Neural Proof Optimization

Riyaz Ahuja

April 16, 2026

Mellon College of Science
Department of Mathematics
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Thesis Committee:

Dr. Prasad Tetali
Dr. Jeremy Avigad
Dr. Sean Welleck

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Mathematics

Acknowledgments

Thank you to my advisors: Jeremy Avigad, Prasad Tetali, and Sean Welleck for their guidance, support, and mentorship throughout this project and over the last few years. Without their encouragement and advice, this work would not have been possible.

I am also grateful to Tate Rowney, whose help on ImProver 2 was invaluable, and to the broader Lean community for their support and feedback on this work.

Finally, I would like to thank my family and friends for their love and encouragement throughout this process.

This work was partially supported by NSF Grant DMS-2434614, DARPA expMath Grant HR0011262E028, Alexander M. Knaster Professorship funds, and a gift from Convergent Research. We additionally thank the L3 Lab, Hoskinson Center for Formal Mathematics, Lean FRO, and the OpenAI Researcher Access Program for their support.

Abstract

Formal mathematics libraries are expanding faster than human maintainers can curate them, driven both by community contributors and by an increasing volume of machine-generated proofs. Proof assistants such as Lean 4 guarantee that these proofs are correct, but offer no guarantees about their quality: proofs may be verbose, opaque, or heavily reliant on obscure library lemmas. This thesis studies automated proof optimization: the problem of rewriting a verified proof to be better with respect to a user-defined criterion while preserving formal correctness.

We present two systems. First, we introduce ImProver as a first approach to agentic proof optimization, built around a frontier language model augmented with an agentic scaffold: Chain-of-States prompting, which annotates intermediate Lean proof states into the generation context; iterative error correction; and retrieval-augmented example and document selection. Across undergraduate, competition, and research-level datasets, ImProver improves on naive GPT-4o by 423–778% on length, declarativity, and mixed metrics, and its evaluation on the completion metric shows empirically that proof optimization strictly generalizes neural theorem proving.

ImProver's reliance on a large, closed-source base model limits its scalability. ImProver 2 addresses this by replacing agentic prompting with a self-improving 7B-parameter small language model trained via iterative reasoning preference optimization, together with two novel structural metrics — modularity (decomposition into reusable subproofs) and dependency (reduction of explicitly named external lemmas) — and a neurosymbolic scaffold that couples the model to Lean's elaboration semantics via context extraction, CoS traces, and auto-informalization. A novel replay buffer stabilizes training across multiple rounds. The resulting model leads all evaluated systems on modularity and ties the strongest frontier models on dependencies, despite being orders of magnitude smaller.

Taken together, ImProver and ImProver 2 establish the task as scalable and learnable to the frontier of research mathematics, and provide a foundation for further work on AI-assisted formal mathematics.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The fundamental virtue of a mathematical proof is that it provides certainty: a deductive argument shows that the assumptions of a mathematical statement logically guarantee the conclusion. In practice, however, informal, natural-language proofs are prone to imprecision, ambiguity, and error. Using a formal language such as Lean 4 [1] removes such ambiguity and imprecision, enabling a proof assistant to verify correctness down to the primitives of a formal axiomatic system.

Formal proof assistants such as Lean, Rocq [2], and Isabelle [3] have transformed mathematical practice by making correctness explicit and mechanized. Community libraries such as Lean's Mathlib [4] now grow at a pace driven both by increasing human contributions and by recent advances in neural theorem proving and autoformalization [5, 6]. This rapid expansion raises multiple concerns about 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. Instructors show their students how to shorten their proofs and structure them better, and the maintainers of Mathlib demand revisions to submissions to improve their robustness and adhere to style guidelines. 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 [7].

Although any two correct formal proofs of a statement equally establish the validity of their conclusion, there are various criteria on which one may be preferred over another. A proof may be more readable, more concise, more modular, or less dependent on obscure library lemmas. When an expert formalizer finishes a proof, they always go back and revise it — aiming, for example, to improve readability and robustness. Unfortunately, the growth rate of formal libraries already exceeds what human reviewers can reliably curate, and this

discrepancy is only expected to widen due to increasing volumes of machine-generated proofs, which even when correct do not carry similar guarantees as to their quality or understandability [8].

Moreover, automated proof optimization is not only useful in its own right, but also for improving AI that can find proofs on its own. At the very least, it provides a form of data augmentation: the limited amount of formal training data is a bottleneck for machine learning, and our methods provide ways of generating additional data automatically. Moreover, our methods also provide a means of optimizing the quality and structure of said training data for an arbitrary metric.

For example, many modern theorem provers (e.g. [6], [9]) rely on the draft-sketch-prove paradigm [10], which generates proofs by first sketching high level proof outlines with "holes" for subproofs/lemmas, and then recursively solving these lemmas. For that purpose, it is useful to have a corpus of proofs written in a structured, modular form that maximizes the number and balance of these subproofs.

1.2 Contributions

We make the following contributions:

- **A formalization of the proof optimization task.** We define proof optimization over a verified proof (c, x, y_0) and an arbitrary user-specified metric μ, separating the problem of finding a proof (neural theorem proving) from the problem of rewriting a proof to be shorter, more structured, or less dependent on external lemmas. We show that neural theorem proving is a strict special case of this framework.

- **ImProver: an agentic system for proof optimization.** We introduce ImProver (Chapters 4–5), a GPT-4o-based agent that rewrites Lean proofs, outperforms a naive GPT-4o baseline by 423–778% across undergraduate-level proofs.

- **ImProver 2: a trained small model for proof optimization.** We introduce ImProver 2 (Chapters 6–7), a pipeline that bootstraps a 7B-parameter model to optimize proofs using a novel IRPO RL pipeline and upgraded neurosymbolic scaffold, outperforming models orders of magnitude larger and extends ImProver to research-level proof optimization.

- **Empirical evidence that formal-structure exposure is the key lever.** Across both systems, gains track neurosymbolic augmentation rather than model scale alone: the scaffold nearly doubles length improvement on DeepSeek-R1 7B before any preference optimization, and also boosts frontier models when applied at inference time.

We argue this supports the thesis that proof optimization is primarily a specialization problem, not a scale problem.

1.3 Thesis Overview

Motivated by the need for more efficient and effective proof optimization, this thesis presents the ImProver and ImProver 2 systems, first exploring a simple yet effective agentic scaffolding approach, and then generalizing to a more powerful bootstrapped RL training pipeline.

ImProver (Chapters 4–5) is the first system for automated proof optimization. It is a large language model agent that rewrites Lean proofs to optimize arbitrary user-defined metrics, incorporating Chain-of-States (CoS) prompting, iterative refinement, and retrieval-augmented generation (RAG). ImProver is capable of rewriting undergraduate, competition, and research-level proofs so that they are substantially shorter and more declarative, outperforming naive GPT-4o by 423–778% depending on the metric. It also demonstrates that proof optimization generalizes neural theorem proving.

ImProver 2 (Chapters 6–7) addresses the core limitation of ImProver: its reliance on expensive, proprietary large language models. Namely, ImProver 2 trains a 7B-parameter small language model (SLM) to optimize Lean proofs via an IRPO-based iterative self-improvement pipeline, as well as extending the agentic scaffolding techniques of ImProver to apply to more complex metrics and research-level datasets. The resulting model outperforms models orders-of-magnitude larger across metrics, and is even competitive with frontier models such as GPT-5-high on particular metrics.

Chapter 2

Related Work

2.1 Formal Theorem Proving and Proof Assistants

Proof assistants such as Lean 4 [1], Rocq [2], and Isabelle [3] have transformed mathematical practice by making the correctness of proofs explicit and mechanized. Community libraries such as Lean's Mathlib [4] now grow at a pace driven by both increasing human contributions and by recent advances in neural theorem proving and autoformalization. This rapid expansion stresses the maintainability, coherence, and long-term usability of libraries, due to proofs that are often heterogeneous in style and clarity. Moreover, the growth rate of formal libraries already exceeds what human reviewers can reliably curate — a discrepancy expected to widen due to increasing volumes of machine-generated proofs, which, even when formally correct, do not carry similar guarantees as to their quality, modularity, or understandability [8].

There is also a rich literature on the varied styles of human formal proofs (e.g., [11, 12]), noting that proofs written for different purposes — readability, brevity, machine-aided search — take structurally different forms. At a high level, proof assistants constitute a sound verifier in a prover-verifier game [13], suggesting that a machine-learning-based prover that interfaces with such a verifier is a natural next step for formal reasoning systems.

2.2 Neural Theorem Proving

A typical approach to developing machine learning provers [14] is to train on a large corpus of mathematical proofs such as Mathlib [15–18]. A model learns from the distribution of proofs in the corpus, such as Mathlib-style proofs. Recently, the AlphaProof [6] system demonstrated solving IMO problems by producing proofs with an arcane, non-human structure and syntax, and subsequent systems have continued to push this frontier [5, 8].

Many systems additionally utilize neurosymbolic augmentation, providing a generative prover model with information gathered from within the proof environment [9, 19, 20]. How-

ever, both formal and informal proofs generated by current LLM-based systems often suffer from stylistic irregularities even when sound, including redundant steps or structures that do not clearly represent the broader logical argument [21].

The miniCTX benchmark [22] evaluates neural provers on theorems drawn from context-rich real-world Lean files, providing a more realistic evaluation setting than miniF2F [23]. ImProver 2 uses the miniCTX-v2 extension of this benchmark as its primary test set, enabling evaluation on genuine research-level mathematics.

2.3 Proof Optimization

Automated proof optimization — the task of rewriting a verified proof to make it better with respect to some criterion while preserving correctness — was introduced as a formal task by ImProver [20] (Chapter 4 of this thesis). ImProver created a system capable of optimizing towards multiple metrics of improvement; however, it relied on general-purpose closed-source models (GPT-4o), leading to substantial deployment costs and limited ability to improve performance beyond these models' baseline.

Subsequent work by [7] focused solely on optimizing the token count of proofs according to a complex tokenizer intended to reduce compilation time. This work does not examine other metrics, leaving out important use cases and limiting its utility to research mathematicians.

Low library quality also 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 [7].

2.4 LLM Agents and Prompting

ImProver draws on a range of techniques from the LLM agent and prompting literature. Chain-of-thought prompting [24] makes intermediate reasoning steps explicit, improving performance on complex tasks; ImProver's Chain-of-States prompting (§4.2) is the formal-proof analogue. Full proof generation from sketches [10, 25] and conditioning on example proofs [10] form the basis of ImProver's retrieval and prompting strategy. Preceding file context [22, 25] improves proof generation quality by providing relevant library context.

2.5 Retrieval-Augmented Generation

ImProver incorporates retrieval-augmented generation (RAG) from both code generation and theorem proving. Documentation retrieval for code generation [26] provides syntactic guidance via the TPiL handbook. Retrieval from the Mathlib library [18, 27] enables the

model to find relevant lemmas to use in its rewrites. ImProver uses Maximum Marginal Relevance [28] to select diverse, relevant examples and documents. ImProver 2 replaces explicit retrieval with context extraction from the proof environment (§6.2.1), which is more targeted and requires no separate retrieval index.

2.6 Preference Optimization and Reinforcement Learning

ImProver 2 builds upon a substantial body of work on preference optimization and reinforcement learning from feedback. Direct Preference Optimization (DPO) [29] provides a stable, offline alternative to RLHF by framing preference learning as a classification problem. Iterative Reasoning Preference Optimization (IRPO) [30] extends DPO with an NLL regularization term that improves training stability and sample efficiency.

Expert iteration [31] bootstraps model capabilities by iteratively generating training data and retraining, enabling self-improvement without external supervision. Group Relative Policy Optimization (GRPO) [32] is an alternative RL policy that learns from groups of candidate solutions; ImProver 2 uses pairwise preference data instead, which provides a denser reward signal for the limited amount of publicly available Lean training data.

Model collapse [33] is a failure mode in which indiscriminate consumption of self-generated training data causes distribution collapse. ImProver 2's replay buffer is specifically designed to prevent this by filtering and combining new and old data across training iterations.

2.7 Neurosymbolic Methods

Neurosymbolic methods combine neural generation with formal symbolic reasoning, leveraging the strengths of both. In the theorem proving context, this typically involves using formal proof checkers to validate generated proofs and extracting symbolic information from the proof environment to guide generation.

ImProver 2 extends neurosymbolic augmentation in three directions. Context extraction (§6.2.1) collects the signatures and documentation of directly referenced lemmas and definitions to inform generation. Chain-of-States prompting (§4.2) provides goal-state traces extracted from Lean's InfoTree structures. Auto-informalization (§6.2) uses natural language translation to provide a higher-level representation of the proof, inspired by the draft-sketch-prove approach [34] and work on auto-formalization [35].

The combination of these three augmentation sources consistently improves performance across both small and frontier models, suggesting that formal structure exposure is complementary to raw model capability.

Chapter 3

Proof Optimization

Given a verified proof of a theorem, a proof optimization agent's objective is to synthesize a semantically equivalent proof that is "better" according to a user-specified objective, while remaining verifiably correct according to the Lean kernel. In this chapter, we formalize this problem and introduce the metrics and evaluation framework used throughout the thesis.

3.1 Setup and Notation

Definition 3.1 (Proof Context, Statement, and Proof)**.** Let $\mathcal{C}$ denote *proof contexts* (imports, local declarations, module metadata, etc.), $\mathcal{X}$ *theorem statements*, and $\mathcal{Y}$ *proofs*. We consider triples $(c, x, y) \in \mathcal{C} \times \mathcal{X} \times \mathcal{Y}$, where y is a purported proof of x in c which may or may not be sound.

Definition 3.2 (Verifier and Valid Proof Set)**.** A *verifier* is a computable function

$$\mathrm{v} : \mathcal{C} \times \mathcal{X} \times \mathcal{Y} \to \{0, 1\},$$

which outputs 1 if y is a syntactically correct and sound proof of x in c. The set of *valid proofs* of x in context c is

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

Two proofs $y, y' \in \mathcal{F}(c, x)$ are said to be *semantically equivalent*: they establish the same mathematical fact, though they may differ arbitrarily in style, structure, or length. We use Lean 4 [1] as our verifier throughout this thesis; its kernel and type-checker provide a strong guarantee of correspondence with a type-theoretic model of modern mathematics.

3.2 Optimization Objective

Definition 3.3 (Optimization Metric)**.** An *optimization metric* is a computable function

$$\mu : \mathcal{C} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}.$$

Original

```
theorem isCoatom_iff [OrderTop A] {K : A} :
    IsCoatom K ↔ K ≠ ⊤ ∧ ∀ H g, K ≤ H → g
      ∉ K → g ∈ H → H = ⊤ := 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]
```

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⟩⟩
```

Original

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

ImProver 2 (dependency)

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

ImProver 2 (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
```

ImProver 2 (modularity)

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

Figure 3.1. ImProver 2 automatically optimizes human-written proofs to reduce explicit dependencies, minimize length, or maximize proof modularity, while maintaining formal correctness.

Definition 3.4 (Improvement Score)**.** Given an input triple (c, x, y_0) with $\mathrm{v}(c, x, y_0) = 1$ and a metric μ, the *improvement score* of a candidate proof y is

$$S_\mu(y, y_0 \mid c, x) := \Delta_\mu(c, x, y, y_0) \cdot \mathrm{v}(c, x, y),$$

where $\Delta_\mu(c, x, y, y_0) := \mu(c, x, y) - \mu(c, x, y_0)$. That is, the improvement score is the gain in metric value when the candidate compiles correctly, and zero otherwise.

The proof optimization problem for a single theorem is then

$$\hat{y} = \arg\max_{y \in \mathcal{Y}} \; S_\mu(y, y_0 \mid c, x). \tag{3.1}$$

Since $S_\mu(y_0, y_0 \mid c, x) = 0$, returning the original proof is always a feasible but trivial solution. Any $\hat{y}$ with $S_\mu > 0$ is a strictly improved proof.

Remark. Proof optimization subsumes neural theorem proving as a special case. If we set y_0 to an empty (incorrect) proof and $\mu := \mu_{\text{comp}}$ (the completion metric, Definition 3.4.1.4), then $S_\mu(y, y_0 \mid c, x) = \mathrm{v}(c, x, y)$, and solving (3.1) reduces to standard theorem proving.

In practice, we approximate (3.1) via language model-based Lean 4 code generation. For a conditional generator G over proofs and an augmentation scaffold Ψ (defined per system

in later chapters), we sample n candidate proofs $y_1, \ldots, y_n \sim G(\cdot \mid \Psi(c, x, y_0), \mu)$ and select the best via the *best-of-n* procedure defined below.

3.3 Datasets

Throughout this thesis, we evaluate proof optimization on Lean 4 datasets spanning a range of mathematical difficulty:

- **Mathematics in Lean (MIL)** [36]: Lean 4 exercises from an introductory textbook on undergraduate-level mathematics. We use a subset of 72 theorems as the primary benchmark for ImProver and its ablations.
- **Compfiles** [37]: Lean 4 formalizations of competition-level mathematics problems. We use a subset of 26 theorems for evaluation.
- **Mathlib** [38]: Research-level theorems from the main Lean 4 mathematics library. We use a subset of 43 theorems drawn from Mathlib proper.
- **miniCTX-v2** [39]: A held-out test set of theorems drawn from a diverse collection of real-world Lean 4 repositories, including Mathlib, HEPLean, PFR, and Carleson. Constructed to test in-context generalization across heterogeneous formal libraries. Used as the primary benchmark for ImProver 2.

3.4 Metrics

We study a collection of metrics designed to be practical, interpretable, and useful for the structural optimization of formal proofs at scale. Each metric is a computable function $\mu : \mathcal{C} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$.

