FORMAL THEOREM PROVING WITH LEAN WORLD MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Most recent neural provers write a whole formal proof before Lean checks it, so they run open-loop: without intermediate observations, a wrong assumption about the proof state can derail the attempt. Many also use informal reasoning distilled from larger, more expensive models before writing Lean proofs. We propose to train a *Lean world model*, where the prover predicts Lean's goal states and error messages alongside its proof steps, learning explicit *Lean-level reasoning* from Lean's feedback rather than from a large LLM teacher. Because it predicts errors, it can repair a proof within the same response, using the failed attempt and its predicted diagnosis. We train 4B and 8B Qwen3 models by supervised fine-tuning on feedback-annotated proofs followed by reinforcement learning (GRPO), and compare them with a standard prover trained through the same stages without Lean annotations. The 8B world-model prover outperforms the standard prover, reaching 75.61% pass@32 on miniF2F and 16.44% on ProofNet against 71.31% and 13.21%. Without informal reasoning, the world-model prover solves as many miniF2F problems as the standard prover further trained on teacher-written reasoning. Adding informal reasoning raises its miniF2F pass@32 to 77.05%, above DeepSeek-Prover-V2-7B and below Goedel-Prover-V2-8B, whose post-training techniques could complement our focus on learning from Lean's feedback.

1 INTRODUCTION

When constructing a formal proof in Lean, users choose the next proof step (called a *tactic*) based on both the proof written so far and the current goal state displayed by the proof assistant. The goal state records the propositions still to be proved and the local hypotheses available for each goal; Lean updates it as tactics are executed (de Moura & Ullrich, 2021). Tactics can introduce assumptions, rewrite a target, or split a goal into several subgoals. The displayed state lets users see what each tactic has changed and what remains to be proved, without reconstructing that information from a long sequence of earlier steps.

Neural provers differ in how they obtain this state information (Figure 1). A common strategy samples many complete proof candidates, checking each script with Lean only after inference is completed. Independent attempts can be batched on GPUs without waiting for Lean between tactics, and reported budgets reach 8,192 attempts per theorem (Lin et al., 2025b; Wang et al., 2025). This explores many candidates, but the model must infer the state of its partial proof from context and generated text, without intermediate observations from Lean. Tactic-level provers instead execute proposed steps and use the returned states to guide search and backtracking (Yang et al., 2023; Lample et al., 2022). ReProver, for example, makes a single search attempt per theorem, exploring multiple tactic branches within a ten-minute wall-time limit (Yang et al., 2023). Within each attempt, every dependent continuation waits for Lean execution on the CPU before generation resumes on the GPU. Predicting feedback within generation offers a way to provide explicit state estimates while sampling proof attempts without intermediate Lean calls.

We train a *Lean world-model prover* to predict the effects of its own tactics: a single language model generates both the tactics and the feedback that Lean would return, so later steps condition on an explicit, approximate account of the proof state. We borrow the term from reinforcement learning, where a world model approximates environment dynamics from observed actions and their consequences (Ha & Schmidhuber, 2018; Hafner et al., 2025); here the environment is Lean.

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

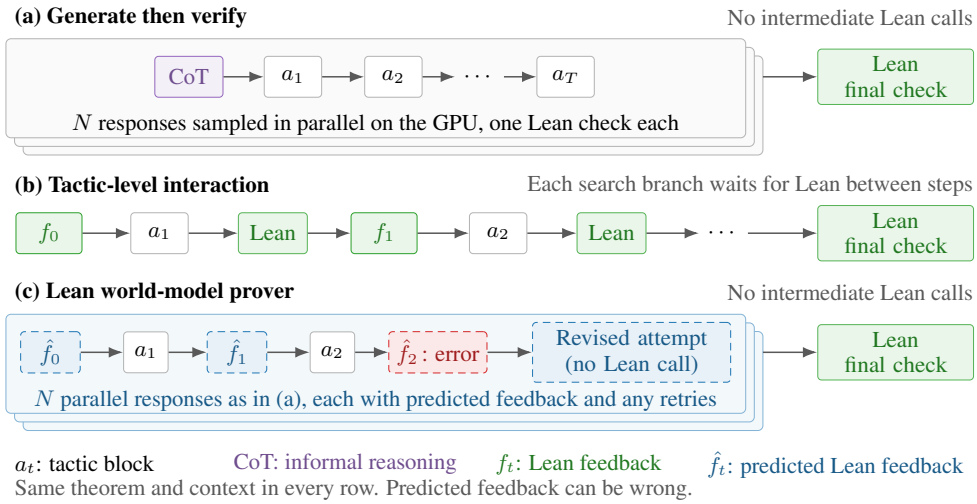


Figure 1: **Feedback during proof generation.** (a) Generating a complete proof before verification, often preceded by informal chain-of-thought (CoT) reasoning, provides no intermediate Lean observations; N responses are sampled in parallel on the GPU, each needing one Lean check. (b) Tactic-level interaction conditions each step on Lean's feedback, so each search branch waits for the verifier; independent branches can be batched. (c) The world-model prover predicts feedback within its response and can begin a new attempt after a predicted error; responses are sampled in parallel and checked once, as in (a). Section 4.3 later adds informal reasoning before each attempt.

Predicting Lean's error messages also enables self-repair: the prover can abandon a failed attempt and start a new proof within the same response, with the failed code and its predicted diagnosis in context; we submit only the last complete code block to Lean (Figure 2).

Informal mathematical planning provides guidance at the level of the argument. Lean-STaR learns informal thoughts before formal steps, DeepSeek-Prover-V2 uses informal reasoning for subgoal decomposition, and Goedel-Prover-V2 trains on proofs paired with reasoning and correction traces (Lin et al., 2024; Ren et al., 2025; Lin et al., 2025b). These traces are usually produced by much larger models: DeepSeek-Prover-V2-7B is fine-tuned on rollouts from the reinforcement-learning phase of its 671B counterpart, while Goedel-Prover-V2 uses DeepSeek-Prover-V2 models for its initial supervised data and reports collecting its self-correction data from the 671B model on 144 H100 GPUs (Ren et al., 2025; Lin et al., 2025b). Producing such traces is therefore a separate and costly stage.

Lean's feedback, by contrast, needs no teacher: our annotator collects goal states and diagnostics in one Lean execution per proof attempt (Section 3.2). Our SFT and GRPO stages therefore use only proof and feedback targets, and a later stage adds teacher-written informal plans to selected models (Section 4.3). This lets us ask two questions: can *Lean-level reasoning*, the states and diagnostics the prover learns to predict from Lean, match informal reasoning written by a teacher, and do the two combine?

We train Qwen3 base models on this data by supervised fine-tuning followed by Group Relative Policy Optimization (GRPO), and compare world-model provers with a standard prover trained through the same stages without feedback (Table 2; Figure 3). Our contributions are: **(1)** a compact feedback format and annotator that turn verified proofs and failed prefixes into proof-state and error traces, and a single prover that predicts these traces alongside its tactics and can repair a failed attempt within the same response (Section 3); **(2)** a corpus of 2,763,895 feedback-annotated direct proofs and repair histories, plus 111,880 teacher-written reasoning annotations, which we will release together with our models (Section 4.1); **(3)** GRPO with Lean-verified rewards combined with an auxiliary feedback loss on Lean-annotated samples and supervised replay (Section 3.3); **(4)** the world-model prover outperforms the standard prover, reaching 75.61% pass@32 on miniF2F and 16.44% on ProofNet against 71.31% and 13.21%; with no teacher-written reasoning, it solves as many miniF2F problems as the standard prover trained on such reasoning (Section 5).

2 Related work

Formal proof generation. LeanDojo's ReProver and HyperTree Proof Search use proof-assistant states to guide tactic search (Yang et al., 2023; Lample et al., 2022). Whole-proof provers such as DeepSeek-Prover-V2 and Goedel-Prover-V2 combine distilled informal reasoning with verifier-based reinforcement learning (Ren et al., 2025; Lin et al., 2025b), while Lean-STaR learns informal thoughts before individual tactics (Lin et al., 2024). Our prover learns explicit predictions of formal execution, which can be combined with informal reasoning about the mathematical argument. DeepSeek-Prover-V1.5 already supervises full tactic-state comments as an auxiliary prediction task and uses verifier-supplied states in truncate-and-resume search (Xin et al., 2024). Proof repair is also established: Goedel-Prover-V2 trains verifier-guided self-correction, and APRIL supervises repair and natural-language diagnosis given compiler feedback (Lin et al., 2025b; Wang et al., 2026). Our contribution is self-repair driven by the model's own predicted diagnosis: it can abandon an attempt and generate another within the same response, without Lean calls between attempts. Only the final complete candidate is checked. We also retain direct supervision of predicted states and diagnostics during reinforcement learning, using Lean annotations of the model's sampled proofs alongside verification rewards.

World models. Learned world models approximate environment dynamics to support control (Ha & Schmidhuber, 2018; Hafner et al., 2025), and language-agent world models co-train action and observation prediction in one model (Lu et al., 2026). We instantiate this idea with textual Lean observations: one autoregressive model generates tactics and compact feedback predictions that condition subsequent steps. Unlike approaches that train policies in imagined environments, our reinforcement-learning rewards come from real Lean execution.

3 The Lean world-model prover

3.1 World model and self-repair

Tactics in a Lean proof act on the goal state: the goals that remain to be proved, each with its own local hypotheses. We divide a proof attempt into tactic blocks $a_1, \ldots, a_T$, separated by feedback annotations; a block may contain several tactics or source lines. The initial feedback f_0 describes the goal state before any tactic runs: the target proposition together with the variables and hypotheses already available. Each later block f_t follows a_t and holds Lean's response to it: the goal state before the next block, together with any diagnostics for the code in a_t (Figure 1b). Feedback is derived from the text Lean displays. While it could reproduce that text in full, we use a compact representation that omits unchanged hypotheses from later state blocks (Section 3.2).

A world model predicts how an environment responds to an agent's actions (Ha & Schmidhuber, 2018; Hafner et al., 2025). Here, the environment is Lean, the actions are tactic blocks, and the observations are feedback blocks. Instead of calling Lean during generation, the prover writes predicted feedback $\hat{f}_t$ and conditions each subsequent tactic block on everything written so far (Figure 1c). An attempt has the form

$$\hat{f}_0,\ a_1,\ \hat{f}_1,\ a_2,\ \ldots,\ a_T,\ [\hat{f}_T],$$

where the optional final block holds diagnostics about the last block. For feedback supervision, the corresponding blocks contain observed feedback f_t collected from Lean. We call these explicit state and diagnostic predictions *Lean-level reasoning*, as distinct from the widely used informal chain-of-thought reasoning about the mathematical argument. A single language model generates both tactics and feedback, with no separate dynamics network or search over predicted states.

