.2.3 Per-Iteration Improvement

Table 2 shows that most improvement occurs within the first two or three IRPO rounds. Dependency peaks at iteration 2, while length and modularity peak at iteration 3, so we select checkpoints separately by metric. These gains suggest that the model is learning useful refactoring behavior from its own filtered generations, and that the replay-buffered preference data remains informative for multiple rounds. Indeed, as shown by the full best@ n curves (Figure 6), the trained checkpoints dominate the untrained and scaffold-only baselines across nearly all sample budgets, indicating that the gains are not caused by a single high-sample outlier.

Improvement is not indefinite, however, as later rounds plateau or regress. We interpret this as indicating saturation rather than instability, as once the most common and highest-yield refactoring patterns have been absorbed, later rounds appear to produce fewer genuinely novel improvements and increasingly focus on narrower or noisier examples.

5.2.4 Effect of Neurosymbolic Scaffolding

Table 3 isolates the effect of the scaffold Ψ from training. Namely, across nearly all evaluated generators and metrics, adding the scaffold Ψ improves mean best@16 performance, often by a large margin. For example, on length, we observe DeepSeek-R1 7B improves from 0.118 to 0.236,

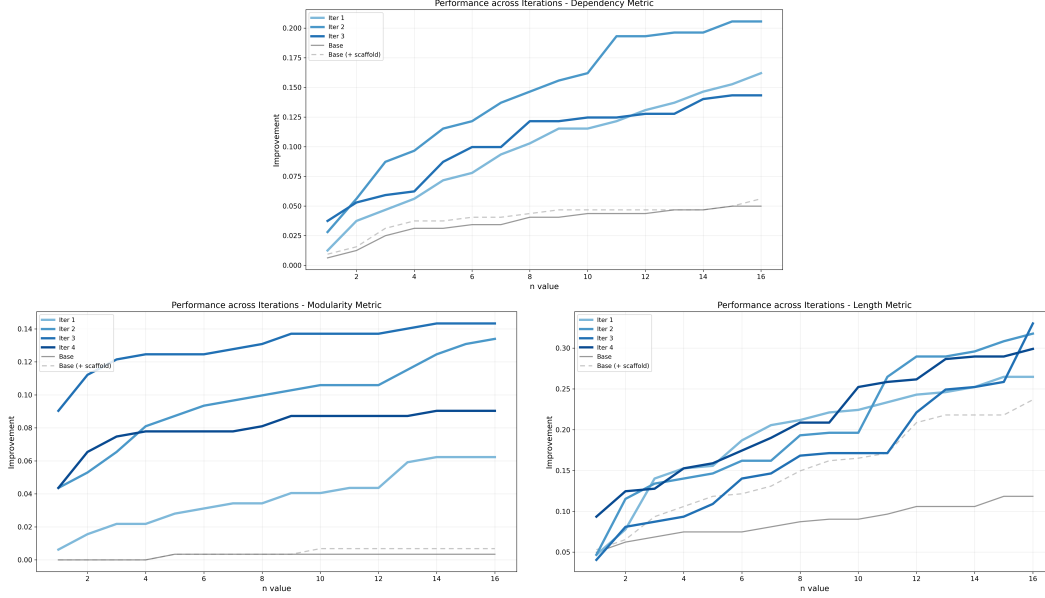


Figure 6: Performance of ImProver 2 as a function of sample budget across training iterations. Gains are largest in early iterations, with saturation by iterations 2–3 and mild regression thereafter.

Table 3: **Scaffold Evaluations.** Effect of neurosymbolic scaffolding. Mean improvement at best@16 with and without the scaffold Ψ , across model families and metrics.

Model	Length		Mod.		Dep.	
	Base	Scaffold	Base	Scaffold	Base	Scaffold
DeepSeek-R1 7B	0.118	0.236	0.003	0.007	0.050	0.056
DeepSeek-R1 14B	0.140	0.202	0.037	0.024	0.093	0.106
DeepSeek-R1 (671B)	0.308	0.355	0.055	0.100	0.153	0.209
GPT-4o	0.336	0.396	0.034	0.092	0.050	0.075
GPT-oss-120B	0.321	0.508	0.075	0.092	0.181	0.406
GPT-5-nano	0.087	0.296	0.065	0.069	0.106	0.108
GPT-5-mini	0.330	0.632	0.109	0.123	0.203	0.267
GPT-5-chat	0.346	0.576	0.118	0.153	0.046	0.087
GPT-5-high	0.660	0.875	0.120	0.183	0.208	0.315

GPT-5-mini from 0.330 to 0.632, and GPT-5-high from 0.660 to 0.875. Dependency also improves for most systems, while modularity gains are smaller but generally positive. This suggests that the scaffold is not merely compensating for weak base models; rather, it changes the representation of the task in a way that makes better rewrites easier to discover under a fixed sampling budget. Overall, the scaffold appears to expand the set of useful rewrites the model can reliably attempt. As such, by exposing goal-state information, relevant context, and informal abstractions, it helps both small and large models move beyond surface-level editing toward more substantive proof refactoring by improving models’ abilities to search over valid higher-value rewrites under a fixed sampling budget.

5.2.5 Additional Analyses

Improvement vs Accuracy Mean improvement alone does not distinguish between broad, reliable gains and a smaller set of high-reward successes. We therefore also report compilation accuracy \mathcal{A} and improved accuracy \mathcal{A}_μ^+ , where \mathcal{A}_μ^+ measures the fraction of test problems for which the model produces a compiling proof that has a strictly positive metric improvement score.

Across both models and training iterations, optimization tends to raise \mathcal{A}_μ^+ faster than \mathcal{A} . This is especially visible in the iteration study (Table 5). For dependency minimization, the base model begins with high compilation accuracy but low improved accuracy (0.754 and 0.037 respectively), indicating that it often produces valid rewrites without meaningfully reducing dependency footprint.

Table 4: **Accuracy Evaluations.** Compilation accuracy and improved accuracy comparison amongst the best@16 mean improvement samples across all three metrics. Each entry is reported as $\mathcal{A}_\mu^+/\mathcal{A}$, where \mathcal{A}_μ^+ is improved accuracy and \mathcal{A} is compilation accuracy.

Model	Length	Mod.	Dep.
DS-R1 7B	0.062 / 0.617	0.003 / 0.570	0.037 / 0.754
DS-R1 14B	0.075 / 0.660	0.034 / 0.775	0.065 / 0.567
DS-R1 671B	0.162 / 0.536	0.048 / 0.536	0.109 / 0.480
ImProver 2	0.131 / 0.657	0.065 / 0.579	0.069 / 0.368
ImProver	0.171 / 0.560	0.068 / 0.597	0.031 / 0.249
GPT-4o	0.158 / 0.595	0.031 / 0.464	0.028 / 0.227
GPT-oss	0.171 / 0.692	0.068 / 0.477	0.097 / 0.676
GPT-5-nano	0.044 / 0.461	0.053 / 0.757	0.065 / 0.894
GPT-5-mini	0.159 / 0.623	0.085 / 0.525	0.111 / 0.834
GPT-5-chat	0.165 / 0.586	0.084 / 0.455	0.025 / 0.185
GPT-5-high	0.264 / 0.757	0.092 / 0.656	0.094 / 0.753

Table 5: **Accuracy Progression.** Compilation accuracy and improved accuracy progression amongst the best@16 mean improvement samples across IRPO training iterations. Each entry is reported as $\mathcal{A}_\mu^+/\mathcal{A}$, where \mathcal{A}_μ^+ is improved accuracy and \mathcal{A} is compilation accuracy.

Model	Length	Mod.	Dep.
Base	0.062 / 0.617	0.003 / 0.570	0.037 / 0.754
Iter. 1	0.125 / 0.720	0.044 / 0.679	0.093 / 0.417
Iter. 2	0.131 / 0.679	0.121 / 0.614	0.106 / 0.464
Iter. 3	0.121 / 0.682	0.118 / 0.579	0.069 / 0.368
Iter. 4	0.131 / 0.657	0.065 / 0.579	N/A

After training, improved accuracy rises sharply, peaking at 0.106 in iteration 2, while compilation accuracy drops to 0.464. This reflects a central tradeoff of proof refactoring: larger structural edits are more likely to yield real gains when they succeed, but they also create more opportunities for compilation failure.

The same pattern appears in cross-model comparisons. For example, GPT-5-nano attains very high compilation accuracy on dependency (0.894) but only moderate improved accuracy (0.065), whereas ImProver 2 attains lower compilation accuracy (0.368) but slightly higher improved accuracy (0.069). This suggests that conservative models often preserve correctness by making safer edits, while specialized optimizers are more willing to attempt riskier transformations that improve the target metric when successful. Accordingly, we view \mathcal{A} and \mathcal{A}_μ^+ as complementary: \mathcal{A} measures stability, while \mathcal{A}_μ^+ more directly captures whether the system is solving the optimization problem rather than merely preserving compilability.

Per-repository heterogeneity Performance varies substantially across repositories (Table 6), suggesting that optimization opportunity may be mediated by project-specific proof style, theorem difficulty distributions, and domain. For example, Mathlib exhibits very small length gains but relatively strong dependency and modularity improvements, consistent with a library whose proofs are often already concise but still admit structural refactoring. By contrast, HepLean and ConNF show much larger length and dependency gains, suggesting that these corpora contain more opportunities for proof compression and simplification. We treat these repository-level results as descriptive rather

Table 6: **Average improvement by project and metric.**

Project	Length	Dependency	Modularity
HepLean	1.283	0.379	0.030
ConNF	0.420	0.278	0.048
Seymour	0.199	0.100	0.359
FLT	0.131	0.133	0.066
Foundation	0.082	0.015	0.219
Carleson	0.048	0.188	0.095
Mathlib	0.016	0.306	0.163

than definitive, since they likely reflect both stylistic differences and variation in theorem composition across projects.

5.3 Overview of Additional Experiments

Ablation Studies Appendix F.1 separates the scaffold channels and reports the hyperparameter searches used during training. The ablation shows that goal-state traces, informalizations, and retrieved context each help (albeit with a majority of improvement stemming from the chain-of-states annotations), while the grid searches motivate per-iteration and per-metric hyperparameter selection and evolution.

Qualitative Examples Appendix F.2 gives representative optimized proofs from the MiniCTX-v2 dataset for all three objectives, including provenance and old/new metric scores. Moreover, we additionally include a case study of evaluation on AI-generated proofs (Alphaproof on IMO 2024 [Hubert et al., 2025]) with a higher compute budget.

6 Limitations and Future Work

In this work, we posit and study a set of particular structural metrics, which may not necessarily align with corpus maintainers’ subjective preferences of proof quality. Namely, the dependency and modularity metrics measure the number of explicitly named dependencies and effective spawned goals respectively, which are structural formal objects and therefore may carry discrepancies between raw dependency scores and maintainer preferences. We do not include a maintainer preference study, and future work may wish to address this gap by engineering maintainer preferences directly (perhaps through informal, LLM-based metrics). Additionally, future work may also wish to study the effects of training dataset optimization on downstream prover performance.

Additionally, we study single-step rewriting rather than full agentic systems. The scaffold can be exposed as a tool, but we do not evaluate Codex- or Claude-Code-style agents in this particular work as this work primarily focuses on the development and evaluation of our training pipeline and scaffold, and as such, we prioritize the robust evaluation of single-step rewriting. However, we do find that stronger optimization often lowers compilation accuracy 5.2.5, suggesting that such agents and agentic iterative repair loops are potential methods to balance this trade-off.

7 Conclusion

We have introduced ImProver 2, a pipeline for boosting the formal proof-optimization ability of small language models. We have demonstrated the utility of our neurosymbolic augmentations to many varieties of models, showcased iterative self-improvement using our replay buffer architecture, and introduced two novel metrics (as well as revisiting an old one) for practical proof improvement. We have found that our system leads all evaluated unscaffolded systems on proof modularity, is competitive with the strongest frontier models on dependency, and matches mid-tier frontier models on length — all from a 7B base model.

Acknowledgements

This research is partially supported by the DARPA expMath program through the DARPA CMO contract number HR0011262E028 and NSF Grant DMS-2434614. We would like to thank Dr. Patrick Shafto, expMath Program Manager, for useful technical discussions.