3.4.1 Metric Definitions

3.4.1.1 Length.

Let $|y|_{\text{tac}}$ denote the number of tactic invocations in a proof y. The *length metric* is

$$\mu_{\text{len}}(c, x, y) := -|y|_{\text{tac}}.$$

Shorter proofs are easier to read and maintain and impose less overhead during compilation. Proof shortening ("proof golfing") is a common activity in formal mathematics [4]. This metric is used in both ImProver (Chapter 4) and ImProver 2 (Chapter 6).

3.4.1.2 Declarativity.

Let $|y|_{\text{have}}$ denote the number of explicitly typed `have` tactics in proof y. The *declarative metric* is

$$\mu_{\text{decl}}(c, x, y) := \frac{|y|_{\text{have}}}{|y|_{\text{tac}}}.$$

Declarative proofs [11, 12] are more readable, explicit, and modular: each `have` tactic names and proves an intermediate claim, making the logical structure of the proof explicit. This metric is used in ImProver (Chapter 4).

3.4.1.3 Mixed.

The *mixed metric* penalizes all tactics and rewards declarative ones:

$$\mu_{\text{mix}}(c, x, y) := 5 \cdot |y|_{\text{have}} - |y|_{\text{tac}}.$$

Assigning $+5$ to each `have` tactic and -1 to every tactic means a declarative tactic "pays for" four additional tactics. This jointly optimizes for brevity and structure. Used in ImProver (Chapter 4).

3.4.1.4 Completion.

The *completion metric* measures proof correctness:

$$\mu_{\text{comp}}(c, x, y) := \text{v}(c, x, y) \in \{0, 1\}.$$

This is a degenerate metric: any correct proof achieves the optimum. It is used in ImProver (Chapter 4) to demonstrate empirically that proof optimization generalizes neural theorem proving.

3.4.1.5 Dependencies.

Let $\text{Deps}(c, x, y)$ denote the set of all theorem or lemma names from outside the local file context explicitly referenced in proof y. The *dependency metric* is

$$\mu_{\text{dep}}(c, x, y) := -\left|\text{Deps}(c, x, y)\right|.$$

Minimizing the dependency footprint encourages self-contained proofs that do not require memorizing large numbers of external lemma names, improving readability and long-term maintainability. Introduced in ImProver 2 (Chapter 6).

3.4.1.6 Modularity.

The modularity metric requires substantially more machinery, rooted in Lean's elaboration semantics. At a high level, we count the number of independent subproofs that are useful towards the main proof, as a more rigorous measure of proof structure quality than the declarative metric with explicit guards against reward hacking. The following definitions formalize this intuition.

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 $A_{?m}$, and σ is a partial assignment mapping metavariables to terms. A *goal* is an unassigned metavariable $?m \in \Delta \setminus \mathrm{dom}(\sigma)$, and at elaboration step i Lean maintains a list of *active goals* $\mathcal{A}_i$. Each tactic step τ_i focuses a goal $?m_i \in \mathcal{A}_i$, produces an assignment $\sigma_{i+1}(?m_i) = t_i$, and 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.

Definition 3.5 (Direct Children)**.** The *direct children* of step τ_i are

$$\mathrm{Children}(\tau_i) := \{\text{metavariables occurring free in } t_i\},$$

representing the direct subgoals of the focused goal $?m_i$.

Lean tactics, however, may surface additional obligations that are *not* direct children of $?m_i$. These arise from nested tactic blocks (`have`, `calc`, `by`), from automation introducing auxiliary lemmas, from goal defocus/refocus patterns, and from tactics that internally elaborate their own subproofs. To capture this distinction, let

$$\mathrm{Current}_i := \{?m_i\} \cup \mathrm{Children}(\tau_i), \qquad S_{i+1} := S_i \cup \mathrm{Children}(\tau_i),$$

so that S_i records all metavariables that have previously appeared as children.

Definition 3.6 (Spawned Goals)**.** The *spawned goals* of step τ_i are

$$\mathrm{Spawned}(\tau_i) := \big(\mathrm{Current}_i \setminus \mathrm{Children}(\tau_i)\big) \setminus S_i.$$

Intuitively, these are goals that first appear at step i, are not logical subgoals of the focused goal, and correspond to independent subproof obligations introduced by tactics such as `have` and `calc`.

Proof graph over steps. We represent a proof as a directed graph over steps rather than metavariables. Let steps be indexed by $i = 1, \ldots, T$. Define a *normal edge* $i \to j$ if the goal solved at step j is a direct child of step i, and a *spawned edge* $i \rightsquigarrow j$ if the goal solved at step j was spawned at step i. This yields a labeled directed forest

$$\mathcal{T}(y) = (V,\ E_{\text{normal}},\ E_{\text{spawned}}),$$

with $E_{\text{spawned}} \subseteq E_{\text{normal}}$. We refer to this structure as the *proof tree*, extracted from Lean's `InfoTree`.

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 (i) a base target hash; (ii) a sorted list of hypothesis type hashes (proof-relevant hypotheses only); (iii) a set of *sequent variant hashes*, obtained by progressively discharging $\forall$-binders and implications; and (iv) 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.

Duplicate and wrapper detection. A spawned goal is considered a *duplicate* and discarded if either (i) its sequent variant hash matches any previously seen goal (global duplicate), or (ii) 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 several classes of reward hacking — trivial $\forall$-introductions, restatements of the parent goal, and redundant lemma spawning.

Nontriviality filter. Let T_g be the subtree of steps rooted at a spawned goal g. We require $|T_g| > 2$ to ensure that the goal is not discharged by a single automation step. Empirically, this excludes trivial goals solvable by tactics such as `simp`, `tauto`, `linarith`, `ring`, `aesop`, and `grind`.

Effectiveness via fixed-point semantics. Let each spawned goal g introduce a set of proof variables $\mathsf{Intro}(g)$ corresponding to hypotheses unavailable in the parent context. We say that a spawned subtree rooted at g is *effective* if its introduced hypotheses are used in the main (non-spawned) proof, or in another spawned subtree that is itself effective. Formally, define the monotone operator Φ on sets of spawned roots by

$$\Phi(S) := \big\{\, g \;\big|\; \mathsf{Intro}(g) \text{ is used outside all spawned subtrees,} \\ \text{or } \exists\, g' \in S,\, g' \neq g,\, \mathsf{Intro}(g) \text{ used in subtree } g' \big\}.$$

Definition 3.7 (Effective Spawned Goals)**.** The set of *effective spawned goals* $\mathcal{E}(y)$ is the least fixed point of Φ, restricted to spawned roots that satisfy the nontriviality, no-duplicates, and no-wrappers guard conditions above. This is computed by standard fixed-point iteration and is guaranteed to terminate, since the set of spawned roots is finite.

Definition 3.8 (Modularity Metric)**.** Given a verified proof y of (c, x), the *modularity metric* is

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

Modular proofs are easier to read, maintain, and reuse. The guard conditions operate at the kernel level on term-level constructions rather than on tactic syntax, which substantially narrows the reward-hacking surface relative to purely syntactic counts like the declarative metric. Figure Figure 3.2 illustrates the proof-tree representation used to compute μ_{mod} on an example proof.

Example Proof Tree

```
lemma KD5_weakerThan_KD45 : (Hilbert.KD5 α) ≤ₛ (Hilbert.KD45 α) := by
  have h₁ : (LO.Modal.Hilbert.KD5 α).axioms ⊆ (LO.Modal.Hilbert.KD45 α).axioms → (Hilbert.KD5 α) ≤ₛ
     (Hilbert.KD45 α) := by
    intro h
    apply normal_weakerThan_of_subset
    <;> assumption
  have h₂ : (LO.Modal.Hilbert.KD5 α).axioms ⊆ (LO.Modal.Hilbert.KD45 α).axioms := by
    intro φ hφ
    cases' hφ with hφ hφ
    <;> simp_all [LO.Modal.Hilbert.KD5, LO.Modal.Hilbert.KD45]
    <;> aesop
  exact h₁ h₂
```

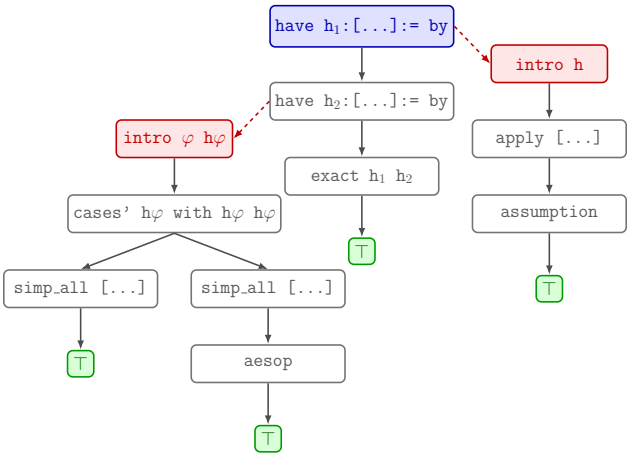


Figure 3.2. Example Lean proof (top) and its generated proof tree (bottom). Note the root node (in blue), and the two effective spawned goals coming from h_1 and h_2 (in red).

3.4.2 Metric Summary and Degenerate Solutions

Table Table 3.1 summarizes which metrics are used by each system.

3.4.3 Failure Modes and the Role of the User

A risk shared by every metric is the possibility of *degenerate solutions*: proofs that score well on the metric without corresponding to its intuitive purpose. We make no claim that the specific metrics in Table Table 3.1 are optimal, canonical, or universally preferred proxies for proof quality. Our systems are tools for optimizing an arbitrary user-specified metric μ; the metrics studied in this thesis are illustrative examples of what can be optimized, not prescriptions for what should be. The choice of metric, and any claim that optimizing it yields better proofs in some external sense, is the responsibility of the user. With that caveat, it is instructive to enumerate the characteristic failure modes of each metric we study.

Table 3.1. Metrics used in ImProver and ImProver 2.

Metric	ImProver	ImProver 2
Length	✓	✓
Declarative	✓	
Mixed	✓	
Completion	✓	
Dependencies		✓
Modularity		✓

3.4.3.1 Length.

The length metric rewards any reduction in tactic count, including rewrites that replace structured proofs with single-line proof terms. Such rewrites are correct and shorter but may obscure proof structure and readability. However, in practice. Length should therefore be interpreted as a toy metric to gauge the effectiveness of our agent, as a measure of syntactic compactness rather than readability.

3.4.3.2 Declarativity.

The declarativity metric is a syntactic count: the ratio of `have` tactics to total tactics. It is vulnerable to `have`-padding, in which a model inserts nominally-typed intermediate statements that play no deductive role. ImProver mitigates this in practice by prompting with human-written declarative examples, but the metric itself offers no kernel-level protection. This limitation is addressed by the modularity metric introduced in ImProver 2.

3.4.3.3 Mixed.

The mixed metric combines length and declarativity additively. It inherits both failure modes — aggressive shortening and `have`-padding — but lessens the impact of each, as such ancillary have statements would decrease the length reward and such proof-term replacement would eliminate such have statements.

3.4.3.4 Completion.

Completion is optimal at any correct proof, so any rewrite that simply preserves correctness is a global optimum. This is the intended behaviour (completion is used only to demonstrate that proof optimization subsumes neural theorem proving), but it should not be interpreted as a quality metric.

3.4.3.5 Dependencies.

The dependency metric measures the explicit dependency footprint — the set of externally named lemmas referenced in the proof text — not the logical dependency structure of the proof. A consequence is that a rewrite from `rw [Nat.add_comm, Nat.add_assoc]` to `omega` scores strictly better, even though the two proofs have identical logical dependencies. This is not a degenerate solution, as it is aligned directly with Mathlib's style guidance [40] for the sake of prevent brittle proofs.

3.4.3.6 Modularity.

The modularity metric's guard conditions (nontriviality, no duplicates, no wrappers, fixed-point effectiveness) explicitly rule out several canonical forms of reward hacking at the kernel level: inserting duplicate `have`s, solving spawned goals with a single automation step, or wrapping the parent goal in a trivially restated sub-obligation. However, this does not guarantee that every modularity-increasing rewrite is mathematically readable. Modularity is therefore best understood as a structural measure aimed at proofs written in a draft-sketch-prove or lemma-decomposition style [10], not a validated proxy for human-maintainer preference. We do not claim that maintainers prefer modularity-optimized proofs, and a human-preference study is identified as important future work.

Across all of these metrics, the consistent pattern is that sharper structural definitions (as in modularity) reduce the reward-hacking surface relative to purely syntactic counts (as in declarativity), but no metric we study is immune. Our position is that robust metric design is itself a research problem: we provide a framework for optimizing any computable μ, and demonstrate it on metrics for which the failure modes are well-characterized, without claiming that these metrics are the correct ones to optimize in every context.

3.5 Evaluation Framework

Since proof optimization is a new task with no established evaluation protocol, we define here the evaluation framework used by both systems in this thesis.

3.5.1 Single-Theorem Evaluation

Given a generator G, an augmentation scaffold Ψ, a metric μ, and an input triple (c, x, y_0), we sample n candidate proofs $y_1, \ldots, y_n \sim G(\cdot \mid \Psi(c, x, y_0), \mu)$ and select the best candidate via the best-of-n procedure.

Definition 3.9 (Best-of-n Selection)**.** Given n candidate proofs $y_0, y_1, \ldots, y_n$, define the *best-of-n output* $\hat{y}$ as

$$\hat{y} = \arg\max_{0 \le i \le n} S_\mu(y_i, y_0 \mid c, x).$$

Since $S_\mu(y_0, y_0 \mid c, x) = 0$, the original proof is always included as a fallback: $\hat{y}$ is never worse than y_0. The expected value $\mathbb{E}[\max_i S_\mu(y_i, y_0 \mid c, x)]$ is nondecreasing in n, so increasing the sample budget monotonically improves the expected selected score.

3.5.2 Dataset-Level Evaluation

Given a test dataset $\mathcal{D} = \{(c_i, x_i, y_{0,i})\}_{i=1}^N$, we track three aggregate quantities. Let $\hat{y}_i$ be the best-of-n output for the i-th theorem.

Definition 3.10 (Mean Improvement Score)**.** The *mean improvement score* over $\mathcal{D}$ is

$$\overline{S}_\mu(\mathcal{D}) := \frac{1}{N} \sum_{i=1}^{N} S_\mu(\hat{y}_i, y_{0,i} \mid c_i, x_i).$$

This is the primary metric throughout this thesis. It jointly rewards generating a compiling proof and achieving a large improvement in μ, and cannot be inflated by returning the original proof (which scores 0).

Definition 3.11 (Compilation Accuracy)**.** The *compilation accuracy* is

$$\mathcal{A}(\mathcal{D}) := \frac{1}{N} \sum_{i=1}^{N} \mathrm{v}(c_i, x_i, \hat{y}_i).$$

This measures the fraction of theorems for which the model produced at least one compiling output.

Definition 3.12 (Improved Accuracy)**.** The *improved accuracy* is

$$\mathcal{A}^+_\mu(\mathcal{D}) := \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}\big[S_\mu(\hat{y}_i, y_{0,i} \mid c_i, x_i) > 0\big].$$

This measures the fraction of theorems for which the model produced a compiling proof with strictly positive metric improvement.

Remark. These three metrics are complementary. $\mathcal{A}$ measures stability (correctness preservation); $\mathcal{A}^+_\mu$ measures whether the system is solving the optimization problem rather than merely preserving compilability; and $\overline{S}_\mu$ captures both by weighting each success by its magnitude. In reported results, $\overline{S}_\mu$ is always the primary selection criterion.

Chapter 4

ImProver

The first question posed by proof optimization is a basic feasibility question: can a language model be prompted to meaningfully rewrite a verified Lean proof into a better one? The challenge is that proof optimization requires the model to simultaneously understand the mathematical content of the proof (to avoid introducing errors) and the structural properties being optimized (to improve the metric). Neither standard prompting nor naive generation is well-suited to this: standard prompting lacks proof-state context, and naive sampling produces too many invalid proofs to be practically useful.

ImProver addresses these challenges by constructing an *agentic scaffold* around a black-box language model generator. Rather than issuing a single prompt and accepting the first output, ImProver augments the generation process with structured proof-state context, iterative error correction, and example-guided metric optimization. This chapter describes the design and components of this scaffold.

4.1 Motivation and Overview

The fundamental difficulty of proof optimization as a generation task is that Lean proofs are sensitive to both syntactic and semantic validity: any change risks introducing a compilation error, and most perturbations of a valid proof are invalid. A language model generating proof rewrites must therefore navigate an extremely sparse feasibility landscape while simultaneously improving a metric.

A natural first approach is to condition a generator G on the theorem statement, context, and original proof, and request an improved proof directly. In practice, this baseline approach is heavily limited by the raw capabilities of the fixed generator G. Moreover, the model lacks access to intermediate proof states, which encode crucial information about the evolution of hypotheses and goals during the proof. Second, the model receives no corrective feedback when it produces invalid outputs. Both limitations can be addressed by augmenting the generation context.

ImProver formalizes this augmentation as an agentic scaffold Ψ_{Imp}. For a given input triple (c, x, y_0) and metric μ, ImProver samples candidates as

$$y_1, \ldots, y_n \sim G\big(\cdot \mid \Psi_{\text{Imp}}(c, x, y_0), \mu\big),$$

and selects $\hat{y}$ via best-of-n (Definition 3.9). The scaffold Ψ_{Imp} conditions G on the following components:

- **Chain-of-States (CoS) annotation**: the original proof y_0 annotated with intermediate Lean proof states (§4.2);
- **Output formatting**: structured output constraints that guide the syntactic form of the generated proof (§4.3);
- **Sampling and refinement**: past history of generated instances and their evaluations (§4.4);
- **Retrieval-augmented generation (RAG)**: relevant examples and documentation retrieved from Mathlib and TPiL (§4.5).