Self-repair within one response. Because the world model predicts errors as well as states, it can notice that an attempt has gone wrong and write the diagnostic Lean would report, its *error block*. A predicted error naturally triggers self-repair: the prover closes the Lean code block of the failed attempt and opens a new one for the same declaration, with the failed code and its predicted diagnosis still in context as a record of what went wrong (Figure 2). A self-repair sequence has the schematic form

$$\underbrace{\hat{f}_0^{(1)}, a_1^{(1)}, \ldots, a_{k_1}^{(1)}, \hat{f}_{k_1}^{(1)}}_{\text{attempt 1}} \longrightarrow \underbrace{\hat{f}_0^{(2)}, a_1^{(2)}, \ldots, a_{k_2}^{(2)}, \hat{f}_{k_2}^{(2)}}_{\text{attempt 2}} \longrightarrow \cdots \longrightarrow \underbrace{\hat{f}_0^{(n)}, a_1^{(n)}, \ldots, a_T^{(n)}, [\hat{f}_T^{(n)}]}_{\text{attempt } n}$$

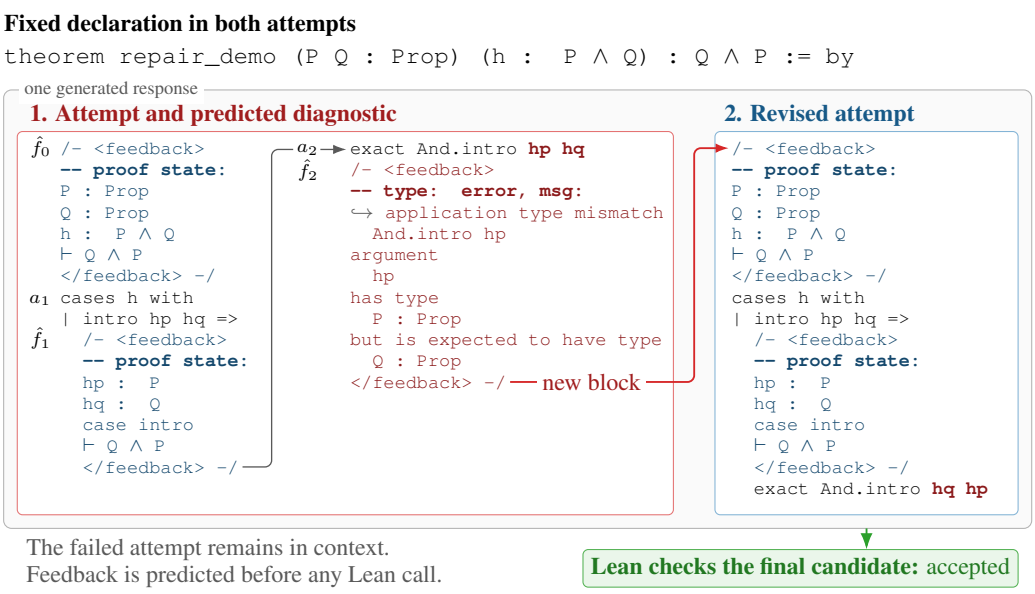


Figure 2: **State prediction and repair within one response.** This constructed example shows two attempts at the same theorem within one generated response. The world-model prover predicts every feedback block itself: after `cases h` it predicts the new hypotheses and the unchanged goal, then predicts that `exact And.intro hp hq` fails with a type mismatch, closes the attempt, and proves the theorem in a second attempt with the arguments swapped.

where every attempt but the last ends with a predicted error, each attempt starts with a fresh prediction of the initial state, and the arrows denote generated text, not Lean calls. The prover keeps re-attempting until it completes an attempt without predicting an error or exhausts its output budget, possibly leaving the last attempt unfinished. Only the last complete code block is submitted to Lean (Appendix E.2).

Proof repair itself is not new: Goedel-Prover-V2 and APRIL train a model to revise a proof given the failed attempt and the diagnostic returned by Lean (Lin et al., 2025b; Wang et al., 2026). The world-model prover instead predicts the diagnostic itself, with no Lean call (Section 3.3; Appendix A.2).

3.2 Constructing compact feedback blocks

We define a feedback block as a Lean comment of the form `/- <feedback> ... </feedback> -/` that contains a proof state, diagnostics, or both. A state block starts with `-- proof state:`, lists hypotheses (local assumptions) as `name : type`, one per line, then gives each goal's case label (when named) and its target, which begins with ⊢ and states what remains to be proved. Diagnostic entries start with `-- type: error, msg:` or `-- type: info, msg:`, followed by Lean's message. Being comments, the blocks do not affect Lean's verification of the proof. Figures 2 and 7 show how they are interleaved with proof code.

We annotate each attempt in one Lean run through LeanInteract (Poiroux et al., 2025). A state block describes the starting state of the tactic that follows it; a diagnostic block follows the tactic that produces its messages. Tactics on the same source line share one state block. We drop warnings and keep errors and informational messages.

Within an attempt, state blocks print only new or changed hypotheses: a name's first occurrence includes its full type, and a later type change prints the new type with the old type in parentheses (Figure 7a). Unchanged and consumed hypotheses are omitted; case labels and targets are always printed. Each block must therefore be read with the preceding blocks and tactics. This keeps targets shorter than with full states: compact feedback makes a target about 3.3 times as long as the proof alone, against 7.3 times with full states (Table 1).

Table 1: Token counts in three target formats for the same 5,000 randomly sampled proofs, measured with the Qwen3-8B tokenizer. Counts cover the whole target, proof and feedback. P50 and P95 denote the median and 95th percentile. Full-state feedback repeats the complete displayed state at each state annotation, whereas compact feedback lists only new or changed hypotheses.

Target format	Mean tokens	P50 tokens	P95 tokens
Clean proof only	657	444	1,893
Proof with compact feedback	2,140	1,103	7,070
Proof with full-state feedback	4,793	1,475	19,327

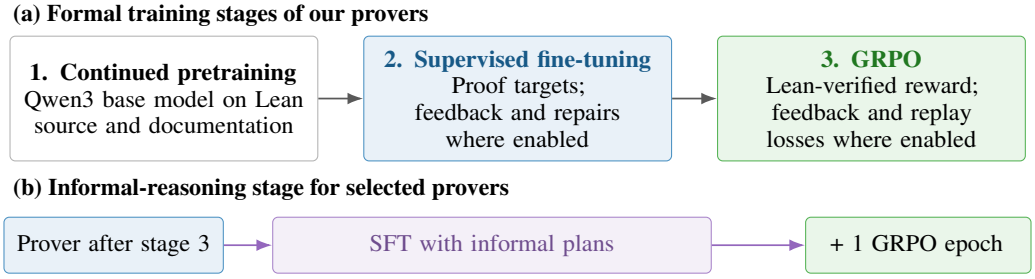


Figure 3: **Training stages.** (a) The three formal stages of each of our provers (Section 3.3); self-distillation and the GRPO problem selection are described in Appendix A.3, and all settings in Appendix B. (b) The additional informal-reasoning stage for selected provers (Section 4.3).

Annotation stops at the first error. The annotated code through the first failing line is the *failed prefix*; that line's diagnostic forms the following *error block*, and later code is omitted. Accepted proofs are annotated in full, with no success marker or state block after the last tactic.

3.3 Training the world model

Continued pretraining. The first of our three training stages (Figure 3a) adapts the base language model to Lean by next-token prediction on Lean source and documentation (Appendix B).

Supervised fine-tuning. SFT teaches the prover to write Lean proofs in its response format by minimizing next-token cross-entropy on the target tokens. Examples are plain prompt–target completions without a chat template, from which the world-model prover learns to predict both the tactics and Lean's feedback (Appendix F.1 shows the templates with examples). SFT examples come in two kinds. In a direct example, the prompt states the problem and the target is the verified proof with its feedback blocks. In a repair example, the prompt includes one or more failed attempts and stops at the line where the latest one fails, and the target begins with the error block for that line and then gives the corrected annotated proof (Appendix F.2). The second kind teaches self-repair: the model learns to recognize that its attempt has failed, to write Lean's diagnosis itself, and to fix the proof in a new attempt.

Reinforcement learning. Reinforcement learning rewards proofs accepted by Lean. Let x be the theorem prompt, $y \sim \pi_\theta(\cdot \mid x)$ a complete response sampled from the prover with parameters θ, and $c(y)$ its final proof candidate. We use GRPO, which compares rewards among responses sampled for the same problem (Shao et al., 2024), with the reward

$$r(x, y) = V(x, c(y)) - \lambda_{\text{time}} T(x, y) - \lambda_{\text{con}} C(x, y), \tag{1}$$

where $V(x, c(y)) \in \{0, 1\}$ indicates whether $c(y)$ proves the original target under the acceptance policy of Appendix E.2. $T(x, y) \in \{0, 1\}$ marks a candidate whose verification exceeds the time budget, so its correctness remains unresolved; the small penalty λ_{time} discourages exhausting the budget (Appendix D.3). $C(x, y) \in \{0, 1\}$ marks a construction violation in an unsuccessful response, such as an altered declaration or an escape hatch, penalized with λ_{con} (Table 9 lists the violations). Ordinary tactic failures incur neither a construction penalty nor a success reward.

We add an auxiliary feedback loss to counteract reinforcement of hallucinated feedback and to keep training the Lean world model during GRPO. Lean ignores comments when verifying a proof, so a rewarded response can contain incorrect state or error predictions. For each sampled candidate with nonempty Lean feedback (Table 8), we build a corrected target z: we remove its predicted feedback, run Lean on the proof code, and insert the resulting states and diagnostics in the format of Section 3.2. Thus z keeps the sampled code, through the first failing line for invalid attempts, with its predictions replaced by Lean's annotations, and we train on it by teacher forcing, conditioning each token z_t on the corrected prefix $z_{<t}$ (here t indexes tokens).

The loss scores only the feedback tokens of z, at positions $F(z)$, each with a fixed weight $w_t > 0$: the error weight if the token's block contains a Lean error message, and the default feedback weight otherwise (Table 8). Error blocks receive greater weight because they guide self-repair and are rarer than state blocks, at most one per annotated attempt. We denote the weighted cross-entropy for one annotated target by $\ell_{\mathrm{fb}}(x, z)$ and its expectation over eligible prompt–target pairs by $\mathcal{L}_{\mathrm{fb}}$:

$$\ell_{\mathrm{fb}}(x, z) = -\frac{\sum_{t \in F(z)} w_t \log \pi_\theta(z_t \mid x, z_{<t})}{\sum_{t \in F(z)} w_t}, \qquad \mathcal{L}_{\mathrm{fb}} = \mathbb{E}_{(x,z)}[\ell_{\mathrm{fb}}(x, z)]. \tag{2}$$

The GRPO loss and the reference-policy KL penalty apply to the entire sampled response y, including its predicted feedback, so feedback generation receives both a proof-level reinforcement signal on y and direct Lean supervision on z; Appendix D.2 describes a variant that masks predicted feedback from the policy-gradient loss.

Supervised replay is added to limit drift from SFT behavior and retain the learned response format. It samples complete targets from a saved supervised pool, including failed attempts followed by repairs, and applies target-token cross-entropy. We call this loss $\mathcal{L}_{\mathrm{replay}}$ and give it a small, fixed coefficient ρ. The combined minimization objective is

$$\mathcal{L} = \mathcal{L}_{\mathrm{GRPO}} + \beta \mathcal{L}_{\mathrm{KL}} + \alpha \mathcal{L}_{\mathrm{fb}} + \rho \mathcal{L}_{\mathrm{replay}}. \tag{3}$$

Here, $\mathcal{L}_{\mathrm{GRPO}}$ is the policy loss, $\mathcal{L}_{\mathrm{KL}}$ penalizes divergence from the reference policy on generated responses, and β and α weight the KL and feedback terms.

