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

Riyaz Ahuja [1] **Tate Rowney** [1] **Jeremy Avigad** [1] **Sean Welleck** [1]

Abstract

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

1. Introduction

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

[1]Carnegie Mellon University. Correspondence to: Riyaz Ahuja <riyaza@andrew.cmu.edu>.

Preprint. February 24, 2026.

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

This motivates automated proof optimization: given a verified proof, produce a proof that is better with respect to a user-defined objective (e.g., shorter, more modular/declarative, having fewer dependencies, or more readable), while preserving formal correctness (Ahuja et al., 2025). Crucially, the optimization objective is task- and user-dependent, and may be informally specified. Therefore, any practical solution must support a variety of metrics and scale to research-level theorems. Additionally, the high-level reasoning required for this task motivates the use of a deep learning-based approach; however, multiple considerations make the use of a small, specialized model preferable over a general LLM for this task, including the scarcity of relevant data in most general training corpora, the reduction of inference costs when deployed across large libraries, and greater accessibility to the formal mathematics community.

We address these problems with **ImProver 2**, a self-improving pipeline that trains small language models (*SLM*s) to optimize Lean proofs under a wide-ranging class of metrics. Our core idea is to leverage *iterative preference optimization*: iterating between generating proof candidates, scoring them according to correctness and the desired optimization criterion, and learning from the resulting pairs of higher and lower scoring proofs. In particular, we extend the Iterative Reasoning Preference Optimization (IRPO) (Pang et al., 2024) algorithm with a new replay buffer that balances

old and newly generated data for use in the next round of training, preventing model collapse and allowing for monotonic improvement over many rounds. We additionally give the model access to rich information from the Lean theorem proving environment, including goal states, informalized summaries, lemma context, and examples, which we term *neurosymbolic augmentation*. We use **ImProver 2** to train models for three different metrics: the *length* of the proof, *modularity* (the ability to divide the proof into a series of smaller lemmas), and the explicit *dependencies* used by the theorem, and show that our trained models can match or outperform much larger models on optimizing research-level theorems. In summary, our contributions are:

1. *Self-improving SLMs that surpass larger models.* We show that iterative preference optimization can bootstrap small language models (SLMs) to significantly increase performance on the task of optimizing proofs in highly specialized research libraries. Moreover, these SLMs can match or outperform much larger models on the research-level proof optimization tasks.

2. *Novel and practical optimization metrics.* In addition to the standard length metric (Ahuja et al., 2025; Gu et al., 2025) we develop new *modularity* and *dependency* metrics that leverage the structure of proofs and their surrounding library. These metrics were chosen in consultation with formal mathematics experts, and each serve an intentional and distinct purpose within the practical task of proof optimization.

3. *Neurosymbolic augmentation for research mathematics.* We introduce techniques for extracting formal and informal context related to a proof, including relevant lemmas or definitions, goal-state traces, and automatically informalized versions of the proof being optimized. This augmentation significantly boosts the performance of both small and large models on proof optimization tasks.

We additionally open-source our code, data, and models[1].

2. Related Work

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

Previous work by (Ahuja et al., 2025; Gu et al., 2025) has attempted to rectify these issues by creating LLM-based agents to refactor formal proofs. (Ahuja et al., 2025) created a system capable of optimizing towards multiple metrics of improvement; however, it relied on general-purpose closed-source models, leading to substantial deployment costs and limited ability to improve performance beyond these models' baseline. (Gu et al., 2025) focused solely on optimizing the token count of proofs according to a complex tokenizer intended to reduce the time required to compile them; they do not examine other metrics, leaving out important use cases and limiting its utility to research mathematicians. Furthermore, the works above do not fully utilize the information available through working in an interactive theorem proving environment via goal-state extraction (Polu & Sutskever, 2020), premise retrieval (Yang et al., 2023), or auto-informalization (Hattori et al., 2025); both of the above works overlook at least one of these aspects.

[1] Github

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]
```

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⟩⟩
```

Original (human-written)

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

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
```

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
```

ImProver 2 (modularity-optimized)

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

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. We present a brief overview of the problem following (Ahuja et al., 2025; Gu et al., 2025), and introduce our objectives of interest, including two novel metrics (*modularity* and *dependencies*).

3.1. Setup and notation

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

We define a *verifier* as a computable function

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

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

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

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

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

3.2. Optimization Objective

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

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

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

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

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

3.2.1. Metrics of Interest

In this work, we focus and evaluate on a collection of three metrics, which are designed to be practical, interpretable,

and useful for the structural optimization of formal proofs at scale.

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

- **Dependencies:** We aim to minimize the dependency footprint of proofs, measured by the number of unique external lemmas explicitly used in the proof. This encourages self-contained proofs that do not require the use or memorization of large numbers of dependency names, improving readability and maintainability[2].

  More specifically, given a theorem and proof (c, x, y), we compute $\text{Deps}_{c,x,y}$, the set of all theorems explicitly named in the proof y (see Section B). The metric is then defined as $\mu_{dep}(c, x, y) := -\left|\text{Deps}_{c,x,y}\right|$.

- **Modularity:** We aim to maximize the *modularity* of proofs, which is intuitively understood to be the number of independent subproofs in our proof. We aim to modify the underlying structure of a proof to contain more independent subcomponents, which are often easier to read, maintain, and reuse.

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

  From here, we define the modularity metric:

$$\mu_{mod}(c, x, y) = |\{\text{effective spawned goals in } y\}|$$

  The designation of which spawned goals are non-trivial and "effective" requires the use of several heuristics; a complete definition is provided in Appendix A.