4.2 Chain-of-States Prompting

Typical formal proofs are a sequence of tactics and *states* showing the hypotheses and goals at each step. The intermediate states often contain valuable information — e.g., an expression after simplification — that is not visible from the tactic text alone. To allow the model to reason about these intermediate goals and hypotheses, we use Lean metaprogramming to automatically annotate each proof state as a comment prior to each tactic.

We refer to this method as *Chain-of-States* (CoS) prompting, since it makes intermediate states explicit, in analogy with how chain-of-thought prompting [24] makes intermediate reasoning steps explicit.

States are extracted directly and symbolically from the underlying Lean compilation steps. In the compiler's elaboration and evaluation stages — where parsed theorem code is converted into concrete syntax trees (`Syntax` objects) and abstract syntax trees (`Expr` objects) — we convert the CST and AST output objects into proof data in the form of proof trees (`Lean.Elab.InfoTree`), as defined in Definition 3.5. These proof trees contain detailed tactic-by-tactic information about proof state modification, metavariable context, and proof correctness [41]. After extraction, we annotate the proof text with the intermediate states as comments.

Figure Figure 4.1 shows an example. This explicit proof-state context aims to help the generator construct more optimized proofs by exposing the symbolic structure of what each

Without Chain-of-States

```
example : s ∩ t ∪ s ∩ u ⊆ s ∩ (t ∪ u)  := by
  rintro x (⟨xs, xt⟩ | ⟨xs, xu⟩)
  · use xs; left; exact xt
  . use xs; right; exact xu
```

With Chain-of-States

```
example : s ∩ t ∪ s ∩ u ⊆ s ∩ (t ∪ u)  := by
  rintro x (⟨xs, xt⟩ | ⟨xs, xu⟩)
  /-
  case inl.intro
  α : Type u_1
  s t u : Set α
  x : α
  xs : x ∈ s
  xt : x ∈ t
  ⊢ x ∈ s ∩ (t ∪ u)
  case inr.intro
  α : Type u_1
  s t u : Set α
  x : α
  xs : x ∈ s
  xu : x ∈ u
  ⊢ x ∈ s ∩ (t ∪ u)
  -/
  · use xs; left; exact xt
  /-
  Goals Solved!
  -/
  . use xs; right; exact xu
  /-
  Goals Solved!
  -/
```

Figure 4.1. A Lean proof (left) with Chain-of-States prompting annotations (right). Intermediate proof states are inserted as comments before each tactic, giving the model symbolic information about the evolving goal state.

tactic does. The ablation study in Chapter 5 confirms that CoS is the most impactful single component of ImProver: enabling CoS nearly doubles the improvement score for declarativity optimization and substantially improves accuracy across all metrics.

4.2.0.1 Formal definition.

More formally, given a verified proof y_0, let its tactic steps be indexed by $i = 1, \ldots, T$. From the `InfoTree` we obtain, for each step, the pair $(\texttt{goalsBefore}_i, \texttt{goalsAfter}_i)$ along with the pretty-printed tactic τ_i. The Chain-of-States transformation is then

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

serialized into the proof as comments adjacent to each τ_i. This turns hidden kernel states into explicit cues the model can condition on.

ImProver 2 (Chapter 6) reuses this same $\Psi_{\cos}$ transformation as one of three neurosymbolic channels, integrating it with additional context-extraction and auto-informalization channels built on the same InfoTree infrastructure.

4.3 Output Formatting

LLM outputs often contain ancillary and syntactically invalid content surrounding the actual proof. Additionally, by applying structure to the output format, we can encourage the model to reason explicitly about proof structure. We introduce two additional output formats alongside the standard `string` output:

- `string list`: the model outputs a tactic sequence as a list of strings, one tactic per element. This isolates individual tactic steps, making parsing and validation straightforward.
- `string tree`: the model outputs the proof as a tree of strings, encouraging explicit reasoning about the hierarchical structure of the proof.

The ablation study in Chapter 5 finds that `string list` output with CoS enabled achieves the best mean improvement score.

4.4 Sampling and Refinement

Beyond best-of-n selection (Definition 3.9), ImProver introduces error correction via iterative refinement, and compound methods combining both.

4.4.1 Error Correction and Refinement

Inspired by self-debugging techniques in code generation [42, 43], ImProver iteratively refines its outputs. The refinement process is parameterized by n (number of iterations) and `prev_num` (number of previous iterations' data to forward). Each iteration receives information from the last `prev_num` iterations, including the input, output, metric score, correctness, and error messages. This allows the model to observe its own mistakes and attempt targeted corrections.

4.4.2 Compound Methods

Compound prompt functions nest best-of-n and refinement within one another:

- `best_of_n(refinement(`m`),` n`)`: runs best-of-n where each of the n calls is an m-step refinement;
- `refinement(best_of_n(`m`),` n`)`: runs an n-step refinement where each iteration is a best-of-m LLM call.

Both methods use a total of mn LLM calls. The ablation study finds that a 5-step refinement with best-of-3 per iteration is the optimal combination for ImProver.

4.5 Retrieval-Augmented Generation

ImProver uses Maximum Marginal Relevance (MMR) [28] retrieval-augmented generation to select relevant examples and documents. Two retrieval types are used:

- **Example retrieval.** For a user-specified k, example retrieval selects the k most relevant examples of proof optimization for the specific metric being optimized. These examples are drawn from a curated set of human-written proof rewrites and provided as multi-shot demonstrations. In the ablation study, $k = 10$ examples was found optimal. Multi-shot prompting helps the model understand the intent of the metric concretely and mitigates the risk of degenerate solutions (see Section 3.4.2).

- **Document retrieval.** Document retrieval extracts information using MMR from two fixed vector databases:

  - The *Theorem Proving in Lean* (TPiL) handbook [44], containing syntax guides and tactic explanations;
  - The Mathlib mathematics library [38], containing thousands of theorems and lemmas.

  The Mathlib retriever finds the top k documents with the highest MMR score against the current theorem. The TPiL retriever finds the top k documents with the highest MMR score against the current theorem and all current error messages. This helps the model find relevant lemmas and correct its own syntax errors.

Chapter 5

ImProver Experiments

We test ImProver on rewriting real-world undergraduate theorems, competition problems, and research-level mathematics, comparing against the base GPT-4o and GPT-4o-mini models. We examine the optimization capabilities of ImProver for the length and declarative metrics, studying the effectiveness in maintaining proof correctness while making proofs more concise or more declarative in structure.

5.1 Experimental Setup

Our experimentation is split into three stages. We first perform ablation testing on the ImProver parameters (§4.1) to find the optimal configuration with respect to correctness and metric improvement. We then evaluate this optimal parameter combination on datasets of varying complexity. Lastly, we benchmark ImProver on neural theorem proving (NTP) tasks to empirically confirm that proof optimization generalizes NTP.

We evaluate on subsets of Mathematics in Lean (MIL, 72 theorems), Compfiles (26 theorems), and Mathlib (43 theorems), as described in Section 3.3. Ablations are performed on a subset of MIL.

Our base generator is GPT-4o [45] (`gpt-4o-2024-08-06`). Since no prior methods exist for automated proof optimization, we use a prompted GPT-4o without the ImProver scaffold as our baseline. Both baseline and ImProver receive a prompt containing metric instructions, the theorem statement, context, and original proof; ImProver additionally augments this prompt with CoS, output formatting, retrieval, and sampling as described in Chapter 4.

We use the evaluation framework defined in Section 3.5: mean improvement score $\overline{S}_\mu(\mathcal{D})$ (primary), compilation accuracy $\mathcal{A}(\mathcal{D})$, and improved accuracy $\mathcal{A}^+_\mu(\mathcal{D})$. The mean improvement score is our primary selection criterion, as it jointly rewards correctness and meaningful metric gains and cannot be inflated by returning the original proof.

5.2 Ablation Studies

We perform ablation studies on a subset of MIL with the length metric fixed, using a factorial testing design that evaluates groups of parameters sequentially, holding the best configuration from each group fixed before varying the next.

5.2.1 Setup.

We define six testing groups:

- *GPT-4o-mini vs. GPT-4o.* This group varies the base model, with `string` output and no other features.
- *Output format and CoS.* We evaluate all output format types (`string`, `string list`, `string tree`) crossed with CoS enabled/disabled, with GPT-4o fixed.
- *Example retrieval.* We vary the number of retrieved examples ($k = 0, 3, 5, 7, 10$) with the best output format and CoS configuration held fixed.
- *Sampling method.* We evaluate best-of-n and refinement for fixed $n = 5$, along with variants of refinement (forwarding all vs. most recent previous iterations, keeping the best vs. most recent output), with model, format, CoS, and examples held fixed.
- *n and model.* We evaluate $n = 3, 5, 7, 10, 15$ for GPT-4o and GPT-4o-mini, and $n = 20$ for GPT-4o-mini, with all other parameters held at their optimal values.
- *Compound methods and RAG.* We evaluate compound sampling (`refinement(best_of_`m'`,` m`)` and `best_of_`m'`(refinement(`m`))`, with mm' equal to the optimal n from the previous group) with and without document retrieval.

For each group, we select the parameter combination maximizing the improvement score. This criterion rewards both generation accuracy and metric improvement magnitude, and avoids the pitfalls of accuracy alone (which can be gamed by returning the input) or raw metric magnitude alone (which ignores incorrect generations).

5.2.2 Results.

As shown in Table Table 5.1, the optimal configuration is GPT-4o with `string list` output, CoS enabled, 10 retrieved examples, 5-step refinement with best-of-3 per iteration, and document retrieval. Changing any one of these parameters leads to a reduction in the improvement score.

Table 5.1. Ablation results. Each cell in the ablation tests shows `best` / `worst`, which are the `best` and `worst` parameter combinations in the test group.

	Improvement	Accuracy	Improved Acc.
GPT-4o-mini	0	3.62%	0%
GPT-4o	7.03	35.77%	15.33%
+ Output and CoS	8.04 / 6.31	64.96% / 44.53%	21.17% / 16.06%
+ Example Retrieval	9.34 / 5.67	63.5% / 67.15%	21.9% / 16.79%
+ Sampling Method	15.35 / 9.34	83.21% / 63.5%	36.5% / 21.9%
+ n and Model	23.51 / 3.65	89.47% / 78.95%	45.61% / 8.77%
+ Combos and RAG	34.88 / 28.25	60.61% / 84.38%	54.55% / 53.12%
ImProver	**34.88**	**100%**	**54.55%**

Table 5.2. CoS Declarativity Ablation results.

	Improvement	Accuracy	Improved Acc.
GPT-4o	4.97	37.5%	12.5%
ImProver, CoS Disabled	9.23	100.0%	28.12%
ImProver	**16.69**	**100.0%**	**46.88%**

5.2.2.1 Declarativity and CoS ablation.

We additionally examine the isolated effect of CoS on declarativity optimization, since we hypothesize that CoS has a particularly high impact on declarative-style rewrites: the proof states embedded by CoS directly reveal the local hypotheses and goals, which are what declarative `have` tactics name explicitly. Table Table 5.2 confirms this: enabling CoS nearly doubles the improvement score and substantially raises improved accuracy, indicating that the symbolic state information is critical to generating genuinely declarative structure.

5.2.2.2 Syntax guidance ablation.

We examine the effect of syntax guidance (error message forwarding) on performance. Without syntax guidance, the ability to improve the metric score is approximately unchanged, but improved accuracy drops by 13%. This confirms that syntax guidance improves the model's ability to generate correct outputs — as expected — but the large performance gap over GPT-4o is not solely due to error correction; CoS, example retrieval, and retrieval are the primary drivers.

Table 5.3. Syntax Guidance Ablation results.

	Improvement	Accuracy	Improved Acc.
GPT-4o	11.00	42.42%	21.21%
ImProver, No Syntax Guidance	23.42	100.0%	46.88%
ImProver	**28.94**	**100.0%**	**59.38%**

Table 5.4. Average Proof optimization results.

Metric	Model	Improvement	Accuracy	Improved Acc.
Length	GPT-4o	3.7	26.36%	8.31%
	ImProver	**20.96**	**100.0%**	**35.44%**
Declarativity	GPT-4o	2.21	18.75%	6.13 %
	ImProver	**9.34**	**100.0%**	**24.56%**
Mixed	GPT-4o	3.51	14.70%	5.11%
	ImProver	**27.31**	**100.0%**	**30.55**

5.3 Main Results

ImProver optimizes proofs across all settings. From Tables Table 5.6, Table 5.7, and Table 5.8, ImProver consistently outperforms the GPT-4o baseline on all datasets for all three metrics. Table Table 5.4 shows that ImProver outperforms GPT-4o by 566% on the length improvement score, 423% on declarativity, and 778% on the mixed metric, aggregated across all datasets.

Length optimization. For the length metric, ImProver guarantees a correct output (accuracy is 100%, as ImProver falls back to the original proof if no valid shorter proof is found). In 35.44% of cases, it generates a strictly shorter proof, and the mean improvement score reaches 20.96 compared to 3.7 for GPT-4o. The baseline achieves both lower accuracy (26.36%) and lower improved accuracy (8.31%), indicating that without the scaffold, the model frequently fails to produce correct rewrites at all.

Declarativity optimization. Declarativity optimization shows a similar pattern: ImProver outperforms GPT-4o by 423% in improvement score. Both models show lower accuracy and improved accuracy for declarativity than length, indicating that generating correct declarative rewrites is harder than length reduction. We attribute this to the fact that declarative rewrites require the model to reorganize proof structure rather than simply eliminating steps, which demands a deeper understanding of the proof's logical flow. CoS is particularly important here, as shown in the ablation.

Table 5.5. Average proof optimization results

	Length		**Declarativity**		**Mixed**	
	GPT-4o	ImProver	GPT-4o	ImProver	GPT-4o	ImProver
Improvement	3.7	20.96	2.21	9.34	3.51	27.31
Accuracy	26.36%	100.0%	18.75%	100.0%	14.70%	100.0%
Improved Acc.	8.31%	35.44%	6.13%	24.56%	5.11%	30.55%

Table 5.6. MIL Proof optimization results.

Metric	Model	Improvement	Accuracy	Improved Acc.
Length	GPT-4o	6.25	37.5%	14.42%
	ImProver	**30.54**	**100.0%**	**50.0%**
Declarativity	GPT-4o	4.18	28.85%	11.54%
	ImProver	**13.45**	**100.0%**	**34.21%**
Mixed	GPT-4o	3.70	13.51%	0.0%
	ImProver	**43.55**	**100.0%**	**45.94%**

Mixed optimization. For the mixed metric (combining length and declarativity), ImProver outperforms GPT-4o by 778%. The improvements and accuracy trends mirror those from the individual metrics, demonstrating that ImProver scales its optimization capabilities to arbitrarily-defined composite metrics without requiring metric-specific engineering beyond the prompt and examples.

Difficulty varies by dataset. The improvement score decreases with dataset difficulty: undergraduate MIL theorems show the highest expected improvement, Compfiles less, and Mathlib theorems the least. This trend holds for both GPT-4o and ImProver. However, ImProver maintains 100% compilation accuracy across all three datasets while GPT-4o's accuracy drops sharply (from 37.5% on MIL to 16.67% on Mathlib for length). This suggests that ImProver's bottleneck is the base model's ability to generate a correct rewrite at all — once the scaffold can secure a valid output, optimization quality degrades much more gracefully with difficulty.

5.4 Neural Theorem Proving Evaluation

We evaluate ImProver's NTP performance using the completion metric on subsets of MIL with empty input proofs. Table Table 5.9 shows that ImProver substantially outperforms GPT-4o across all subsets: 45.45% vs. 18.18% on MIL-C04, 33.33% vs. 25% on MIL-C08, and 39.13% vs. 21.73% overall. On MiniF2F, ImProver achieves 16.39% compared to GPT-

Table 5.7. Compfiles Proof optimization results.

Metric	Model	Improvement	Accuracy	Improved Acc.
Length	GPT-4o	2.75	11.54%	5.13%
	ImProver	**18.86**	**100.0%**	**34.62%**
Declarativity	GPT-4o	0.39	14.1%	1.28%
	ImProver	**5.74**	**100.0%**	**19.23%**
Mixed	GPT-4o	3.96	26.9%	20.0%
	ImProver	**20.60**	**100.0%**	**23.07%**

Table 5.8. Mathlib Proof optimization results.

Metric	Model	Improvement	Accuracy	Improved Acc.
Length	GPT-4o	0.0	16.67%	0.0%
	ImProver	**6.19**	**100.0%**	**11.54%**
Declarativity	GPT-4o	0.0	4.65%	0.0%
	ImProver	**4.63**	**100.0%**	**11.63%**
Mixed	GPT-4o	2.92	9.30%	4.65%
	ImProver	**4.16**	**100.0%**	**9.30%**

4o's 9.02%.

Specialized NTP models such as Lean Expert Iteration [46] outperform ImProver on MiniF2F (34.5% vs. 16.39%), but these systems are specifically trained on large corpora of Lean code and designed exclusively for theorem proving rather than proof optimization. The purpose of this experiment is to confirm empirically that proof optimization as formulated in Chapter 3 is a strict generalization of NTP, and that the ImProver scaffold achieves meaningful performance on this harder task without any NTP-specific training.

5.5 Qualitative Examples

To complement the aggregate metrics, we examine representative rewrites produced by ImProver on each of the three metrics. These examples illustrate the qualitative behaviours that the scaffold encourages; the worked Chain-of-States annotation in Figure Figure 4.1 in Chapter 4 shows the intermediate symbolic state information that ImProver receives during generation. Additional examples across MIL, Compfiles, and Mathlib are collected in Appendix B.

Table 5.9. Proof generation results. Each cell shows percent accuracy.