References

- Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathis Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladimir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. Aristotle: Imo-level automated theorem proving, 2025. URL <https://arxiv.org/abs/2510.01346>.
- Riyaz Ahuja, Jeremy Avigad, Prasad Tetali, and Sean Welleck. Improver: Agent-based automated proof optimization. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, editors, *International Conference on Representation Learning*, volume 2025, pages 29521–29543, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/4864005cfdea7ebd07086ed1b9846825-Paper-Conference.pdf.
- Kevin Buzzard and Richard Taylor. Fermat’s last theorem, 2026. URL <https://imperialcollegelondon.github.io/FLT/blueprint.pdf>.
- Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, Zhihong Wang, Mingxuan Wang, Chenrui Wei, Shufa Wei, Huajian Xin, Fan Yang, Weihao Gao, Zheng Yuan, Tianyang Zhan, Zeyu Zheng, Tianxi Zhou, and Thomas Hanwen Zhu. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience, 2025. URL <https://arxiv.org/abs/2512.17260>.
- DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL <https://arxiv.org/abs/2501.12948>.
- Simon Frieder, Jonas Bayer, Sam Looi, Jacob Loader, Julius Berner, Katherine M. Collins, András Juhász, Fabian Ruehle, Sean Welleck, Gabriel Poesia, Ryan-Rhys Griffiths, Adrian Weller, Anirudh Goyal, Cameron Freer, Thomas Lukasiewicz, and Timothy Gowers. Data for mathematical copilots: Better ways of presenting proofs for machine learning, 2025. URL <https://arxiv.org/abs/2412.15184>.
- Alex Gu, Bartosz Piotrowski, Fabian Gloeckle, Kaiyu Yang, and Aram H. Markosyan. Proofoptimizer: Training language models to simplify proofs without human demonstrations. In *The 5th Workshop on Mathematical Reasoning and AI at NeurIPS 2025*, 2025. URL <https://openreview.net/forum?id=ghxS7M35FU>.
- Seiji Hattori, Takuya Matsuzaki, and Makoto Fujiwara. Natural language translation of formal proofs through informalization of proof steps and recursive summarization along proof structure. In Lucie Flek, Shashi Narayan, Lê Hồng Phuong, and Jiahuan Pei, editors, *Proceedings of the 18th International Natural Language Generation Conference*, pages 376–389, Hanoi, Vietnam, October 2025. Association for Computational Linguistics. URL <https://aclanthology.org/2025.inlg-main.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 Sonne, 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*, 2025. doi: 10.1038/s41586-025-09833-y. URL <https://www.nature.com/articles/s41586-025-09833-y>. Published: 12 November 2025.
- Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, Sketch, and Prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations*, 2023. URL <https://doi.org/10.48550/arXiv.2210.12283>.
- Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. A survey on deep learning for theorem proving. In *First Conference on Language Modeling*, 2024. URL <https://openreview.net/forum?id=zlw6AHwukB>.

- 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, 2025. URL <https://arxiv.org/abs/2508.03613>.
- Pan Lu, Liang Qiu, Wenhao Yu, Sean Welleck, and Kai-Wei Chang. A survey of deep learning for mathematical reasoning. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14605–14631, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.817. URL <https://aclanthology.org/2023.acl-long.817/>.
- Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5.
- Richard Yuanzhe Pang, Weizhe Yuan, Kyunghyun Cho, He He, Sainbayar Sukhbaatar, and Jason Weston. Iterative reasoning preference optimization, 2024. URL <https://arxiv.org/abs/2404.19733>.
- Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020. URL <https://arxiv.org/abs/2009.03393>.
- Shogo Saito and Mashu Noguchi. Foundation, 2026. URL <https://github.com/FormalizedFormalLogic/Foundation>.
- Ivan Sergeev, Martin Dvorak, Tristan Figuerola-Reid, Rida Hamadani, Byung-Hak Hwang, Evgenia Karunus, Vladimir Kolmogorov, Alex Meiburg, Peter Nelson, and Mark Sandey. Regularity of 1-, 2-, and 3-sums of matroids, 2026. URL <https://ivan-sergeyev.github.io/seymour/blueprint.pdf>.
- Ilya Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. Ai models collapse when trained on recursively generated data. *Nature*, page 755–759, 2024. URL <https://doi.org/10.1038/s41586-024-07566-y>.
- Terrance Tao. Pfr blueprint, 2026. URL <https://teorth.github.io/pfr/blueprint.pdf>.
- The Coq Development Team. The coq proof assistant, September 2024. URL <https://doi.org/10.5281/zenodo.14542673>.
- The Mathlib Community. 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, 2020. Association for Computing Machinery. ISBN 9781450370974. doi: 10.1145/3372885.3373824. URL <https://doi.org/10.1145/3372885.3373824>.
- Joseph Tooby-Smith. Hephlean: Digitalising high energy physics. *Computer Physics Communications*, 308:109457, 2025. ISSN 0010-4655. doi: <https://doi.org/10.1016/j.cpc.2024.109457>. URL <https://www.sciencedirect.com/science/article/pii/S0010465524003801>.
- Floris van Doorn, Michael Rothgang, Pietro Monticone, Jeremy Tan Jie Rui, James Sundstrom, María Inés de Frutos-Fernández, Ruben Van de Velde, Sebastien Gouezel, Leo Diederling, Jim Portegies, and Joris Roos. Formalizing carleson’s theorem in lean, 2026. URL <https://florisvandoorn.com/carleson/>.
- Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The isabelle framework. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 33–38, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-71067-7.
- Sky Wilshaw. New foundations is consistent, 2026. URL <https://leanprover-community.github.io/con-nf/print/print.pdf>.
- Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*, 2023. URL <https://icml.cc/virtual/2023/27190>.

A Formal Definition of the Modularity Metric

This appendix presents a precise, proof-theoretic and type-theoretic definition of the modularity metric μ_{mod} used throughout the paper, matching its concrete implementation in Lean 4. We proceed from first principles, beginning with Lean’s elaboration semantics, and culminate in the exact algorithm used to count effective spawned goals.

A.1 Lean Elaboration Semantics

Lean elaborates tactic proofs by incrementally constructing and solving metavariables. At any point in elaboration, the system maintains a metavariable context $M = (\Delta, \sigma)$ where Δ is a finite set of metavariable declarations of the form $?m : (\Gamma_{?m} \vdash A_{?m})$, with local context $\Gamma_{?m}$ and target type $A_{?m}$. Similarly, σ is a partial assignment mapping metavariables to terms.

A goal is an unassigned metavariable $?m \in \Delta \setminus \text{dom}(\sigma)$. At elaboration step i , Lean maintains a list of *active goals* \mathcal{A}_i .

Each tactic step τ_i :

1. Focuses a goal $?m_i \in \mathcal{A}_i$,
2. Produces an assignment $\sigma_{i+1}(?m_i) = t_i$,
3. Potentially introduces new metavariables corresponding to subgoals.

The set of newly created metavariables is exactly the set of free metavariables occurring in t_i after assignment.

A.2 Direct Children vs. Spawned Goals

Let $\text{Children}(\tau_i)$ denote the set of metavariables that occur free in t_i and therefore represent *direct subgoals* of the focused goal $?m_i$.

However, Lean tactics may surface additional obligations that are *not* direct children of $?m_i$. These arise from nested tactic blocks (e.g. `have`, `calc`, `by`), automation introducing auxiliary lemmas, goal defocus/refocus patterns, tactics that internally elaborate subproofs, etc.

To capture this distinction, define:

$$\text{Current}_i = \{?m_i\} \cup \text{Children}(\tau_i),$$

and let S_i be the set of all metavariables that have previously appeared as children: $S_{i+1} = S_i \cup \text{Children}(\tau_i)$.

Definition 1 (Spawned Goals). *The spawned goals of step τ_i are defined as:*

$$\text{Spawned}(\tau_i) = (\text{Current}_i \setminus \text{Children}(\tau_i)) \setminus S_i.$$

Intuitively, these are goals that first appear at step i , are not logical subgoals of the focused goal, correspond to independent subproof obligations.

A.3 Proof Graph over Steps

We represent a proof as a directed graph over steps rather than metavariables.

Let steps be indexed by $i = 1, \dots, T$. We define:

- A *normal edge* $i \rightarrow j$ if the goal solved at step j is a direct child of step i .
- A *spawned edge* $i \rightsquigarrow j$ if the goal solved at step j was spawned at step i .

This yields a labeled directed forest

$$(V, E_{\text{normal}}, E_{\text{spawned}}),$$

with $E_{\text{spawned}} \subseteq E_{\text{normal}}$. We refer to this as the proof tree.

A.4 Canonical Goal Representation

To prevent adversarial or spurious inflation of modularity, goals are compared modulo definitional equality, α -equivalence, and universal abstraction.

Each goal is assigned a canonical representation consisting of:

- A base target hash,
- A sorted list of hypothesis type hashes (proof-relevant hypotheses only),
- A set of sequent variant hashes, obtained by progressively discharging \forall -binders and implications,
- A set of target-only variant hashes, used for wrapper detection.

All hashes are computed after $\beta\delta\iota$ -normalization, metadata erasure, binder name scrubbing (for α -invariance), and replacement of free variables by deterministic canonical constants.

This representation ensures that goals differing only by irrelevant syntactic structure or parameter order are identified.

A.5 Duplicate and Wrapper Detection

A spawned goal is considered duplicate and discarded if either:

1. Its sequent variant hash matches any previously seen goal (global duplicate), or
2. Its target is a definitional wrapper of its parent goal, i.e. $\text{target}(g_{\text{child}}) \in \text{targetVariants}(g_{\text{parent}})$ or vice versa.

This eliminates the potential for many types of reward hacking cases, such as trivial \forall -introductions, restatements of the parent goal, and redundant lemma spawning.

A.6 Nontriviality Filter

Let T_g be the subtree of steps rooted at a spawned goal g .

We require $|T_g| > 2$, ensuring that the goal is not solved immediately, and namely, the goal is not discharged by a single automation step.

This empirically excludes trivial goals solvable by tactics such as `simp`, `tauto`, `linarith`, `ring`, `aesop`, or `grind`.

A.7 Effectiveness via Fixed-Point Semantics

Let each spawned goal g introduce a set of proof variables $\text{Intro}(g)$ corresponding to hypotheses unavailable in the parent context.

We define a spawned subtree rooted at g to be effective if its introduced hypotheses are used in the main (non-spawned) proof, or in another spawned subtree that is itself effective.

Formally, define a monotone operator Φ on sets of spawned roots:

$$\Phi(S) = \{g \mid \text{Intro}(g) \text{ is used outside all spawned subtrees} \\ \vee \exists g' \in S, g' \neq g, \text{Intro}(g) \text{ used in subtree } g'\}.$$

Definition 2 (Effective Spawned Goals). *The set of effective spawned goals is the least fixed point of Φ .*

This is computed by standard fixed-point iteration, guaranteed to terminate since the set of spawned roots is finite.

A.8 Modularity Metric

Definition 3 (Modularity Metric). *Given a verified proof y of (c, x) , let $\mathcal{E}(y)$ be the set of effective spawned goals. The modularity metric is defined as:*

$$\mu_{\text{mod}}(c, x, y) = |\mathcal{E}(y)|.$$

B Context Extraction

To facilitate extraction of relevant context from the proof environment, we consider the Lean 4 concrete syntax tree (CST) and abstract syntax tree (AST). The CST preserves surface tokens and references locations in source code; the AST resolves names, binds variables, and canonicalizes declarations (e.g., definitions/theorems/structure/etc. nodes). Context extraction uses both:

- CST view (textual reachability): harvest all surface identifiers and their source spans that occur in x and in the proof text of y_0 .
- AST view (semantic reachability): resolve those identifiers to fully qualified symbols under the language’s environment (imports, namespaces, instances, modules); record declaration categories (e.g., definition, theorem) and provenance (module/package).

Graphs and the slice. From the AST we build two standard graphs: (i) an import DAG over files/modules, and (ii) an entity graph whose vertices are declarations (constants, lemmas, definitions) with edges for semantic references (uses in types/bodies). Let $\text{touch}(x, y_0)$ be the set of AST nodes directly referenced by the CST identifiers collected from x and y_0 (optionally augmented by a dynamic trace of proof states, if available). The context slice is the subgraph

$$S(c, x, y_0) = \text{Reach}(G_{\text{ent}}, (\text{touch}(x, y_0)))$$

optionally restricted by the import DAG to a budgeted neighborhood. We then serialize each element of S with metadata as to its object type (e.g. lemma, definition, etc.), as well as a stable snippet of its source content.

Because selection is driven by AST resolution, Ψ_{ctx} is invariant to superficial edits (whitespace, formatting) and robust to local refactors (α -renaming within a module). The slice size grows with the reachable subgraph, not raw file size, yielding a compact, minimal, and deterministic bundle of proof context that is read-only with respect to the program state.