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

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
```

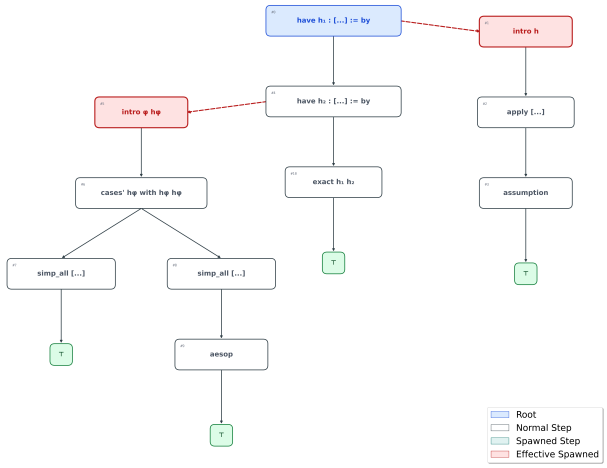


Figure 2. Example Lean proof (top) and corresponding proof tree visualization (bottom). Note the two effective spawned goals h_1 and h_2.

4. 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 (*SLM*s) 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 4.3.2), a replay buffer to balance old and newly generated data (4.3.1), and multiple sources of neurosymbolic augmentation to boost generation capabilities (4.2). Each are described in detail below.

4.1. Overview

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

1. For some budget hyperparameter $n \in \mathbb{N}$, generate n

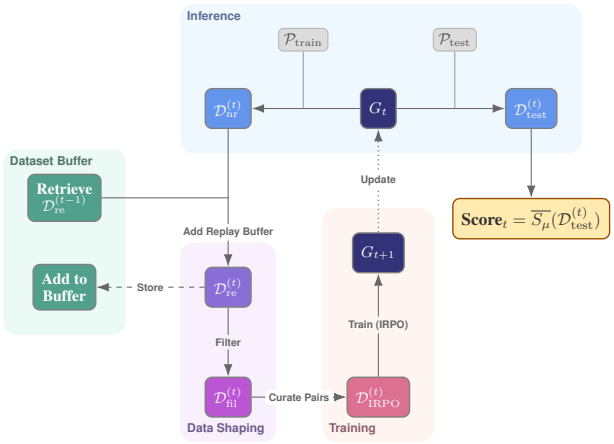


Figure 3. **ImProver^2 training loop.** The diagram illustrates the iterative process of generation, retrieval, filtering, and training. Node colors represent the evolution of data from initial sampling (Blue) through processing (Purple) to training (Magenta).

candidate proofs per problem in the training set using the current model G_t (4.2), providing the model with neurosymbolic augmentation (4.2.1) and a description of the metric to assist in generation. These new potential proofs form a new dataset (denoted $\mathcal{D}_{\text{nr}}^{(t)}$).

2. The previous iteration's dataset, $\mathcal{D}_{\text{re}}^{(t-1)}$ (our *replay buffer*), is interleaved with $\mathcal{D}_{\text{nr}}^{(t)}$ to get $\mathcal{D}_{\text{re}}^{(t)}$ (see 4.3.1).

3. The model G_t is trained to obtain G_{t+1} (4.3.2). Our reinforcement learning policy of choice, known as Iterative Reasoning Preference Optimization or IRPO (Pang et al., 2024), relies on preference pairs of desirable/undesirable proofs, so $\mathcal{D}_{\text{re}}^{(t)}$ is filtered to remove low-quality solutions and pairs are created to form $\mathcal{D}_{\text{IRPO}}^{(t)}$ (again described in 4.3.1).

4. G_{t+1} is evaluated with n samples on the test set, again similarly to 4.2. The improvement in metric score of the new proofs over the originals is calculated.

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

4.2. Generation

A central feature of 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.

4.2.1. Neurosymbolic Augmentation

Formal 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.

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.

More specifically, the signatures of all definitions and theorems that are directly referenced by name in the theorem statement x or the original proof y_0 are collected. Each of these is provided to the model, along with any associated documentation comments (which generally contain summaries or explanations of the associated declaration). The process of collecting and filtering these is provided in Appendix B.

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

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

More concretely, we prompt a language model using the proof's chain-of-states information as described above to translate a Lean proof into natural language by explaining the effect of each tactic on the proof state and providing this to the proof optimizer model. We emphasize that this informalization is a secondary channel for the generator and

serves simply as an additional representation of the target theorem; outputs are still prompted to be generated formally, and correctness is still judged formally.

4.3. Training

We build upon the Iterative Reasoning Preference Optimization algorithm (Pang et al., 2024) 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.

4.3.1. 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 (Shumailov et al., 2024). 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 sort them based on the degree of their improvement for use during training.

Specifically, following the generation of samples $\mathcal{D}_{\text{nr}}^{(t)}$ at iteration t (see 4.2), we utilize this and the previous iteration's dataset $\mathcal{D}_{\text{re}}^{(t-1)}$ to obtain the new dataset $\mathcal{D}_{\text{re}}^{(t)}$ via the following steps:

1. Filter $\mathcal{D}_{\text{nr}}^{(t)}$ for "replay-eligible" examples (those that contain improved, compiling solutions from previous iterations).
2. For each proof in the filtered set, find its counterpart in $\mathcal{D}_{\text{re}}^{(t-1)}$, and combine the sets of proof candidates from each one, creating our new dataset $\mathcal{D}_{\text{re}}^{(t)}$.
3. Filter $\mathcal{D}_{\text{re}}^{(t)}$ by removing problems that had a high overall improvement rate (i.e. those that represent trivial tasks for the model).
4. For each theorem T in $\mathcal{D}_{\text{re}}^{(t)}$, partition its set of proofs into a "winners" or "losers" category (denoted $W_T^{(t)}$ and $L_T^{(t)}$), with "winners" being those which compile and have a sufficiently high improvement score, and "losers" containing all other proofs. We call this filtered and partitioned dataset $\mathcal{D}_{\text{fil}}^{(t)}$.

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

This process is described in more detail in Appendix C.

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 (Shao et al., 2024), 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.

4.3.2. IRPO (Iterative Reasoning Preference Optimization)

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