	MIL-C04 Pass@15	MIL-C08 Pass@15	MIL Pass@15	MiniF2F-test Pass@8
GPT-4o	18.18%	25.00%	21.73%	9.02%
ImProver	**45.45%**	**33.33%**	**39.13%**	16.39%
Lean Expert Iteration	—	—	—	**34.5%**

5.5.0.1 Length optimization (MIL).

Figure Figure 5.1 optimizes an exercise solution from MIL Chapter 8 Section 1 (group theory) for length. ImProver restructures the two `rintro` branches into single-line chains using semicolons and replaces a `calc` block with a proof term `(congr_arg ψ hz).trans hy`, shortening the proof from 9 to 7 tactic invocations without altering the overall case split.

Original (human-written)

```
example (φ : G →* H) (ψ : H →* K) (S : Subgroup G)
    :
    map (ψ.comp φ) S = map ψ (S.map φ)  := by
  ext x
  simp only [mem_map]
  constructor
  · rintro ⟨y, y_in, hy⟩
    exact ⟨φ y, ⟨y, y_in, rfl⟩, hy⟩
  · rintro ⟨y, ⟨z, z_in, hz⟩, hy⟩
    use z, z_in
    calc ψ.comp φ z = ψ (φ z) := rfl
    _               = ψ y := by congr
```

ImProver (length-optimized)

```
example (φ : G →* H) (ψ : H →* K) (S : Subgroup G)
    :
    map (ψ.comp φ) S = map ψ (S.map φ)   := by
  ext x
  simp only [mem_map]
  constructor
  rintro ⟨y, y_in, hy⟩; exact ⟨φ y, ⟨y, y_in, rfl⟩, hy⟩
  rintro ⟨y, ⟨z, z_in, hz⟩, hy⟩; exact ⟨z, z_in,
    (congr_arg ψ hz).trans hy⟩
```

Figure 5.1. Length optimization: a group-theoretic lemma from MIL Chapter 8 Section 1.

5.5.0.2 Declarativity optimization (Compfiles).

Figure Figure 5.2 shows a lemma from IMO 2019 Problem 1 optimized for declarativity. The original proof threads a chain of rewrites through a single `refine` block. ImProver exposes two named intermediate hypotheses — `linear_property` and `g_smul` — that correspond to reusable algebraic facts about g, and then completes the proof by instantiating those lemmas at specific arguments. The resulting proof is longer but its logical structure is explicit: each `have` is a candidate for extraction into a standalone lemma.

5.5.0.3 Mixed optimization (MIL).

Figure Figure 5.3 optimizes a group-theoretic lemma from MIL for the mixed length-/declarativity metric. ImProver collapses the case analysis inside `suffices` into a single

Original (human-written)

```
lemma additive_to_int_linear (f : ℤ → ℤ) (h: ∀ (x y
    : ℤ), f (x + y) = f x + f y):
  ∃ c, ∀ a, f a = c * a  := by
 let g := AddMonoidHom.toIntLinearMap <|
    AddMonoidHom.mk' f h
 refine ⟨f 1, fun a => ?_⟩
 change g a = g 1 * a
 rw [mul_comm, ← smul_eq_mul, ←
    LinearMap.map_smul, smul_eq_mul, mul_one]
```

ImProver (declarativity-optimized)

```
lemma additive_to_int_linear (f : ℤ → ℤ) (h: ∀ (x y
    : ℤ), f (x + y) = f x + f y):
  ∃ c, ∀ a, f a = c * a   := by
 let g := AddMonoidHom.toIntLinearMap <|
    AddMonoidHom.mk' f h
 have linear_property : ∀ a, f a = g a := by
   intro a
   rfl
 have g_smul : ∀ a, g a = g 1 * a := by
   intro a
   rw [mul_comm, ← smul_eq_mul, ←
    LinearMap.map_smul, smul_eq_mul, mul_one]
 refine ⟨f 1, fun a => ?_⟩
 have f_eq_g : f a = g a := linear_property a
 have g_a_eq : g a = g 1 * a := g_smul a
 rw [f_eq_g, linear_property 1, g_a_eq]
```

Figure 5.2. Declarativity optimization: a lemma from IMO 2019 P1 (Compfiles).

declarative `have` that proves the biconditional directly via anonymous constructor syntax, and then discharges the goal with a single `simpa ... using this`. The rewrite simultaneously reduces tactic count and introduces a named intermediate fact — exactly the behaviour the mixed metric targets.

Original (human-written)

```
lemma eq_bot_iff_card {G : Type*} [Group G] {H :
    Subgroup G} [Fintype H] :
   H = ⊥ ↔ card H = 1  := by
 suffices (∀ x ∈ H, x = 1) ↔ ∃ x ∈ H, ∀ a ∈ H, a =
    x by
   simpa [eq_bot_iff_forall, card_eq_one_iff]
 constructor
 · intro h
   use 1, H.one_mem
 · rintro ⟨y, -, hy'⟩ x hx
   calc x = y := hy' x hx
   _      = 1 := (hy' 1 H.one_mem).symm
```

ImProver (mix-optimized)

```
lemma eq_bot_iff_card {G : Type*} [Group G] {H :
    Subgroup G} [Fintype H] :
   H = ⊥ ↔ card H = 1     := by
 have : (∀ x ∈ H, x = 1) ↔ ∃ x ∈ H, ∀ a ∈ H, a = x
    :=
   ⟨λ h => ⟨1, H.one_mem, h⟩, λ ⟨y, _, hy'⟩ x hx =>
    (hy' 1 H.one_mem).symm ▷ hy' x hx⟩
 simpa [eq_bot_iff_forall, card_eq_one_iff] using
   this
```

Figure 5.3. Mixed length/declarativity optimization: a lemma from MIL Chapter 8 Section 1.

These three rewrites — a length shortening by structural compression, a declarativity rewrite that names reusable intermediates, and a mixed rewrite that achieves both simultaneously — illustrate the qualitative range of transformations the ImProver scaffold produces. They also highlight that optimization for a given metric can produce genuinely different proof styles on the same theorem, rather than just mechanical shortenings or `have`-padding.

5.6 Limitations

ImProver establishes that agentic scaffolding enables meaningful proof optimization, but several limitations restrict its practical scope.

Cost. Each theorem requires numerous API calls with document retrieval, making library-scale deployment expensive. At current API pricing, running ImProver over all of Mathlib would be prohibitively costly.

Closed-source dependence. ImProver's performance is tightly coupled to GPT-4o, a closed-source, proprietary model. This limits reproducibility, prohibits local deployment, and means that the system's behavior is subject to model deprecation.

Small evaluation datasets. We evaluate only on $72 + 26 + 43$ theorems on relatively toy datasets, providing statistically limited signal. Namely, it is not evident that such datasets are representative of standard research repositories in formal mathematics, lacking many of the inter-file dependencies and novel datastructures/API as true mathematics research has.

Weak metrics. The declarativity metric is a coarse proxy for the true goal of generating more structured proofs, but is vulnurable to gaming by superficial rewrites that increase the number of `have` statements without meaningfully improving readability. Moreover, our evaluated set of metrics is sparse; although length and declarativity are reasonable proxies for proof efficiency and structure, we do not define any metrics that lend themselves to a notion of maintainability.

These limitations motivate the development of ImProver 2, described in the following chapters, which replaces the expensive agentic approach with a specialized small model trained to perform proof optimization directly.

Chapter 6

ImProver 2

We present ImProver 2, a training pipeline for proof optimization towards a specific metric. Its core purpose is to bootstrap the capabilities of small language models (SLMs) through repeated applications of training, generation, and evaluation/filtering of generated data for use in further iterations. To do so, it utilizes three core components: a reinforcement learning-based training stage (described in 6.3.2), a replay buffer to balance old and newly generated data (6.3.1), and multiple sources of neurosymbolic augmentation to boost generation capabilities (6.2). Each are described in detail below.

6.1 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 Figure 6.1):

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 (6.2), providing the model with neurosymbolic augmentation (6.2) 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 6.3.1).

3. The model G_t is trained to obtain G_{t+1} (6.3.2). Our reinforcement learning policy of choice, known as Iterative Reasoning Preference Optimization or IRPO [30], 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 6.3.1).

4. G_{t+1} is evaluated with n samples on the test set, again similarly to 6.2. The improvement in metric score of the new proofs over the originals is calculated.

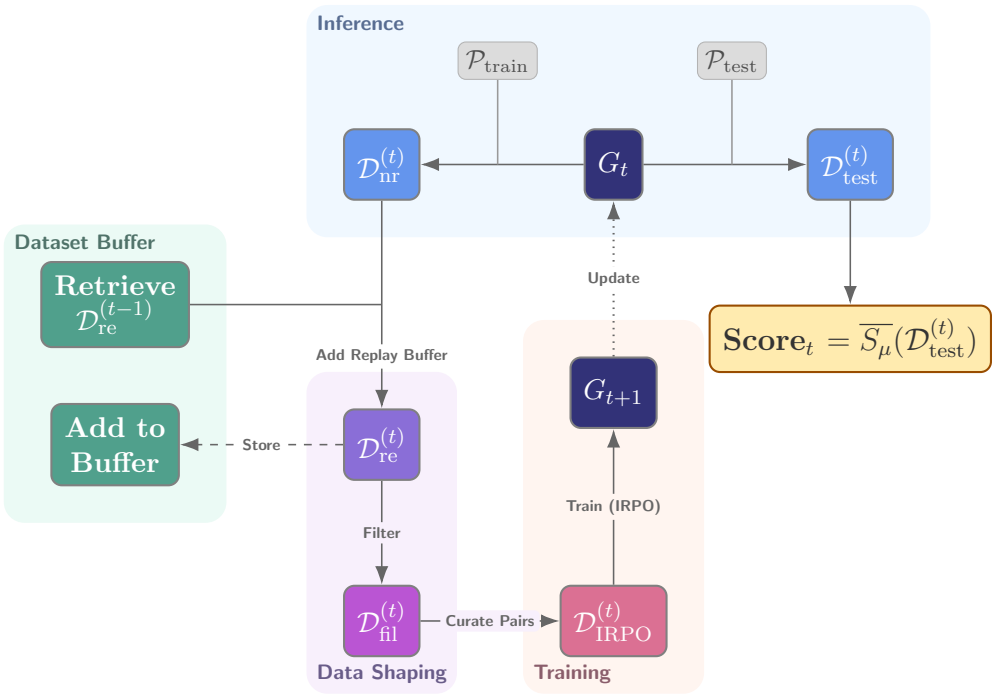


Figure 6.1. **ImProver² training loop.** The diagram illustrates the iterative process of generation, retrieval, filtering, and training. Node colors represent the evolution of data from initial sampling (Blue) through processing (Purple) to training (Magenta).

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

6.2 Neurosymbolic Augmentation

A central feature of the ImProver 2 pipeline is its ability to generate its own training data for future iterations. This is performed by running inference on the current model $G^{(t)}$, providing it with the theorem statement and the original proof, and requesting an improved version; this is repeated n times per theorem. Additionally, the model's generation capabilities are augmented with information from the proof environment as described in the following sections.

Namly, theseformal proof environments provide substantial opportunities to obtain relevant information about a proof. We prompt our language model not only with serialized information on the original theorem statement x, proof y_0, and metric μ, but also with additional neurosymbolic context that exposes structure and dependencies at both a formal and informal level. This augmentation 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.

6.2.1 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.

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., definition/theorem/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).

6.2.1.1 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 $\mathsf{touch}(x, y_0)$ 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 then the subgraph

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

optionally restricted by the import DAG to a budgeted neighborhood. We then serialize each element of S with metadata describing its object type (e.g. lemma, definition), along with 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 with raw file size, yielding a compact, minimal, and deterministic bundle of proof context that is read-only with respect to the program state.

6.2.2 Chain-of-States (CoS)

We again employ the Chain-of-States technique described in 4.2 to expose the internal proof states of the original proof y_0 to the model. As shown in ImProver, his provides a richer signal about the structure of the proof and the effect of each tactic than is available from the original proof text alone, which may be more difficult for the model to parse and understand.

6.2.3 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 [34]. 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.

The details of the prompting and generation process for each of these channels are described in Appendix C.3.

6.3 Training

We build upon the Iterative Reasoning Preference Optimization algorithm [30] to improve proof optimization capabilities during each round of training. However, our architecture differs from the original implementation in several ways. First, we use the improvement in score (i.e. $\mu(s, x, y_1) > \mu(s, x, y_2)$) in addition to binary correctness to rank candidates and form preference pairs, creating a denser reward signal. We furthermore maintain a replay buffer that mixes newly-generated solutions with old data to enable stable improvement over many training rounds. Finally, unlike the original IRPO construction, where training is done on both the reasoning trace as well as the final output, we mask the reasoning trace during training and train solely on the final output.

6.3.1 Datasets and Replay Buffer

Indiscriminate consumption of self-generated training data has been shown to harm performance in language models, collapsing the tails of their distribution [33]. To avoid this issue and enable self-improvement, we introduce a novel replay buffer design that filters new samples, combines them with existing training data, and sorts them based on the degree of their improvement for use during training.

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

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

6.3.1.1 Improvement rate and eligibility.

For problem T, we define its improvement rate at iteration j by

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

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 $j < t$).

6.3.1.2 Replay buffer construction.

We build the post-replay dataset in two main steps.

1. *Mark.* 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 the highest-π_T items (i.e. the "easiest") until

the replay fraction is ρ (capped at $\pi_{\max}$).

(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 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 $\widetilde{\mathcal{Y}}_T^{(t-1)}$. Then we case on `mode`:

   - `join`: set $\widetilde{\mathcal{Y}}_T^{(t)} = \mathcal{Y}_T^{(t)} \cup \widetilde{\mathcal{Y}}_T^{(t-1)}$, deduplicated by normalized proof text. This increases candidate diversity for IRPO.
   - `replace`: set $\widetilde{\mathcal{Y}}_T^{(t)} = \widetilde{\mathcal{Y}}_T^{(t-1)}$, overwriting the current candidates with the previous iteration's.
   - `mark`: skip this step (no candidate-level splice). In other words, set $\widetilde{\mathcal{Y}}_T^{(t)} = \mathcal{Y}_T^{(t)}$.

With this, we obtain the replay dataset

$$\mathcal{D}_{\text{re}}^{(t)} = \Big\{ \big(c_T, x_T, y_{T,0},\ \widetilde{\mathcal{Y}}_T^{(t)}\big) \ \Big|\ T \in \mathcal{P}_{\text{train}} \Big\}.$$

6.3.1.3 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 $\pi_T^{(t)} > \pi_{\max}$. That is, we define

$$\widetilde{\mathcal{P}}^{(t)} = \mathcal{P}_{\text{train}} \setminus \{T \mid \pi_T^{(t)} > \pi_{\max}\}.$$

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

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

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. The finalized filtered post-replay dataset is

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

6.3.1.4 IRPO Dataset

As IRPO is a preference-based training method, $\mathcal{D}_{\text{fil}}^{(t)}$ is partitioned into a set of preference pairs. By mixing and matching many potential proofs of a single theorem, we glean a higher quantity of data points per theorem than would be available through group-based RL policies such as GRPO [32], compensating for the limited amount of publicly available high-quality Lean training data. Additionally, by creating pairs of compiling/non-compiling examples as well as better/worse metric score, we balance the two parallel goals of learning to write correct Lean syntax and actually making improvements to proofs; this balance is dictated by the hyperparameters W and L in Algorithm 1.

Concretely, this process is parameterized by $W, L \in \mathbb{N}$, which control the number of winners and losers sampled per problem. For a given problem T in $\widetilde{\mathcal{P}}^{(t)}$, we first deduplicate $W_T^{(t)}$ by equal scores $\widehat{s}_{T,i} = \widehat{s}_{T,j}$ and deduplicate $L_T^{(t)}$ by string equality. We then form two families of preference pairs:

$$\mathfrak{P}_T^{\ell \to w} = \{(b, g) : b \in L_T^{(t)},\ g \in W_T^{(t)}\}, \qquad \mathfrak{P}_T^{w \to w} = \{(g', g) : g, g' \in W_T^{(t)},\ \widehat{s}_{T,g} > \widehat{s}_{T,g'}\}.$$

The IRPO training dataset is then

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

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

6.3.2 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}.

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

Chapter 7

ImProver 2 Experiments

7.1 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.

7.1.1 Dataset and split

We utilize Lean proofs from several open-source projects formalizing research-level mathematics across multiple domains [4, 47–53]. 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 believe miniCTX-v2 represents a reasonable approximation of the problems that would be encountered during deployment for optimizing human-written research-level math. Although we hypothesize that ImProver 2 could perform similar optimization in other domains, such as long machine-generated proofs, we omit testing on these domains as they fall outside the scope of the aforementioned application.

7.1.2 Evaluation protocol

For all main results, we evaluate with best@16 sampling (Definition 3.9) and report the mean improvement score $\overline{S}_\mu(\mathcal{D}_{\text{test}})$ (Definition 3.10) for $\mu \in \{\mu_{\text{len}}, \mu_{\text{dep}}, \mu_{\text{mod}}\}$. We also report compilation accuracy $\mathcal{A}(\mathcal{D}_{\text{test}})$ (Definition 3.11) and improved accuracy $\mathcal{A}^+_\mu(\mathcal{D}_{\text{test}})$ (Definition 3.12). We operate with a base model of $G_0 =$ `DeepSeek-R1-Distill-Qwen-7B` [54]. Hyperparameters and configuration are described in C.

7.1.3 Systems compared

Table 7.1. Inference cost comparison, based on public API pricing at time of submission, and given as the Input/Output cost per 1M tokens.

Model	Type	Cost
DeepSeek-R1-7B	Open-weight	Locally Hosted
DeepSeek-R1-14B	Open-weight	Locally Hosted
DeepSeek-R1-671B	Open-weight	\$0.70/ \$2.50
GPT-4o	API	\$2.50 / \$10
GPT-5-nano	API	\$0.05 / \$0.40
GPT-5-mini	API	\$0.25 / \$2
GPT-oss-120B	Open-weight	Locally Hosted
GPT-5-chat	API	\$1.25 /\$10
GPT-5 (High)	API	\$1.25 / \$10

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`. Current inference costs for each system is displayed in Table Table 7.1.