C Replay Buffer and Dataset Construction Details

We provide a more detailed overview of the replay buffer algorithm. Given the problem set $\mathcal{P} = \mathcal{P}_{\text{train}} \cup \mathcal{P}_{\text{test}}$ and the current iteration t model G_t , we run generation as described in §4.2 to obtain n candidate proofs per problem in $\mathcal{P}_{\text{train}}$. This yields the raw (no-replay) dataset:

$$\begin{aligned} \mathcal{D}_{\text{nr}}^{(t)} &= \{(c_T, x_T, y_{T,0}, \{(y_{T,i}, \widehat{s}_{T,i})\}_{i=1}^n)\}_{T \in \mathcal{P}_{\text{train}}} \\ &= \{(c_T, x_T, y_{T,0}, \mathcal{Y}_T^{(t)})\}_{T \in \mathcal{P}_{\text{train}}} \end{aligned}$$

where $\widehat{s}_{T,i} = S_\mu(y_{T,i}, y_{T,0} \| c_T, x_T)$ is the improvement score of candidate $y_{T,i}$ over the baseline $y_{T,0}$.

We then construct the solutions (replay) dataset $\mathcal{D}_{\text{re}}^{(t)}$ by post-processing $\mathcal{D}_{\text{nr}}^{(t)}$ together with the previous iteration’s already post-processed dataset $\mathcal{D}_{\text{re}}^{(t-1)}$. The procedure is parameterized by a target replay proportion $\rho \in [0, 1]$, replay mode $\text{mode} \in \{\text{mark}, \text{join}, \text{replace}\}$, an improvement-rate cap $\pi_{\text{max}} \in [0, 1]$, and a minimum gap $\gamma \in [0, 1]$.

Improvement rate and eligibility. Additionally, for problem T , define its improvement rate at iteration j by

$$\pi_T^{(j)} = \frac{1}{n} \sum_{i=1}^n \mathbb{I}[\mathbf{v}(c_T, x_T, y_{T,i}^{(j)}) = 1 \wedge \widehat{s}_{T,i}^{(j)} > 0].$$

A problem is replay-eligible at iteration t iff it had at least one improved, compiling solution in some previous iteration $j < t$. We maintain a reservoir \mathcal{E} of replay-eligible problems with preference for “easy” items (largest $\pi_T^{(j)}$ across $j < t$).

Replay buffer construction We proceed now to construct the post-replay buffer dataset in two main steps:

1. *Mark*: We first mark problems in $\mathcal{D}_{\text{nr}}^{(t)}$ as REPLAY or FRONTIER so that the fraction of replay equals ρ :
 - (a) Start with the subset of problems that are replay-eligible; if their fraction exceeds ρ , downsample them by removing highest- π_T items (i.e. the “easiest”) until the replay fraction is ρ (cap at π_{\max}).
 - (b) If their fraction is below ρ , replace uniformly chosen frontier items by items sampled from the reservoir \mathcal{E} (taken from $\mathcal{D}_{\text{re}}^{(t-1)}$) until the replay fraction reaches ρ as best as possible. This keeps dataset size fixed while achieving the target mix.

After this step, every item in the dataset is tagged as REPLAY or FRONTIER and the target mix is satisfied.

2. *Merge*

For each item marked REPLAY with key $(c_T, x_T, y_{T,0})$, find its counterpart in $\mathcal{D}_{\text{re}}^{(t-1)}$, say with candidate (multi)set $\tilde{\mathcal{Y}}_T^{(t-1)}$. Then, we case on mode as follows:

- *join*: set the current candidates to the union $\tilde{\mathcal{Y}}_T^{(t)} = \mathcal{Y}_T^{(t)} \cup \tilde{\mathcal{Y}}_T^{(t-1)}$ (deduplicated by normalized proof text). This increases candidate diversity for IRPO.
- *replace*: set $\tilde{\mathcal{Y}}_T^{(t)} = \tilde{\mathcal{Y}}_T^{(t-1)}$, i.e., overwrite the current candidates by the previous iteration’s.

If mode is mark, skip this step (no candidate-level splice). In other words, set $\tilde{\mathcal{Y}}_T^{(t)} = \mathcal{Y}_T^{(t)}$

With this, we obtain the replay dataset:

$$\mathcal{D}_{\text{re}}^{(t)} = \left\{ (c_T, x_T, y_{T,0}, \tilde{\mathcal{Y}}_T^{(t)}) \mid T \in \mathcal{P}_{\text{train}} \right\}$$

Filtering Finally, we post-process $\mathcal{D}_{\text{re}}^{(t)}$ by filtering out low-quality candidates and separating the samples into winner (W) and loser (L) sets. Specifically, we filter the high improvement rate problems (those which were sufficiently “easy” for the model to improve on many candidates) by removing problems T with $\pi_T^{(t)} > \pi_{\max}$. Namely, we define $\tilde{\mathcal{P}}^{(t)} = \mathcal{P}_{\text{train}} \setminus \{T \mid \pi_T^{(t)} > \pi_{\max}\}$ to be the filtered problem set.

Then, for each problem T in $\tilde{\mathcal{P}}^{(t)}$, we partition $\tilde{\mathcal{Y}}_T^{(t)} = W_T^{(t)} \cup L_T^{(t)}$ such that:

- $W_T^{(t)} = \{y \in \tilde{\mathcal{Y}}_T^{(t)} \mid v(c_T, x_T, y) = 1 \wedge \hat{s}_{T,y} > \delta^{(t)}\}$, where $\delta^{(t)}$ is the γ -th percentile of all scores $\{\hat{s}_{T,i} : T \in \tilde{\mathcal{P}}^{(t)}, y_i \in \tilde{\mathcal{Y}}_T^{(t)}\}$.
- $L_T^{(t)} = \tilde{\mathcal{Y}}_T^{(t)} \setminus W_T^{(t)}$.

In other words, winners are those candidates that compile and have an improvement score above the γ -th percentile threshold across all samples in the dataset, while losers are the rest.

With this, we finalize the filtered post-replay buffer dataset as:

$$\mathcal{D}_{\text{fil}}^{(t)} = \left\{ (c_T, x_T, y_{T,0}, W_T^{(t)}, L_T^{(t)}) \mid T \in \tilde{\mathcal{P}}^{(t)} \right\}$$

IRPO Dataset As IRPO is a preference-based training method, we now convert $\mathcal{D}_{\text{fil}}^{(t)}$ into a set of preference pairs. This process is parameterized by two hyperparameters $W, L \in \mathbb{N}$, which control the number of winners and losers to sample per problem, respectively.

First, for a given problem T in $\tilde{\mathcal{P}}^{(t)}$, we deduplicate $W_T^{(t)}$ by equal scores $\hat{s}_{T,i} = \hat{s}_{T,j}$ and deduplicate $L_T^{(t)}$ by string equality.

With this, we form two families of preference pairs:

$$\begin{aligned} \mathfrak{P}_T^{\ell \rightarrow w} &= \{(b, g) : b \in L_T^{(t)}, g \in W_T^{(t)}\} \\ \mathfrak{P}_T^{w \rightarrow w} &= \{(g', g) : g, g' \in W_T^{(t)}, \hat{s}_{T,g} > \hat{s}_{T,g'}\}. \end{aligned}$$

And with this, we form our IRPO training dataset as the following set:

$$\mathcal{D}_{\text{IRPO}}^{(t)} = \bigcup_{T \in \tilde{\mathcal{P}}^{(t)}} (\mathfrak{P}_T^{\ell \rightarrow w} \cup \mathfrak{P}_T^{w \rightarrow w})$$

Which can be reinterpreted elementwise as a collection of tuples $(c_T, x_T, y_{T,0}, y_{T,\ell}, y_{T,w})$.

Algorithm 1 Preference Pair Creation

Input: filtered dataset $\mathcal{D}_{\text{fil}}^{(t)}$, hyperparameters $W \in \mathbb{N}$ and $L \in \mathbb{N}$
Initialize $\mathcal{D}_{\text{IRPO}}^{(t)} = []$
for T in $\mathcal{D}_{\text{fil}}^{(t)}$ **do**
 De-duplicate $W_T^{(t)}$ by total improvement score
 De-duplicate $L_T^{(t)}$ by string equality
 Discard or duplicate elements uniformly at random until $|W_T^{(t)}| = W$ and $|L_T^{(t)}| = L$
 for w_1 in $W_T^{(t)}$ **do**
 for w_2 in $W_T^{(t)}$ **do**
 if $\mu(c, x, w_1) > \mu(c, x, w_2)$ **then**
 $\mathcal{D}_{\text{IRPO}}^{(t)} := (w_1, w_2) : \mathcal{D}_{\text{IRPO}}^{(t)}$
 end if
 end for
 end for
 for w in $W_T^{(t)}$ **do**
 for l in $L_T^{(t)}$ **do**
 $\mathcal{D}_{\text{IRPO}}^{(t)} := (w, l) : \mathcal{D}_{\text{IRPO}}^{(t)}$
 end for
 end for
end for

D Chain-of-States Implementation

Given a verified proof y_0 , let its tactic steps be indexed by $i = 1, \dots, T$. From the InfoTree we obtain for each step $(\text{goalsBefore}_i, \text{goalsAfter}_i)$ and a pretty-printed tactic. We define

$$\Psi_{\text{cos}}(y_0) = \langle (\text{goalsBefore}_i, \tau_i, \text{goalsAfter}_i) \rangle_{i=1}^T,$$

and serialize each triple as a short snippet where goal lists are pretty-printed into comments adjacent to τ_i . This turns hidden kernel states into explicit cues the model can condition on.

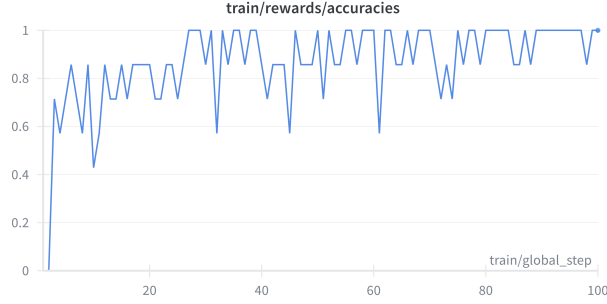
E System Configuration

E.1 Compute Resources

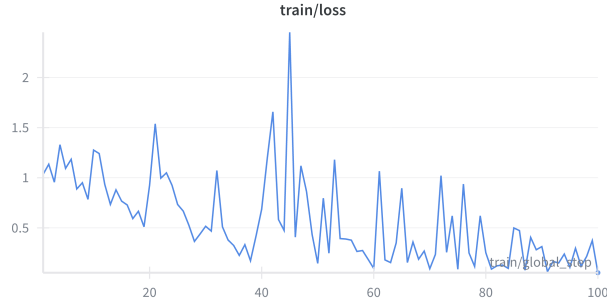
All local model training and evaluation runs were performed on a machine with 8 NVIDIA L40S GPUs. Local evaluations used this same GPU pool for batched best@ n generation and Lean verification. Closed-source and hosted API baselines were accessed through OpenRouter; for these models, compute was provided by the API provider rather than by our local hardware. We report the inference token costs used for the API baselines in Table 1.

E.2 Training

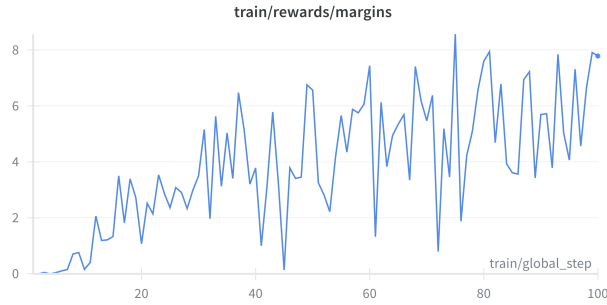
Training uses the same fixed systems configuration across metrics and IRPO rounds, while the main optimization hyperparameters are selected separately for each metric and iteration by validation grid search. In particular, the preference gap, replay-buffer ratio, learning rate, and related selection thresholds are tuned per round; the resulting searches are reported in Figures 9 and 10. Unless otherwise stated, each training run uses the following fixed configuration:



(a) Training accuracy over time on iteration 3 of the dependency metric.



(b) Training loss over time on iteration 3 of the dependency metric. Loss exhibits an overall downward trend.



(c) Average margin over time during training on iteration 3 of the dependency metric.

Figure 7: Training statistics during iteration 3 of the dependency metric. Model shows stable performance despite overall regression on this iteration.

- micro-batch size: 1 example per GPU, with gradient accumulation 1, for an effective batch size of 8 examples across 8 GPUs;
- epochs: 1;
- optimizer: `adamw_torch`;
- learning-rate scheduler: cosine, with 5 warmup steps;
- weight decay: 0.0 and maximum gradient norm: 1.0;
- precision and memory settings: bf16, FlashAttention-2, gradient checkpointing, and DeepSpeed ZeRO-3 CPU offload.

Figure 7a–7c shows representative training traces for the third dependency-optimization round.

E.3 System Prompts

During generation, ImProver 2 is prompted by combining the following prompts: depending on the metric, we take one of the **length**, **modularity**, or **dependency** prompts, and append the **annotation**, **context**, and **examples** prompts along with the relevant neurosymbolic augmentation in the correct locations.