4 Data and provers

4.1 Training data

Continued pretraining uses Lean source and documentation from Mathlib and Lean Pool, about 130M tokens (Appendix B). For SFT, we build a shared collection of direct proofs and repair histories for standard and world-model provers. Its main sources are Goedel-Pset-v1 and the Goedel-Prover-V2 SFT corpus (Lin et al., 2025a;b), supplemented by NuminaMath-LEAN (Wang et al., 2025) and APRIL (Wang et al., 2026). To expand Goedel-Pset-v1, we initialize the collection with Lean-verified solutions generated by DeepSeek-Prover-V2 (Ren et al., 2025). Each subsequent round fine-tunes a standard prover for one epoch on the collected proofs, samples four attempts per remaining unsolved problem, and adds newly verified solutions. Collection continues until a round no longer solves a significant number of new problems. Failed attempts and their Lean diagnostics are retained to construct repair histories that end in a verified proof. The collection contains 2,763,895 training examples: 1,884,514 direct proofs and 879,381 repair examples.

We run Lean on the collected code to obtain compact proof-state and diagnostic annotations (Section 3.2). These annotated examples form the world-model SFT targets. The standard prover, our baseline (Section 4.2), is trained on the same proof scripts and repair histories with feedback-free targets (Appendix A.2). Both provers therefore learn from the same underlying proofs; only the world-model prover learns to generate feedback blocks. Appendix A details the sources and the construction of the targets.

After initial SFT, each prover undergoes verified self-distillation, completing stage 2 of Figure 3a, followed by GRPO on a prover-specific problem pool; Appendix A.3 describes the self-distillation targets and the GRPO problem selection.

Table 2: The four provers, each trained through the stages of Figure 3a. "Repair" denotes explicit repair supervision; "Aux. losses" indicates whether both the auxiliary feedback loss and the supervised replay loss are used during GRPO (Section 3.3).

ID	Base model	Feedback	Repair	Aux. losses	Purpose
WM8	Qwen3-8B-Base	Yes	Yes	Yes	Main method
WM8-NR	Qwen3-8B-Base	Yes	No	Yes	Repair supervision
WM4	Qwen3-4B-Base	Yes	Yes	Yes	Model scale
S8	Qwen3-8B-Base	No	Yes	No	Standard pipeline

4.2 Prover variants

All four provers of Table 2 start from Qwen3 base models (Yang et al., 2025) (Qwen3-8B-Base or Qwen3-4B-Base, not the post-trained variants) and receive the same continued pretraining on Lean (Figure 3a). WM denotes world-model provers and S the standard prover; the number gives the parameter count in billions, and NR denotes no repair supervision. None of the four initially learns informal reasoning. Our baseline, S8, is trained on the same proofs and repair histories as WM8 with the feedback blocks removed, so it writes tactics only, and it omits the feedback and replay losses ($\alpha = \rho = 0$ in Equation 3). Feedback blocks are removed only from S8's targets: its repair prompts retain Lean's annotations of failed attempts, including the error block, so Lean's diagnosis is available to it. Appendix F.1 gives the prompt and target templates, and Appendix F.2 shows the repair examples. WM8-NR excludes repair examples from SFT, self-distillation, and replay, and therefore sees no error-prediction target before GRPO, while retaining feedback prediction and the auxiliary feedback loss. Training settings are in Appendix B.

4.3 Adding informal mathematical reasoning

We add informal chain-of-thought (CoT) reasoning to WM8 and S8 to test its benefit with and without feedback prediction. Their post-GRPO checkpoints receive reasoning SFT followed by one further GRPO epoch, with the reward and losses of Section 3.3 unchanged (Figure 3b; Appendix B), yielding WM8 + reasoning and S8 + reasoning (Appendix F.5 shows the prompts and complete targets of this stage). At inference, the reasoning precedes each proof attempt. We obtain the CoT data by annotating 111,880 training examples with Kimi-K2.7-Code, at a cost of about US$1,200 (Appendix C). Unlike Lean feedback, which the verifier returns for every proof attempt at the cost of one Lean execution, these annotations require additional teacher-model inference.

5 Experimental evaluation

5.1 Evaluation protocol

We evaluate on all 488 problems of miniF2F (244 validation and 244 test problems from mathematics competitions) and all 371 problems of ProofNet (validation and test; undergraduate mathematics) (Zheng et al., 2022; Azerbayev et al., 2023). Appendix E.1 lists the benchmark versions. Acceptance requires the last complete Lean code block of a response to pass Lean verification; Appendix E.2 gives the full acceptance policy and the sampling and verification settings. Pass@32 is the percentage of problems with at least one accepted final candidate among 32 responses. Mean tokens (Tok.) averages generated tokens over all responses, including predicted feedback, abandoned attempts, and informal reasoning where present, but excluding prompt tokens.

5.2 Sharpening versus self-repair under GRPO

The standard prover sharpens. Under GRPO, S8's miniF2F pass@1 rises from 53.8% to 62.4%, but its pass@32 falls from 72.75% to 71.31% and stays at 13.21% on ProofNet (Table 3). This pattern is consistent with distribution sharpening toward proofs the model can already find (Yue et al., 2025), as observed for GRPO in Lean (He et al., 2025) and for late-stage SFT and RL in Goedel-Prover-V2 (Lin et al., 2025b).

Table 3: **Verified success and response length.** Pass: pass@32 (%, Section 5.1). Tok.: mean generated tokens per response. IDs follow Table 2. *Before GRPO* and *After GRPO*: the checkpoints entering and leaving stage 3 of Figure 3a. *After informal reasoning*: after the stage of Figure 3b (Section 4.3). Like our "+ reasoning" models, the public provers (Lin et al., 2025b; Ren et al., 2025) write informal reasoning before the proof.

	miniF2F		ProofNet	
Model	Pass ↑	Tok.	Pass ↑	Tok.
Before GRPO				
WM8 (world model)	**73.98**	5,372	**14.29**	4,951
WM8-NR (no repair)	73.57	5,296	12.67	6,651
WM4 (world model, 4B)	72.95	5,263	12.67	5,832
S8 (standard)	72.75	1,956	13.21	1,354
After GRPO				
WM8 (world model)	**75.61**	6,153	**16.44**	6,727
WM8-NR (no repair)	73.77	4,894	16.17	5,725
WM4 (world model, 4B)	74.59	5,722	**16.44**	6,339
S8 (standard)	71.31	1,095	13.21	897
After informal reasoning				
WM8 + reasoning	**77.05**	6,850	**16.17**	11,399
S8 + reasoning	75.61	2,703	14.02	2,750
Public provers				
Goedel-Prover-V2-8B	**82.58**	6,700	17.52	16,988
DeepSeek-Prover-V2-7B	75.61	3,875	**22.4**	4,946

World-model provers keep improving. WM8's miniF2F pass@1 rises from 54.9% to 65.2%, and its pass@32 rises from 73.98% to 75.61% on miniF2F and from 14.29% to 16.44% on ProofNet. WM4 also improves its pass@32, from 72.95% to 74.59% and from 12.67% to 16.44%, surpassing S8 on both benchmarks. These gains belong to the world-model recipe as a whole: S8 also omits the feedback and replay losses, and replay could itself slow sharpening (Section 4.2).

Self-repair contributes to success. GRPO reinforces an entire response whose final candidate succeeds, including predicted errors and retries generated without intermediate Lean calls (Section 3.3). WM8's repair rate, the share of responses with at least two attempts, rises from 0.4% to 18.3% on miniF2F and from 0.3% to 22.6% on ProofNet. These rates count attempts, not successful recoveries, but 4.3% and 7.7% of the problems WM8 solves after GRPO, and none before, are solved only through repair: a later attempt succeeds while every first attempt across the problem's samples has a known invalid verdict (Appendix E.4). Moreover, WM8-NR keeps feedback prediction and both auxiliary losses but excludes repair examples, so it sees no error-prediction target before GRPO and never repairs (Section 4.2). After GRPO, 7.9% and 5.4% of its responses repair, so repair supervision is not required for retries to emerge, although only 0.9% and 2.3% of the problems it solves are solved only through repair. Its pass@32 barely changes on miniF2F (73.57% to 73.77%), where WM8 gains, and rises on ProofNet (12.67% to 16.17%).

5.3 Adding informal reasoning

The third block of Table 3 adds informal reasoning distilled from a teacher model (Section 4.3) to the post-GRPO checkpoints of WM8 and S8. Without informal reasoning, WM8 matches S8 + reasoning on miniF2F. The reasoning stage raises S8's miniF2F pass@32 from 71.31% to 75.61%, a gain that its first GRPO stage does not produce, and its ProofNet pass@32 from 13.21% to 14.02%. WM8 reaches the same 75.61% on miniF2F after GRPO and a higher 16.44% on ProofNet, without informal reasoning and without the additional SFT and GRPO epoch. S8 + reasoning generates shorter responses than WM8: reasoning lengthens S8's miniF2F responses from 1,095 to 2,703 tokens, whereas WM8's predicted states and retries bring its responses to 6,153. The two solve partly different problems: each solves 4.5% of miniF2F that the other does not, so together, with 32 responses each, they solve 80.1%. Adding informal reasoning to WM8 raises its miniF2F pass@32

from 75.61% to 77.05%, our best result; its responses grow from 6,153 to 6,850 tokens. On miniF2F, informal reasoning improves a prover that already reasons about Lean states.

5.4 Comparison with public provers

The last block of Table 3 lists two public provers of similar size, Goedel-Prover-V2-8B and DeepSeek-Prover-V2-7B (Lin et al., 2025b; Ren et al., 2025). Both are trained to write informal reasoning before the proof, and Goedel-Prover-V2-8B also builds on a Qwen3 model (Yang et al., 2025), although its initialization and training data differ from ours. We evaluate both on the same benchmarks and splits as our provers, following each prover's own generation pipeline and verifying its proofs with Lean v4.9.0, the version used to train both.

Even without informal reasoning, WM8 matches DeepSeek-Prover-V2-7B on miniF2F and comes within about one percentage point of Goedel-Prover-V2-8B on ProofNet, so a prover with Lean-level reasoning can compete with provers that use informal reasoning. Its larger gaps are to DeepSeek-Prover-V2-7B on ProofNet, whose validation problems appear in varied form in DeepSeek-Prover-V2's supervised fine-tuning data (Ren et al., 2025), and to Goedel-Prover-V2-8B on miniF2F. Goedel-Prover-V2-8B is a particularly challenging baseline, as its training also uses scaffolded data synthesis and model averaging (Lin et al., 2025b), techniques that could be combined with Lean-level reasoning but are beyond the scope of this work.