We also evaluate all generators with and without the neurosymbolic scaffold 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.

7.2 Main Results

Tables Table 7.2 and Table 7.3 show that ImProver 2 substantially improves over its 7B base generator on all three objectives, leads all evaluated unscaffolded systems on modularity, and is competitive with frontier models on dependency and length.

After IRPO training, ImProver 2 achieves mean best@16 improvements of 0.330 on length, 0.143 on modularity, and 0.206 on dependency, compared to 0.118, 0.003, and 0.050 for the untrained `DeepSeek-R1 7B` baseline. These correspond to roughly 2.8× on length, 4.1× on dependency, and a large absolute gain on modularity (0.003 → 0.143). We note that the dependency multiplier reflects both the effectiveness of the pipeline and the low absolute baseline of the untrained model. Together, these gains indicate that iterative self-improvement combined with neurosymbolic conditioning yields substantial improvements even from a 7B base model.

Table 7.2. **Frontier Evaluations.** Comparison with frontier models and prior proof optimization systems. Mean improvement at best@16 on MiniCTX-v2

Model	Length	Mod.	Dep.
GPT-4o	0.336	0.034	0.050
GPT-oss-120B	0.321	0.075	0.181
GPT-5-nano	0.087	0.065	0.106
GPT-5-mini	0.330	0.109	0.203
GPT-5-chat	0.346	0.118	0.046
GPT-5-high	**0.660**	0.120	**0.208**
ImProver	0.355	0.088	0.047
ImProver 2	0.330	**0.143**	0.206

Table 7.3. **Intra-Family Evaluations.** Intra-family comparison across DeepSeek-R1 model scales. Mean improvement at best@16 on MiniCTX-v2 for all three metrics.

Model	Length	Mod.	Dep.
DeepSeek-R1 7B	0.118	0.003	0.050
DeepSeek-R1 14B	0.140	0.037	0.093
DeepSeek-R1 671B	0.308	0.055	0.153
ImProver 2	**0.330**	**0.143**	**0.206**

These gains are not confined to a single objective. While length improves steadily with model scale, ImProver 2 also delivers strong gains on the more structural objectives of dependency and modularity, which require more than local proof compression. In particular, its dependency score of 0.206 exceeds both the 14B and 671B DeepSeek baselines, suggesting that for structural proof refactoring, task-specific training can matter at least as much as raw parameter count.

Comparison with frontier systems further sharpens this picture. On modularity, ImProver 2 leads all unscaffolded systems evaluated at 0.143, ahead of the next-best frontier models (`GPT-5` high reasoning at 0.120, `GPT-5-chat` at 0.118), suggesting that iterative task-specific training can catch up to raw model scaling for learning to decompose proofs into reusable subproof structure. Similarly, on dependency, ImProver 2 is competitive with the strongest frontier systems: it achieves 0.206, essentially tied with `GPT-5-high` (0.208) and ahead of `GPT-5-mini` (0.203). On length, ImProver 2 (0.330) is competitive with mid-tier frontier models, matching `GPT-5-mini` (0.330) and exceeding `GPT-oss-120B` (0.321), while trailing only `GPT-5-high` (0.660) — a full-size high-reasoning variant operating at substantially greater inference cost — by a significant margin. The prior `ImProver` system (0.355) also leads on length, consistent with its use of an agentic prompting strategy on a substantially

larger `GPT-4o` base model, specifically optimized for proof shortening.

Overall, the results suggest that task-specific iterative training is particularly effective for structural refactoring: ImProver 2 leads all systems on modularity and is competitive with the strongest frontier models on dependency, while remaining within range of mid-tier frontier models on length — all from just a completely bootstrapped training pipeline on a 7B base model.

Table 7.4. **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

7.3 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 $\mathcal{A}_\mu^+$, where $\mathcal{A}_\mu^+$ 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 $\mathcal{A}_\mu^+$ faster than $\mathcal{A}$. This is especially visible in the iteration study (Table Table 7.6). 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. After training, improved accuracy rises sharply, peaking at 0.106 in iteration 2, while compilation accuracy drops to 0.464. This reflects a central trade-off 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 $\mathcal{A}_\mu^+$ as complementary: $\mathcal{A}$ measures stability, while $\mathcal{A}_\mu^+$ more directly captures whether the system is solving the optimization problem rather than merely preserving compilability.

Table 7.5. **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 $\mathcal{A}_\mu^+$ 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 7.6. **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 $\mathcal{A}_\mu^+$ 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

7.4 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 Figure 7.1 shows that within the `DeepSeek-R1` model family, larger models generally achieve higher mean improvement under a fixed prompting and 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.

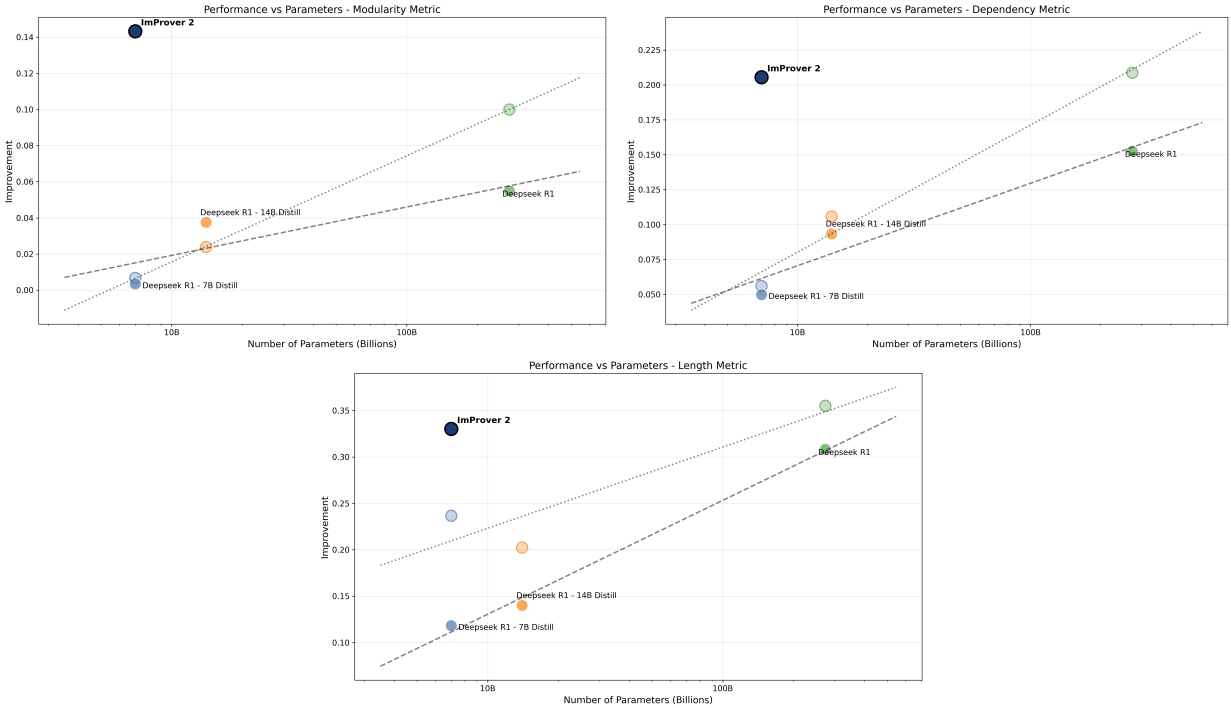


Figure 7.1. Effect of parameter count on model performance on mean improvement at best@16 across all three metrics, with ImProver 2 marked.

7.5 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), demonstrating that iterative task-specific training surpasses generalist frontier models on the modularity objective, even at much larger parameter counts. 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 `GPT-oss-120B` (0.321), trailing only `GPT-5-high` (0.660), which represents a qualitatively different operating point in terms of inference cost.

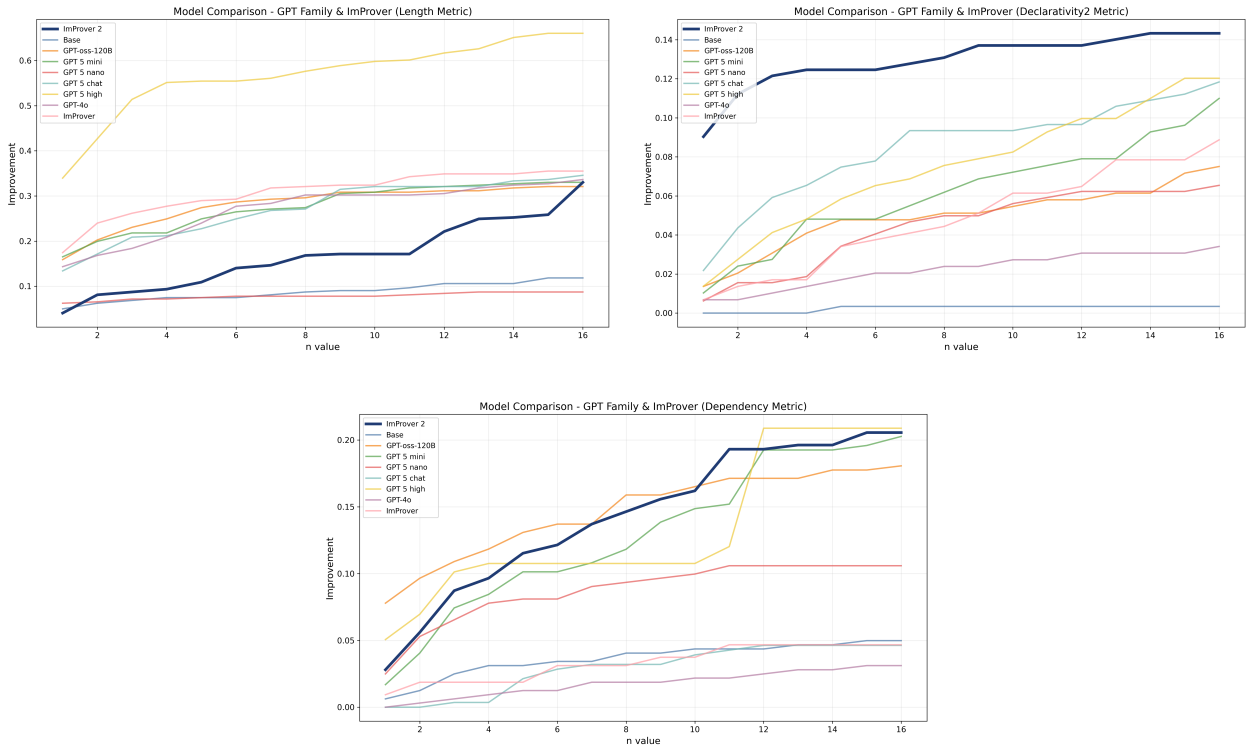


Figure 7.2. Comparison of ImProver 2 against frontier GPT-based models and ImProver, evaluated on mean improvement at best@n on all metrics, from $n = 1$ to $n = 16$.

Relative to large open-weight baselines, ImProver 2 is also strong. It matches or exceeds `GPT-oss-120B` on all three metrics despite starting from a much 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 computational gap; nonetheless, `GPT-oss-120B` is a significantly more powerful base model compared to the `DeepSeek-R1-Distill-Qwen-7B` base that is used by ImProver 2 (as shown by the comparative evaluation of [55],[54]). As such, these results demonstrate that metric-specific iterative self-improvement can substitute for large increases in both parameter count and

Table 7.7. **Scaffold Evaluations.** Effect of neurosymbolic scaffolding. Mean improvement at best@16 with and without the scaffold Ψ, across model families and metrics.

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

base model capability for this task.

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 an agentic 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), demonstrating that on these more complex metrics, iterative structural supervision learns refactoring capabilities that agentic prompting alone does not reliably acquire.

7.6 Effect of Neurosymbolic Scaffolding

A central claim of this work is that proof optimization is limited not only by model size, but by how well the model is conditioned on the structure of the formal task. Table Table 7.7 strongly supports this claim. Across nearly all evaluated generators and metrics, adding the scaffold Ψ improves mean best@16 performance, often by a large margin.

The effect is especially striking on length. For `DeepSeek-R1 7B`, scaffolding nearly doubles performance from 0.118 to 0.236 even before iterative training. For stronger frontier models the gains are larger still in absolute terms: `GPT-5-mini` rises from 0.330 to 0.632, `GPT-5-chat` from 0.346 to 0.576, and `GPT-oss-120B` from 0.321 to 0.508. 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.

This pattern extends even to the strongest evaluated system: `GPT-5-high` improves from 0.660 to 0.875 on length under scaffolding, suggesting that Lean-aware conditioning provides complementary information even to large high-reasoning models.

The dependency metric shows a similar pattern. Most models benefit substantially from scaffolding, with especially large gains for `GPT-oss-120B` (0.181 $\rightarrow$ 0.406), `GPT-5-mini` (0.203 $\rightarrow$ 0.267), and `GPT-5-chat` (0.046 $\rightarrow$ 0.087). Modularity gains are smaller and less uniform, but still mostly positive. The main exception is `DeepSeek-R1 14B`, where modularity decreases slightly under scaffolding (0.037 $\rightarrow$ 0.024). Given the broader pattern across

both open and closed models, we view this as an outlier rather than evidence against the scaffold.

Overall, the scaffold appears to expand the set of useful rewrites the model can reliably attempt. 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.

7.6.1 Neurosymbolic Ablation

In addition to the above results, we study the effect of each of our sources of neurosymbolic scaffolding on model performance, finding that each one added increases length-metric performance (Figure Figure 7.3), though with diminishing returns.

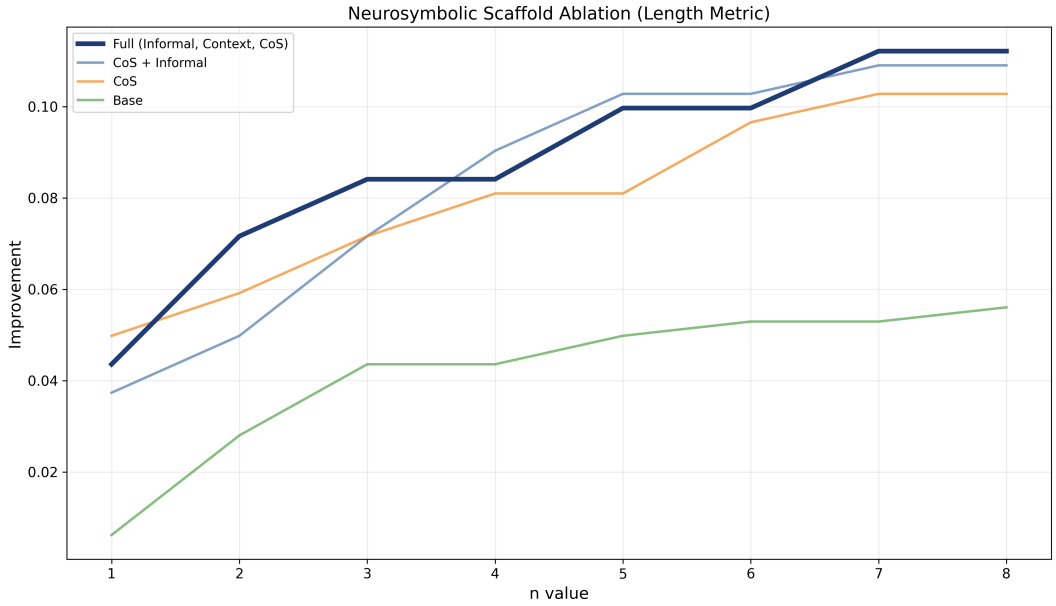


Figure 7.3. Effect of neurosymbolic augmentation on base model performance (`DeepSeek-R1-Distill-Qwen-7B`) on the length metric, vs. number of samples generated. Each source of augmentation shows noticeable improvement in average score on some n values.

7.7 Training and Iteration

Table Table 7.4 and Fig. Figure 7.4 show rapid gains in the first few rounds of self-improvement, followed by saturation. For all three metrics, performance improves sharply

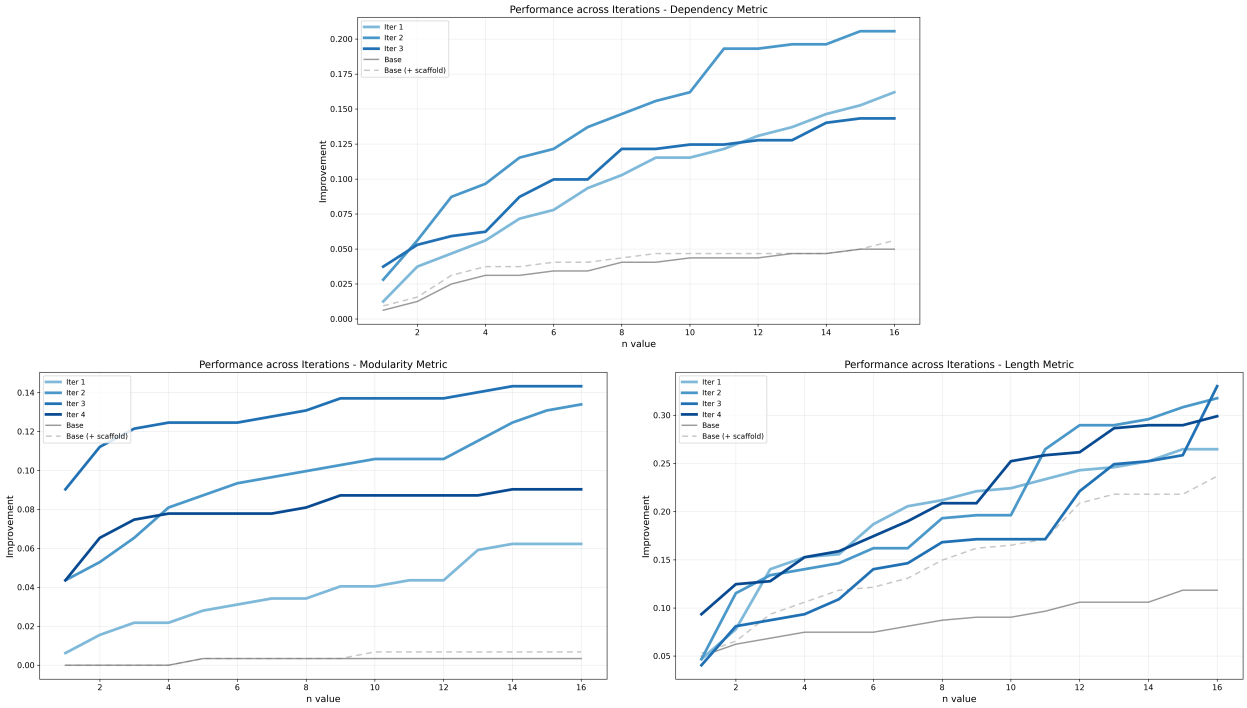


Figure 7.4. Performance of ImProver 2 as a function of sample budget across training iterations. Gains are largest in early iterations, with saturation by iterations 2–3 and mild regression thereafter.

from the base model to the first two or three IRPO iterations. Length rises from 0.118 to 0.330, modularity from 0.003 to 0.143, and dependency from 0.050 to a peak of 0.206 at iteration 2. These gains indicate 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.