Length: You are an expert Lean4 theorem rewriting assistant. Shorten the current Lean4 theorem (wrapped in `<CURRENT>...</CURRENT>`) to be as short as possible in length - measured in the number of tactics in the proof - while also ensuring that the output is still a correct proof of the theorem. Be sure to output your final response as a Lean4 theorem wrapped in `<IMPROVED>...</IMPROVED>` tags, as shown in the example. Namely, only return the statement and proof of the current theorem in Lean4 code, wrapped in `<IMPROVED>...</IMPROVED>` tags. Do not include any other text or comments.

Modularity: You are an expert Lean4 theorem rewriting assistant. Given a Lean4 theorem enclosed in `<CURRENT>...</CURRENT>` tags, rewrite the theorem to be as modular and declarative as possible. Modularity is defined by the number of independent, meaningful subproofs, measured by the occurrence of tactics that spawn new goals (such as 'have' statements, case splits, automation tactics, or 'calc' blocks) - with the caveat that these subproofs must be nontrivial and contribute to the overall proof structure. That means any sort of duplicate, unused, or trivial spawned goals will be ignored and/or penalized; this includes trivial modifications to spawned goals such as changing binders into forall statements. etc. Your objective is to maximize the number of these useful, nontrivial, and interesting spawned subproofs, while ensuring that the theorem remains correct and is clearly structured and readable. Optimize and rewrite the proof structure based on genuine sub-arguments, not superficial goal spawning. Validate that the revised theorem remains correct and that improvements in modularity are nontrivial and significant. In your output, only provide the improved statement and proof, wrapped in `<IMPROVED>...</IMPROVED>` tags, with no additional text or comments. Under no circumstances should you create artificial or superficial modularity in order to optimize for or maximize a reward metric; prioritize exclusively genuine mathematical quality and proof clarity over any gamified optimization.

Dependency: You are an expert Lean4 theorem rewriting assistant. Rewrite the current Lean4 theorem (wrapped in `<CURRENT>...</CURRENT>`) to be as independent of external theorems and lemmas as possible. Namely, you aim to rewrite the proof to minimize the number of external dependencies - while also ensuring that the output is still a correct proof of the theorem. Be sure to output your final response as a Lean4 theorem wrapped in `<IMPROVED>...</IMPROVED>` tags, as shown in the example. Namely, only return the statement and proof of the current theorem in Lean4 code, wrapped in `<IMPROVED>...</IMPROVED>` tags. Do not include any other text or comments.

Annotation: A version of the current theorem with the goal states annotated has also been provided for reference (wrapped in `<ANNOTATED>...</ANNOTATED>`). Namely, the goal states have been interleaved between tactics as comments to help you better understand the proof and ensure the correctness of your response. Do not include such state comments in your final response.

Context: The proof context, with relevant definitions and theorems, has additionally been provided to help you better understand the proof and ensure the correctness of your response. It is wrapped in `<CONTEXT>...</CONTEXT>`, with each item wrapped in `<ITEM>...</ITEM>`.

Examples: Here are some examples of such optimization, as wrapped in `<EXAMPLES>...</EXAMPLES>`. Note that these examples are for illustrative purposes only and should not be copied directly. Instead, use them to understand the kind of optimization expected and apply similar techniques to the current theorem, using these positive examples as an intuition and guidance on what kinds of optimizations you may do on your current target theorems. Additionally, you will also be provided with a negative example which will be marked as such. Use it to understand common pitfalls and avoid them in your response. Use both the positive and negative examples to guide your optimization of the current theorem.

E.4 Autoinformalization

During the generation round at iteration $t = 0$, we create informal statements of each theorem for use in neurosymbolic augmentation. This is carried out by prompting the base model G_0 with the following:

You are an expert informalizer of formal mathematics to natural language. Namely, given a formal theorem and proof in Lean4, you will generate an informalized statement of this same theorem in natural language, as well as (2) an informalized, natural language version of the same formal proof that is aligned with the informal statement. Namely, when informalizing the proof, you should convert each tactic of the formal proof into a natural language step in the informal proof, and thereby, your informal proof should be written as a sequence of steps. Consider the following example:

(examples omitted)

Now, with these examples in mind, it is now your turn to informalize the following formal statement and proof, which is wrapped in `<FORMAL>...</FORMAL>` tags.

You may think and reason as much as you want, but ensure that your final answer for (1): the informal statement is wrapped in `<STATEMENT>...</STATEMENT>` tags, and (2): the informal proof is wrapped in `<PROOF>...</PROOF>` tags. Your final answer should have both a `<STATEMENT>...</STATEMENT>` tag and a `<PROOF>...</PROOF>` tag, and if there is no formal proof provided in the input, you may simply output `<PROOF></PROOF>` for the proof after informalizing the statement (i.e. if you are given a theorem without a proof, or a definition/class/etc.). Input: `<FORMAL>`

(formal statement of proof is placed here)

`</FORMAL>`

The final informal statements are then parsed out for use in neurosymbolic augmentation.

F Additional Experiments and Analyses

F.1 Ablation Study

In addition to experiments presented in 5.2, we study the effect of each source of neurosymbolic augmentation on model performance, finding that each added channel increases length-metric performance (Figure 8). Additionally, we perform hyperparameter searches at each iteration for both the length (Figure 9) and dependency metric (Figure 10).

F.1.1 Neurosymbolic Ablations

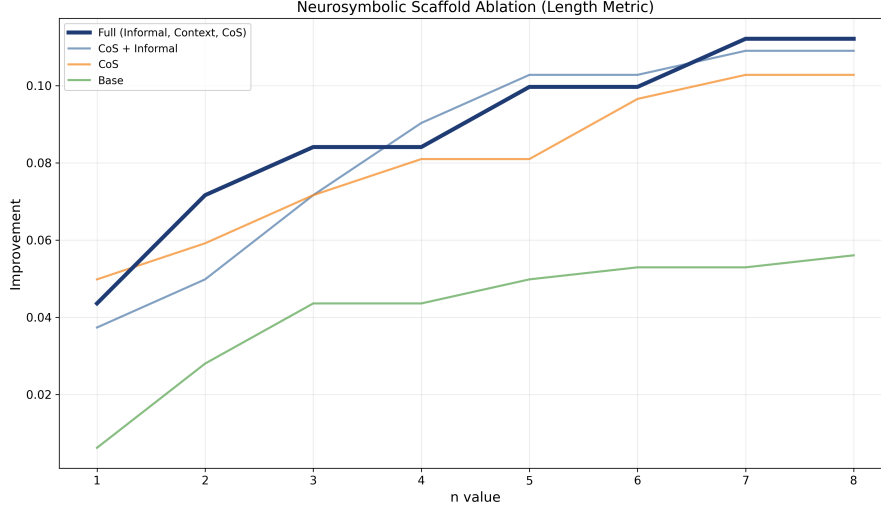


Figure 8: Effect of neurosymbolic augmentation on base model performance (DeepSeek-R1-Distill-Qwen-7B) on the length metric, vs. number of samples generated. Each source of augmentation shows noticeable improvement in average score on some n values.

Figure 8 shows that the unaugmented base model is consistently the weakest generator across sample budgets. Adding chain-of-states information alone produces a substantial improvement, indicating that exposing intermediate Lean goals gives the model useful local structure for proof rewriting. Adding informalized statements and proofs marginally improves performance across much of the curve, and the full scaffold, which additionally includes retrieved context, reaches the best performance at larger values of n . However, we observe that the majority of this improvement comes from the initial chain-of-states annotation. The gap is especially clear at best@8, where the full scaffold roughly doubles the improvement of the unaugmented base model. These results support the claim that neurosymbolic augmentation is not merely a prompting detail, but a central part of making proof optimization learnable and sample-efficient for small models.

F.1.2 Grid Search Results

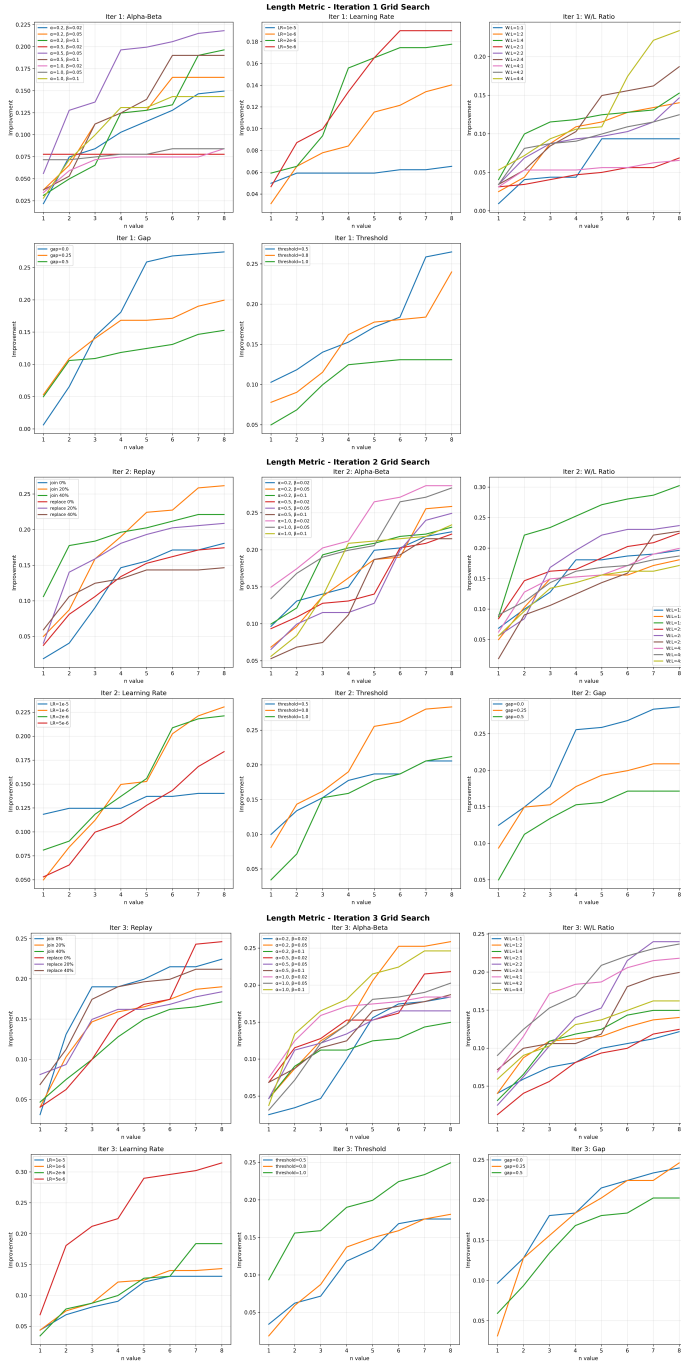


Figure 9: Hyperparameter grid searches for the **length** metric across iterations 1–3.

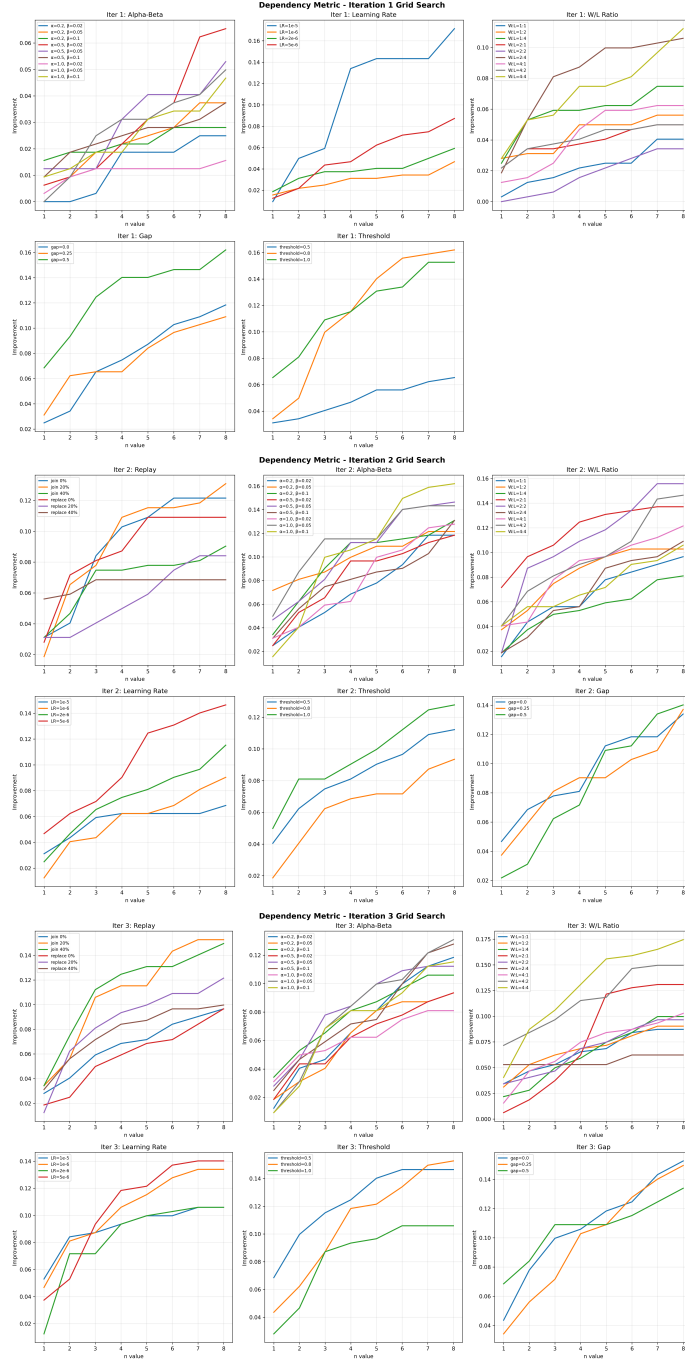


Figure 10: Hyperparameter grid searches for the **dependency** metric across iterations 1–3.