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

In the above, we represent the relevant neurosymbolic aug-

mentation with the function Ψ. After training with respect to this objective for one epoch, we obtain G_{t+1}.

5. Experiments

5.1. Setup

We now proceed to evaluate the ImProver 2 pipeline on all three of the aforementioned metrics on public research-level mathematics repositories to benchmark the optimization performance both qualitatively and quantitatively against a variety of open-source and closed-source baselines at varying parameter counts. We show that the ImProver 2 pipeline can indeed significantly improve the optimization performance of small-parameter models to outperform far larger and more complex models with a low number of training iterations, across all tested metrics.

Dataset and split We utilize Lean proofs from several open-source projects formalizing research-level mathematics across multiple domains (The Mathlib Community, 2020; Tooby-Smith, 2025; Tao, 2026; van Doorn et al., 2026; Wilshaw, 2026; Buzzard & Taylor, 2026; Saito & Noguchi, 2026; Sergeev et al., 2026). We reserve theorems in the miniCTX-v2 dataset (Hu et al., 2025) — a heterogeneous collection of curated theorems from each of these repositories except for (Tao, 2026) — as a test set, along with theorems in each of the files in which a miniCTX-v2 theorem is defined. Training and validation sets are drawn from theorems in all remaining files, and theorems are discarded from each set to achieve an $80\%/20\%$ train/test split. All results presented are based on evaluation on this test set.

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.

Evaluation protocol For all main results, we evaluate with best@16 sampling (4.2) and report the improvement score $\mu(c, x, y) - \mu(c, x, y_0)$ for all problems in the test set and $\mu \in \{\mu_{len}, \mu_{dep}, \mu_{mod}\}$. With this we compute the mean of the improvement scores, as well as ancillary metrics such as the compilation accuracy $\mathcal{A}(\mathcal{D}^{(t)}_{test})$ and improved accuracy $\mathcal{A}^{+}_{\mu}(\mathcal{D}^{(t)}_{test})$, defined as the percentage of compiling theorems that have a strictly positive improvement score. We operate with a base model of $G_0 =$`DeepSeek-R1-Distill-Qwen-7B` (DeepSeek-AI, 2025).

Systems compared We formulate our main results to compare several iterations of ImProver 2's IRPO pipeline on all three metrics to the best-of-16 performance of `GPT-5-chat` (a nonreasoning variant of GPT-5, chosen due to resource constraints), `GPT-5-mini`, `GPT-5-nano`, `DeepSeek-R1`, and `GPT-oss-120B`. Each metric's instance of ImProver 2 was trained until performance plateaued as detailed in 4.3 (4 iterations for length, 3 iterations for dependency, and 4 iterations for modularity). We additionally perform a robust ablation study of the affects of the neurosymbolic augmentation on various metrics, as well as a full grid search evaluation and optimization of all parameters in our training pipeline.

5.2. Main results

Table 1. **Main Results.** Comparison at best@16 of ImProver 2 against baselines across the length, modularity, and dependency metrics. We compare the mean improvement across the MiniCTX 2 test set against open source models, frontier baselines, and prior work on agentic proof optimization.

Model	Length	Mod.	Dep.
DeepSeek-R1 7B	0.118	0.003	0.050
+ *Scaffold*	0.236	0.007	0.056
DeepSeek-R1 14B	0.140	0.037	0.093
+ *Scaffold*	0.202	0.024	0.106
DeepSeek-R1 (full)	0.308	0.055	0.153
+ *Scaffold*	0.355	0.100	0.209
ImProver 2	0.330	0.143	0.206
ImProver	0.417	0.088	0.047
GPT-4o	0.361	0.034	0.050
+ *Scaffold*	0.470	0.092	0.075
GPT-oss-120B	0.321	0.178	0.181
+ *Scaffold*	0.508	0.234	0.406
GPT-5-nano	0.087	0.065	0.106
+ *Scaffold*	0.296	0.069	0.108
GPT-5-mini	0.330	0.271	0.203
+ *Scaffold*	0.632	0.290	0.267
GPT-5-chat	0.346	0.118	0.046
+ *Scaffold*	0.576	0.153	0.087

Table 2. **Per-iteration improvements.** We present the progression of ImProver 2 performance across IRPO training iterations on the length, modularity, and dependency metrics, stopping at the first regression in mean improvement score. For each metric, performance increases with diminishing returns up to a saturation point, at which the model begins to regress.

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

Table 1 reports mean improvement at best@16 on MiniCTX-v2. Across all three objectives, ImProver 2 achieves substantial gains over its base generator and matches or exceeds

significantly larger models. After three IRPO iterations, a 7B model trained with ImProver 2 removes an average of 0.330 tactics per proof, introduces 0.143 effective modular subproofs, and eliminates 0.206 explicit dependencies (Table 2). These correspond to $2.8\times$, $8.9\times$, and $4.3\times$ gains over the untrained base model on length, modularity, and dependency respectively.