We note that the public provers learn informal reasoning largely from larger models: Goedel-Prover-V2 trains on proofs with reasoning traces generated at scale with DeepSeek-Prover-V2-7B and DeepSeek-Prover-V2-671B (Lin et al., 2025b), and its released SFT corpus (Table 4) holds 1,745,010 examples, about 15.6 times our reasoning dataset; DeepSeek-Prover-V2-7B is fine-tuned on rollouts from the reinforcement-learning phase of DeepSeek-Prover-V2-671B (Ren et al., 2025). Neither paper reports the full cost of this data, whereas WM8's feedback supervision costs one Lean execution per proof attempt (Section 4.3). All of these provers, ours included, still learn from verified proofs, and ours start from proofs generated by DeepSeek-Prover-V2 and from the Goedel-Prover-V2 corpus (Section 4.1); the difference lies in the supervision added to those proofs.

5.5 Accuracy of predicted feedback

We compare the feedback blocks predicted by WM8 before and after GRPO and by WM4 and WM8-NR after GRPO with Lean's annotation of the same code, using block F1, which counts exact matches of whitespace-normalized blocks, and token F1, which gives partial credit for wording (Appendix E.3 defines the metrics; Appendix E.4 reports all results). Proof states are predicted almost exactly already before GRPO: on valid programs, block F1 is at least 0.94 and token F1 at least 0.98 in every evaluated checkpoint and on both benchmarks. Invalid programs are harder, because the model must foresee where the proof fails and reproduce Lean's message; there, WM8's block F1 rises under GRPO from 0.54 to 0.72 on miniF2F and from 0.26 to 0.29 on ProofNet.

GRPO teaches the model to foresee failures. WM8's per-attempt error recall rises from 3.3% to 70.6% on miniF2F and from 0.6% to 58.7% on ProofNet, and after GRPO 98.5% and 100% of the attempts it flags do fail; this is the ability self-repair builds on (Section 5.2). SFT supplies error targets only through repair examples, whereas during GRPO the auxiliary feedback loss adds targets from the model's own failed candidates (Section 3.3; Appendix D.1). High precision does not imply that the predicted diagnosis or its position is correct, and many failures, especially on ProofNet, still go undetected.

6 Conclusion

Our Lean world-model prover predicts Lean's states and errors alongside its tactics. Under GRPO, the standard prover (S8) sharpens on miniF2F, improving pass@1 but not pass@32, whereas WM8 improves both and retries within its responses, reaching 75.61% and 16.44% pass@32 on miniF2F and ProofNet against S8's 71.31% and 13.21% (Table 3); even the 4B world-model prover surpasses S8 on both. Predicted states are nearly exact on valid proofs, and GRPO teaches WM8 to foresee most failing attempts with almost no false alarms (Section 5.5). Without teacher reasoning, WM8 matches S8 + reasoning on miniF2F; adding reasoning raises its pass@32 to 77.05%.

REPRODUCIBILITY STATEMENT

Section 3 specifies the sequence format and learning objectives. Sections 4 and 5 define the provers and metrics; Appendix E.2 specifies the acceptance policy, sampling settings, and verification budgets. Appendices A, B, and C document the data sources, dataset sizes, target construction, the training settings of every stage, and the informal annotation prompts. Appendix D analyzes GRPO training, Appendix E lists the benchmark versions and defines and reports the feedback and repair metrics, and Appendix F gives example prompts and targets for every training format. We will release the annotated corpora, the annotation and training code, and the model checkpoints. Final checkpoint and evaluation manifests, uncertainty estimates, and the remaining experimental results are pending.

AI USE STATEMENT

Generative AI assistants Claude and Codex help draft and revise the text, find and summarize literature, suggest analyses and evaluation controls, and check reported numbers against logs and code. Kimi-K2.7-Code generates the informal reasoning annotations used for training (Appendix C), and the SFT collection includes Lean-verified proofs generated by DeepSeek-Prover-V2 and by our own provers (Section 4.1). We have reviewed all AI-assisted work and take full responsibility for the paper's content: the authors checked AI-edited claims and numbers against the experiment logs, the code, and the cited papers; AI-written code was tested before use; and every proof counted as solved is verified by Lean.

REFERENCES

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. arXiv:2302.12433, 2023. URL https://arxiv.org/abs/2302.12433.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse control tasks through world models. *Nature*, 640:647–653, 2025.

Andre Wang He, Daniel Fried, and Sean Welleck. Rewarding the unlikely: Lifting GRPO beyond distribution sharpening. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 25548–25560, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.1298. URL https://aclanthology.org/2025.emnlp-main.1298/.

Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurelien Rodriguez, and Timothee Lacroix. Hypertree proof search for neural theorem proving. In *Advances in Neural Information Processing Systems*, volume 35, pp. 26337–26349, 2022.

Haohan Lin, Zhiqing Sun, Yiming Yang, and Sean Welleck. Lean-STaR: Learning to interleave thinking and proving. arXiv:2407.10040, 2024. URL https://arxiv.org/abs/2407.10040.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover: A frontier model for open-source automated theorem proving. arXiv:2502.07640, 2025a. URL https://arxiv.org/abs/2502.07640.

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. arXiv:2508.03613, 2025b. URL https://arxiv.org/abs/2508.03613.

Ning Lu, Baijiong Lin, Shengcai Liu, Jiahao Wu, Haoze Lv, Yanbin Wei, Lingting Zhu, Shengju Qian, Xin Wang, Ying-Cong Chen, Qi Wang, and Ke Tang. Policy and world modeling co-training for language agents. arXiv:2606.02388, 2026. URL https://arxiv.org/abs/2606.02388.

Auguste Poiroux, Viktor Kuncak, and Antoine Bosselut. LeanInteract: A python interface for Lean 4. Software, 2025. URL https://github.com/augustepoiroux/LeanInteract.

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

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

Evan Wang, Simon Chess, Daniel Lee, Siyuan Ge, Ajit Mallavarapu, Jarod Alper, and Vasily Ilin. Learning to repair Lean proofs from compiler feedback. arXiv:2602.02990, 2026. URL https://arxiv.org/abs/2602.02990.

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

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search. arXiv:2408.08152, 2024. URL https://arxiv.org/abs/2408.08152.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, et al. Qwen3 technical report. arXiv:2505.09388, 2025. URL https://arxiv.org/abs/2505.09388.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. arXiv:2306.15626, 2023. URL https://arxiv.org/abs/2306.15626.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? In *Advances in Neural Information Processing Systems*, volume 38, pp. 57654–57689, 2025. doi: 10.52202/085713-1933. URL https://arxiv.org/abs/2504.13837.

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.

A Data construction

A.1 Supervised sources

Table 4 lists the public sources and their roles in the shared SFT collection. Section 4.1 describes how iterative proof generation expands this collection and supplies failed attempts for repair histories.

Table 4: Public sources and their roles in the shared SFT proof collection.

Source	Role in the collection
`Goedel-LM/Goedel-Pset-v1`	Formal statements for initial and iterative proof generation.
`Goedel-LM/SFT_dataset_v2`	Supplied proofs and repair histories, without their informal reasoning.
`AI-MO/NuminaMath-LEAN`	Supplied direct proofs.
`uw-math-ai/APRIL`	Direct proofs and failed/corrected proof pairs.

A.2 Feedback and repair targets

We write SFT examples in the notation of Section 3. Let x be the base prompt, with the instruction, the theorem statement, and its Lean context (the instruction differs between the two provers; Appendix F.1), $a_1, \ldots, a_T$ the tactic blocks of a verified proof, and $f_0, \ldots, f_T$ the compact feedback blocks that Lean returns for them (Section 3.2), where f_0 is the initial state and the optional $[f_T]$ holds diagnostics for the last block. In a repair history, the latest failed attempt has blocks $a'_1, \ldots, a'_k$ up to its first failing line, feedback $f'_0, \ldots, f'_{k-1}$, and the error block f'_k for that line. With $\Rightarrow$ separating prompt and target, and omitting code fences and surrounding text, the main prompt–target formats are

$$\begin{aligned}
\text{world model, direct:} \quad & x \;\Rightarrow\; f_0, a_1, f_1, \ldots, a_T, [f_T], \\
\text{world model, repair:} \quad & x, \underbrace{f'_0, a'_1, \ldots, f'_{k-1}, a'_k}_{\text{failed prefix (input)}} \;\Rightarrow\; \underbrace{f'_k}_{\text{error block (target)}}, \; f_0, a_1, f_1, \ldots, a_T, [f_T], \\
\text{standard, direct:} \quad & x \;\Rightarrow\; a_1, \ldots, a_T, \\
\text{standard, repair:} \quad & x, \underbrace{f'_0, a'_1, \ldots, a'_k, f'_k}_{\text{failed prefix and error block (input)}} \;\Rightarrow\; a_1, \ldots, a_T.
\end{aligned}$$

The world-model repair prompt stops at the failing line, so the target begins with the error block, which the model learns to write before the corrected proof. The standard repair prompt keeps Lean's annotation of the failed attempt, including the error block, so the standard prover receives Lean's diagnosis, and its target is the bare corrected proof. In training, all these blocks are Lean's; when the world-model prover repairs at inference, its failed prefix carries its own predicted blocks $\hat{f}$ (Section 3.1), whereas the standard prover sees Lean's blocks only in training, since our provers never call Lean during generation. When a history contains several failed attempts, the earlier ones precede the latest in the prompt, each with its annotation and error block. The world-model collection also keeps some direct proofs without annotations, so that the prover sees both formats. Appendix F.1 gives the prompt and target templates with complete examples, and Appendix F.2 shows repair examples.

A.3 Model-specific self-distillation and RL selection

We use 94,701 problems from the Goedel-Prover-V2 RL problem set (Lin et al., 2025b), reserving a random subset of 29,480 for self-distillation and a separate pool of 65,221 for GRPO selection (Table 5).

For self-distillation, each prover's initial SFT checkpoint samples 16 responses per problem. Every Lean-verified solution is retained as a supervised target, so one problem may contribute multiple targets. Responses of the world-model provers whose final attempt succeeds after an earlier failure are retained as repair examples, except for WM8-NR, which excludes them. For world-model targets, predicted feedback is replaced with annotations returned by Lean; standard targets have inline feedback removed. Each model then trains on its retained targets by SFT at a learning rate of 2×10^{-6}.

Each self-distilled checkpoint generates eight responses per problem in the shared GRPO candidate pool. For each model independently, let $\hat{p}_{\mathrm{pre}}(x)$ be the number of responses accepted by Lean within the verification budget divided by eight. We retain problems satisfying

$$0 < \hat{p}_{\mathrm{pre}}(x) \leq 0.75. \tag{4}$$

Equation 4 selects problems with some verified successes but room for improvement. It follows the $(0, 0.75]$ range used by Goedel-Prover-V2 and the moderate-success-rate selection of DeepSeek-Prover-V1.5 (Lin et al., 2025b; Xin et al., 2024). Groups with equal rewards have zero group-relative

advantages; unsuccessful responses can nevertheless receive different penalties under Equation 1. Appendix B specifies the continued-pretraining, supervised-training, sampling, and GRPO settings.

Table 5: Dataset sizes. GRPO candidates are counted before model-specific filtering.

Data	Count
Initial SFT training examples	2,763,895
Of which repair examples	879,381
Self-distillation input problems	29,480
GRPO candidate problems	65,221

B TRAINING CONFIGURATION

WM8, WM8-NR, and S8 use Qwen3-8B-Base; WM4 uses Qwen3-4B-Base. Before supervised fine-tuning, the base models undergo continued pretraining (CPT) on Lean source and documentation. The subsequent stages are SFT, consisting of initial SFT and verified self-distillation, and GRPO (stages 2 and 3 of Figure 3a). Selected post-GRPO checkpoints additionally receive informal-reasoning SFT followed by a further GRPO stage. Tables 6–9 list the settings of every stage; entries marked *as configured* are default values of our training scripts.