Improvement is not indefinite, however. After the peak, later iterations plateau or regress slightly: dependency falls from 0.206 at iteration 2 to 0.165 at iteration 3, and both length and modularity decline at iteration 4. We interpret this as saturation rather than instability. 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.

These results also justify selecting checkpoints separately for each metric. Dependency peaks earlier, while length and modularity benefit from one additional round. This difference is plausible given the nature of the objectives: dependency minimization often admits relatively direct simplifications, whereas modularity improvements require more global restructuring and therefore appear to be learned more gradually.

7.8 Per-repository heterogeneity

Table 7.8. **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

Performance varies substantially across repositories (Table Table 7.8), 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 than definitive, since they likely reflect both stylistic differences and variation in theorem composition across projects.

7.9 Qualitative optimization behaviors

Beyond the aggregate metrics, ImProver 2 learns distinct families of refactoring behaviors that align with the intended objective. Figure Figure 3.1 in Chapter 3 previews three such behaviors on a representative example per metric, and Appendix D collects additional rewrites across the miniCTX-v2 test repositories.

7.9.1 Length minimization

The model frequently merges local tactic sequences into shorter automation-heavy proofs, replacing explicit intermediate steps with more compact combinations of simplification and search. The `summerCommute_jacobi_ofCrAnListF` rewrite from HepLean (Figure Figure D.3) is illustrative: the original proof case-splits on eight combinations of bosonic/fermionic status and discharges each with a bespoke `simp only` lemma list, whereas ImProver 2 collapses the case split into a single `simp_all` over a shared lemma set followed by `by_cases` chained with `<;>` combinators, reducing the tactic count by 19. The `mem_cross_iff` rewrite from ConNF

(Figure Figure D.4) similarly replaces a nested `rintro/obtain` skeleton with a two-line `simp_all; aesop` script, eliminating eight tactics while remaining semantically equivalent.

7.9.2 Dependency minimization

The model rewrites proofs to rely on more local reasoning, generic automation, or broader lemmas, reducing the number of explicitly named external facts. In `isCoatom_iff` (Figure Figure D.1) ImProver 2 replaces a hand-tuned `simp_rw` chain that names five Mathlib lemmas with a combinator-driven `constructor; simp_all; tauto` pattern that names only two, eliminating three explicit dependencies. This behaviour also often replaces brittle, version-specific `simp only` or `rw` chains with broader automation tactics such as `simp`, which is consistent with Mathlib's own style guidance [40]. We emphasise that the dependency metric measures the explicit dependency footprint of a proof rather than every dimension of proof quality: a tactic like `simp` still relies implicitly on its simp set, and the metric should be read accordingly.

7.9.3 Modularity maximization

The model introduces intermediate claims and clearer proof boundaries, turning flat tactic scripts into proofs with more reusable substructure. The `KD_weakerThan_KDB` rewrite in modal logic (Figure Figure D.5) transforms a single-line `aesop` finisher into a proof that first states the reduction lemma $\texttt{h}_1$ ("a subset of axioms implies weaker-than"), then proves the required axiom subset $\texttt{h}_2$, and finally applies the two.

The `hilbertPoly_eq_zero_of_le_rootMultiplicity_one` rewrite from Mathlib (Figure Figure D.6) likewise pulls the two branches of a `by_cases` apart into named sub-lemmas before recombining them. Both produce nontrivial spawned goals that satisfy the modularity metric's duplicate-detection and wrapper-detection filters defined in Chapter 3.

We note that the modularity metric is a structural proxy motivated by draft-sketch-prove workflows and lemma decomposition in neural provers; it is not a validated proxy for human maintainer preference, and a study of whether maintainers prefer modularity-optimized proofs remains an important direction for future work. With that caveat, these behaviors are qualitatively important because they show that the model is not merely exploiting superficial quirks of the metrics. Instead, the resulting proofs often reflect recognizable mathematical refactoring patterns of the kind a human maintainer might intentionally perform, which supports the interpretation of the quantitative results: ImProver 2 is learning reusable structural transformations, rather than memorising a small set of syntactic shortcuts.

Chapter 8

Conclusion

This thesis has presented two systems for automated proof optimization in Lean 4, developed as a natural two-act progression from an agentic prompting approach to a specialized trained model.

8.1 Summary of Contributions

ImProver introduced the proof optimization task and demonstrated its feasibility using agentic prompting around a frontier language model. Its contributions span the formulation and the machinery needed to make optimization practical: it formalizes proof optimization as a task with a flexible metric framework, introduces Chain-of-States prompting as a mechanism for annotating intermediate proof states as comments in the generation context, and adds error correction and refinement loops that iteratively rewrite generated proofs. Retrieval-augmented generation over the Mathlib library and the *Theorem Proving in Lean* handbook supplies the model with relevant examples and API documentation. Together these components show that proof optimization strictly generalizes neural theorem proving, and that agentic prompting alone is already capable of producing meaningful gains on length, declarativity, and mixed objectives.

ImProver 2 addresses the cost and scalability limitations of the first system by replacing the frontier-model agent with a small, specialized model trained to optimize proofs directly. Its contributions are both methodological and empirical: an iterative self-improvement pipeline built on iterative reasoning preference optimization (IRPO), stabilized by a novel replay buffer that prevents model collapse across rounds; a neurosymbolic augmentation strategy (context extraction, Chain-of-States, auto-informalization) that consistently improves performance across model sizes; and two novel metrics, modularity and dependencies, that target the structural quality and external-lemma footprint of a proof rather than merely its length. The resulting 7B-parameter model outperforms models orders of magnitude larger on structural metrics and matches frontier models on dependencies.

8.2 Limitations

Both systems have limitations that scope the claims made above and that reviewers of the underlying papers have surfaced.

ImProver is limited by its per-theorem cost (many GPT-4o calls with document retrieval) and its dependence on a closed-source frontier model, which together make library-scale deployment impractical at current API pricing and harm reproducibility. The evaluation datasets are also small ($72 + 26 + 43$ theorems), which bounds the statistical strength of the reported gains.

ImProver 2 has its own distinct set of limitations. The modularity and dependency metrics are structural proxies, not optimized for human-maintainer preferences. The system is also a single-step generator with no internal refinement loop, so when a rewrite fails to compile or regresses on the metric the model has no recovery mechanism analogous to ImProver's error-correction pass. However, this was a concious design choice in ImProver2, as we intend to focus on studying the dynamics of bootstrapped training before the additional complexity of an agentic system like ImProver.

8.3 Future Directions

The limitations above motivate a cluster of follow-ups. Most directly, the single-step generation in ImProver 2 can be replaced with a trained iterative-revision policy that internalizes ImProver's refinement loop into the weights, so that the model itself decides when to stop, when to revise, and how to incorporate compiler feedback. On the metric side, the structural proxies can be grounded against human preference through a small, controlled study in which formal mathematicians compare optimized rewrites against their originals; such a study is also a natural testbed for richer, more subjective metrics such as LLM-as-judge readability scoring.

Scaling out from per-theorem optimization to library-level optimization is the largest remaining challenge. Maintaining consistency across inter-dependent rewrites — so that a modular refactor of one theorem does not break downstream proofs that referenced its intermediate names — is both an engineering problem and a research problem, and it is the regime in which the practical value of automated proof optimization is likely to be largest. Finally, a core motivation for this line of work is improving the training data on which neural theorem provers are bootstrapped; empirically showing that provers trained on ImProver- or ImProver 2-optimized corpora outperform those trained on unoptimized data would confirm the training-data-quality hypothesis that underlies both systems.

8.4 Closing Remarks

Formal mathematics libraries are growing faster than human maintainers can curate them. This thesis shows that automated proof optimization — rewriting verified proofs to be shorter, more modular, or less dependent on external lemmas — is a feasible and learnable task. The central insight is simple but powerful: giving a language model access to the formal structure of the proof environment, whether at inference time through Chain-of-States prompting or at training time through neurosymbolic augmentation, enables far more substantive proof transformations than prompting alone.

ImProver and ImProver 2 together establish proof optimization as a well-defined task with practical applications to library maintenance, training data quality, and the broader agenda of making formal mathematics more accessible to both humans and machines. We hope these systems serve as a foundation for further work on AI-assisted formal mathematics.

Bibliography

[1] 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.

[2] The Coq Development Team. The coq proof assistant, September 2024. URL https://doi.org/10.5281/zenodo.14542673.

[3] 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.

[4] 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.

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

[6] AlphaProof and AlphaGeometry Teams. AI achieves silver-medal standard solving international mathematical olympiad problems. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/, 2024.

[7] 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.

[8] 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.

[9] 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.

[10] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=SMa9EAovKMC.

[11] Serge Autexier and Dominik Dietrich. A tactic language for declarative proofs. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 99–114, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[12] Freek Wiedijk. Formal proof sketches. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, pages 378–393, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24849-1.

[13] Cem Anil, Guodong Zhang, Yuhuai Wu, and Roger Grosse. Learning to give checkable answers with prover-verifier games, 2021. URL https://arxiv.org/abs/2108.12099.

[14] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

[15] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=rpxJc9j04U.

[16] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning, 2022.

[17] Guillaume Lample, Timothee Lacroix, Marie anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=J4pX8Q8cxHH.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Jiewen Hu, Thomas Zhu, and Sean Welleck. minictx: Neural theorem proving with (long-)contexts, 2024. URL https://arxiv.org/abs/2408.03350.

[23] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

[24] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_VjQlMeSB_J.

[25] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models, 2023.

[26] Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ZTCxT2t2Ru.

[27] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving, 2024.

[28] Jaime Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98, page 335–336, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130155. doi: 10.1145/290941.291025. URL https://doi.org/10.1145/290941.291025.

[29] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2023.

[30] 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.

[31] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *Advances in Neural Information Processing Systems*, 30, 2017.

[32] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu nd Junxiao Song, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

[33] Ilia 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.

[34] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timoth'ee 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.

[35] 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, Le Hong 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/.

[36] leanprover-community. mathematics_in_lean. https://github.com/leanprover-community/mathematics_in_lean, 2024.

[37] David Renshaw. compfiles. https://github.com/dwrensha/compfiles, 2024.

[38] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, POPL '20. ACM, January 2020. doi: 10.1145/3372885.3373824. URL http://dx.doi.org/10.1145/3372885.3373824.

[39] Jiewen Hu, Thomas Zhu, and Sean Welleck. miniCTX: Neural theorem proving with (long-)contexts. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=KIgaAqEFHW.

[40] The mathlib Community. Library style guidelines. https://leanprover-community.github.io/contribute/style.html, 2024. Accessed: 2026-04-18.

[41] Kim Morrison. lean-training-data. https://github.com/kim-em/lean-training-data, 2024.

[42] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=S37hOerQLB.

[43] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL https://arxiv.org/abs/2304.05128.

[44] Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. *Theorem Proving in Lean 4*. Carnegie Mellon University, 2024. Available at https://leanprover.github.io/theorem_proving_in_lean4/.

[45] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry

Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

[46] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. *CoRR*, abs/2202.01344, 2022. URL https://arxiv.org/abs/2202.01344.

[47] Joseph Tooby-Smith. Heplean: 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.

[48] Terrance Tao. Pfr blueprint, 2026. URL https://teorth.github.io/pfr/blueprint.pdf.

[49] 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 Diedering, Jim Portegies, and Joris Roos. Formalizing carleson's theorem in lean, 2026. URL https://florisvandoorn.com/carleson/.

[50] Sky Wilshaw. New foundations is consistent, 2026. URL https://leanprover-community.github.io/con-nf/print/print.pdf.

[51] Kevin Buzzard and Richard Taylor. Fermat's last theorem, 2026. URL https://imperialcollegelondon.github.io/FLT/blueprint.pdf.

[52] Shogo Saito and Mashu Noguchi. Foundation, 2026. URL https://github.com/FormalizedFormalLogic/Foundation.

[53] Ivan Sergeev, Martin Dvorak, Tristan Figueroa-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.

[54] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

[55] OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Appendix A
System Configuration (ImProver)

In this appendix, we note the prompts used by ImProver both for general LLM prompting, as well as the metric-specific prompts.

A.1 Template

For the main prompt sent to the LLM on each sample, we build a prompt string using a chat prompt template that is then invoked at runtime to fill in the variables.

Namely, these variables include the set of metric prompts, previous results, input theorem, context, a syntax documents, Mathlib documents, and examples.

The prompt template is a conversation of the format:

> **Placeholder:** *All metric prompts with a 'System' role*
>
> **System:** You will be given the proof context (i.e. the lean file contents/imports leading up to the theorem declaration) wrapped by ¡CONTEXT¿...¡/CONTEXT¿.
>
> You will be given the previous *num_prev* input/output pairs as well as their metric (metric.name) score and correctness score, as well as any error messages, for your reference to improve upon. Each of these previous results will be wrapped with ¡PREV I=0¿¡/PREV I=0¿,...,¡PREV I=*num_prev-1*¿¡/PREV I=*num_prev-1*¿, with I=*num_prev-1* being the most recent result.
>
> Remember to use lean 4 syntax, which has significant changes from the lean 3 syntax. To assist with the syntax relating to the current theorem and current error messages, you will be given *num_syntax_docs* documents to refer to for fixing these syntax issues. Each of these documents will be wrapped with ¡SYNTAX_DOC¿...¡/SYNTAX_DOC¿.
>
> You will also receive *num_mathlib_docs* documents relevant to the current theorem to help with formulating your modified proof. Each of these will be wrapped with ¡CONTENT_DOC¿...¡/CONTENT_DOC¿
>
> You will also receive *num_examples* examples of input-output pairs of proofs that were optimized for the *metric* metric. Each of these will be wrapped with ¡EXAM-

PLE¿...¡/EXAMPLE¿

You will be given the tactic states as comments for reference. The current theorem will be wrapped in ¡CURRENT¿...¡/CURRENT¿

System: *Output format instructions*

Placeholder: *All retrieved syntax documentation*

Placeholder: *All retrieved mathlib documentation*

Placeholder: *All retrieved examples*

User: ¡CONTEXT¿ *context* ¡/CONTEXT¿

Placeholder: *Previous results and inputs/outputs*

Placeholder: *All metric prompts with a 'User' role*

User: ¡CURRENT¿ *theorem* ¡/CURRENT¿

This prompt is then invoked and sent to the language model by filling in all the variables and placeholders. Notably, when we invoke the chain given by `chain|llm|parser`, we throttle the invocation with a randomized exponential rate limit throttling to account for API rate limits, especially in highly-parallelized requests like when benchmarking over a large number of theorems.

A.2 Metric Prompts

Length Metric

System: You are an AI assistant who shortens Lean 4 proofs while ensuring their correctness. You will aim to reduce the number of lines of the tactic proof while ensuring that it properly compiles in Lean 4.

User: Shorten the current theorem (wrapped in ¡CURRENT¿...¡/CURRENT¿) to be as short in length—measured in the number of lines of the proof—as possible, while also ensuring that the output is still syntactically correct."

Declarativity Metric

System: You are an AI assistant who rewrites Lean 4 proofs to be more readable while ensuring their correctness. We measure readablity by considering the ratio of the number of explicitly typed `have` tactics against the total number of tactics in the proof, as this is proportional to whether a proof is declarative in style, and thus, readable.

User: Rewrite the current theorem (wrapped in ¡CURRENT¿...¡/CURRENT¿) so it is more readable and declarative and modular.

Mixed Metric

System: You are an AI assistant who rewrites Lean 4 proofs to be higher quality, namely, more concise and more readable/declarative in style and structure. We measure the length of a proof by the number of tactics, and readability/declarativity by the number of explicitly typed `have` tactics.

User: Rewrite the current theorem (wrapped in ¡CURRENT¿...¡/CURRENT¿) so it is more readable and declarative and concise, while also being correct. Namely, we penalize 1 point for every tactic, and reward 5 points for every declarative tactic (namely, `have` statements). Your goal is to maximize that reward function as much as possible while generating a correct proof using the provided template as a starting point.

Completion Metric

System: You are an AI assistant who automatically solves Lean 4 proofs (as in, generates the tactic proof) and ensures its correctness. You will receive a Lean 4 proof you must modify to eliminate any errors so that it compiles correctly and eliminate any "sorry"s with full proofs.