Correctness vs. improvement. Mean improvement alone can hide whether gains come from a small subset of easy problems or from broad, reliable refactoring. We therefore also report compilation accuracy $\mathcal{A}$ and improved accuracy $\mathcal{A}_\mu^+$ (Latter shown in Appendix F, Table 4). A consistent pattern is that optimization increases $\mathcal{A}_\mu^+$ faster than $\mathcal{A}$: early IRPO iterations push the model toward more aggressive rewrites that are more likely to improve the metric when they compile, but may temporarily reduce overall compilation rate. For example, on dependency minimization, ImProver 2 increases improved accuracy from 0.037 (base) to 0.106 (iteration 2) while compilation accuracy drops from 0.754 to 0.464 (Table 3). This illustrates the central tension of proof refactoring: making larger structural edits increases the probability of meaningful improvement, but also increases the chance of breaking compilation.

Table 3. **Accuracy Results.** Comparison at best@16 of ImProver 2 against baselines across the length, modularity, and dependency metrics. We compare the mean accuracy across the MiniCTX 2 test set against open source models, frontier baselines, and prior work on agentic proof optimization.

Model	Length	Mod.	Dep.
DeepSeek-R1 7B	0.617	0.570	0.754
+ *Scaffold*	0.757	0.686	0.748
DeepSeek-R1 14B	0.660	0.775	0.567
+ *Scaffold*	0.695	0.587	0.551
DeepSeek-R1 (full)	0.536	0.536	0.480
+ *Scaffold*	0.595	0.603	0.595
ImProver 2 - iter 1	0.720	0.679	0.417
ImProver 2 - iter 2	0.679	0.614	0.464
ImProver 2 - iter 3	0.682	0.579	0.368
ImProver 2 - iter 4	0.657	0.579	N/A
ImProver	0.738	0.597	0.249
GPT-4o	0.601	0.464	0.227
+ *Scaffold*	0.586	0.502	0.312
GPT-oss-120B	0.692	0.576	0.676
+ *Scaffold*	0.785	0.645	0.822
GPT-5-nano	0.461	0.757	0.894
+ *Scaffold*	0.947	0.838	0.957
GPT-5-mini	0.623	0.664	0.834
+ *Scaffold*	0.754	0.745	0.878
GPT-5-chat	0.586	0.455	0.185
+ *Scaffold*	0.648	0.514	0.243

Saturation and iteration selection. Performance improves rapidly in the first one to three iterations before plateauing or slightly regressing (Table 2; Appendix F; Fig. 7). Importantly, the regressions occur in the *proxy improvement metrics* rather than in training stability indicators: training loss decreases and margins remain well-behaved even in later iterations (Appendix F; Fig. 9c). We interpret this as objective saturation and dataset-induced narrowing (the replay buffer concentrates on already-improvable problems), rather than collapse or reward hacking.

Per-repository heterogeneity. MiniCTX-v2 spans multiple research-grade projects with different conventions and baseline proof styles. Improvements are not uniform: repositories that are already heavily optimized for brevity (notably Mathlib) exhibit smaller length gains, while projects with more verbose or case-heavy styles tend to admit larger structural refactors. Per-repository breakdowns are given in Appendix F (Fig. 10).

5.3. Scaling with parameter count

A central question is whether proof optimization performance is primarily a function of model scale or task-specific specialization. Across model families, parameter count improves mean optimization performance under a fixed sampling budget (Fig. 4). Within the `DeepSeek-R1` family, larger variants consistently improve mean length and dependency gains, indicating that generic reasoning capacity does translate into better refactoring when prompts and sampling are held fixed.

However, ImProver 2 trained from a 7B base outperforms substantially larger models on multiple objectives (Table 1), demonstrating that neurosymbolic conditioning plus iterative self-improvement can substitute for large increases in parameter count. This effect is especially pronounced when comparing *improved accuracy* $\mathcal{A}_\mu^+$: ImProver 2 increases the fraction of problems for which the model finds a compiling *and improved* proof, not merely a compiling one (Table 3).

5.4. Effect of Neurosymbolic Scaffolding

A key claim of this work is that proof optimization is often bottlenecked not by raw parameter count, but by whether the model is conditioned on the *right representation* of the formal task. We therefore evaluate the neurosymbolic scaffold Ψ as a standalone intervention: for each base generator we compare a minimal prompt (no scaffold) against the augmented input $\Psi(c, x, y_0)$ at a fixed sampling budget (best@16).

Across nearly all tested generators, scaffolding increases mean improvement in length, modularity, and dependency (Table 1), with gains observed for both open-weight and closed-source models and persisting across a wide range of sampling budgets (Appendix F; Fig. 8). The improvement is especially large for smaller or non-specialized models, such as GPT-5-nano, which exhibits substantial improvements

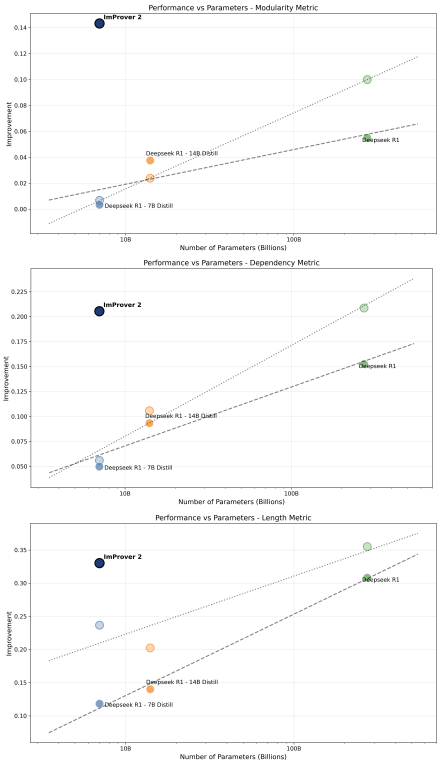


Figure 4. Effect of parameter count on model performance on improvement across all metrics, with ImProver 2 marked.

in compilation accuracy under scaffolding across multiple metrics (Table 3). Although scaffolding enables deeper, higher-risk refactors that can raise improved accuracy $\mathcal{A}_{\mu}^{+}$ without strictly increasing $\mathcal{A}$ (Table 3), the net effect is an expansion of the attainable improvement frontier. Finally, ablation results show that each component of the scaffold contributes measurably, with the full scaffold performing best overall (Appendix F; Fig. 11).

5.5. Comparison to Frontier and Prior Systems

We compare ImProver 2 to (i) frontier closed-source models, (ii) large open-weight baselines, and (iii) prior agentic proof-optimization systems. Across all baselines we evaluate best@16 and report mean improvement (Table 1) as well as correctness-sensitive metrics (Table 3).