Each prover uses its own self-distillation targets and filtered GRPO problem pool. WM8-NR excludes repair examples from initial SFT, self-distillation SFT, and supervised replay. The standard prover uses neither the auxiliary feedback loss nor supervised replay.

Table 6: Continued pretraining.

Stage	Setting	Value
CPT	Corpus	Lean source and documentation from Mathlib and Lean Pool; 469,585 documents, 408 MB of text, approximately 130M tokens
CPT	Objective	Next-token prediction; full fine-tuning
CPT	Context and packing	32,768 tokens; packing enabled
CPT	Learning rate and duration	5×10^{-6}; one epoch
CPT	Optimizer	AdamW
CPT	Schedule and warmup	Cosine decay; warmup over 3% of training steps
CPT	Batch and accumulation	One packed sequence per device; one accumulation step
CPT	Precision	BF16; TF32 enabled
CPT	Gradient clipping	Maximum gradient norm 1.0
CPT	Implementation	LLaMA-Factory; DeepSpeed ZeRO-2; FlashAttention-2; gradient checkpointing

Table 7: Supervised stages, the sampling used for self-distillation, and the pre-RL problem filter. Common SFT settings apply to initial, self-distillation, and informal-reasoning SFT.

Stage	Setting	Value
All SFT	Implementation	LLaMA-Factory; full fine-tuning; DeepSpeed ZeRO-2; FlashAttention-2
All SFT	Context	32,768 tokens
All SFT	Packing	Packed sequences with per-sequence attention masks
All SFT	Formatting and loss	No chat template; prompt tokens excluded from the loss
All SFT	Optimizer and schedule	Fused AdamW; cosine decay
All SFT	Batch and accumulation	One packed sequence per device; one accumulation step
All SFT	Precision	BF16; TF32 enabled
All SFT	Gradient clipping	Maximum gradient norm 1.0
All SFT	Random seeds	Training seed 42; data seed 42
Initial SFT	Starting checkpoint	Corresponding CPT checkpoint
Initial SFT	Learning rate and duration	10^{-5}; one epoch
Initial SFT	Warmup	5% of training steps
Self-distillation	Starting checkpoint	Each prover's initial SFT checkpoint
Self-distillation	Generation	16 responses per problem; temperature 0.5; top-p 0.95
Self-distillation	Target selection	Retain verified solutions; WM8-NR excludes repair examples
Self-distillation	Feedback targets	Replace predicted feedback with Lean annotations for world-model targets
Self-distillation	SFT learning rate and duration	2×10^{-6}; one epoch
Self-distillation	SFT warmup and weight decay	3% of training steps; weight decay 0
Pre-RL filter	Sampling and selection	Eight responses per problem and model; retain $0 < \hat{p}_{\mathrm{pre}} \leq 0.75$
Informal reasoning	Starting checkpoint	Selected post-GRPO checkpoint
Informal reasoning	SFT learning rate and duration	10^{-5}; two epochs
Informal reasoning	SFT warmup	5% of training steps

Table 8: Auxiliary supervision during world-model GRPO. These losses are absent from the standard prover.

Stage	Setting	Value
Feedback loss	Coefficient	$\alpha = 0.1$
Feedback loss	Token weights	0.3 for non-error feedback; 0.5 for feedback blocks containing an error
Feedback loss	Reduction	Weighted token cross-entropy normalized by the sum of token weights, as in Section 3.3
Feedback loss	Eligible code	Successful and failed candidates; failed annotations stop at the first error, as configured
Feedback loss	Payload budgets	4,000 characters per feedback-block body and a nominal 16,000-character aggregate budget, with oversized feedback elided, as configured
Replay	Coefficient	$\rho = 0.1$
Replay	Sampling	128 global replay examples per optimizer step
Replay	Pool	161,030 direct and repair examples drawn at random from the SFT collection; WM8-NR excludes repair examples
Replay	Loss	Cross-entropy on target tokens only
Replay	Length filter	Prompt plus target at most 32,768 tokens, as configured
Replay	Microbatch and seed	Microbatch size 1; sampling seed 1234, as configured

Table 9: Shared GRPO settings. The informal-reasoning continuation uses the same settings except for its training duration. Validation settings refer to monitoring during training, not to the final benchmark protocol.

Stage	Setting	Value
GRPO	Implementation	Customized verl; vLLM rollouts
GRPO	Starting checkpoint	Self-distillation SFT checkpoint; informal-reasoning SFT checkpoint for the later continuation
GRPO	Rollout batch	128 prompts, eight responses per prompt: 1,024 responses
GRPO	Initial duration	Two epochs
GRPO	Reasoning continuation	One additional epoch
GRPO	Learning rate and warmup	3×10^{-6}; 25 warmup steps
GRPO	Prompt and response limits	2,048 prompt tokens; 30,720 response tokens; 32,768 total context
GRPO	Training sampling	Temperature 1.0; top-p 0.95
GRPO	KL regularization	Coefficient $\beta = 0.1$; low-variance KL estimator; separate loss, not part of the scalar reward
GRPO	Entropy coefficient	0
GRPO	Loss aggregation	Sequence mean of token means
GRPO	PPO minibatch	128 prompts before expansion into rollouts
GRPO	Actor batching	One response per micro-batch on each GPU; dynamic batching off, as configured
GRPO	Advantage normalization	Standard-deviation floor $(1 - 10^{-6})/\sqrt{8}$, as configured
GRPO	Data seed	1234, as configured
GRPO	Verification	Lean v4.15.0; 60 s candidate-execution timeout; 800,000 heartbeats, as configured
GRPO	Success reward	1 for a verified proof; 0 for an ordinary proof failure
GRPO	Reward penalties	Construction: 0.2; candidate timeout: 0.02; nontermination: 0
GRPO	Construction violations	Missing or altered declaration; changed context; an admission, unauthorized axiom, or other escape hatch; an unusable code block; an exact repeated candidate; penalized only in unsuccessful responses
GRPO	Validation frequency	Every 25 training steps
GRPO	Validation sampling	Held-out validation split of the RL problem set; temperature 0.3; top-p 0.95; one response per problem; at most 500 problems, as configured

C INFORMAL REASONING ANNOTATIONS

This appendix documents the informal-reasoning data of Section 4.3 (Figure 3b). Each annotated example receives one reasoning segment per attempt: a solution path with the first seven fields of Table 10 for a direct proof or for the first attempt of a repair history, and a repair segment with the last three fields for each later attempt. Appendix C.1 summarizes the annotated examples, and Appendix C.2 gives the teacher prompts.

Table 10: Informal annotation fields and their intended roles.

Field	Role
Problem Restatement	Mathematical task.
Useful Observations	Facts motivating the approach.
Reasoning Path	Development of the mathematical argument.
Key Derivation	Central equations or deductions.
Proof Type	Form of argument, including vacuity when applicable.
Formal Anchors	Connections between key steps and the supplied proof.
Final Solution	Compact, self-contained explanation.
Error Analysis	Diagnosis of the preceding attempt's failure.
Revised Reasoning	Development of the next proposed argument.
Updated Proof Plan	Outline of the next attempt.

C.1 REASONING DATASET

We annotate 111,880 training examples from the SFT collection (Section 4.1): NuminaMath-LEAN proofs and a random sample of examples from the other sources, 83,177 direct proofs and 28,703 repair histories in total, each with one reasoning segment per attempt (Table 11). Most repair histories (25,636) contain one failed attempt before the final proof; the remaining 3,067 contain two to five. Producing the annotations costs about US$1,200.

Table 11: Annotated training examples by source.

Source	Direct	Repair	Total
Goedel-Pset-v1	29,140	9,921	39,061
NuminaMath-LEAN	33,120	0	33,120
Goedel-Prover-V2 SFT corpus	14,517	10,190	24,707
APRIL	6,400	8,592	14,992
Total	83,177	28,703	111,880

C.2 ANNOTATION PROMPTS

Kimi-K2.7-Code with high reasoning effort generates every annotation; Appendix F.4 shows generated examples. The token ranges in the prompts are budgets, not measured output lengths. For a direct proof, the model receives the verified proof on standard input together with the prompt below and writes the seven-section solution path.

Direct-proof prompt

```
Produce training-data-friendly mathematical reasoning for the solved Lean proof in
stdin.

Goal:
Write an informal solution path that reads like a human deriving the proof from the
problem statement, not like a commentary on existing code.

Requirements:
- Write 700-5000 tokens, scaling length to the complexity of the proof.
- Prioritize mathematical reasoning, useful observations, and why the key moves are
natural.
- Use the Lean proof only as a ground-truth source for the verified argument.
- Do not present the text as a line-by-line explanation of the Lean code.
- Do not over-emphasize tactics, temporary hypothesis names, or implementation
details.
- Short Lean snippets are allowed only in the [Formal Anchors] section.
- Explain how one might discover the key substitutions or special cases.
- Do not invent a different proof or add unsupported claims.
- Do not include fake failed attempts, private chain-of-thought, or speculation.
- If the proof works because the hypotheses are inconsistent, clearly flag that as a
 vacuous proof.
- Avoid repeating the same derivation across sections.

Output format:

[Problem Restatement]
Restate the mathematical claim in plain language.

[Useful Observations]
List the observations that make the proof direction natural.

[Reasoning Path]
Give the main derivation as if solving the problem from scratch. Focus on why each
special point, substitution, or transformation is chosen.

[Key Derivation]
Present the central equations cleanly and sequentially.

[Proof Type]
```

```
State whether this is a direct proof, contradiction proof, vacuous proof from
inconsistent hypotheses, case split, etc.

[Formal Anchors]
Briefly connect the reasoning to the Lean proof using only a few key snippets or
theorem steps.

[Final Solution]
Give a polished compact solution that could stand alone without Lean.
```

For a repair history, the first segment is produced with the same direct-proof prompt applied to the first attempt, extended by the hidden-context block below, which asks for the confident reasoning behind that attempt rather than a commentary on its failure.

Hidden context appended for the first segment of a repair history

```
Hidden context for this specific input, which you must never reveal, reference, or
hint at in the output:
- The proof in stdin was written by a solver who fully believed it was correct and
complete. Treat it as the complete correct solution and write the reasoning
accordingly, in the solver's confident voice.
- If the code looks unfinished or ends abruptly, it was merely cut off during
copying: silently infer the most natural intended completion of the argument and
write the reasoning as if that full argument were given. Never mention that anything
 is missing or unfinished.
- Never question, verify, or comment on the correctness, completeness, or
compilability of the proof or of the statement, even if you privately doubt a step;
present the argument the attempt embodies as a confident derivation.
- Never address the reader with requests, apologies, or caveats.
- Fill every required section with confident mathematical content exactly as
specified above; never leave a section empty or state that content is unavailable.
```

Each later segment receives the problem statement, the previous attempt with its error block, the segment already written for it, and the new attempt, together with the prompt below. The sentence "The new attempt is the final proof and it is verified correct." is used for the verified final attempt; for an intermediate attempt that also fails, it is replaced by the hidden-context block shown after the prompt.