Figures 9 and 10 show that the training pipeline is sensitive to optimization hyperparameters, but that several qualitative patterns are stable. Increasing the sample budget generally improves scores for nearly all settings, so the relative comparisons are not artifacts of a single best@ n choice. For length optimization, lower preference gaps and moderate replay settings tend to perform well, suggesting that proof-shortening benefits from keeping a broad set of successful rewrites in the training signal. For dependency optimization, larger filtering gaps and more selective thresholds are often stronger, consistent with the metric rewarding more targeted structural changes. Learning-rate and winner/loser-ratio effects vary across iterations, which is why we tune these settings separately for each metric and training round rather than reusing a single global configuration.

F.2 Qualitative Examples

Figures 11–19 show representative examples from the held-out evaluation set. For each example, we report the source module and declaration together with the original and optimized metric values. The AlphaProof qualitative case studies use best@64 sampling rather than the best@16 budget used for the main MiniCTX-v2 evaluations. The original AlphaProof IMO 2024 release targets Lean v4.10.0; for these examples, we manually bumped the AlphaProof proofs to Lean v4.17.0. The bumped version is available at [Redacted], and the original release is available at <https://storage.googleapis.com/deepmind-media/DeepMind.com/Blog/imo-2024-solutions/index.html>.

F.2.1 Dependency Optimization

Original (human-written)

```
theorem isCoatom_iff [OrderTop A] {K : A} :
  IsCoatom K ↔ K ≠ T ∧ ∀ H g, K ≤ H → g ∉ K → g ∈
  H → H = T := by
  simp_rw [IsCoatom, lt_iff_le_not_le,
    SetLike.not_le_iff_exists,
    and_comm (a := _ ≤ _), and_imp, exists_imp, ← and_imp,
    and_comm]
```

ImProver 2 (dependency-optimized)

```
theorem isCoatom_iff [OrderTop A] {K : A} :
  IsCoatom K ↔ K ≠ T ∧ ∀ H g, K ≤ H → g ∉ K → g ∈
  H → H = T := by
  constructor <|> intro h
  <|> simp_all [IsCoatom, lt_iff_le_not_le,
    SetLike.not_le_iff_exists]
  <|> tauto
```

Figure 11: ImProver 2 optimizes a proof for dependencies, reducing explicit dependencies by 3.

Figure 11 comes from `Mathlib.Order.Atoms`, declaration `SetLike.isCoatom_iff`. The original proof has explicit dependency count 5, while the optimized proof has count 2. The rewrite replaces a long `simp_rw` chain naming several transformations with a case split, broader simplification, and propositional reasoning via `tauto`; this reduces the explicit dependency footprint while preserving the same theorem statement.

Original (human-written)

```
theorem mem_cross_iff (x y : TSet γ) :
  ∀ a, a ∈ 'cross hβ hγ hδ x y ↔ ∃ b c, a = ⟨b, c⟩' ∧
  b ∈ 'x ∧ c ∈ 'y := by
  intro a
  rw [cross, mem_inter_iff, vCross_spec]
  constructor
  · rintro ⟨h₁, b, c, rfl, h₂⟩
    simp only [op_mem_converse_iff, vCross_spec, op_inj] at
    h₁
    obtain ⟨b', c', ⟨rfl, rfl⟩, h₁⟩ := h₁
    exact ⟨b, c, rfl, h₁, h₂⟩
  · rintro ⟨b, c, rfl, h₁, h₂⟩
    simp only [op_mem_converse_iff, vCross_spec, op_inj]
    exact ⟨⟨c, b, ⟨rfl, rfl⟩, h₁⟩, ⟨b, c, ⟨rfl, rfl⟩, h₂⟩⟩
```

ImProver 2 (dependency-optimized)

```
theorem mem_cross_iff (x y : TSet γ) :
  ∀ a, a ∈ 'cross hβ hγ hδ x y ↔ ∃ b c, a = ⟨b, c⟩' ∧
  b ∈ 'x ∧ c ∈ 'y := by
  intro a
  -- Use the definition of cross and simplify the membership
  -- conditions directly
  constructor <|> intro h
  -- First direction: Assume membership in cross, construct
  -- the pair
  <|> simp [cross] at h ⊢
  -- Second direction: Decompose the pair existence claim
  -- and verify conditions
  <|> aesop
  -- Handle remaining simple cases with basic reasoning
```

Figure 12: ImProver 2 optimizes a proof for dependencies, reducing explicit dependencies by 2.

Figure 12 comes from `ConNF.External.Basic`, declaration `ConNF.mem_cross_iff`. The dependency count decreases from 2 to 0. The optimized proof avoids explicitly naming the original rewrite lemmas and instead unfolds `cross` and delegates the remaining elementary cases to automation. This illustrates a common dependency-optimization behavior: replacing brittle named rewrite sequences with more local simplification and search.

Original (AlphaProof)

```

theorem imo_2024_p6
  (IsAquaesulian : (Q → Q) → Prop)
  (IsAquaesulian_def : ∀ f, IsAquaesulian f ↔
    ∀ x y, f (x + f y) = f x + y ∨ f (f x + y) = x + f y)
  :
  IsLeast {(c : Z) | ∀ f, IsAquaesulian f → {(f r + f (-r)) | (r : Q)}.Finite ∧
    {(f r + f (-r)) | (r : Q)}.ncard ≤ c} 2 := by
  exists0?_
  · use λ u b => if j : u 0 = 0 then by_contra λ c => ?_ else ?_
  ·
    suffices: ({J | ∃ k, u k + u (-k) = J} ⊆ {0})
    · simp_all [this.antisymm]
    · rintro - (a, rfl)
    · contrapose! c
    · simp_all
    suffices: {U | ∃ examples6 : Q, u examples6 + u (-examples6) = U} ⊆ {0, (u (a : Rat) + (u <| 0 0 ↑ ((-a) )))) }
    ..
    · constructor
    · exact (Set.toFinite ( _ ) ).subset (by simp using this)
    · exact (Set.ncard_le_ncard (by simp using this)).
      trans (Set.ncard_pair (Ne.symm (↑ ( (c) ) ) ) ).le
      rintro - (hz, rfl)
      induction b @hz a
      · have := b (-a) $ hz + u a
        · have := b hz hz
        · simp_all [add_comm]
        · have := b (-hz) (hz + u ↑ (hz))
        · simp_all [add_assoc, C]
        · induction this
        ·
          simp_all
          have := b hz (hz + (u a + u (-a)))
          have := b (hz + (u a + u (-a))) $ hz + (u a + u (-a))
          use .inr $ by_contra $ by hint
          have := b hz $ hz + (u hz + u (-hz))
          cases b (hz + (u hz + u (-hz))) $ hz + (u hz + u (-hz)) with | _
            => hint
          have := b (-hz) (u hz + a)
          have := b $ -a
          specialize this (u hz + a)
          simp_all [← add_assoc]
          have := b 0
          have := b
          specialize b a a
          simp_all [add_comm]
          have := (this <| -a) (↑ a + ((u a)) : (↑ _ : (( _ ) ) ) )
          ..
          simp_all [add_assoc]
          cases this
          ·
            simp_all
            contrapose! IsAquaesulian_def
            simp_all
            ex falso
            have := this a (a + (u hz + u (-hz)))
            simp_all [Ne.symm, Bool]
            have := (∀ congr_arg G, _) (a + (u hz + u (-hz))) $ a + (u ↑ hz + u ↑ (-hz))
            simp_all
            have := this a (a + (u a + u (-a)))
            cases (forall Jd S, _) (a + (u a + u (-a))) ( a + (u a + u ↑ (-a))) with | _ => hint
            simp_all
            cases b 0 0 with | _ => exact absurd (b 0 $ (0 + (1 * (u ↑ ((0)))))) ~ 01 : ↑ (( _ ) ) (id $ (by (cases ( b (u 0) ( (u 0) )) with | _ => continuity)))
            rintro K V
            specialize V $ λ N => -N + 2 * Int.ceil N
            specialize ( V $ (IsAquaesulian_def _).mpr _ )
            · simp_rw [← eq_sub_iff_add_eq']
            · ring_nf
            · use mod_cast0?_
            · norm_num [← add_mul, Int.ceil_eq_iff]
            · use λ c K => (em _).imp ((by linarith [Int.ceil_lt_add_one c, Int.le_ceil K],.)) (by repeat use by linarith [., Int.le_ceil c, or, Int.ceil_lt_add_one $ K])
            let S : Set Q := {x | ∃ r : Q, (-r + 2 * ↑[r]) + (-(-r) + 2 * ↑[-r]) = x}
            have hsub : {(0 : Q), 2} : Set Q ⊆ S := by
              intro x hx
              simp at hx
              rcases hx with rfl | rfl
              · exact ((-1 : Q), by norm_num)

```

ImProver 2 (dependencies-optimized)

```

theorem imo_2024_p6
  (IsAquaesulian : (Q → Q) → Prop)
  (IsAquaesulian_def : ∀ f, IsAquaesulian f ↔
    ∀ x y, f (x + f y) = f x + y ∨ f (f x + y) = x + f y)
  :
  IsLeast {(c : Z) | ∀ f, IsAquaesulian f → {(f r + f (-r)) | (r : Q)}.Finite ∧
    {(f r + f (-r)) | (r : Q)}.ncard ≤ c} 2 := by
  exists0?_
  · use λ u b => if j : u 0 = 0 then by_contra λ c => ?_ else ?_
  · suffices: ({J | ∃ k, u k + u (-k) = J} ⊆ {0})
    · simp_all [this.antisymm]
    · rintro - (a, rfl)
    · contrapose! c
    · simp_all
    suffices: {U | ∃ examples6 : Q, u examples6 + u (-examples6) = U} ⊆ {0, (u (a : Rat) + (u <| 0 0 ↑ ((-a) )))) }
    ..
    · constructor
    · exact (Set.toFinite ( _ ) ).subset (by simp using this)
    · exact (Set.ncard_le_ncard (by simp using this)).
      trans (Set.ncard_pair (Ne.symm (↑ ( (c) ) ) ) ).le
      rintro - (hz, rfl)
      induction b @hz a
      · have := b (-a) $ hz + u a
        · have := b hz hz
        · simp_all [add_comm]
        · have := b (-hz) (hz + u ↑ (hz))
        · simp_all [add_assoc, C]
        · induction this
        · simp_all
          have := b hz (hz + (u a + u (-a)))
          have := b (hz + (u a + u (-a))) $ hz + (u a + u (-a))
          use .inr $ by_contra $ by hint
          have := b hz $ hz + (u hz + u (-hz))
          cases b (hz + (u hz + u (-hz))) $ hz + (u hz + u (-hz)) with | _
            => hint
          have := b (-hz) (u hz + a)
          have := b $ -a
          specialize this (u hz + a)
          simp_all [← add_assoc]
          have := b 0
          have := b
          specialize b a a
          simp_all [add_comm]
          have := (this <| -a) (↑ a + ((u a)) : (↑ _ : (( _ ) ) ) )
          ..
          simp_all [add_assoc]
          cases this
          · simp_all
            contrapose! IsAquaesulian_def
            simp_all
            ex falso
            have := this a (a + (u hz + u (-hz)))
            simp_all [Ne.symm, Bool]
            have := (∀ congr_arg G, _) (a + (u hz + u (-hz))) $ a + (u ↑ hz + u ↑ (-hz))
            simp_all
            have := this a (a + (u a + u (-a)))
            cases (forall Jd S, _) (a + (u a + u (-a))) ( a + (u a + u ↑ (-a))) with | _ => hint
            simp_all
            cases b 0 0 with | _ => exact absurd (b 0 $ (0 + (1 * (u ↑ ((0)))))) ~ 01 : ↑ (( _ ) ) (id $ (by (cases ( b (u 0) ( (u 0) )) with | _ => continuity)))
            rintro K V
            specialize V $ λ N => -N + 2 * Int.ceil N
            specialize ( V $ (IsAquaesulian_def _).mpr _ )
            · simp_rw [← eq_sub_iff_add_eq']
            · ring_nf
            · use mod_cast0?_
            · norm_num [← add_mul, Int.ceil_eq_iff]
            · use λ c K => (em _).imp ((by linarith [Int.ceil_lt_add_one c, Int.le_ceil K],.)) (by repeat use by linarith [., Int.le_ceil c, or, Int.ceil_lt_add_one $ K])
            let S : Set Q := {x | ∃ r : Q, (-r + 2 * ↑[r]) + (-(-r) + 2 * ↑[-r]) = x}
            have hsub : {(0 : Q), 2} : Set Q ⊆ S := by
              intro x hx
              simp at hx
              rcases hx with rfl | rfl
              · exact ((-1 : Q), by norm_num)