Frontier models. Frontier models are strong proof rewriters out-of-the-box, but their performance remains highly metric-dependent. On length and dependency, ImProver 2 trained from a 7B base is competitive with the GPT-5 variants in mean improvement (Table 1) while using orders-of-magnitude fewer parameters. On modularity, the largest frontier models maintain an advantage in mean improvement, plausibly because modularity rewards deeper structural editing and more robust handling of edge cases in goal management (Table 1). However, ImProver 2 closes much of this gap via iterative training and achieves large gains over its own base generator (Table 2).

Large open-weight baselines. We also compare to `GPT-oss-120B` as a representative large open-weight model. While scale helps (Fig. 4; Appendix F), ImProver 2 demonstrates that task-specialized neurosymbolic conditioning plus iterative self-improvement can substitute for large increases in parameter count. In particular, ImProver 2 matches or exceeds `GPT-oss-120B` on length and dependency despite starting from a 7B base model (Table 1).

Prior agentic optimization systems (ImProver). Finally, we compare to the prior ImProver system (Ahuja et al., 2025), which is agentic and prompt-driven and uses a strong proprietary base model (GPT-4o) plus example retrieval. We adapt ImProver to our expanded metric suite by providing hand-written metric descriptions and examples, mirroring its intended usage (details in Appendix E). On structural objectives (dependency and modularity), ImProver 2 yields substantially stronger gains than the prior system while using a smaller open-weight base model (Table 1). On length, both systems perform competitively, with improvements depending on how aggressively each method trades off refactoring for compilation stability (Table 3).

5.6. Qualitative optimization behaviors

Beyond aggregate scores, ImProver 2 exhibits consistent families of refactoring strategies that align with each metric. We include representative before/after proofs in Appendix H, noting that ImProver 2 learns sophisticated strategies for optimization, such as tactic fusion and automation consolidation for length minimization, local reasoning and generic library automation for dependency minimization, and introduction of meaningful subproof boundaries for modularity maximization. Appendix H contains examples where a one-shot proof is rewritten into a clearer sequence of intermediate claims that later drive the main proof.

6. Conclusion

We have introduced ImProver 2, a pipeline for boosting the formal proof-optimization ability of small language models. We have demonstrated the utility of our neurosymbolic augmentations to many varieties of models, showcased iterative self-improvement using our replay buffer architecture, and introduced two novel metrics (as well as revisiting an old one) for practical proof improvement. We have found that our system matches or exceeds state-of-the-art generalist models on these tasks, while using orders of magnitude fewer parameters.

ImProver 2 currently generates improved proofs in a single

generation step, limiting its capacity for in-context learning and fixing trivial syntax errors; future work could focus on training proof optimizers to perform this revision process. Additionally, further research could examine the possibility of more subjective or probabilistic metrics to better align with human preferences, for example by utilizing an LLM-as-a-judge system to score proofs.

References

Achim, T., Best, A., Bietti, A., Der, K., Fédérico, M., Gukov, S., Halpern-Leistner, D., Henningsgard, K., Kudryashov, Y., Meiburg, A., Michelsen, M., Patterson, R., Rodriguez, E., Scharff, L., Shanker, V., Sicca, V., Sowrirajan, H., Swope, A., Tamas, M., Tenev, V., Thomm, J., Williams, H., and Wu, L. Aristotle: Imo-level automated theorem proving, 2025. URL https://arxiv.org/abs/2510.01346.

Ahuja, R., Avigad, J., Tetali, P., and Welleck, S. Improver: Agent-based automated proof optimization. In Yue, Y., Garg, A., Peng, N., Sha, F., and Yu, R. (eds.), *International Conference on Representation Learning*, volume 2025, pp. 29521–29543, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/4864005cfdea7ebd07086ed1b9846825-Paper-Conference.pdf.

Buzzard, K. and Taylor, R. Fermat's last theorem, 2026. URL https://imperialcollegelondon.github.io/FLT/blueprint.pdf.

Chen, J., Chen, W., Du, J., Hu, J., Jiang, Z., Jie, A., Jin, X., Jin, X., Li, C., Shi, W., Wang, Z., Wang, M., Wei, C., Wei, S., Xin, H., Yang, F., Gao, W., Yuan, Z., Zhan, T., Zheng, Z., Zhou, T., and Zhu, T. H. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience, 2025. URL https://arxiv.org/abs/2512.17260.

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

Frieder, S., Bayer, J., Looi, S., Loader, J., Berner, J., Collins, K. M., Juhász, A., Ruehle, F., Welleck, S., Poesia, G., Griffiths, R.-R., Weller, A., Goyal, A., Freer, C., Lukasiewicz, T., and Gowers, T. Data for mathematical copilots: Better ways of presenting proofs for machine learning, 2025. URL https://arxiv.org/abs/2412.15184.

Gu, A., Piotrowski, B., Gloeckle, F., Yang, K., and Markosyan, A. H. 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.