User: Rewrite the current theorem (wrapped in ¡CURRENT¿...¡/CURRENT¿) so it is a formal, complete, and correct Lean 4 proof by filling in its tactic proof.

A.3 Metric Examples

In this section, we illustrate side-by-side examples of metric optimization. These examples are part of a larger set of examples provided to the model as described in §*A.1*.

Length Metric As shown in Figure Figure A.1, we provide the model an example of using more advanced tactics like `rintro` and inlining `apply` statements to shorten the proof from 5 tactics to 2.

Suboptimal

```
example : (P → Q) ∧ (Q → R) → P → R := by
  intro h p
  rcases h with ⟨a,b⟩
  apply b
  apply a
  exact p
```

Length Optimized

```
example : (P → Q) ∧ (Q → R) → P → R := by
  rintro (⟨hpq,hqr⟩) hp
  exact hqr (hpq hp)
```

Figure A.1. A human-written example of length optimization.

Declarative Metric

As shown in Figure Figure A.2, we provide the model an example of adding an intermediate result `hp_nq` with an explicitly written type of `P → ¬Q`. Additionally, we show the model an example of simplifying tactics and external lemmas and dependencies to solve the problem in a more direct, declarative, and readable manner.

Suboptimal

```
example (h : ¬ (P ∧ Q)) : ¬ P ∨ ¬ Q := by
  push_neg at h
  exact not_or_of_imp h
```

Declarativity Optimized

```
example (h : ¬ (P ∧ Q)) : ¬ P ∨ ¬ Q := by
  have hp_nq : P → ¬ Q := by
    intro p q
    exact h ⟨p,q⟩
  by_cases hp:P
  . right
    exact hp_nq hp
  . left
    exact hp
```

Figure A.2. A human-written example of declarativity optimization.

Mixed Metric

As shown in Figure Figure A.3, we provide the model an example of adding multiple intermediate results as well as using proof terms and structural modifications to shorten the number of tactics, resulting in a more optimized proof.

We note that although the "Suboptimal" example is taken from MIL C04, it is ensured to not be in our testing split to avoid data contamination.

Completion Metric

As shown in Figure Figure A.4, we provide the model an example of showing a property about `Set`'s, an externally defined datastructure, using simple tactics and forward reasoning, without external lemmas.

Suboptimal

```
example : s \ t ∪ t \ s = (s ∪ t) \ (s ∩ t)  := by
  ext x; constructor
  · rintro (⟨xs, xnt⟩ | ⟨xt, xns⟩)
    · constructor
      left
      exact xs
      rintro ⟨_, xt⟩
      contradiction
    . constructor
      right
      exact xt
      rintro ⟨xs, _⟩
      contradiction
  rintro ⟨xs | xt, nxst⟩
  · left
    use xs
    intro xt
    apply nxst
    constructor <;> assumption
  . right; use xt; intro xs
    apply nxst
    constructor <;> assumption
```

Joint Length/Readability Optimized

```
example : s \ t ∪ t \ s = (s ∪ t) \ (s ∩ t)   := by
  ext x
  constructor
  have in_union_not_inter : ∀ x, x ∈ s ∪ t → x ∉ s ∩
      t → x ∈ (s ∪ t) \ (s ∩ t) := by
    intro x hx hnx
    exact ⟨hx, hnx⟩
  have in_diff_union : ∀ x, (x ∈ s \ t ∨ x ∈ t \ s)
     → x ∈ (s ∪ t) \ (s ∩ t) := by
    rintro x (⟨xs, xnt⟩ | ⟨xt, xns⟩)
    . exact in_union_not_inter x (Or.inl xs) (fun ⟨_,
     ht⟩ => xnt ht)
    . exact in_union_not_inter x (Or.inr xt) (fun ⟨
     hs, _⟩ => xns hs)
  exact in_diff_union x
  have in_diff_or : ∀ x, x ∈ (s ∪ t) \ (s ∩ t) → (x
     ∈ s \ t ∨ x ∈ t \ s) := by
    rintro x ⟨hx, hnx⟩
    by_cases xs : x ∈ s
    . left; exact ⟨xs, fun ht => hnx ⟨xs, ht⟩⟩
    . right; exact ⟨hx.resolve_left xs, fun hs => hnx ⟨
     hs, hx.resolve_left xs⟩⟩
  exact in_diff_or x
```

Figure A.3. A human-written example of mixed length/declarativity optimization optimization.

Suboptimal

```
example {α : Type*} (s : Set α) : s ∩ s = s := by
    sorry
```

Completion Optimized

```
example {α : Type*} (s : Set α) : s ∩ s = s := by
    ext x
    constructor
    . intro h
        rcases h with ⟨hs,_⟩
        exact hs
    . intro h
        constructor
        . exact h
        . exact h
```

Figure A.4. A human-written example of proof completion.

Appendix B
Qualitative Results (ImProver)

In this section, we provide additional qualitative examples demonstrating the improvements ImProver achieves in proof optimization.

B.0.0.1 Compfiles: Length Optimization

Consider **??**, a lemma from the 2022 IMO Question 2 (Compfiles) that we optimize for length. ImProver halves thr proof from 12 tactics to 6. Here, ImProver makes multiple nontrivial optimizations, such as eliminating the `h2'` and `h4` and `hxw` hypotheses, as well as fully generating proof terms for specific rewrites and other tactics.

B.0.0.2 Compfiles: Declarativity Optimization

Consider Figure Figure B.1, in which a lemma from the 2019 IMO problem 1 (from the Compfiles dataset) is optimized for declarativity. This introduces multiple new hypotheses, which generalize a `linear_property` of the functions, and then reuses and instantiates that (and others, too) hypothesis throughout the proof, creating a significantly more declarative proof.

B.0.0.3 Compfiles: Mixed Optimization

Consider Figure Figure B.2, in which a lemma from the 2023 USAMO problem 2 (from the Compfiles dataset) is optimized for mixed declarativity and length. This introduces a new hypothesis, which declares a powerful intermediate lemma, which is then applied to solve the problem. Moreover, this declarativity is introduced in such a way that it makes the proof more concise than the original, with 3 tactics rather than 5.

Original (human-written)

```
lemma additive_to_int_linear (f : ℤ → ℤ) (h: ∀ (x y
    : ℤ), f (x + y) = f x + f y):
  ∃ c, ∀ a, f a = c * a  := by
 let g := AddMonoidHom.toIntLinearMap <|
    AddMonoidHom.mk' f h
 refine ⟨f 1, fun a => ?_⟩
 change g a = g 1 * a
 rw [mul_comm, ← smul_eq_mul, ←
    LinearMap.map_smul, smul_eq_mul, mul_one]
```

ImProver (declarativity-optimized)

```
lemma additive_to_int_linear (f : ℤ → ℤ) (h: ∀ (x y
    : ℤ), f (x + y) = f x + f y):
  ∃ c, ∀ a, f a = c * a   := by
 let g := AddMonoidHom.toIntLinearMap <|
    AddMonoidHom.mk' f h
 have linear_property : ∀ a, f a = g a := by
   intro a
   rfl
 have g_smul : ∀ a, g a = g 1 * a := by
   intro a
   rw [mul_comm, ← smul_eq_mul, ←
    LinearMap.map_smul, smul_eq_mul, mul_one]
 refine ⟨f 1, fun a => ?_⟩
 have f_eq_g : f a = g a := linear_property a
 have g_a_eq : g a = g 1 * a := g_smul a
 rw [f_eq_g, linear_property 1, g_a_eq]
```

Figure B.1. Optimizing a lemma from IMO 2019 P1 for declarativity

Original (human-written)

```
lemma lemma_3 {a b c : ℝ+} (h : a = b + c) : c < a  :=
     by
 rw [h]
 obtain ⟨b, hb⟩ := b
 obtain ⟨c, hc⟩ := c
 rw [←Subtype.coe_lt_coe, Positive.coe_add]
 exact lt_add_of_pos_left c hb
```

ImProver (mix-optimized)

```
lemma lemma_3 {a b c : ℝ+} (h : a = b + c) : c < a
    := by
 have : ↑c < ↑b + ↑c := lt_add_of_pos_left c.val
    b.property
 rw [h, ←Subtype.coe_lt_coe, Positive.coe_add]
 exact this
```

Figure B.2. Optimizing a lemma from USAMO 2023 P2 for mixed declarativity/length

B.0.0.4 MIL: Length Optimization

Consider Figure Figure B.3, which optimizes an exercise solution from MIL Chapter 8, Section 1 (Group theory) for length, modifying the proof structure and introducing proof terms into the structure of the proof to shorten it from 9 tactic invocations to 7.

B.0.0.5 MIL Declarativity Optimization.

Consider Figure Figure B.4, a result from MIL that we optimize for the declarative metric. This original proof carried a score of 0, as it has no `have` statements. In comparison, after applying ImProver, the model explicitly defines hypotheses for use in the latter half of the proof; these hypotheses can easily be converted into standalone lemmas for reuse.

B.0.0.6 MIL: Mixed Optimization

Consider Figure Figure B.5, which optimizes an exercise solution from MIL Chapter 8, Section 1 (Group theory) for mixed length/declarativity. ImProver significantly modifies

Original (human-written)

```
example (φ : G →* H) (ψ : H →* K) (S : Subgroup G)
    :
   map (ψ.comp φ) S = map ψ (S.map φ)  := by
  ext x
  simp only [mem_map]
  constructor
  · rintro ⟨y, y_in, hy⟩
    exact ⟨φ y, ⟨y, y_in, rfl⟩, hy⟩
  · rintro ⟨y, ⟨z, z_in, hz⟩, hy⟩
    use z, z_in
    calc ψ.comp φ z = ψ (φ z) := rfl
    _               = ψ y := by congr
```

ImProver (length-optimized)

```
example (φ : G →* H) (ψ : H →* K) (S : Subgroup G)
    :
   map (ψ.comp φ) S = map ψ (S.map φ)   := by
  ext x
  simp only [mem_map]
  constructor
  rintro ⟨y, y_in, hy⟩; exact ⟨φ y, ⟨y, y_in, rfl⟩, hy⟩
  rintro ⟨y, ⟨z, z_in, hz⟩, hy⟩; exact ⟨z, z_in,
    (congr_arg ψ hz).trans hy⟩
```

Figure B.3. Optimizing a lemma from the solutions of MIL CH08 S01 for length

Original (human-written)

```
def iso₁ [Fintype G] (h : Disjoint H K) (h' : card G =
     card H * card K)
: K ≃* G ⧸ H  := by
  apply MulEquiv.ofBijective ((QuotientGroup.mk'
    H).restrict K)
  rw [bijective_iff_injective_and_card]
  constructor
  · rw [← ker_eq_bot_iff, (QuotientGroup.mk'
    H).ker_restrict K]
    simp [h]
  · symm
    exact aux_card_eq h'
```

ImProver (declarativity-optimized)

```
def iso₁ [Fintype G] (h : Disjoint H K) (h' : card G =
     card H * card K)
: K ≃* G ⧸ H   := by
  have injectivity : Function.Injective
    ((QuotientGroup.mk' H).restrict K) := by
   rw [← ker_eq_bot_iff, (QuotientGroup.mk'
    H).ker_restrict K]
   simp [h]
  have card_eq : card (G ⧸ H) = card K := by
    exact aux_card_eq h'
  apply MulEquiv.ofBijective ((QuotientGroup.mk'
    H).restrict K)
  rw [bijective_iff_injective_and_card]
  constructor
  exact injectivity
  symm
  exact card_eq
```

Figure B.4. Optimizing a group-theoretic result from MIL Chapter 8 Section 1 for declarativity.

the structure of the proof, applying more general-purpose tactics like `simpa` to simplify the proof (via eliminating the need for manual step-by-step calculations). Moreover, the proof introduces declarativity by a intermediate result, which is used by the `simpa` call to finish the proof in an efficient and declarative manner.

B.0.0.7 Mathlib: Length Optimization

Consider Figure Figure B.6, which optimizes a theorem in algebraic topology from mathlib for length, eliminating `simp` calls and combining tactics to shorten it from 3 tactic invocations to 1.

Original (human-written)

```
lemma eq_bot_iff_card {G : Type*} [Group G] {H :
    Subgroup G} [Fintype H] :
   H = ⊥ ↔ card H = 1  := by
  suffices (∀ x ∈ H, x = 1) ↔ ∃ x ∈ H, ∀ a ∈ H, a =
    x by
    simpa [eq_bot_iff_forall, card_eq_one_iff]
  constructor
  · intro h
    use 1, H.one_mem
  · rintro ⟨y, -, hy'⟩ x hx
    calc x = y := hy' x hx
    _      = 1 := (hy' 1 H.one_mem).symm
```

ImProver (mix-optimized)

```
lemma eq_bot_iff_card {G : Type*} [Group G] {H :
    Subgroup G} [Fintype H] :
   H = ⊥ ↔ card H = 1     := by
  have : (∀ x ∈ H, x = 1) ↔ ∃ x ∈ H, ∀ a ∈ H, a = x
    :=
    ⟨λ h => ⟨1, H.one_mem, h⟩, λ ⟨y, _, hy'⟩ x hx =>
     (hy' 1 H.one_mem).symm ▷ hy' x hx⟩
  simpa [eq_bot_iff_forall, card_eq_one_iff] using
    this
```

Figure B.5. Optimizing a lemma from MIL CH08 S01 solution for mixed declarativity/length

Original (human-written)

```
/-- If `f(p(t) = g(q(t))` for two paths `p` and `q`,
    then the induced path homotopy classes
`f(p)` and `g(p)` are the same as well, despite
    having a priori different types -/
theorem heq_path_of_eq_image : HEq ((πₘ f).map ⟦ p ⟧)
    ((πₘ g).map ⟦ q ⟧)  := by
  simp only [map_eq, ← Path.Homotopic.map_lift];
    apply Path.Homotopic.hpath_hext; exact hfg
```

ImProver (length-optimized)

```
/-- If `f(p(t) = g(q(t))` for two paths `p` and `q`,
    then the induced path homotopy classes
`f(p)` and `g(p)` are the same as well, despite
    having a priori different types -/
theorem heq_path_of_eq_image : HEq ((πₘ f).map ⟦ p ⟧)
    ((πₘ g).map ⟦ q ⟧)    := by
  exact Path.Homotopic.hpath_hext hfg
```

Figure B.6. Optimizing a theorem from `Mathlib/FundamentalGroupoid/InducedMaps` for length

B.0.0.8 Mathlib: Declarativity Optimization

Consider Figure Figure B.7, a theorem from Mathlib that we optimize for declarativity.

This original proof carried a score of 0, as it does not contain any declarative statements. It is concise and efficient, however, it is difficult to understand and read.

After optimizing for declarativity, we see that the model did not change the structure of the proof. Rather, it added an intermediate declaration so that users can better understand the state after the `convert`. This intermediate tactic greatly helps in the understandability and clarity of the proof.

B.0.0.9 Mathlib: Mixed Optimization

Consider Figure Figure B.8, a theorem from Mathlib that we optimize for mixed length-/declarativity.

We observe that ImProver applies more complex tactics such as `all_goals` and `split_ifs` to significantly decrease the number of tactics in the proofs, while maintaining the overall structure.

Original (human-written)

```
/-- Another version of
    `simply_connected_iff_paths_homotopic` -/
theorem simply_connected_iff_paths_homotopic' {Y :
    Type*} [TopologicalSpace Y] :
    SimplyConnectedSpace Y ↔
      PathConnectedSpace Y ∧ ∀ {x y : Y} (p₁ p₂ :
    Path x y), Path.Homotopic p₁ p₂  := by
  convert simply_connected_iff_paths_homotopic (Y :=
    Y)
  simp [Path.Homotopic.Quotient, Setoid.eq_top_iff];
    rfl
```

ImProver (declarativity-optimized)

```
/-- Another version of
    `simply_connected_iff_paths_homotopic` -/
theorem simply_connected_iff_paths_homotopic' {Y :
    Type*} [TopologicalSpace Y] :
    SimplyConnectedSpace Y ↔
      PathConnectedSpace Y ∧ ∀ {x y : Y} (p₁ p₂ :
    Path x y), Path.Homotopic p₁ p₂   := by
  convert simply_connected_iff_paths_homotopic (Y :=
    Y)
  have h1 : ∀ x y : Y, Subsingleton
    (Path.Homotopic.Quotient x y) ↔ ∀ {p₁ p₂ : Path
    x y}, Path.Homotopic p₁ p₂ := by
    intro x y
    simp [Path.Homotopic.Quotient, Setoid.eq_top_iff]
    rfl
  simp only [h1]
```

Figure B.7. Optimizing a theorem from `Mathlib/FundamentalGroupoid/SimplyConnected` for declarativity

B.0.0.10 Full Proof Generation (MIL).

We analyze the application of ImProver to neural theorem proving in the MIL example from Figure Figure B.9. This theorem relating to group theory originally has no proof, however, ImProver generates one from scratch. This generated proof is verified to be correct by Lean, utilizing all the included hypotheses as well as a retrieved mathlib theorem.

B.0.0.11 Full Proof Generation (MiniF2F).

We analyze the application of ImProver to neural theorem proving in the MiniF2F example from Figure Figure B.10. This theorem relating to number theory originally has no proof, however, ImProver generates one from scratch by calculating step by step, applying theorems retrieved from Mathlib, as well as general tactics like `simp`.