```

Original (continued)

```

suffices:2<=V.1.toFinset.card
· let M:=V.1.toFinset
  have h2nat : 2 ≤ ({x | ∃ r : ℚ, 2 * ↑[r] + -(2 * ↑[r])
    = x} : Set ℚ).ncard := by
    rwa [Set.ncard_eq_toFinset_card _ V.1]
  have h2int : (2 : ℤ) ≤ ({x | ∃ r : ℚ, 2 * ↑[r] + -(2 *
    ↑[r]) = x} : Set ℚ).ncard := by
    exact_mod_cast h2nat
  exact h2int.trans V.2
use Finset.one_lt_card.2$ by
  refine (0, V.1.mem_toFinset.2 ?_, 2, V.1.mem_toFinset.2 ?
    _, by norm_num)
· exact (-1, by norm_num)
· exact ((1 / 2 : ℚ), by norm_num)

```

ImProver 2 (dependencies-optimized; continued)

```

· exact ((1 / 2 : ℚ), by norm_num)
have h2nat : 2 ≤ S.ncard := by
  have hp : ({(0 : ℚ), 2} : Set ℚ).ncard = 2 := by
    simpa using (Set.ncard_pair (by norm_num : (0 : ℚ) ≠
      2))
  calc
    2 = ({(0 : ℚ), 2} : Set ℚ).ncard := hp.symm
    _ ≤ S.ncard := Set.ncard_le_ncard hsub V.1
have h2int : (2 : ℤ) ≤ S.ncard := by
  exact_mod_cast h2nat
exact h2int.trans V.2

```

Figure 13: ImProver 2 optimizes an AlphaProof proof for dependencies, reducing explicit dependencies by 3.

Figure 13 comes from `AlphaProof.P6`, declaration `imo_2024_p6`. The score changes from 18 to 15. The optimized proof keeps the same global proof search structure but removes a few explicit named dependencies by relying more on local facts and consolidated arithmetic/set-cardinality reasoning. However, overall, this case demonstrates negligible impact by ImProver2 on such a large-scale, complex problem.

F.2.2 Length Optimization

Original (human-written)

```

lemma summerCommute_jacobi_ofCrAnListF (ψ1 ψ2 ψ3 : List
  ℱ.CrAnFieldOp) :
  [ofCrAnListF ψ1, [ofCrAnListF ψ2, ofCrAnListF ψ
    s3]_sca]_sca =
  S (ℱ |>_s ψ1, ℱ |>_s ψ3) •
  (- S (ℱ |>_s ψ2, ℱ |>_s ψ3) • [ofCrAnListF ψ3,
    [ofCrAnListF ψ1, ofCrAnListF ψ2]_sca]_sca -
  S (ℱ |>_s ψ1, ℱ |>_s ψ2) • [ofCrAnListF ψ2,
    [ofCrAnListF ψ3, ofCrAnListF ψ1]_sca]_sca) := by
  repeat rw [superCommuteF_ofCrAnListF_ofCrAnListF]
  simp only [instCommGroup, map_sub, map_smul, neg_smul]
  repeat rw [superCommuteF_ofCrAnListF_ofCrAnListF]
  simp only [instCommGroup.eq_1, ofList_append_eq_mul,
    List.append_assoc]
  by_cases h1 : (ℱ |>_s ψ1) = bosonic <;>
  by_cases h2 : (ℱ |>_s ψ2) = bosonic <;>
  by_cases h3 : (ℱ |>_s ψ3) = bosonic
  · simp only [h1, h2, h3, mul_self, bosonic_exchangeSign,
    one_smul, exchangeSign_bosonic, neg_sub]
    abel
  · simp only [h1, h2, bosonic_exchangeSign, one_smul,
    mul_bosonic, mul_self, map_one,
    exchangeSign_bosonic, neg_sub]
    abel
  · simp only [h1, h3, mul_bosonic, bosonic_exchangeSign,
    one_smul, exchangeSign_bosonic, neg_sub,
    mul_self, map_one]
    abel
  · simp only [neq_bosonic_iff_eq_fermionic] at h1 h2 h3
    simp only [h1, h2, h3, mul_self, bosonic_exchangeSign,
    one_smul,
    fermionic_exchangeSign_fermionic, neg_smul, neg_sub,
    bosonic_mul_fermionic, sub_neg_eq_add,
    mul_bosonic, smul_add, exchangeSign_bosonic]
    abel
  · simp only [neq_bosonic_iff_eq_fermionic] at h1 h2 h3
    simp only [h1, h2, h3, bosonic_mul_fermionic,
    fermionic_exchangeSign_fermionic, neg_smul,
    one_smul, sub_neg_eq_add, bosonic_exchangeSign,
    mul_bosonic, smul_add, exchangeSign_bosonic,
    neg_sub, mul_self]
    abel
  · simp only [neq_bosonic_iff_eq_fermionic] at h1 h2 h3
    simp only [h1, h2, h3, mul_bosonic,
    fermionic_exchangeSign_fermionic, neg_smul, one_smul,
    sub_neg_eq_add, exchangeSign_bosonic,
    bosonic_mul_fermionic, smul_add, mul_self,
    bosonic_exchangeSign, neg_sub]
    abel
  · simp only [neq_bosonic_iff_eq_fermionic] at h1 h2 h3

```

ImProver 2 (length-optimized)

```

lemma summerCommute_jacobi_ofCrAnListF (ψ1 ψ2 ψ3 : List
  ℱ.CrAnFieldOp) :
  [ofCrAnListF ψ1, [ofCrAnListF ψ2, ofCrAnListF ψ
    s3]_sca]_sca =
  S (ℱ |>_s ψ1, ℱ |>_s ψ3) •
  (- S (ℱ |>_s ψ2, ℱ |>_s ψ3) • [ofCrAnListF ψ3,
    [ofCrAnListF ψ1, ofCrAnListF ψ2]_sca]_sca -
  S (ℱ |>_s ψ1, ℱ |>_s ψ2) • [ofCrAnListF ψ2,
    [ofCrAnListF ψ3, ofCrAnListF ψ1]_sca]_sca) := by
  simp_all [superCommuteF_ofCrAnListF_ofCrAnListF,
    instCommGroup, map_sub, map_smul, neg_smul,
    superCommuteF_ofCrAnListF_ofCrAnListF,
    instCommGroup.eq_1, ofList_append_eq_mul,
    List.append_assoc,
    neq_bosonic_iff_eq_fermionic]
  <;> by_cases h1 : (ℱ |>_s ψ1) = bosonic <;>
  by_cases h2 : (ℱ |>_s ψ2) = bosonic <;>
  by_cases h3 : (ℱ |>_s ψ3) = bosonic
  <;> simp_all [h1, h2, h3, mul_self, bosonic_exchangeSign,
    one_smul, exchangeSign_bosonic, neg_sub,
    fermionic_exchangeSign_fermionic, neg_smul,
    bosonic_mul_fermionic, sub_neg_eq_add,
    mul_bosonic, smul_add, exchangeSign_bosonic,
    neg_sub, mul_self] <;> abel

```



```

simp only [h1, h2, h3, mul_self, map_one, one_smul,
  fermionic_exchangeSign_fermionic, neg_smul,
  neg_sub]
abel

```

Figure 14: ImProver 2 optimizes a proof for length, reducing tactic count by 19.

Figure 14 comes from `HepLean.PerturbationTheory.FieldOpFreeAlgebra.SuperCommute`, declaration `summerCommute_jacobi_ofCrAnListF`. The tactic count decreases from 43 to 24. The optimized proof collapses repeated rewriting and many case-specific simplification branches into a single larger `simp_all` call followed by the same case split structure and `abel`. This preserves the human proof’s algebraic strategy while eliminating repeated local boilerplate.

Original (human-written)

```

theorem mem_cross_iff (x y : TSet  $\gamma$ ) :
   $\forall a, a \in 'cross\ h\beta\ h\gamma\ h\delta\ x\ y \leftrightarrow \exists b\ c, a = \langle b, c \rangle' \wedge$ 
   $b \in 'x \wedge c \in 'y := by$ 
  intro a
  rw [cross, mem_inter_iff, vCross_spec]
  constructor
  · rintro  $\langle h_1, b, c, rfl, h_2 \rangle$ 
    simp only [op_mem_converse_iff, vCross_spec, op_inj] at
      h1
    obtain  $\langle b', c', \langle rfl, rfl \rangle, h_1 \rangle := h_1$ 
    exact  $\langle b, c, rfl, h_1, h_2 \rangle$ 
  · rintro  $\langle b, c, rfl, h_1, h_2 \rangle$ 
    simp only [op_mem_converse_iff, vCross_spec, op_inj]
    exact  $\langle \langle c, b, \langle rfl, rfl \rangle, h_1 \rangle, \langle b, c, \langle rfl, rfl \rangle, h_2 \rangle \rangle$ 

```

ImProver 2 (length-optimized)

```

theorem mem_cross_iff (x y : TSet  $\gamma$ ) :
   $\forall a, a \in 'cross\ h\beta\ h\gamma\ h\delta\ x\ y \leftrightarrow \exists b\ c, a = \langle b, c \rangle' \wedge$ 
   $b \in 'x \wedge c \in 'y := by$ 
  simp_all [cross, mem_inter_iff, vCross_spec,
    op_mem_converse_iff, op_inj]
  <|> aesop

```

Figure 15: ImProver 2 optimizes a proof for length, reducing tactic count by 8.

Figure 15 is another optimization of `ConNF.External.Basic`, declaration `ConNF.mem_cross_iff`, this time under the length metric. The tactic count decreases from 10 to 2. The optimized proof replaces the explicit bidirectional constructor proof with a compact simplification over the relevant definitions followed by `aesop`; this is the characteristic length-optimization pattern of compressing routine structural reasoning into a small number of automation-heavy tactics.