Hattori, S., Matsuzaki, T., and Fujiwara, M. Natural language translation of formal proofs through informalization of proof steps and recursive summarization along proof structure. In Flek, L., Narayan, S., Phng, L. H., and Pei, J. (eds.), *Proceedings of the 18th International Natural Language Generation Conference*, pp. 376–389, Hanoi, Vietnam, October 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.inlg-main.23/.

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

Hubert, T., Mehta, R., Sartran, L., Horváth, M. Z., Žužić, G., Wieser, E., Huang, A., Schrittwieser, J., Schroecker, Y., Masoom, H., Bertolli, O., Zahavy, T., Mandhane, A., Yung, J., Beloshapka, I., Ibarz, B., Veeriah, V., Yu, L., Nash, O., Lezeau, P., Mercuri, S., Sönne, C., Mehta, B., Davies, A., Zheng, D., Pedregosa, F., Li, Y., von Glehn, I., Rowland, M., Albanie, S., Velingker, A., Schmitt, S., Lockhart, E., Hughes, E., Michalewski, H., Sonnerat, N., Hassabis, D., Kohli, P., and Silver, D. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 2025. doi: 10.1038/s41586-025-09833-y. URL https://www.nature.com/articles/s41586-025-09833-y. Published: 12 November 2025.

Jiang, A. Q., Welleck, S., Zhou, J. P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., and Lample, G. Draft, Sketch, and Prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations*, 2023. URL https://doi.org/10.48550/arXiv.2210.12283.

Li, Z., Sun, J., Murphy, L., Su, Q., Li, Z., Zhang, X., Yang, K., and Si, X. A survey on deep learning for theorem proving. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=zlw6AHwukB.

Lin, Y., Tang, S., Lyu, B., Yang, Z., Chung, J.-H., Zhao, H., Jiang, L., Geng, Y., Ge, J., Sun, J., Wu, J., Gesi, J., Lu, X., Acuna, D., Yang, K., Lin, H., Choi, Y., Chen, D., Arora, S., and Jin, C. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction, 2025. URL https://arxiv.org/abs/2508.03613.

Lu, P., Qiu, L., Yu, W., Welleck, S., and Chang, K.-W. A survey of deep learning for mathematical reasoning. In Rogers, A., Boyd-Graber, J., and Okazaki, N. (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14605–14631, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.817. URL https://aclanthology.org/2023.acl-long.817/.

Moura, L. d. and Ullrich, S. The lean 4 theorem prover and programming language. In Platzer, A. and Sutcliffe, G. (eds.), *Automated Deduction – CADE 28*, pp. 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5.

Pang, R. Y., Yuan, W., Cho, K., He, H., Sukhbaatar, S., and Weston, J. Iterative reasoning preference optimization, 2024. URL https://arxiv.org/abs/2404.19733.

Polu, S. and Sutskever, I. Generative language modeling for automated theorem proving, 2020. URL https://arxiv.org/abs/2009.03393.

Saito, S. and Noguchi, M. Foundation, 2026. URL https://github.com/FormalizedFormalLogic/Foundation.

Sergeev, I., Dvorak, M., Figueroa-Reid, T., Hamadani, R., Hwang, B.-H., Karunus, E., Kolmogorov, V., Meiburg, A., Nelson, P., and Sandey, M. Regularity of 1-, 2-, and 3-sums of matroids, 2026. URL https://ivan-sergeyev.github.io/seymour/blueprint.pdf.

Shao, Z., Wang, P., Zhu, Q., nd Junxiao Song, R. X., Zhang, M., Li, Y., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Shumailov, I., Shumaylov, Z., Zhao, Y., Papernot, N., Anderson, R., and Gal, Y. Ai models collapse when trained on recursively generated data. *Nature*, pp. 755–759, 2024. URL https://doi.org/10.1038/s41586-024-07566-y.

Tao, T. Pfr blueprint, 2026. URL https://teorth.github.io/pfr/blueprint.pdf.

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

The Mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pp. 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.

Tooby-Smith, J. 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.

van Doorn, F., Rothgang, M., Monticone, P., Rui, J. T. J., Sundstrom, J., de Frutos-Fernández, M. I., de Velde, R. V., Gouezel, S., Diedering, L., Portegies, J., and Roos, J. Formalizing carleson's theorem in lean, 2026. URL https://florisvandoorn.com/carleson/.

Wenzel, M., Paulson, L. C., and Nipkow, T. The isabelle framework. In Mohamed, O. A., Muñoz, C., and Tahar, S. (eds.), *Theorem Proving in Higher Order Logics*, pp. 33–38, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-71067-7.

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

Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*, 2023. URL https://icml.cc/virtual/2023/27190.

A. Formal Definition of the Modularity Metric

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

A.1. Lean Elaboration Semantics

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

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

Each tactic step τ_i:

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

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

A.2. Direct Children vs. Spawned Goals

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

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

To capture this distinction, define:

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

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

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

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

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

A.3. Proof Graph over Steps

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

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

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

This yields a labeled directed forest

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