Repair-segment prompt (new attempt is the final, verified proof)

```
Write the NEXT segment of a training-data-friendly reasoning trace for the Lean 4
proof repair given in stdin.

stdin contains:
1. the problem statement,
2. the previous proof attempt, annotated with the compiler error that ended it,
3. the reasoning segment already written for that previous attempt,
4. the new proof attempt.

The segment you write must read as the same person continuing the same train of
thought: they just saw the compiler error, they diagnose it, and they derive the new
 attempt.

The new attempt is the final proof and it is verified correct.

Requirements:
- Write 300-2000 tokens, scaling with the complexity of the change.
- Open by reacting naturally to the compiler error (for example noticing that a
lemma application was rejected) and explain concretely what was wrong in the
previous reasoning.
- Paraphrase the error in mathematical terms; do not quote the raw feedback blocks
at length.
- Then derive the new attempt as a confident, forward-looking continuation, not as a
 diff description of the code change.
- Discuss only the error that actually occurred; never foreshadow later failures.
- Stay consistent with the previous reasoning segment; refer back to it naturally
where helpful.
- Do not write a line-by-line commentary on the code; short Lean snippets only where
 genuinely clarifying.
- Do not describe moves the new attempt does not actually make.
```

```
Output format:

[Error Analysis]
What went wrong in the previous attempt and why.

[Revised Reasoning]
The corrected derivation and why the new key moves are right.

[Updated Proof Plan]
A compact, confident summary of the plan the new attempt implements.
```

Hidden context used instead when the new attempt also fails

```
Hidden context you must never reveal: the new attempt will also fail to compile later
. Write as someone who does not know that. The new attempt's code may be cut off at
the point where the compiler stopped; if it looks incomplete, infer the most natural
 intended continuation of the plan and present it seamlessly.
```

D GRPO TRAINING ANALYSIS

D.1 AUXILIARY FEEDBACK LOSS

Figure 4 shows the auxiliary feedback loss (Section 3.3) during the first of WM8's two GRPO epochs, 170 steps. Averaged over the first and the last ten steps of the epoch, the cross-entropy on error blocks falls from 0.088 to 0.029, and the cross-entropy over all feedback blocks, including proof-state blocks, falls from 0.0093 to 0.0037. The ratio of the error-block to the all-feedback cross-entropy is about 9.5 over the first ten steps and 7.9 over the last ten, in line with the gap between predicted states and predicted failures in Section 5.5.

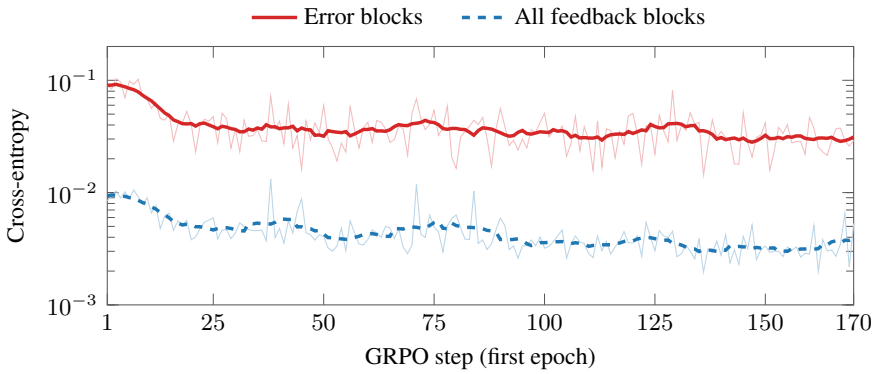


Figure 4: **Auxiliary feedback loss during WM8's first GRPO epoch.** Teacher-forced cross-entropy on the Lean annotations of the model's own sampled candidates, over the tokens of all feedback blocks and of error blocks only (log scale). Thin lines show per-step values; thick lines show an 11-step centered moving average.

D.2 FEEDBACK MASKING

Section 3.3 applies the GRPO loss and the KL penalty to the whole response, including predicted feedback. We also train a variant that masks predicted feedback from the policy-gradient loss, separating policy and feedback supervision, analogous to the action and observation losses in language-agent world-model co-training (Lu et al., 2026); the KL penalty still covers these tokens. Our runs with this masking become unstable, as in the run of Figure 5. After about 70 steps, the model begins to write gibberish, and training degrades. We flag gibberish with a simple heuristic for scripts we do not expect in these Lean responses: a response is flagged if it contains, anywhere, at least 30 characters from CJK, Hangul, Kana, Cyrillic, Arabic, Hebrew, Devanagari, or Thai. Greek letters and

mathematical symbols do not count, since Lean proofs use them. The heuristic can flag meaningful text and misses gibberish in Latin script. Around the same time, the model stops repairing: the share of responses that complete a second attempt, about 20% over steps 60–67, falls to nearly zero by step 90 and does not recover. This is not a controlled ablation: besides masking, the run uses one GRPO epoch instead of two, additional reward penalties, a different replay pool, and an older version of our training code, and we have not identified the cause. Several masked runs collapse in this way. Gibberish also appears in unmasked runs without supervised replay: among our world-model runs, avoiding it requires both training without masking and supervised replay (Section 3.3), which all our world-model provers use.

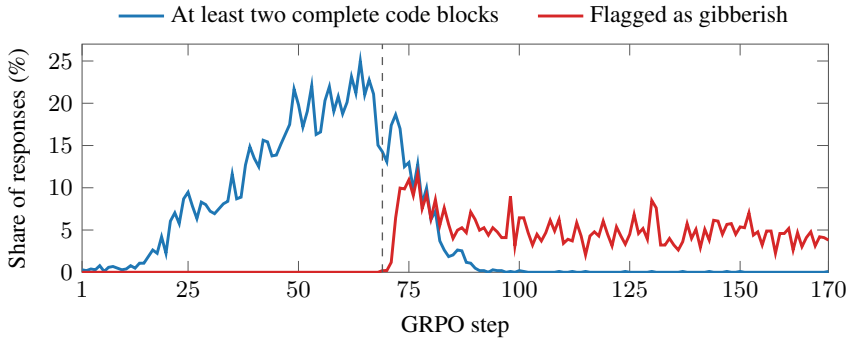


Figure 5: **A GRPO run with predicted feedback masked from the policy-gradient loss.** Per step, the share of the 1,024 sampled responses that complete at least two code blocks (a stricter criterion than that used for the repair rate in Appendix E.4, which also counts unfinished attempts) and the share flagged by the gibberish heuristic of this subsection (at least 30 characters from the listed scripts in a response). The dashed line marks the first step with a flagged response (step 69).

D.3 Verifier timeouts

During GRPO, Lean checks the final candidate of each sampled response with a time limit of 60 s (Table 9). A candidate that exceeds it receives no verdict: its response stays in its GRPO group but earns no success reward, receives the small penalty $\lambda_{\text{time}} = 0.02$ of Equation 1, and yields no target for the auxiliary feedback loss. By contrast, a response whose verification fails for infrastructure reasons, not because of its proof, is dropped from its group. In WM8's first GRPO epoch, 4.6% of the 1,024 rollouts per step time out on average (Figure 6). We keep the 60 s limit because such candidates often contain tactics whose checking runs far longer, so a longer limit would slow training considerably for little expected benefit. The penalty is meant to discourage these costly tactics. Consistent with this aim, the timeout rate falls slightly, from 4.8% over steps 1–25 to 4.3% over steps 146–170, and to 3.5% over steps 229–253 in the second epoch.

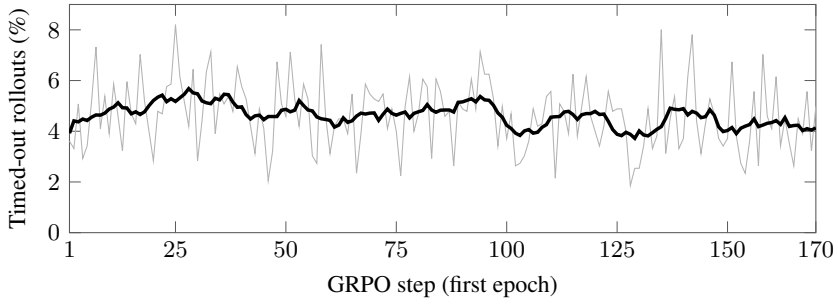


Figure 6: **Verifier timeouts during WM8's first GRPO epoch.** Share of the 1,024 rollouts per step whose final candidate exceeds Lean's 60 s limit. The thin line shows per-step values; the thick line shows an 11-step centered moving average.

E EVALUATION DETAILS

E.1 BENCHMARK VERSIONS

Table 12 lists the benchmark files we use. For miniF2F, we take the statements from the Goedel-Prover repository and apply the corrections released with Goedel-Prover-V2 (Lin et al., 2025a;b). We evaluate both the validation and the test split of miniF2F and ProofNet.

Table 12: Benchmark versions.

Benchmark	Problems	Source
miniF2F	488	`Goedel-LM/Goedel-Prover`, `datasets/minif2f.jsonl`, with corrections from `Goedel-LM/Goedel-Prover-V2`, `dataset/minif2f.jsonl`
ProofNet	371	`deepseek-ai/DeepSeek-Prover-V1.5`, `datasets/proofnet.jsonl`

E.2 ACCEPTANCE POLICY, SAMPLING, AND VERIFICATION

Acceptance policy. Predicted feedback blocks are Lean comments; they are removed before verification and play no role in the verdict (Section 3.2). A final candidate is accepted if it keeps the benchmark's imports, Lean compiles it without errors, and it contains a declaration whose elaborated signature, as Lean prints it, equals that of the original statement. Candidates that contain `sorry` or `admit` (placeholders that close a goal without a proof), or that declare axioms, opaque constants, or unsafe definitions, are rejected before compilation.

Sampling and verification. For pass@32, each of our provers samples 32 responses per problem at temperature 0.7 and top-p 0.95, with at most 32,768 generated tokens per response. The last complete Lean code block of a response is its final candidate (Section 3.1); a response without one fails. Lean v4.15.0 with Mathlib checks each final candidate with a time limit of 180 s and a limit of 800,000 heartbeats (Lean's deterministic measure of computation); a candidate that exceeds either limit counts as unsolved. Public provers are evaluated with the same sample count, following their own generation pipelines, with their proofs verified using Lean v4.9.0, the version used to train them (Section 5.4).

A response counts as one sample even when it contains several attempts: only its final candidate is checked, and earlier attempts never count as additional samples. For pass@1 on miniF2F, each of our provers samples one response per problem at temperature 0.3.

E.3 FEEDBACK-ACCURACY METRICS

Section 5.5 compares the feedback that the world-model provers predict, without informal reasoning, with the feedback Lean returns for the same programs. This subsection defines the metrics; Appendix E.4 reports the results.