Original (AlphaProof)

```

theorem imo_2024_p2 : {(a, b) | 0 < a ∧ 0 < b ∧ ∃ g N, 0 <
  g ∧ 0 < N ∧ ∀ n ≥ N, Nat.gcd (a ^ n + b) (b ^ n + a)
  = g} = {(1, 1)} := by
  induction (10)+2
  · use Set.eq_singleton_iff_unique_mem.2 ⟨?, λb g =>
    by_contra! $ g.2.2.rec λY S i => S.rec λL D => ?_⟩
  ·
    exact (by left, by left, 2, 3, by simp_all)
  have : b.1 + b.2 | Y := ?_
  ·
    suffices : b.1 = b.2
    ·
      norm_num [b.ext_iff, <- D.2.2 L, this] at *
      use (Nat.pow_lt_pow_right (g.1.nat_succ.le.lt_of_ne' i
        ) L.lt_succ_self).ne' (D.2.2 _ L.le_succ)
      suffices : b.1 + b.2 | b.fst ^ (2 * L) + b.2 ^ (b.fst + (b).snd |
        b.snd ^ (2 * L) + b.1
      ·
        suffices : b.1 ^ 2 | (b.1 + b.2) = b.2 ^ 2 | (b.1 + b.snd)
        ·
          norm_num [Nat.add_mod, pow_mul, this, Nat.
            dvd_iff_mod_eq_zero, Nat.pow_mod] at *
          norm_num [add_comm, b.ext_iff, sq _, <- Nat.pow_mod, <-
            Nat.dvd_iff_mod_eq_zero] at *
          zify at *
          cases this.1.sub this.2.with! _ Z => nlinarith [ (by (
            nlinarith) : Z = 0 ) ]
          apply (Nat.modEq_of_dvd
            use (b.snd) - b.fst , (by ring : ( (b.snd) : Z) ^ 2 - b.fst
              ^ 2 = (b.fst + (b).2) ^ 2 - _ )
            norm_num [(2).le_mul_of_pos_left, Nat.gcd_dvd, <- D.2.2
              (2 * L), this.trans, (D.right.1 : _)]
            suffices : b.1 + b.2 | b.1 ^ (2 * L) + b.2 ^ (b.1 + b.2 | b.snd ^ (2 * L) + b
              .1
          ·
            exact D.2.2 (2 * L) ( (le_mul_of_one_le_left' (by
              decide ) ) > dvd_gcd (this.left) (this).2
            ex falso
            suffices : b.1 * b.2 + 1 | Y
            ·
              suffices : b.1 ^ (b.1 * b.2 + 1) % (b.1 * b.2 + 1) = 1 % (b.1 * b.2 + 1) ∧
                b.2 ^ (b.1 * b.snd + 1) % ((b).1 * ↑(b.snd) + 1) = 1 % (b.1 * b.
                  snd + 1)
              ·
                absurd D.2.2 ( (b.1 * b.2 + 1) * L) (by nlinarith [((b.
                  fst * b.2 + 1).totient_pos).2 ↑ Fin.size_pos'])
                apply mt (.> Nat.gcd_dvd _ _)
                use AH => absurd (⟦_⟧Y).trans H.1) (λv => absurd (⟦_⟧Y).
                  trans H.2) ? _
                norm_num [pow_mul, b.ext_iff, (1).mod_eq_of_lt, g.symm,
                  this, Nat.add_mod, Nat.dvd_iff_mod_eq_zero, Nat.pow_mod]
                  at (i) v
                norm_num [add_comm, pow_mul, <- Nat.dvd_iff_mod_eq_zero]
                  at *
                contrapose! i
                zify at *
                repeat use by nlinarith [Int.le_of_dvd (by linarith) v
                  , Int.le_of_dvd (by linarith) i]
                repeat use ↑(Nat.ModEq.pow_totient (by norm_num))
            by_contra! H
            suffices : b.1 ^ (b.1 * b.2 + 1) % (b.1 * b.2 + 1) = 1 % (b.1 * b.2 + 1) ∧ b
              .2 ^ (b.1 * b.2 + 1) % (b.1 * b.2 + 1) = 1 % (b.fst * ↑(b.snd) + 1)
            ·
              simp_all
              suffices : b.1 * b.2 + 1 | b.1 ^ ( (b.1 * b.2 + 1) * (L + 1) - 1) + b.2 ^ (b.1
                * b.2 + 1) | b.2 ^ ( (b.1 * b.2 + 1) * (L + 1) - 1) + (b.fst)
              ·
                use H $ D.2.2 ( (b.1 * b.2 + 1) * (L + 1) - 1) (L.le_sub_of_add_le (by
                  nlinarith [((b.1 * b.2 + 1).totient_pos).2 Nat.succ_pos'])
                  > ((Nat.dvd_gcd) ( this).1) ) this.right
                  cases B : Nat.exists_eq_add_of_lt $ ( (b.1 * b.2 + 1).
                    totient_pos).2 (by continuity)
                    norm_num [*, g, ( (b.1 * b.2 + 1) * (L + 1) - 1) + b.2 ^ (b.1
                      * b.2 + 1) | b.1 ^ ( (b.1 * b.2 + 1) * (L + 1) - 1) + (b.fst)
                    mod_eq_of_lt, pow_add, Nat.add_mod, pow_mul, Nat.
                      dvd_iff_mod_eq_zero, Nat.mul_mod] at this
                    simp_all
                    suffices : b.1 * b.2 + 1 | b.1 * ( (b.1 % (b).1 * ( (b.snd) + 1) :
                      _) ^ (Nat) + b.snd) ∧ (b.fst * ↑(b.snd) + 1) | (b).snd * ( (b.
                        snd % ((b).fst * b.snd + 1)) ^ (Nat) + b.fst)
                    ·
                      norm_num [ <- Nat.dvd_iff_mod_eq_zero, g, (1).mod_eq_of_lt
                        , Nat.dvd_mul] at this
                      exists @?_
                      ·
                        cases this.1 with! _ Q r => simp_all [(Q.dvd_gcd r.1 ( _
                          , symm r.right.choose_spec.2)).antisymm]
                        cases @this.2 with! _ F X => simp_all [(F.dvd_gcd X.1 ( _
                          , symm X.2.choose_spec.2)).antisymm]
                    simp_all [mul_comm, mul_add, add_comm, Nat.add_mod, Nat.
                      dvd_iff_mod_eq_zero]
                    repeat use (Nat.ModEq.pow_totient (by . . . norm_num) )
            congr 26

```

ImProver 2 (length-optimized)

```

theorem imo_2024_p2 : {(a, b) | 0 < a ∧ 0 < b ∧ ∃ g N, 0 <
  g ∧ 0 < N ∧ ∀ n ≥ N, Nat.gcd (a ^ n + b) (b ^ n + a)
  = g} = {(1, 1)} := by
  refine Set.eq_singleton_iff_unique_mem.2 ⟨(Nat.zero_lt_one,
    Nat.zero_lt_one, 2, 1, Nat.succ_pos 1, Nat.
      zero_lt_one, by simp), λb g => by_contra! $ g.2.2.rec λY S
        i => S.rec λL D => ?_⟩
  suffices : b.1 * b.2 + 1 | Y
  ·
    suffices : b.1 ^ (b.1 * b.2 + 1) % (b.1 * b.2 + 1) = 1 % (b.1 * b.2 + 1) ∧ b
      .2 ^ (b.1 * b.snd + 1) % ((b).1 * ↑(b.snd) + 1) = 1 % (b.1 * b.
        snd + 1)
    ·
      absurd D.2.2 ( (b.1 * b.2 + 1) * L) (by nlinarith [((b.fst *
        b.2 + 1).totient_pos).2 ↑ Fin.size_pos'])
      apply mt (.> Nat.gcd_dvd _ _)
      use AH => absurd (⟦_⟧Y).trans H.1) (λv => absurd (⟦_⟧Y).
        trans H.2) ? _
      norm_num [pow_mul, b.ext_iff, (1).mod_eq_of_lt, g.symm, this,
        Nat.add_mod, Nat.dvd_iff_mod_eq_zero, Nat.pow_mod] at (i)
      v
      norm_num [add_comm, pow_mul, <- Nat.dvd_iff_mod_eq_zero] at *
      contrapose! i
      zify at *
      repeat use by nlinarith [Int.le_of_dvd (by linarith) v,
        Int.le_of_dvd (by linarith) i]
      repeat use ↑(Nat.ModEq.pow_totient (by norm_num))
  by_contra! H
  suffices : b.1 ^ (b.1 * b.2 + 1) % (b.1 * b.2 + 1) = 1 % (b.1 * b.2 + 1) ∧ b.2 ^
    (b.1 * b.2 + 1) % (b.1 * b.2 + 1) = 1 % (b.fst * ↑(b.snd) + 1)
  ·
    simp_all
    suffices : b.1 * b.2 + 1 | b.1 ^ ( (b.1 * b.2 + 1) * (L + 1) - 1) + b.2 ^ (b.1 *
      b.2 + 1) | b.2 ^ ( (b.1 * b.2 + 1) * (L + 1) - 1) + (b.fst)
    ·
      use H $ D.2.2 ( (b.1 * b.2 + 1) * (L + 1) - 1) (L.le_sub_of_add_le (by
        nlinarith [((b.1 * b.2 + 1).totient_pos).2 Nat.succ_pos'])
        > ((Nat.dvd_gcd) ( this).1) ) this.right
        cases B : Nat.exists_eq_add_of_lt $ ( (b.1 * b.2 + 1).totient_pos
          ).2 (by continuity)
          norm_num [*, g, ( (b.1 * b.2 + 1) * (L + 1) - 1) + b.2 ^ (b.1 *
            b.2 + 1) | b.1 ^ ( (b.1 * b.2 + 1) * (L + 1) - 1) + (b.fst)
          mod_eq_of_lt, pow_add, Nat.add_mod, pow_mul, Nat.
            dvd_iff_mod_eq_zero, Nat.mul_mod] at this
          simp_all
          suffices : b.1 * b.2 + 1 | b.1 * ( (b.1 % (b).1 * ( (b.snd) + 1) :
            _) ^ (Nat) + b.snd) ∧ (b.fst * ↑(b.snd) + 1) | (b).snd * ( (b.
              snd % ((b).fst * b.snd + 1)) ^ (Nat) + b.fst)
          ·
            norm_num [ <- Nat.dvd_iff_mod_eq_zero, g, (1).mod_eq_of_lt,
              Nat.dvd_mul] at this
            exists @?_
            ·
              cases this.1 with! _ Q r => simp_all [(Q.dvd_gcd r.1 ( _
                , symm r.right.choose_spec.2)).antisymm]
              cases @this.2 with! _ F X => simp_all [(F.dvd_gcd X.1 ( _
                , symm X.2.choose_spec.2)).antisymm]
          simp_all [mul_comm, mul_add, add_comm, Nat.add_mod, Nat.
            dvd_iff_mod_eq_zero]
          repeat use (Nat.ModEq.pow_totient (by . . . norm_num) )

```

Figure 16: ImProver 2 optimizes an AlphaProof proof for length, reducing tactic count by 26.

Figure 16 comes from AlphaProof.P2, declaration imo_2024_p2. The score changes from 80 to 54. The optimized proof removes a discarded intermediate lemma and folds repeated divisibility and

Euler-theorem reasoning into the main argument. The resulting proof is still dense, but it is materially shorter while preserving the same number-theoretic spine: prove the singleton characterization, force divisibility by $ab + 1$, and derive $a = b = 1$.

F.2.3 Modularity Optimization

Original (human-written)

```
lemma KD_weakerThan_KDB : (Hilbert.KD  $\alpha$ )  $\leq_s$  (Hilbert.KDB  $\alpha$ )
:= normal_weakerThan_of_subset $ by intro; aesop;
```

ImProver 2 (modularity-optimized)

```
lemma KD_weakerThan_KDB : (Hilbert.KD  $\alpha$ )  $\leq_s$  (Hilbert.KDB  $\alpha$ )
:= by
  -- Introduce a lemma to handle the subset relationship
  -- between the axioms of KD and KDB
  have h1 : (LO.Modal.Hilbert.KD  $\alpha$ ).axioms  $\subseteq$ 
    (LO.Modal.Hilbert.KDB  $\alpha$ ).axioms  $\rightarrow$  (Hilbert.KD  $\alpha$ )  $\leq_s$ 
    (Hilbert.KDB  $\alpha$ ) := by
    intro h
    -- Apply the lemma that establishes the weakening
    -- relation given the subset of axioms
    apply normal_weakerThan_of_subset
    -- Use the given subset condition
    apply h
  -- Prove the subset relationship between the axioms of KD
  -- and KDB
  have h2 : (LO.Modal.Hilbert.KD  $\alpha$ ).axioms  $\subseteq$ 
    (LO.Modal.Hilbert.KDB  $\alpha$ ).axioms := by
    -- Prove that all axioms of KD are included in KDB
    intro  $\varphi$  h $\varphi$ 
    cases' h $\varphi$  with h $\varphi$  h $\varphi$ 
    · -- Case: KD's basic modal logic axioms are included in
      KDB
      simp_all [LO.Modal.Hilbert.KD]
    · -- Case: KD's additional axiom (K) is included in KDB
      simp_all [LO.Modal.Hilbert.KDB]
  -- Combine the results to conclude the weakening relation
  exact h1 h2
```

Figure 17: ImProver 2 optimizes a proof for modularity, increasing modularity score (see A) by 2.

Figure 17 comes from `Foundation.Modal.Hilbert.WeakerThan.KD_KDB`, declaration `LO.Modal.Hilbert.KD_weakerThan_KDB`. The modularity score increases from 0 to 2. The original proof is a one-line use of `normal_weakerThan_of_subset`; the optimized proof introduces two explicit intermediate claims, one reducing the theorem to an axiom-subset statement and one proving that subset statement. This raises the modularity score by exposing reusable subproof structure.