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

A.4. Canonical Goal Representation

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

Each goal is assigned a canonical representation consisting of:

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

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

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

A.5. Duplicate and Wrapper Detection

A spawned goal is considered duplicate and discarded if either:

1. Its sequent variant hash matches any previously seen goal (global duplicate), or

2. Its target is a definitional wrapper of its parent goal, i.e. $\text{target}(g_{\text{child}}) \in \text{targetVariants}(g_{\text{parent}})$ or vice versa.

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

A.6. Nontriviality Filter

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

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

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

A.7. Effectiveness via Fixed-Point Semantics

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

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

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

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

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

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

A.8. Modularity Metric

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

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

B. Context Extraction

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

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

Graphs and the slice. From the AST we build two standard graphs: (i) an import DAG over files/modules, and (ii) an entity graph whose vertices are declarations (constants, lemmas, definitions) with edges for semantic references (uses in types/bodies). Let $\mathsf{touch}(x, y_0)$ 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 the subgraph

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

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

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

C. Replay Buffer and Dataset Construction Details

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

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

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

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

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

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

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 $j < t$).

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

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

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

2. *Merge*

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

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

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

With this, we obtain the replay dataset:

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

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

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

- $W_T^{(t)} = \{y \in \widetilde{\mathcal{Y}}_T^{(t)} \mid \mathrm{v}(c_T, x_T, y) = 1 \wedge \widehat{s}_{T,y} > \delta^{(t)}\}$, 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.

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

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

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

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

With this, we form two families of preference pairs:

$$\mathfrak{P}_T^{\ell \to w} = \{(b, g) : b \in L_T^{(t)}, \ g \in W_T^{(t)}\}$$

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

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

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

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

D. Chain-of-States Implementation

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 $(\texttt{goalsBefore}_i, \texttt{goalsAfter}_i)$ and a pretty-printed tactic. We define

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

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

E. System Configuration

E.1. System Prompts

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

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

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

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

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

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

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

E.2. Autoinformalization

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

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

(examples omitted)

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

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

(formal statement of proof is placed here)

`</FORMAL>`

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

F. Additional Results

How to read Appendix F. We report (i) compilation and improved accuracies, (ii) scaling curves (improvement vs. parameter count), (ii) sampling curves (mean improvement vs. best-of-n), (iii) scaffold ablations, (iv) score distributions over replay datasets across iterations, (v) per-repository breakdowns, and (vi) representative training dynamics (loss, accuracy, margin). Together these diagnose whether gains come from scale, sampling, representation, or training, and whether iteration selection reflects instability or objective saturation.