Programs and references. We evaluate every feedback block in the responses that four world-model checkpoints generate for miniF2F and ProofNet problems: WM8 before and after GRPO, WM4 after GRPO, and WM8-NR after GRPO. For a program i, $\hat{z}_i$ is the code together with the feedback blocks the model predicts for it. To build the reference z_i, we remove these blocks, run Lean on the remaining code, and insert Lean's states and messages in the format of Section 3.2. As in training, annotation stops at the first error, so the reference of an invalid program contains no blocks after the error block of its first failing line. The annotator marks a program valid if Lean reports no error and no `sorry` for it, unresolved if the check times out or hits an infrastructure failure, and invalid otherwise.

Blocks, anchors, and tokens. A feedback block is a complete comment that opens with `/-<feedback>` and closes with `</feedback> -/`; the tags match regardless of case and surrounding whitespace, and an unclosed block is not extracted. Its content is the text between the tags, with leading and trailing whitespace removed and every internal run of whitespace replaced by a single

space, so that indentation and line wrapping do not affect matching; case, identifiers, and punctuation are kept. Let B_i and $\hat{B}_i$ be the multisets of block contents in z_i and $\hat{z}_i$, in which repeated blocks keep their multiplicity. The anchor of a block is the number of nonempty lines of code before it in its own program, counted after earlier feedback blocks have been removed, so that they do not shift later anchors; A_i and $\hat{A}_i$ are the multisets of (anchor, content) pairs. Finally, T_i and $\hat{T}_i$ pool the tokens of all block contents on each side, where the regular expression `\w+|[^\w\s]` splits each block's content into maximal runs of Unicode word characters and single characters that are neither word characters nor whitespace.

Multiset F1. For a reference multiset G and a predicted multiset P, let $c_G(u)$ and $c_P(u)$ count the occurrences of an item u, and let $O(G,P) = \sum_u \min\{c_G(u), c_P(u)\}$ be the size of their overlap. The F1 score between them is

$$F(G,P) = \begin{cases} 1, & |G| + |P| = 0, \\ \dfrac{2\,O(G,P)}{|G| + |P|}, & \text{otherwise,} \end{cases}$$

so predicting no feedback where Lean returns none counts as agreement, and a program with feedback on only one side scores 0. The three per-program scores are

$$\mathrm{BlockF1}_i = F(B_i, \hat{B}_i), \qquad \mathrm{AnchoredF1}_i = F(A_i, \hat{A}_i), \qquad \mathrm{TokenF1}_i = F(T_i, \hat{T}_i).$$

Block F1 counts a predicted block only if its normalized content equals that of a reference block and ignores the order of the blocks; position-anchored F1 also requires the same anchor, so a block counts only if it has the right content and follows the same number of lines of code; token F1 gives partial credit when the predicted wording differs from Lean's and ignores block boundaries, order, and positions.

Error-presence detection. Error presence is scored over all labelled attempts, using Lean's errors as ground truth: an attempt is a reference positive if Lean reports an error for it, and a predicted positive if its text contains an error entry (Section 3.2), that is, a match for `--\s*type\s*:\s*error\b` regardless of case. Precision is the fraction of predicted positives that are reference positives, recall is the fraction of reference positives that are predicted positives, accuracy is the fraction of attempts classified correctly, and F1 is the harmonic mean of precision and recall. These metrics ignore the text and position of the predicted error.

Aggregation. The three feedback F1 scores are averaged over programs, overall and separately for valid and invalid programs: for a set S of programs and a per-program score m_i, we report $\bar{m}_S = \frac{1}{|S|}\sum_{i \in S} m_i$. Each program receives equal weight; counts are not pooled across programs. The error-presence metrics are computed over all labelled attempts.

E.4 Feedback prediction and repair across provers

This subsection compares four world-model checkpoints, WM8 before and after GRPO, WM4 after GRPO, and WM8-NR after GRPO; the repair statistics also include WM8-NR before GRPO. Table 13 reports the feedback accuracy defined in Appendix E.3, Table 14 the per-attempt error presence, and Table 15 how often responses repair. A response *repairs* if it contains at least two attempts, counting an attempt whose code block is left incomplete, and *attempts* is the mean number of attempts per response. A problem is *solved only through repair* if a later attempt succeeds while every first attempt across all its samples has a known invalid verdict; we report these problems as a percentage of the problems solved.

States are learned in SFT. On valid programs, block F1 is at least 0.94 and token F1 at least 0.98 in every checkpoint, and WM8's valid-program block F1 barely changes under GRPO (0.98 to 0.97 on miniF2F, 0.94 to 0.95 on ProofNet). Requiring the correct anchor lowers mean per-program F1 by at most 0.003 in every group.

Errors are learned in GRPO. Before GRPO, WM8 flags almost none of the attempts that Lean rejects (recall 3.3% on miniF2F and 0.6% on ProofNet), and its invalid programs reach block F1 of

only 0.54 and 0.26. After GRPO, recall rises to 70.6% and 58.7% at a precision of at least 98.5%, and invalid-program block F1 rises to 0.72 and 0.29. These gains do not extend to every feedback metric: WM8's overall token F1 on miniF2F falls from 0.82 to 0.79, and its overall ProofNet block F1 stays at 0.32. SFT offers few error targets: state blocks recur along every proof, whereas each repair target supplies a single error block and direct proofs supply none (Appendix A.2). GRPO adds error targets from the model's own failed candidates, annotated by Lean through the first error, and the teacher-forced cross-entropy on error blocks falls from 0.088 to 0.029 during WM8's first epoch (Appendix D.1). One likely reason for the failures WM8 still misses is their diversity: a tactic may not prove its goal, a lemma may not apply, a name may be unavailable, or an expression may have the wrong type, and many such failures are rare in any given context.

Repair follows error prediction. Before GRPO, WM8 repairs in 0.4% and 0.3% of its responses, and no problem is solved only through repair. After GRPO, it repairs in 18.3% and 22.6% of its responses, with 1.84 and 2.23 attempts per response on average, and 4.3% and 7.7% of the problems it solves are solved only through repair. WM4 behaves alike: it flags 70.7% and 64.9% of failing attempts, repairs in 17.4% and 27.9% of its responses, and solves 3.5% and 9.3% of its solved problems only through repair; on ProofNet, it predicts invalid programs more accurately than WM8 (block F1 0.49 against 0.29).

Repair without repair supervision. WM8-NR, which receives no repair examples, never repairs before GRPO. After GRPO, it flags 53.1% and 39.2% of failing attempts, without false alarms, and repairs in 7.9% and 5.4% of its responses. Repair thus emerges during GRPO without repair supervision, but more weakly than with it: only 0.9% and 2.3% of the problems WM8-NR solves are solved only through repair. It also has the highest valid-program block F1 of all checkpoints, 0.98 on both benchmarks.

The repair rates count attempts and do not check that each earlier attempt fails; the share of problems solved only through repair is the measure that ties repair to success.

Table 13: Accuracy of the predicted feedback (without informal reasoning), for all, valid, and invalid programs. Each score is the mean per-program F1 of Appendix E.3: block, position-anchored, and token F1.

		miniF2F			ProofNet		
Checkpoint	Programs	Block	Position	Token	Block	Position	Token
WM8, before GRPO	All	0.7995	0.7988	0.8239	0.3189	0.3187	0.4358
	Valid	0.9784	0.9784	0.9957	0.9447	0.9447	0.9921
	Invalid	0.5383	0.5366	0.5732	0.2625	0.2623	0.3857
WM8, after GRPO	All	0.8094	0.8077	0.7884	0.3188	0.3188	0.4637
	Valid	0.9661	0.9661	0.9894	0.9485	0.9485	0.9950
	Invalid	0.7228	0.7201	0.6773	0.2906	0.2905	0.4398
WM4, after GRPO	All	0.8057	0.8043	0.7937	0.5014	0.4989	0.5772
	Valid	0.9693	0.9693	0.9833	0.9692	0.9692	0.9934
	Invalid	0.7210	0.7189	0.6955	0.4874	0.4849	0.5647
WM8-NR, after GRPO	All	0.8211	0.8204	0.8056	0.3177	0.3176	0.4681
	Valid	0.9837	0.9837	0.9937	0.9762	0.9762	0.9980
	Invalid	0.6823	0.6810	0.6450	0.2863	0.2862	0.4428

Table 14: Error-presence detection per attempt (Appendix E.3), in percent: precision, recall, accuracy, and F1.

	miniF2F				ProofNet			
Checkpoint	Prec.	Rec.	Acc.	F1	Prec.	Rec.	Acc.	F1
WM8, before GRPO	100.00	3.27	60.67	6.33	100.00	0.62	8.82	1.22
WM8, after GRPO	98.51	70.59	80.37	82.25	100.00	58.65	60.42	73.93
WM4, after GRPO	98.17	70.71	79.84	82.21	100.00	64.87	65.89	78.69
WM8-NR, after GRPO	100.00	53.06	74.68	69.34	100.00	39.15	41.92	56.27

Table 15: Repair across checkpoints: the share of responses that repair (at least two attempts), the mean number of attempts per response, and the share of solved problems that are solved only through repair.

	miniF2F			ProofNet		
Checkpoint	Repair (%)	Attempts	Only repair (%)	Repair (%)	Attempts	Only repair (%)
WM8, before GRPO	0.44	1.01	0.00	0.27	1.01	0.00
WM8-NR, before GRPO	0.00	1.00	0.00	0.00	1.00	0.00
WM8, after GRPO	18.34	1.84	4.32	22.57	2.23	7.69
WM4, after GRPO	17.39	1.89	3.51	27.86	2.69	9.26
WM8-NR, after GRPO	7.86	1.40	0.88	5.42	1.68	2.27

F TRAINING EXAMPLES

This appendix presents training examples: the SFT prompts and targets of the standard and world-model provers, repair examples, further annotated excerpts, the teacher-generated informal-reasoning annotations of Appendix C, and the prompts and targets of the reasoning stage (Section 4.3).