Original (human-written)

```
theorem trans_refl_reparam (p : Path x₀ x₁) :
    p.trans (Path.refl x₁) =
      p.reparam (fun t => ⟨transReflReparamAux t,
     transReflReparamAux_mem_I t⟩) (by continuity)
        (Subtype.ext transReflReparamAux_zero)
     (Subtype.ext transReflReparamAux_one)  := by
      p.reparam (fun t => ⟨transReflReparamAux t,
     transReflReparamAux_mem_I t⟩) (by continuity)
        (Subtype.ext transReflReparamAux_zero)
    (Subtype.ext transReflReparamAux_one) := by
  ext
  unfold transReflReparamAux
  simp only [Path.trans_apply, not_le, coe_reparam,
     Function.comp_apply, one_div, Path.refl_apply]
  split_ifs
  · rfl
  · rfl
  · simp
  · simp
```

ImProver (mix-optimized)

```
theorem trans_refl_reparam (p : Path x₀ x₁) :
    p.trans (Path.refl x₁) =
      p.reparam (fun t => ⟨transReflReparamAux t,
     transReflReparamAux_mem_I t⟩) (by continuity)
        (Subtype.ext transReflReparamAux_zero)
     (Subtype.ext transReflReparamAux_one)   := by
      p.reparam (fun t => ⟨transReflReparamAux t,
     transReflReparamAux_mem_I t⟩) (by continuity)
        (Subtype.ext transReflReparamAux_zero)
    (Subtype.ext transReflReparamAux_one)  := by
  ext t
  simp only [Path.trans_apply, Path.refl_apply,
     coe_reparam, Function.comp_apply]
  unfold transReflReparamAux
  split_ifs
  all_goals { simp }
```

Figure B.8. Optimizing a theorem from `Mathlib/FundamentalGroupoid/SimplyConnected` for mixed length/declarativity

Original (human-written)

```
example (φ : G →* H) (S T : Subgroup H) (hST : S ≤
    T) : comap φ S ≤ comap φ T  := by
  sorry
```

ImProver (completeness-optimized)

```
example (φ : G →* H) (S T : Subgroup H) (hST : S ≤
    T) : comap φ S ≤ comap φ T   := by
  intro g
  simp only [mem_comap]
  intro hS
  exact hST hS
```

Figure B.9. Solving a group theorem exercise from MIL Chapter 8 Section 1.

Original (human-written)

```
theorem mathd_numbertheory_345 :
  (2000 + 2001 + 2002 + 2003 + 2004 + 2005 + 2006) %
    7 = 0 := by sorry
```

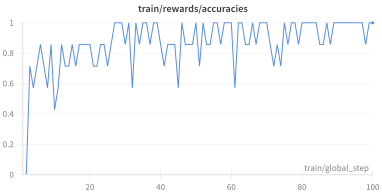
ImProver (completeness-optimized)

```
theorem mathd_numbertheory_345 :
  (2000 + 2001 + 2002 + 2003 + 2004 + 2005 + 2006) %
    7 = 0   := by
  calc
    (2000 + 2001 + 2002 + 2003 + 2004 + 2005 + 2006)
     % 7 = (0 + 1 + 2 + 3 + 4 + 5 + 6) % 7 := by
      simp [Nat.add_mod, Nat.mod_eq_zero_of_dvd,
     Nat.dvd_refl, Nat.mod_add_mod, Nat.mod_self]
    _ = 21 % 7 := by simp
    _ = 0 := by simp
```

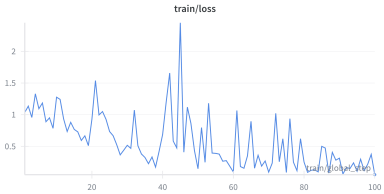
Figure B.10. Solving a number theoretic theorem from MiniF2F.

Appendix C
System Configuration (ImProver2)

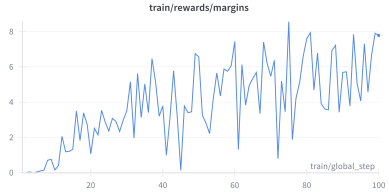
C.1 Training



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



Training loss over time on iteration 3 of the dependency metric.



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

Figure C.1. Training statistics during iteration 3 of the dependency metric. Model shows stable performance despite overall regression on this iteration.

C.2 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.

C.2.0.1 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.
```

C.2.0.2 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.
```

C.2.0.3 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.

C.2.0.4 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.

C.2.0.5 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>.

C.2.0.6 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.

C.3 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.

C.4 Hyperparameter Grid Search

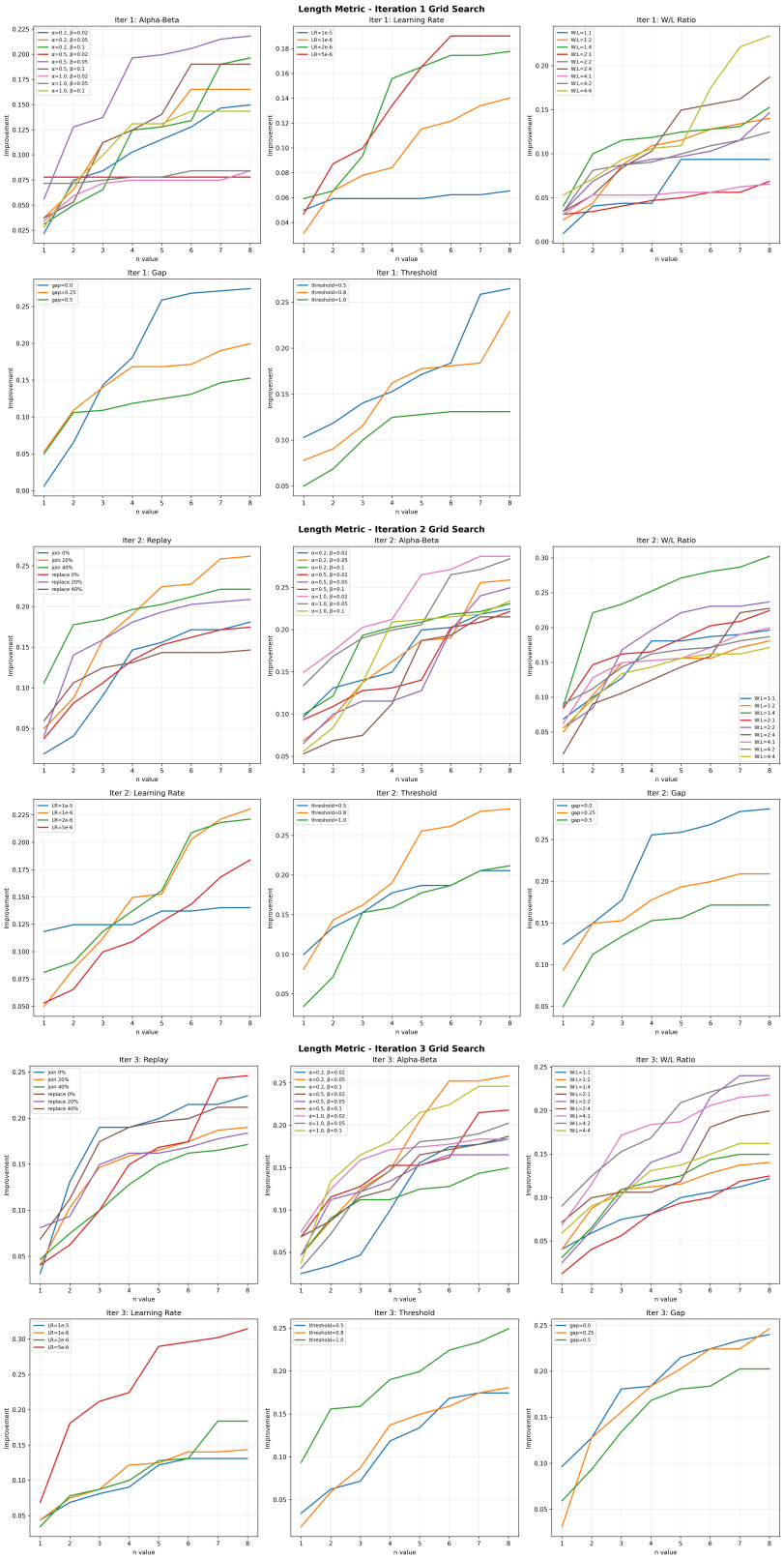


Figure C.2. Hyperparameter grid searches for the **length** metric across iterations 1–3.

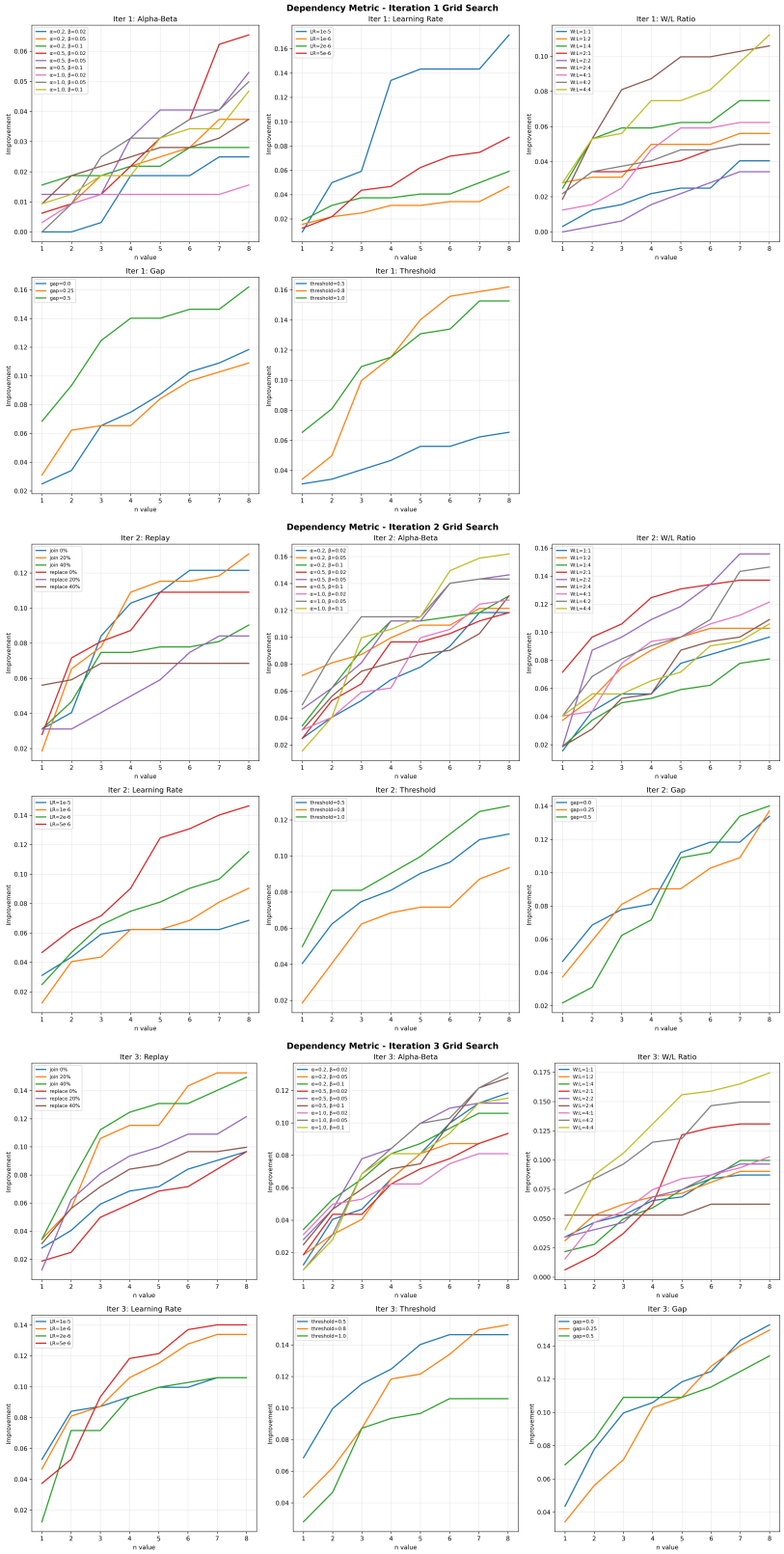


Figure C.3. Hyperparameter grid searches for the **dependency** metric across iterations 1–3.

Appendix D
Qualitative Results (ImProver2)

D.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 D.1. ImProver 2 optimizes a proof for dependencies, reducing explicit dependencies by 3.

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 D.2. ImProver 2 optimizes a proof for dependencies, reducing explicit dependencies by 2.

D.2 Length Optimization

Original (human-written)

```
lemma summerCommute_jacobi_ofCrAnListF (φs1 φs2 φs3 : List 𝓕.
      CrAnFieldOp) :
    [ofCrAnListF φs1, [ofCrAnListF φs2, ofCrAnListF φs3]ₛca]ₛca =
    𝓢 (𝓕 |>ₛ φs1, 𝓕 |>ₛ φs3) •
    (- 𝓢(𝓕 |>ₛ φs2, 𝓕 |>ₛ φs3) • [ofCrAnListF φs3, [ofCrAnListF φ
      s1, ofCrAnListF φs2]ₛca]ₛca -
    𝓢(𝓕 |>ₛ φs1, 𝓕 |>ₛ φs2) • [ofCrAnListF φs2, [ofCrAnListF φs3,
      ofCrAnListF φs1]ₛca]ₛca) := by
  repeat rw [superCommuteF_ofCrAnListF_ofCrAnListF]
  simp only [instCommGroup, map_sub, map_smul, neg_smul]
  repeat rw [superCommuteF_ofCrAnListF_ofCrAnListF]
```

ImProver 2 (length-optimized)

```
lemma summerCommute_jacobi_ofCrAnListF (φs1 φs2 φs3 : List 𝓕.
      CrAnFieldOp) :
    [ofCrAnListF φs1, [ofCrAnListF φs2, ofCrAnListF φs3]ₛca]ₛca =
    𝓢(𝓕 |>ₛ φs1, 𝓕 |>ₛ φs3) •
    (- 𝓢(𝓕 |>ₛ φs2, 𝓕 |>ₛ φs3) • [ofCrAnListF φs3, [ofCrAnListF φ
      s1, ofCrAnListF φs2]ₛca]ₛca -
    𝓢(𝓕 |>ₛ φs1, 𝓕 |>ₛ φs2) • [ofCrAnListF φs2, [ofCrAnListF φs3,
      ofCrAnListF φs1]ₛca]ₛca) := by
  simp_all [superCommuteF_ofCrAnListF_ofCrAnListF,
    instCommGroup, map_sub, map_smul, neg_smul,
    superCommuteF_ofCrAnListF_ofCrAnListF,
```

```
  simp only [instCommGroup.eq_1, ofList_append_eq_mul, List.
      append_assoc]
  by_cases h1 : (𝓕 |>ₛ φs1) = bosonic <;>
    by_cases h2 : (𝓕 |>ₛ φs2) = bosonic <;>
    by_cases h3 : (𝓕 |>ₛ φs3) = 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, mul_self, map_one, one_smul,
      exchangeSign_bosonic, mul_bosonic,
      bosonic_exchangeSign, bosonic_mul_fermionic, neg_sub]
    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
    simp only [h1, h2, h3, mul_self, map_one, one_smul,
      fermionic_exchangeSign_fermionic, neg_smul,
      neg_sub]
    abel
```

```
  instCommGroup.eq_1, ofList_append_eq_mul, List.append_assoc,
  neq_bosonic_iff_eq_fermionic]
<;> by_cases h1 : (𝓕 |>ₛ φs1) = bosonic <;>
  by_cases h2 : (𝓕 |>ₛ φs2) = bosonic <;>
  by_cases h3 : (𝓕 |>ₛ φs3) = 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
```

Figure D.3. ImProver 2 optimizes a proof for length, reducing tactic count by 19.

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 (length-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
  simp_all [cross, mem_inter_iff, vCross_spec,
    op_mem_converse_iff, op_inj]
  <;> aesop
```

Figure D.4. ImProver 2 optimizes a proof for length, reducing tactic count by 8.

Original (human-written)

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

ImProver 2 (modularity-optimized)

```
lemma KD_weakerThan_KDB : (Hilbert.KD α) ≤ₛ (Hilbert.KDB α) := by
  -- Introduce a lemma to handle the subset relationship between the
      axioms of KD and KDB
```

```
have h₁ : (LO.Modal.Hilbert.KD α).axioms ⊆ (LO.Modal.Hilbert.KDB α).
    axioms → (Hilbert.KD α) ≤ₛ (Hilbert.KDB α) := 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 α).axioms ⊆ (LO.Modal.Hilbert.KDB α).
    axioms := by
  -- Prove that all axioms of KD are included in KDB
  intro φ hφ
  cases' hφ with hφ hφ
  · -- 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 D.5. ImProver 2 optimizes a proof for modularity, increasing modularity score (see Definition 3.8) by 2.

Original (human-written)

```
lemma hilbertPoly_eq_zero_of_le_rootMultiplicity_one
    {p : F[X]} {d : ℕ} (hdp : d ≤ 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 ⟨
      q, hq1, hq2⟩
    have heq : p = q * (- 1) ^ p.rootMultiplicity 1 * (1 - X) ^ p.
      rootMultiplicity 1 := by
      simp only [mul_assoc, ← mul_pow, neg_mul, one_mul, neg_sub]
      exact hq1.trans (mul_comm _ _)
    rw [heq, ← zero_add d, ← Nat.sub_add_cancel hdp, pow_add (1 -
      X), ← 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 ≤ p.rootMultiplicity 1) :
    hilbertPoly p d = 0 := by
  -- Consider the case where p is the zero polynomial
  have h₁ : p = 0 → 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 ≠ 0 → d ≤ p.rootMultiplicity 1 → 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 ⟨
      q, hq1, hq2⟩
    -- 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, ← 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, ← zero_add d, ← Nat.sub_add_cancel hdp, pow_add (1 -
      X), ← mul_assoc,
      hilbertPoly_mul_one_sub_pow_add, hilbertPoly]
  -- Combine both cases to conclude the proof
  by_cases hp : p = 0 <;> simp_all
  <;> simp_all [h₁, h₂, hdp]
```

Figure D.6. ImProver 2 optimizes a proof for modularity, increasing modularity score (see Definition 3.8) by 2.