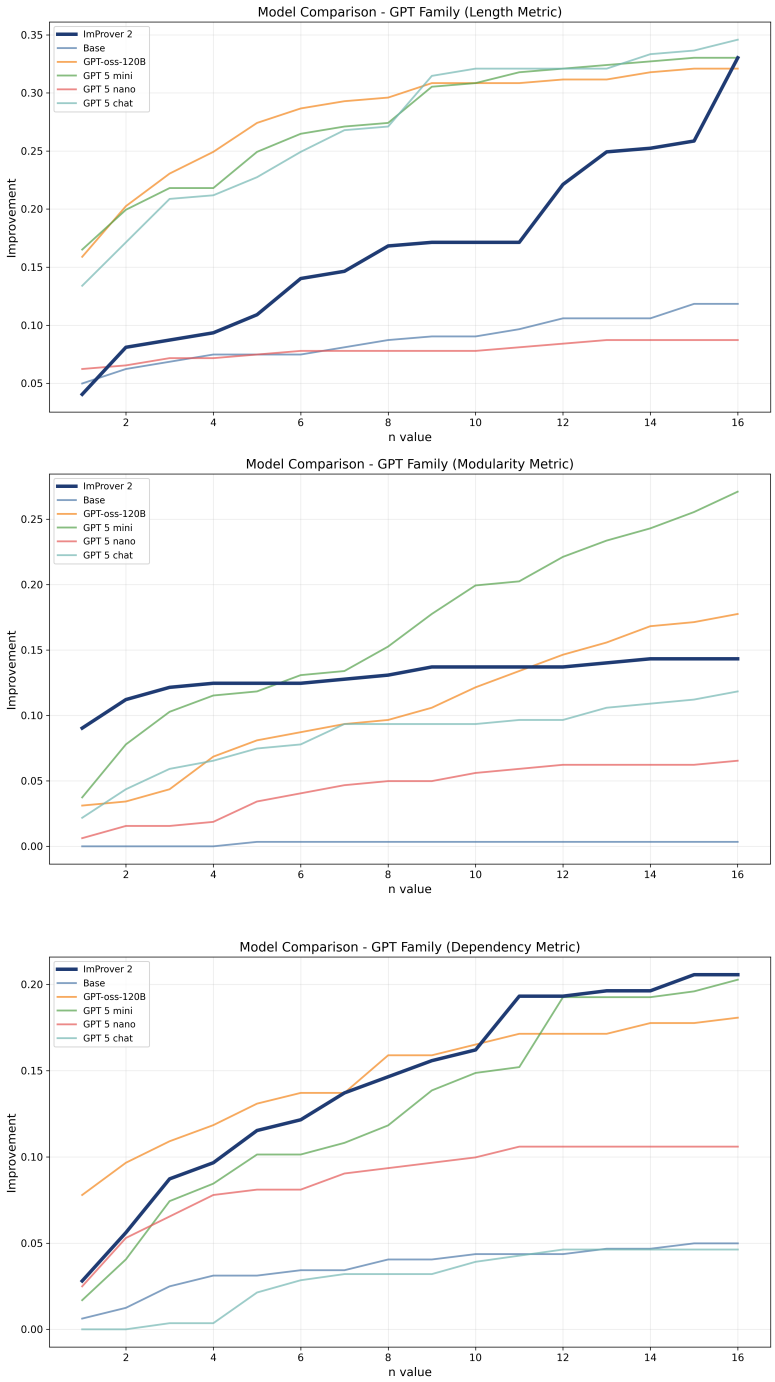


Figure 5. Comparison of ImProver 2 against frontier GPT-based models on all metrics, in best of n samples vs. mean metric score improvement.

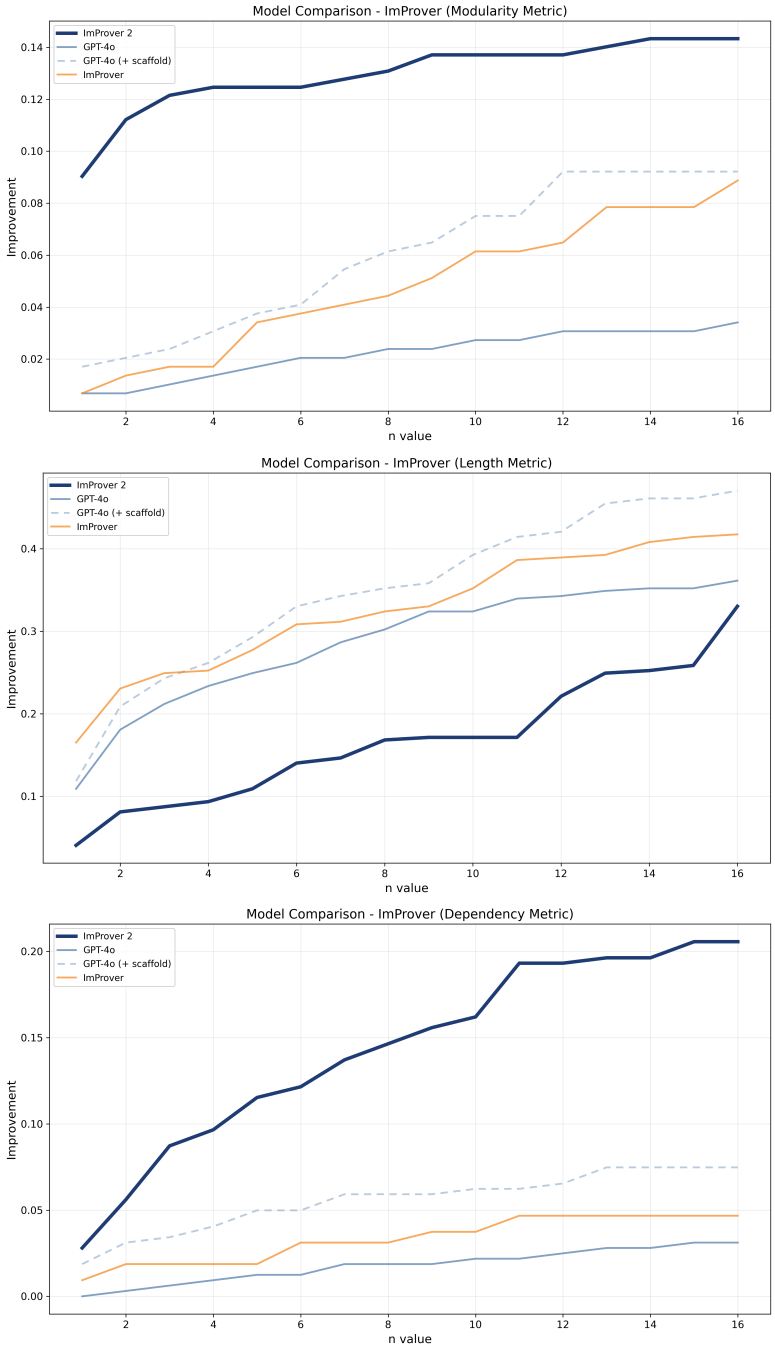


Figure 6. Comparison of ImProver 2 against prior ImProver models and base GPT-models on all metrics. Remaining metrics show similar behavior, and can be found in Appendix F.

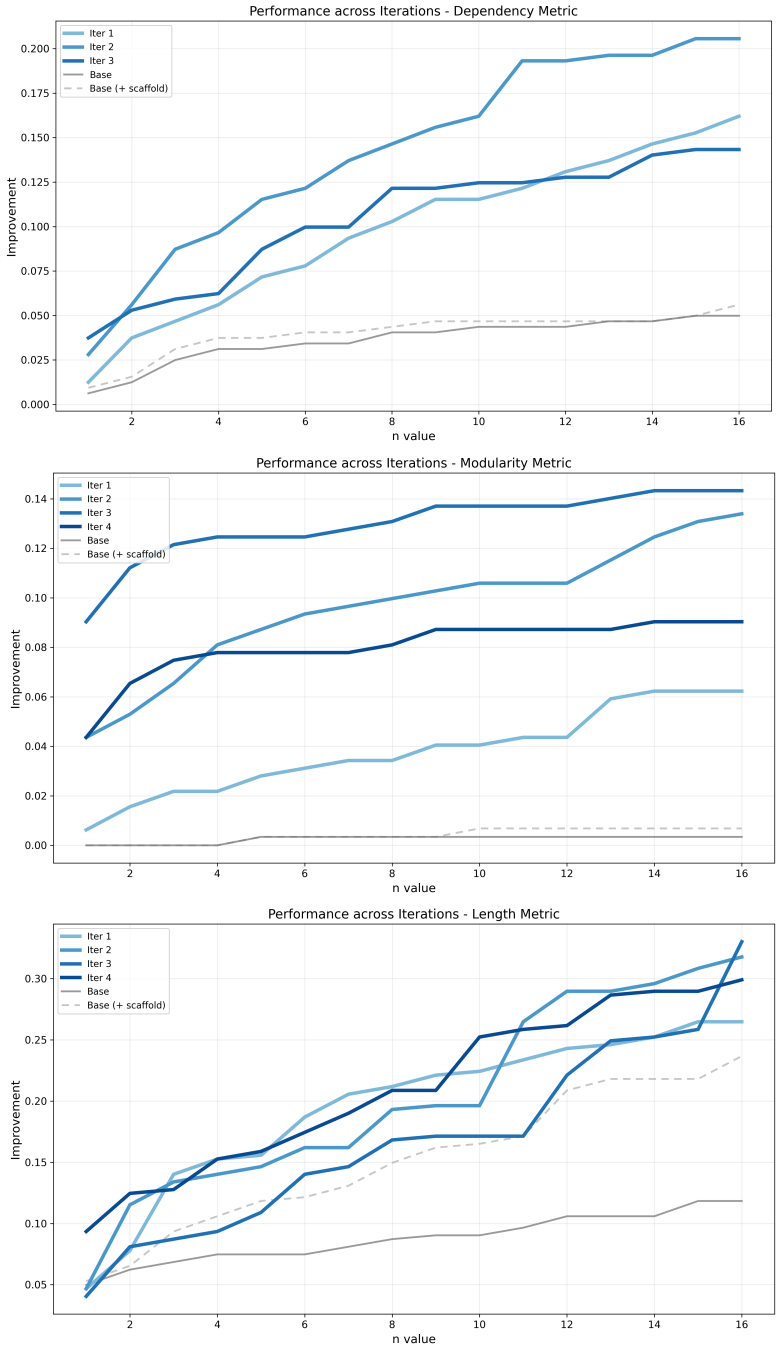


Figure 7. Performance of ImProver 2 (in mean length improvement per theorem) vs. number of samples generated at training iterations 0-3; significant improvement is shown from iterations 0-2 before plateauing at 3 and dropping off at 4 (2 and 3 for dependency metric).

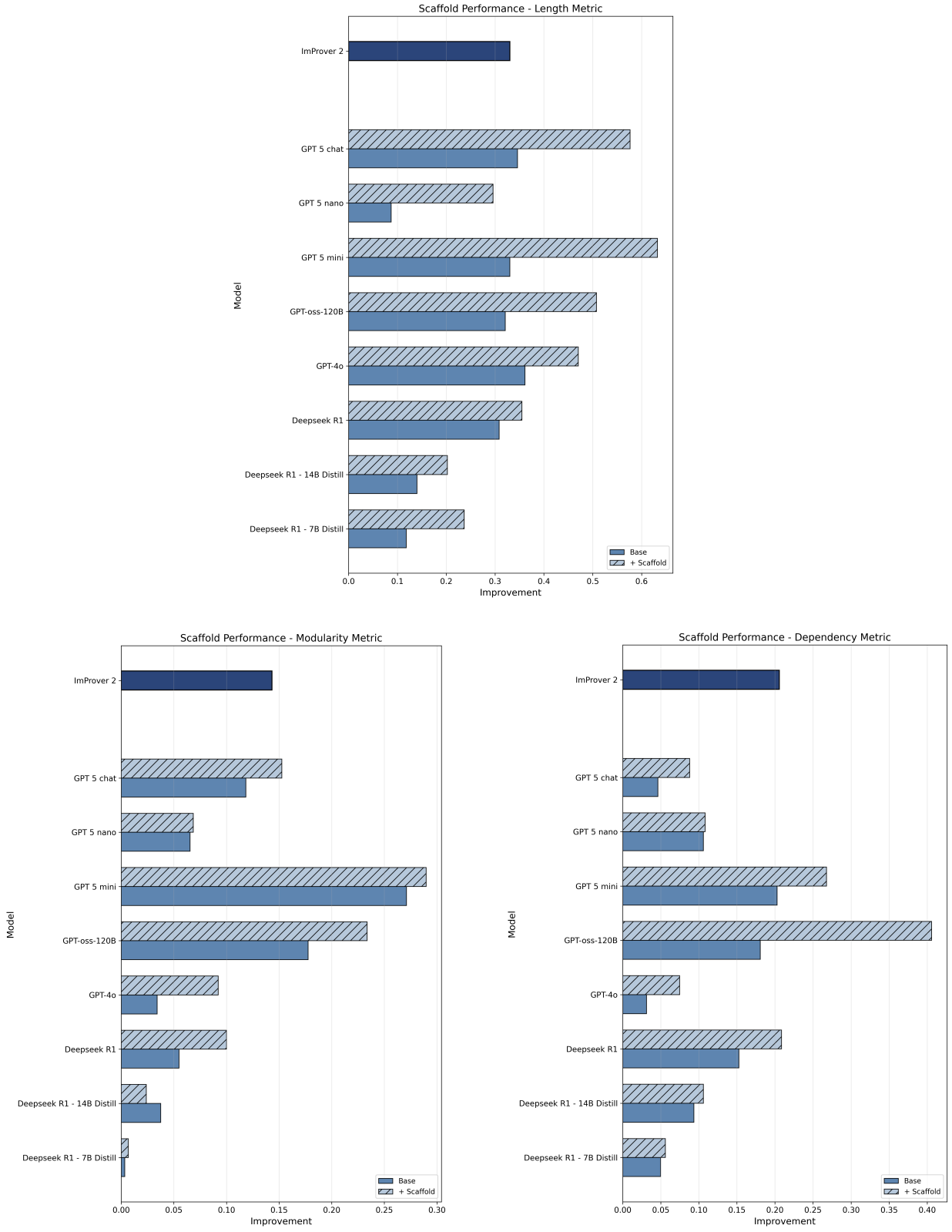
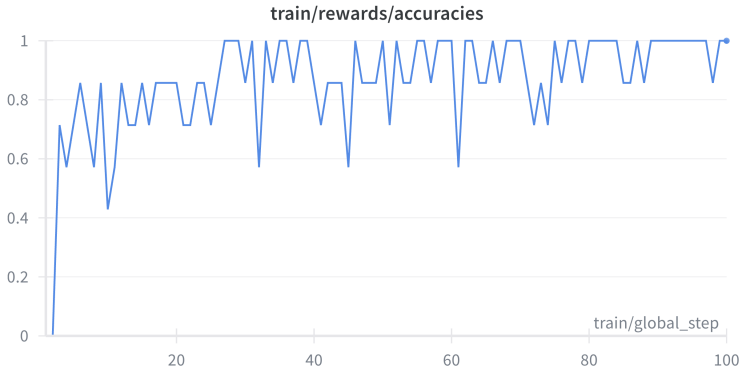
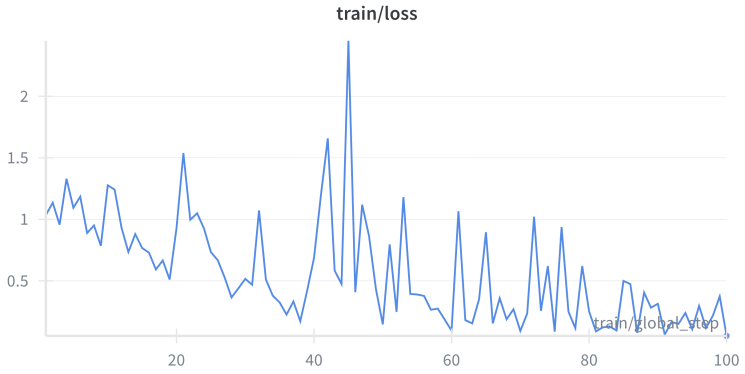


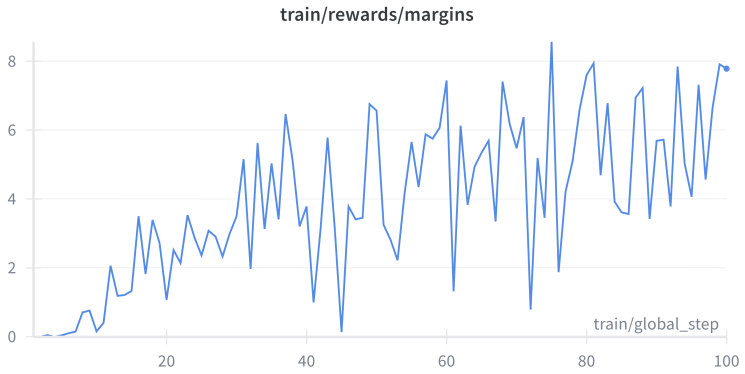
Figure 8. Effect of neurosymbolic scaffolding on models from various sources evaluated on all metrics, with blue representing no-scaffold performance (in mean metric improvement) and orange representing scaffolded improvement.



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



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



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

Figure 9. Training statistics during iteration 3 of the dependency metric. Model shows stable performance despite overall regression on this iteration.

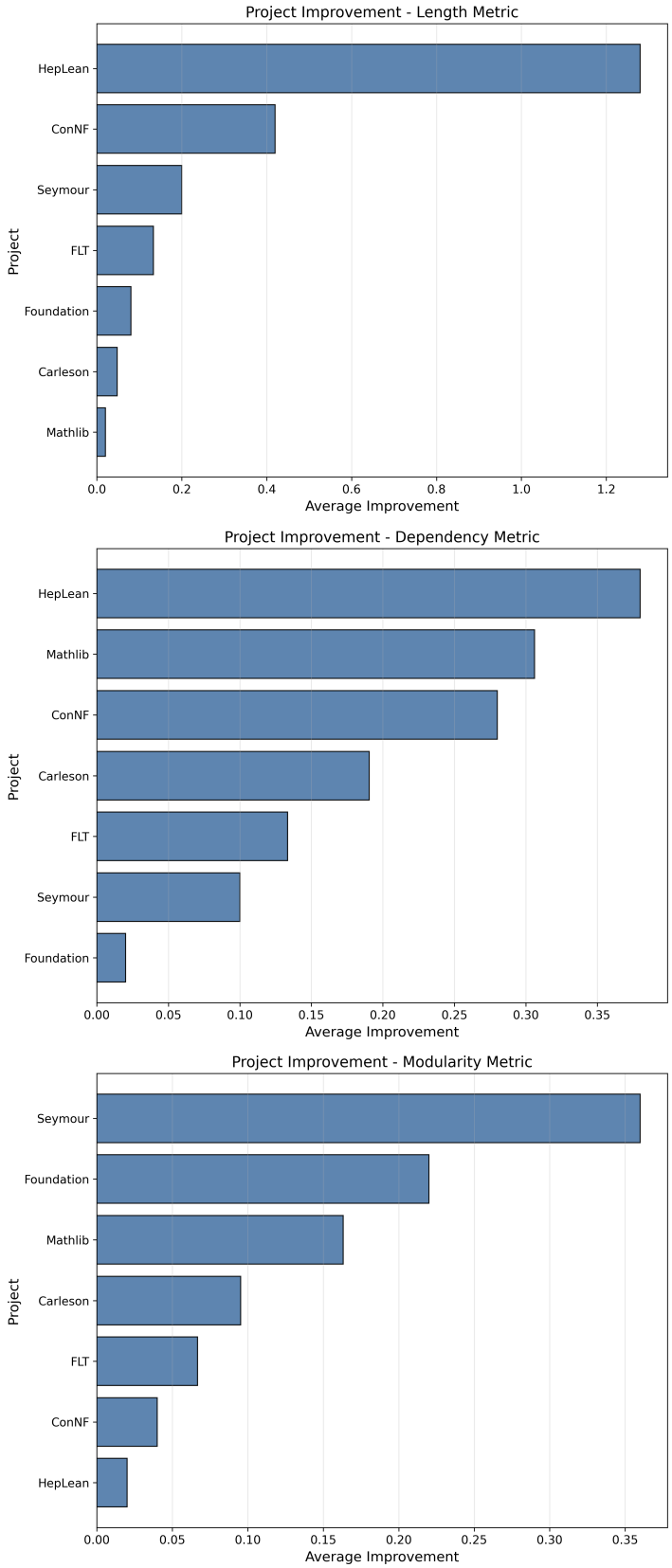


Figure 10. Performance of the various metrics by data source, as tested with the final ImProver 2dependency model.

Table 4. **Improved Accuracy Results.** Comparison at best@16 of ImProver 2 against baselines across the length, modularity, and dependency metrics. We compare the improved accuracy across the MiniCTX 2 test set against open source models, frontier baselines, and prior work on agentic proof optimization.

Model	Length	Mod.	Dep.
DeepSeek-R1 7B	0.062	0.003	0.037
+ *Scaffold*	0.093	0.007	0.044
DeepSeek-R1 14B	0.075	0.034	0.065
+ *Scaffold*	0.090	0.024	0.056
DeepSeek-R1 (full)	0.162	0.048	0.109
+ *Scaffold*	0.178	0.072	0.115
ImProver 2 - iter 1	0.125	0.044	0.093
ImProver 2 - iter 2	0.131	0.121	0.106
ImProver 2 - iter 3	0.121	0.118	0.069
ImProver 2 - iter 4	0.131	0.065	N/A
ImProver	0.196	0.068	0.031
GPT-4o	0.181	0.031	0.028
+ *Scaffold*	0.196	0.078	0.050
GPT-oss-120B	0.171	0.125	0.097
+ *Scaffold*	0.215	0.150	0.207
GPT-5-nano	0.044	0.053	0.065
+ *Scaffold*	0.187	0.062	0.054
GPT-5-mini	0.159	0.174	0.111
+ *Scaffold*	0.265	0.193	0.156
GPT-5-chat	0.165	0.084	0.025
+ *Scaffold*	0.240	0.118	0.047

G. Ablation Study

In addition to experiments mentioned above, we study the effect of each of our sources of neurosymbolic on model performance, finding that each one added increases length-metric performance (Figure 11). Additionally, we perform hyperparameter searches at each iteration for both the length (Figure 12) and dependency metric (Figure 13).

G.1. Hyperparameter Grid Search

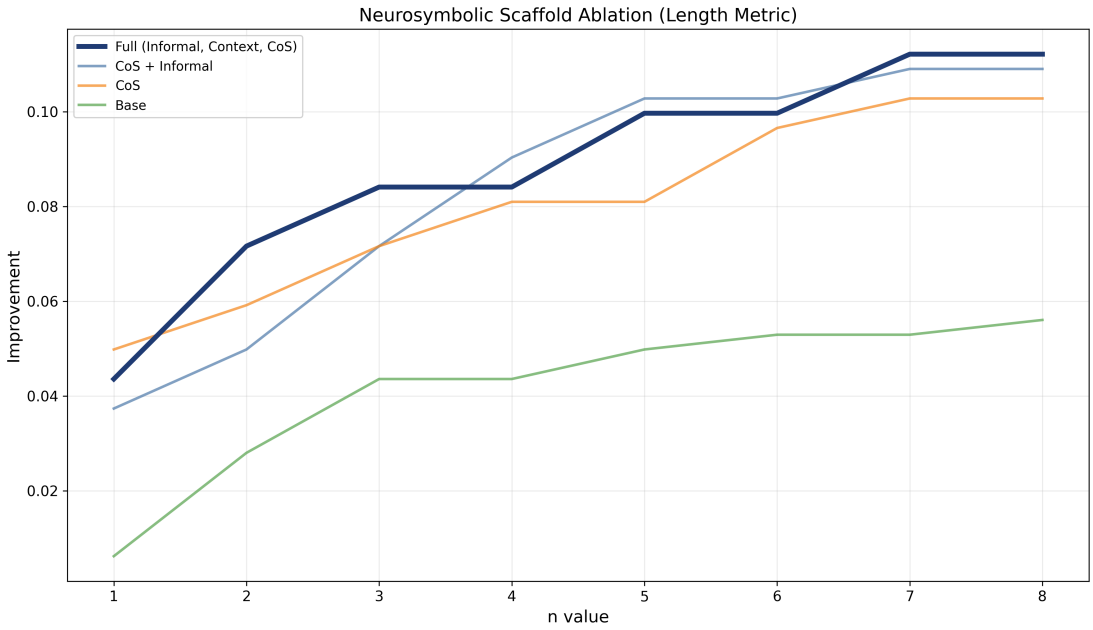


Figure 11. Effect of neurosymbolic augmentation on base model performance (`DeepSeek-R1-Distill-Qwen-7B`) on the length metric, vs. number of samples generated. Each source of augmentation shows noticeable improvement in average score on some n values.

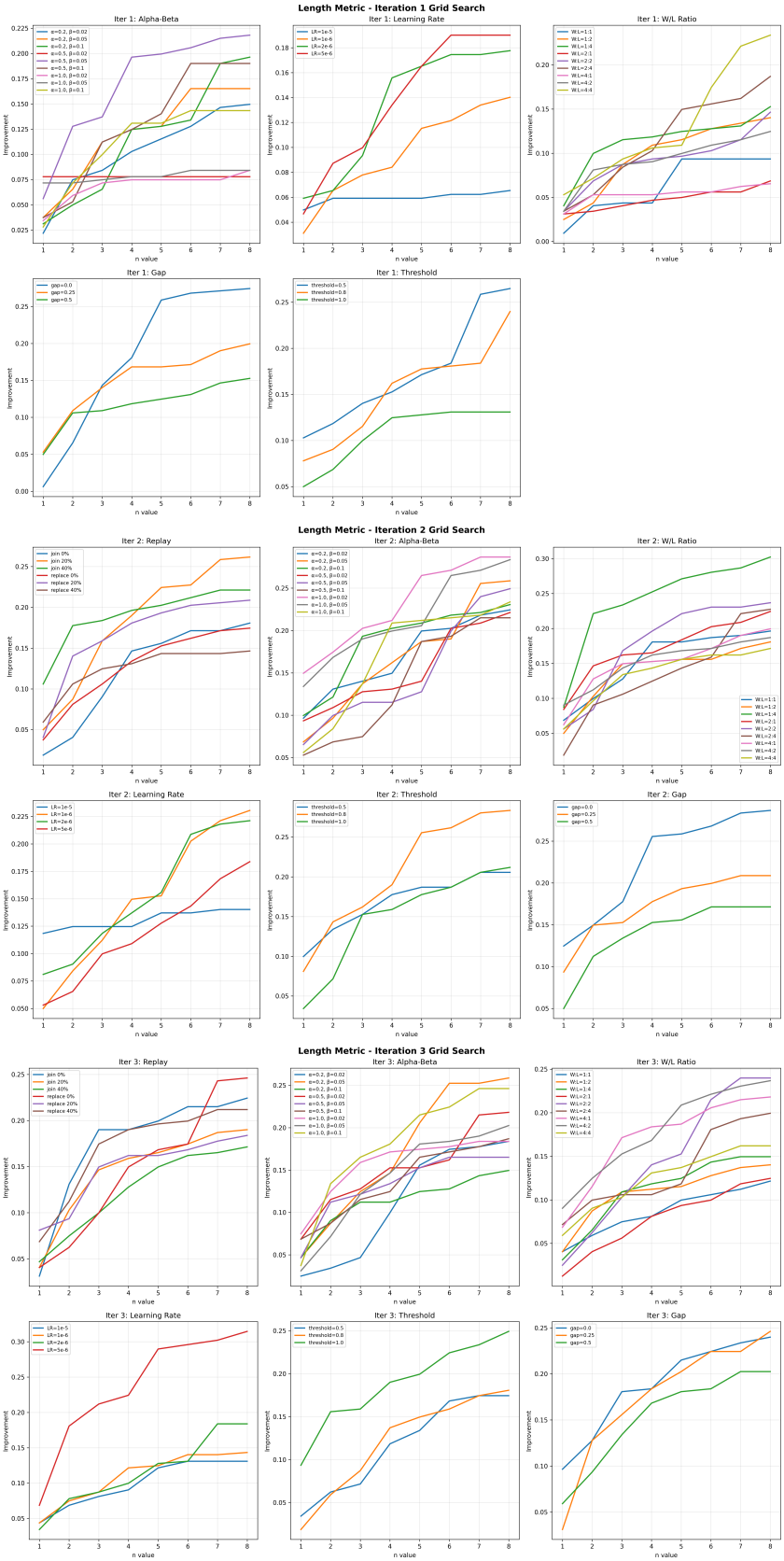


Figure 12. Hyperparameter grid searches for the **length** metric across iterations 1–3.

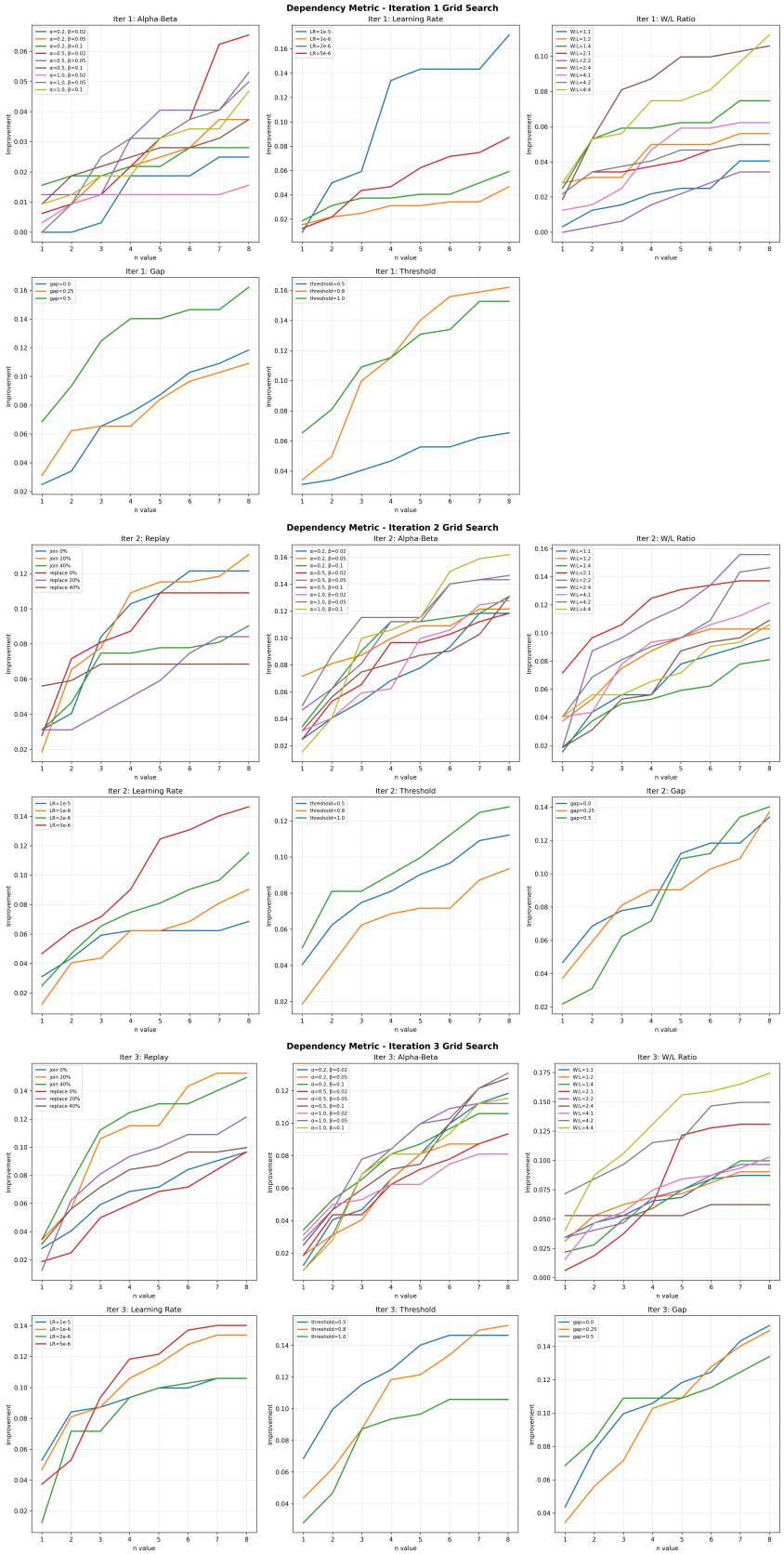


Figure 13. Hyperparameter grid searches for the **dependency** metric across iterations 1–3.

H. Qualitative Examples

H.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 14. 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 15. ImProver 2 optimizes a proof for dependencies, reducing explicit dependencies by 2.

H.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 φs1, ofCrAnListF φs2]ₛca]ₛca -
    𝓢(𝓕 |>ₛ φs1, 𝓕 |>ₛ φs2) • [ofCrAnListF φs2, [ofCrAnListF φ
      s3, ofCrAnListF φs1]ₛca]ₛca) := by
  repeat rw [superCommuteF_ofCrAnListF_ofCrAnListF]
  simp only [instCommGroup, map_sub, map_smul, neg_smul]
  repeat rw [superCommuteF_ofCrAnListF_ofCrAnListF]
  simp only [instCommGroup.eq_1, ofList_append_eq_mul, List.
      append_assoc]
  by_cases h1 : (𝓕 |>ₛ 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
```

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 φs1, ofCrAnListF φs2]ₛca]ₛca -
    𝓢(𝓕 |>ₛ φs1, 𝓕 |>ₛ φs2) • [ofCrAnListF φs2, [ofCrAnListF φ
      s3, ofCrAnListF φs1]ₛca]ₛca) := by
  simp_all [superCommuteF_ofCrAnListF_ofCrAnListF,
    instCommGroup, map_sub, map_smul, neg_smul,
    superCommuteF_ofCrAnListF_ofCrAnListF,
    instCommGroup.eq_1, ofList_append_eq_mul, List.append_assoc,
    neq_bosonic_iff_eq_fermionic]
  <;> by_cases h1 : (𝓕 |>ₛ 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
```

```
    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
```

Figure 16. 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 17. 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 18. ImProver 2 optimizes a proof for modularity, increasing modularity score (see A) 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 19. ImProver 2 optimizes a proof for modularity, increasing modularity score (see A) by 2.