Original (human-written)

```
lemma hilbertPoly_eq_zero_of_le_rootMultiplicity_one
{p : F[X]} {d : N} (hdp : d  $\leq$  p.rootMultiplicity 1) :
hilbertPoly p d = 0 := by
  by_cases hp : p = 0
  · rw [hp, hilbertPoly_zero_left]
  · rcases exists_eq_pow_rootMultiplicity_mul_and_not_dvd p
    hp 1 with <math>\langle q, hq1, hq2 \rangle</math>
    have heq : p = q * (-1) ^ p.rootMultiplicity 1 * (1 - X)
      ^ p.rootMultiplicity 1 := by
      simp only [mul_assoc,  $\leftarrow$  mul_pow, neg_mul, one_mul,
        neg_sub]
      exact hq1.trans (mul_comm _ _)
    rw [heq,  $\leftarrow$  zero_add d,  $\leftarrow$  Nat.sub_add_cancel hdp,
      pow_add (1 - X),  $\leftarrow$  mul_assoc,
      hilbertPoly_mul_one_sub_pow_add, hilbertPoly]
```

ImProver 2 (modularity-optimized)

```
lemma hilbertPoly_eq_zero_of_le_rootMultiplicity_one
{p : F[X]} {d : N} (hdp : d  $\leq$  p.rootMultiplicity 1) :
hilbertPoly p d = 0 := by
  -- Consider the case where p is the zero polynomial
  have h1 : p = 0  $\rightarrow$  hilbertPoly p d = 0 := by
    intro hp
    rw [hp, hilbertPoly_zero_left]
  -- Consider the case where p is not the zero polynomial
  have h2 : p  $\neq$  0  $\rightarrow$  d  $\leq$  p.rootMultiplicity 1  $\rightarrow$ 
    hilbertPoly p d = 0 := by
    intro hp hdp
    -- Express p as a product of a polynomial, a power of
    -- -1, and a power of (1 - X)
    rcases exists_eq_pow_rootMultiplicity_mul_and_not_dvd p
      hp 1 with <math>\langle q, hq1, hq2 \rangle</math>
    -- Derive the equality of p with the product form
    have heq : p = q * (-1) ^ p.rootMultiplicity 1 * (1 - X)
      ^ p.rootMultiplicity 1 := by
      simp only [mul_assoc,  $\leftarrow$  mul_pow, neg_mul, one_mul,
        neg_sub]
      exact hq1.trans (mul_comm _ _)
    -- Substitute the product form into the Hilbert
    -- polynomial expression and simplify
    rw [heq,  $\leftarrow$  zero_add d,  $\leftarrow$  Nat.sub_add_cancel hdp,
      pow_add (1 - X),  $\leftarrow$  mul_assoc,
      hilbertPoly_mul_one_sub_pow_add, hilbertPoly]
  -- Combine both cases to conclude the proof
  by_cases hp : p = 0 <math>\langle;></math> simp_all
  <math>\langle;></math> simp_all [h1, h2, hdp]
```

Figure 18: ImProver 2 optimizes a proof for modularity, increasing modularity score (see A) by 2.

Figure 18 comes from `Mathlib.RingTheory.Polynomial.HilbertPoly`, declaration `Polynomial.hilbertPoly_eq_zero_of_le_rootMultiplicity_one`. The modularity score increases from 0 to 2. The optimized proof factors the original `by_cases` proof into separate

claims for the zero and nonzero polynomial cases before recombining them. The result is longer, but it makes the case structure explicit, which is the behavior targeted by the modularity metric.

Original (AlphaProof)

```

theorem imo_2024_p6
  (IsAquaesulian : (ℚ → ℚ) → Prop)
  (IsAquaesulian_def : ∀ f, IsAquaesulian f ↔
    ∀ x y, f (x + f y) = f x + y ∨ f (f x + y) = x + f y)
  :
  IsLeast {(c : ℤ) | ∀ f, IsAquaesulian f → {(f r + f (-r)) | (r : ℚ)}.Finite ∧
    {(f r + f (-r)) | (r : ℚ)}.ncard ≤ c} 2 := by
exists0?_
·
  use λ u b => if j : u 0 = 0 then by_contra λ c => ?_ else ?_
  ·
    suffices: {(λ | ∃ k, u k + u (-k) = J)} ⊆ {0}
    · simp_all [this.antisymm]
    rintro - ⟨a, rfl⟩
    contrapose! c
    simp_all
    suffices: {U | ∃ examples6 : ℚ, u examples6 + u (-examples6) = U} ⊆ {0, (u (a : Rat) + (u <| @0†((-a))))} }
    ..
    · constructor
      · exact (Set.toFinite ( _ )).subset (by simp using this)
      · exact (Set.ncard_le_ncard (by simp using this)).trans (Set.ncard_pair (Ne.symm (↑ (c)))).le
    rintro - ⟨hz, rfl⟩
    induction b @hz a
    ·
      have := b (-a) $ hz + u a
      have := b hz hz
      simp_all [add_comm]
      have := b (-hz) (hz + u ↑(hz))
      simp_all [add_assoc, C]
      induction this
      ·
        simp_all
        have := b hz (hz + (u a + u (-a)))
        have := b (hz + (u a + u (-a))) $ hz + (u a + u (-a))
        use .inr $ by_contra $ by hint
        have := b hz $ hz + (u hz + u (-hz))
        cases b (hz + (u hz + u (-hz))) $ hz + (u hz + u (-hz)) with | _ => hint
      have := b (-hz) (u hz + a)
      have := b $ -a
      specialize this (u hz + a)
      simp_all [← add_assoc]
      have := b 0
      have := b
      specialize b a
      simp_all [add_comm]
      have := (this <| -a) (↑a + (((u a))) : (↑_ : (( _ ) ) ) )
      ..
      simp_all [add_assoc]
      cases this
      ·
        simp_all
        contrapose! IsAquaesulian_def
        simp_all
        ex falso
        have := this a (a + (u hz + u (-hz)))
        simp_all [Ne.symm, Bool]
        have := ⟨Vcongr_arg G, _⟩ (a + (u hz + u (-hz))) $ a + (u ↑hz + u ↑(-hz))
        simp_all
        have := this a (a + (u a + u (-a)))
        cases (forall Jd S, _) (a + (u a + u (-a))) (a + (u a + u ↑(-a))) with | _ => hint
      simp_all
      cases b 0 0 with | _ => exact absurd (b 0 $ (0 + (1 * (u ↑((0)))))) ~ 01 : ↑(( _ )) (id $ (by (cases (b (u 0) (u 0))) with | _ => continuity)))
    rintro K V
    specialize V $ λ N => -N + 2 * Int.ceil N
    specialize (V $ (IsAquaesulian_def _).mpr _)
    · simp_rw [← eq_sub_iff_add_eq']
    ring_nf
    use mod_cast0?_
    norm_num [← add_mul, Int.ceil_eq_iff]
    use λ c K => (em _).imp ((by linarith [Int.ceil_lt_add_one c, Int.le ceil K, .]) (by repeat use by linarith [Int.le ceil c, or, Int.ceil_lt_add_one $ K])
    simp_all [Int.ceil_neg, ← add_assoc]

```

ImProver 2 (modularity-optimized)

```

theorem imo_2024_p6
  (IsAquaesulian : (ℚ → ℚ) → Prop)
  (IsAquaesulian_def : ∀ f, IsAquaesulian f ↔
    ∀ x y, f (x + f y) = f x + y ∨ f (f x + y) = x + f y)
  :
  IsLeast {(c : ℤ) | ∀ f, IsAquaesulian f → {(f r + f (-r)) | (r : ℚ)}.Finite ∧
    {(f r + f (-r)) | (r : ℚ)}.ncard ≤ c} 2 := by
exists0?_
·
  use λ u b => if j : u 0 = 0 then by_contra λ c => ?_ else ?_
  ·
    suffices h_zero_image : {(λ | ∃ k, u k + u (-k) = J)} ⊆ {0}
    · simp_all [h_zero_image.antisymm]
    rintro - ⟨a, rfl⟩
    contrapose! c
    simp_all
    suffices h_two_values :
      {U | ∃ examples6 : ℚ, u examples6 + u (-examples6) = U} ⊆
      {0, (u (a : Rat) + (u <| @0†((-a))))} } ..
    · constructor
      · exact (Set.toFinite ( _ )).subset (by simp using h_two_values)
      · exact (Set.ncard_le_ncard (by simp using h_two_values)).trans (Set.ncard_pair (Ne.symm (↑ (c)))).le
    rintro - ⟨hz, rfl⟩
    induction b @hz a
    ·
      have h_step_i := b (-a) $ hz + u a
      have h_diag := b hz hz
      simp_all [add_comm]
      have h_step_iv := b (-hz) (hz + u ↑(hz))
      simp_all [add_assoc, C]
      induction h_step_iv
      · simp_all
        have h_shift_x := b hz (hz + (u a + u (-a)))
        have h_shift_self := b (hz + (u a + u (-a))) $ hz + (u a + u (-a))
        use .inr $ by_contra $ by hint
      · have h_zero_case := b hz $ hz + (u hz + u (-hz))
        cases b (hz + (u hz + u (-hz))) $ hz + (u hz + u (-hz)) with | _ => hint
      · have h_neg_x := b (-hz) (u hz + a)
        have h_neg_a_all := b $ -a
        specialize h_neg_a_all (u hz + a)
        simp_all [← add_assoc]
        have h_zero_all := b 0
        have h_all := b
        specialize b a
        simp_all [add_comm]
        have h_a_shift := (h_all <| -a) (↑a + (((u a))) : (↑_ : (( _ ) ) ) ) ..
        simp_all [add_assoc]
        cases h_a_shift
        · simp_all
          contrapose! IsAquaesulian_def
          simp_all
          ex falso
          have h_test_a := h_all a (a + (u hz + u (-hz)))
          simp_all [Ne.symm, Bool]
          have h_test_self := ⟨Vcongr_arg G, _⟩ (a + (u hz + u (-hz))) $
            a + (u ↑hz + u ↑(-hz))
          simp_all
          · have h_test_a := h_all a (a + (u a + u (-a)))
            cases (forall Jd S, _) (a + (u a + u (-a))) (a + (u a + u ↑(-a))) with | _ => hint
          · simp_all
            cases b 0 0 with | _ => exact absurd (b 0 $ (0 + (1 * (u ↑((0)))))) ~ 01 : ↑(( _ )) (id $ (by (cases (b (u 0) (u 0) (u 0))) with | _ => continuity)))
    rintro K V
    specialize V $ λ N => -N + 2 * Int.ceil N
    have h_model_aqua :
      IsAquaesulian (λ N : ℚ => -N + 2 * Int.ceil N) := by
      refine (IsAquaesulian_def _).mpr ?_
      simp_rw [← eq_sub_iff_add_eq']
      ring_nf
      use mod_cast0?_
      norm_num [← add_mul, Int.ceil_eq_iff]
      use λ c K => (em _).imp

```

Original (continued)

```

suffices:2<=V.1.toFinset.card
· let M:=V.1.toFinset
  have h2nat : 2 ≤ ({x | ∃ r : ℚ, 2 * ↑[r] + -(2 * ↑[r])
    = x} : Set ℚ).ncard := by
    rwa [Set.ncard_eq_toFinset_card _ V.1]
  have h2int : (2 : ℤ) ≤ ({x | ∃ r : ℚ, 2 * ↑[r] + -(2 *
    ↑[r]) = x} : Set ℚ).ncard := by
    exact_mod_cast h2nat
  exact h2int.trans V.2
use Finset.one_lt_card.2$ by
  refine ⟨0, V.1.mem_toFinset.2 ?_, 2, V.1.mem_toFinset.2 ?
    _, by norm_num⟩
  · exact ⟨-1, by norm_num⟩
  · exact ⟨(1 / 2 : ℚ), by norm_num⟩

```

ImProver 2 (modularity-optimized; continued)

```

    (⟨by linarith[Int.ceil_lt_add_one c,Int.le_ceil K],.⟩)
    (by repeat use by linarith[.,Int.le_ceil c,or,Int.
      ceil_lt_add_one$ K])
specialize V h_model_aqua
simp_all[Int.ceil_neg, ←add_assoc]
have h_card_toFinset :
  2<=V.1.toFinset.card := by
  use Finset.one_lt_card.2$ by
    refine ⟨0, V.1.mem_toFinset.2 ?_, 2, V.1.mem_toFinset.2
      ?_, by norm_num⟩
  · exact ⟨-1, by norm_num⟩
  · exact ⟨(1 / 2 : ℚ), by norm_num⟩
have h_card_ncard :
  2 ≤ ({x | ∃ r : ℚ, 2 * ↑[r] + -(2 * ↑[r]) = x} : Set
    ℚ).ncard := by
  rwa [Set.ncard_eq_toFinset_card _ V.1]
have h_card_int :
  (2 : ℤ) ≤ ({x | ∃ r : ℚ, 2 * ↑[r] + -(2 * ↑[r]) = x}
    : Set ℚ).ncard := by
  exact_mod_cast h_card_ncard
exact h_card_int.trans V.2

```

Figure 19: ImProver 2 optimizes an AlphaProof proof for modularity, increasing modularity score by 3.

Figure 19 comes from AlphaProof.P6, declaration imo_2024_p6. The score changes from 0 to 3. The modularity-optimized version introduces named intermediate claims such as `h_zero_image`, `h_two_values`, and `h_model_aqua`. These names expose the proof’s main subgoals: bounding the image set, verifying the model function is aquaesulian, and converting cardinality facts back to the final integer bound.