F.1 SFT PROMPTS AND TARGETS

Every supervised example is a prompt followed by a target and the end-of-text token, with no chat template. Both provers share the base prompt; only its last sentence differs. For the direct example below, a Goedel-Pset-v1 problem, the standard prover's prompt is

````
Solve the following problem with Lean 4 code and explanatory comments:

```lean4
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat


 /-A vat of orange juice was some pints. If you wanted to pour the vat into five
glasses with the same amount in each glass, and there would be 30 pints in each
glass, how many pints were in the vat?-/
theorem lean_workbook_plus_41504 (x : ℝ) (hx : x = 30 * 5) : x = 150   := by sorry
```

Replace every sorry statement with an appropriate proof. Provide a complete solution
 in the lean4 code block.
````

and the world-model prover's prompt ends instead with

```
Replace every sorry statement with an appropriate proof. Provide a complete solution
 in the lean4 code block, annotated with Lean 4 compiler feedback blocks in the form
 `/- <feedback> ... </feedback> -/` at the appropriate locations.
```

The standard target is the verified proof (the header lines repeat those of the prompt and are abbreviated here):

````
The complete typechecked Lean 4 proof is:

```lean4
<imports and options as in the prompt>

theorem lean_workbook_plus_41504 (x : ℝ) (hx : x = 30 * 5) : x = 150   := by
  rw [hx]
  linarith
```
````

The world-model target is the same proof with Lean's feedback blocks, so the model learns to predict both the tactics and the states between them:

````
Here is the complete Lean 4 proof annotated with Lean 4 compiler feedback blocks:

```lean4
<imports and options as in the prompt>

theorem lean_workbook_plus_41504 (x : ℝ) (hx : x = 30 * 5) : x = 150   := by
  /- <feedback>
  -- proof state:
  x : ℝ
  hx : x = 30 * 5
  ⊢ x = 150
  </feedback> -/
  rw [hx]
  /- <feedback>
  -- proof state:
  ⊢ 30 * 5 = 150
  </feedback> -/
  linarith
```
````

F.2 Repair examples

A repair prompt appends a failed attempt to the base prompt for the same problem; the example below is from APRIL. For the standard prover, the failed attempt appears annotated by Lean, including its error block, and the target is the clean corrected proof:

Standard prover: repair prompt (continuation after the instruction)

````
The complete typechecked Lean 4 proof is:

```lean4
import Mathlib

theorem number_theory_254098 : 2222^5555 + 5555^2222 ≡ 0 [MOD 7] := by
  /- <feedback>
  -- proof state:
  ⊢ 2222 ^ 5555 + 5555 ^ 2222 ≡ 0 [MOD 7]
  </feedback> -/
  linarith [Nat.ModEq]
  /- <feedback>
  -- type: error, msg: linarith failed to find a contradiction
  ⊢ False
  failed
  </feedback> -/
```
````

Standard prover: repair target

````
The corrected Lean 4 proof is:

```lean4
import Mathlib

theorem number_theory_254098 : 2222^5555 + 5555^2222 ≡ 0 [MOD 7] := by

  norm_num [Nat.ModEq]
```
````

For the world-model prover, the prompt continues with the failed attempt only through its failing line, so the error block is not given; the target begins with that error block, closes the code fence, and continues with the corrected annotated proof. This trains the world-model prover to predict Lean's diagnosis before revising the proof, supporting self-repair without intermediate Lean calls, whereas the standard prover learns to revise a proof given Lean's diagnosis:

World-model prover: repair prompt (continuation after the instruction)

````
Here is the complete Lean 4 proof annotated with Lean 4 compiler feedback blocks:

```lean4
import Mathlib

theorem number_theory_254098 : 2222^5555 + 5555^2222 ≡ 0 [MOD 7] := by
  /- <feedback>
  -- proof state:
  ⊢ 2222 ^ 5555 + 5555 ^ 2222 ≡ 0 [MOD 7]
  </feedback> -/
  linarith [Nat.ModEq]
```
````

World-model prover: repair target

````
  /- <feedback>
  -- type: error, msg: linarith failed to find a contradiction
  ⊢ False
  failed
  </feedback> -/
```

Here is the corrected Lean 4 proof annotated with Lean 4 compiler feedback blocks:

```lean4
import Mathlib

theorem number_theory_254098 : 2222^5555 + 5555^2222 ≡ 0 [MOD 7] := by
  /- <feedback>
  -- proof state:
  ⊢ 2222 ^ 5555 + 5555 ^ 2222 ≡ 0 [MOD 7]
  </feedback> -/
  norm_num [Nat.ModEq]
```
````

Repair histories with several failed attempts (up to five) chain the attempts in the prompt in the same way: the standard prompt lists each annotated attempt, later ones under "The corrected Lean 4 proof is:"; the world-model prompt closes each earlier attempt with its error block and introduces the next under the corrected-proof header.

F.3 Annotated excerpts

Figure 7 shows excerpts from two more world-model training examples: an accepted proof whose later state blocks omit unchanged hypotheses and mark a rewritten one, and a repair example whose error block combines an informational message with two errors for the failing line.
````

(a) Accepted proof: state blocks with omitted, new, and changed hypotheses

```
theorem lean_workbook_plus_78827 (current additional total : ℕ) :
    current + additional = total → current = 212 ∧ additional = 68 →
    total = 280   := by
  /- <feedback>
  -- proof state:
  current : ℕ
  additional : ℕ
  total : ℕ
  ⊢ current + additional = total → current = 212 ∧ additional = 68 → total = 280
  </feedback> -/
  rintro h₁ ⟨h₂, h₃⟩
  /- <feedback>
  -- proof state:
  h₁ : current + additional = total
  h₂ : current = 212
  h₃ : additional = 68
  case intro
  ⊢ total = 280
  </feedback> -/
  rw [h₂, h₃] at h₁
  /- <feedback>
  -- proof state:
  h₁ : 212 + 68 = total (was current + additional = total)
  case intro
  ⊢ total = 280
  </feedback> -/
  linarith
```

(b) Repair example, end of the prompt

```
theorem algebra_12691 : (4 - 5 * .I : ℂ) * (-5 + 5 * .I) = 5 + 45 * .I := by
  /- <feedback>
  -- proof state:
  ⊢ (4 - 5 * Complex.I) * (-5 + 5 * Complex.I) = 5 + 45 * Complex.I
  </feedback> -/
  ring [Complex.ext_iff, Complex.I_mul_I]
```

(b) Repair example, training target

```
  /- <feedback>
  -- type: info, msg: Try this: ring_nf
  -- type: error, msg: unsolved goals
  ⊢ -20 + (Complex.I * 45 - Complex.I ^ 2 * 25) = 5 + Complex.I * 45
  -- type: error, msg: unexpected token '['; expected command
  </feedback> -/
```

Here is the corrected Lean 4 proof annotated with Lean 4 compiler feedback blocks:

```lean4
theorem algebra_12691 : (4 - 5 * .I : ℂ) * (-5 + 5 * .I) = 5 + 45 * .I := by
  /- <feedback>
  -- proof state:
  ⊢ (4 - 5 * Complex.I) * (-5 + 5 * Complex.I) = 5 + 45 * Complex.I
  </feedback> -/
  simp [Complex.ext_iff, Complex.I_mul_I]
  /- <feedback>
  -- proof state:
  ⊢ -(4 * 5) + 5 * 5 = 5 ∧ 4 * 5 + 5 * 5 = 45
  </feedback> -/
  all_goals norm_num
```<|endoftext|>

Figure 7: **Verbatim excerpts from two world-model training examples.** (a) An accepted proof from our iterative proof collection: block 2 omits the unchanged `current`, `additional`, `total`, adds the new h_1–h_3 and `case intro`; block 3 prints only the rewritten h_1 among the hypotheses, with its old type in `(was ...)`; no terminal `no goals` block follows `linarith`. (b) A repair example derived from APRIL, in the format of Appendix F.2: the error block of the failing `ring [...]` combines Lean's info suggestion with two errors. Within the excerpts, only header lines (`import`, `set_option`, `open`) are removed and (a)'s signature is wrapped.

F.4 TEACHER-GENERATED REASONING ANNOTATIONS

The two examples below come from the training data: a direct proof and a two-attempt repair history. The Lean code and the error block are shown as is. The teacher's reasoning is reproduced with its wording and section structure unchanged, but its Markdown and LaTeX markup is rendered for readability.

Direct example: verified proof supplied to the annotator

```
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat


theorem candy_sum : 28 + 42 + 63 = 133 := by
  -- First add the first two numbers (28 + 42)
  have h1 : 28 + 42 = 70 := by
  -- begin proof
    norm_num
  -- end proof

  -- Then add the result to 63
  have h2 : 70 + 63 = 133 := by
  -- begin proof
    norm_num
  -- end proof

  -- Combine the steps to prove the final result
  have h3 : 28 + 42 + 63 = 70 + 63 := by
  -- begin proof
    rw [h1]
  -- end proof

  -- Complete the proof
  calc
    28 + 42 + 63 = 70 + 63 := h3
    _ = 133  := h2
```

Direct example: generated reasoning

[Problem Restatement]

Show that the total of three numbers, 28, 42, and 63, is 133:

$$28 + 42 + 63 = 133.$$

[Useful Observations]

- The expression is a straightforward addition problem, so it is natural to evaluate it in smaller stages.
- The first two numbers have a convenient sum:

  $$28 + 42 = 70.$$

  Reaching 70 makes the remaining addition especially simple.
- After replacing $28 + 42$ by 70, the goal becomes

  $$70 + 63 = 133.$$

- These two elementary arithmetic facts completely determine the result.

[Reasoning Path]

To calculate the sum $28 + 42 + 63$, begin with the first two addends. This is a natural grouping because 28 and 42 combine neatly:

Direct example: generated reasoning (continued)

$$28 + 42 = 70.$$

Thus the original expression can be rewritten without changing its value:

$$28 + 42 + 63 = 70 + 63.$$

Now only one addition remains. Adding 63 to 70 gives

$$70 + 63 = 133.$$

Since the original sum is equal to $70 + 63$, and $70 + 63$ is equal to 133, it follows that the original sum is 133.

[Key Derivation]

$$\begin{aligned} 28 + 42 + 63 &= 70 + 63 && \text{because } 28 + 42 = 70, \\ &= 133 && \text{because } 70 + 63 = 133. \end{aligned}$$

Therefore,

$$28 + 42 + 63 = 133.$$

[Proof Type]

Direct proof by evaluating the sum in two stages.

[Formal Anchors]

The verified proof records the two arithmetic facts and then rewrites the larger expression using the first one:

```
have h1 : 28 + 42 = 70 := by norm_num
have h2 : 70 + 63 = 133 := by norm_num
```

The final calculation chains the rewrite and the remaining addition:

```
calc
  28 + 42 + 63 = 70 + 63 := h3
  _ = 133 := h2
```

[Final Solution]

First compute the sum of the first two numbers:

$$28 + 42 = 70.$$

Substituting this into the full expression gives

$$28 + 42 + 63 = 70 + 63.$$

Finally,

$$70 + 63 = 133.$$

Hence,

$$\boxed{28 + 42 + 63 = 133}.$$

Repair example (uw-math-ai/APRIL): first attempt and its error block

```
import Mathlib

/- Prove that for any integer $n$ the number $n (n^2 + 5)$ is divisible by $6$. -/
theorem number_theory_47061 (n : ℤ) : 6 ∣ n * (n ^ 2 + 5) := by
```

```
  refine Int.dvd_of_emod_eq_zero
-- type: error, msg: type mismatch
    Int.dvd_of_emod_eq_zero
  has type
    ?m.406 % ?m.405 = 0 → ?m.405 ∣ ?m.406 : Prop
  but is expected to have type
    6 ∣ n * (n ^ 2 + 5) : Prop
```

Repair example: first segment, written for the first attempt

[Problem Restatement]

For every integer n, prove that

$$6 \mid n(n^2 + 5).$$

Equivalently, we must show that the product is divisible by both 2 and 3.

[Useful Observations]

- Since $6 = 2 \cdot 3$ and $2, 3$ are coprime, it is natural to establish divisibility by 2 and by 3 separately.
- Modulo 2, every integer satisfies $n^2 \equiv n$. Thus

  $$n^2 + 5 \equiv n + 1 \pmod 2,$$

  so the product is congruent to $n(n + 1)$, the product of consecutive integers.
- Modulo 3, the possible residues of an integer are $0, 1, 2$, and their squares are congruent to $0, 1, 1$. Hence $n^2 + 5 \equiv n^2 + 2$ is divisible by 3 whenever $n \not\equiv 0 \pmod 3$. If $n \equiv 0 \pmod 3$, the leading factor n already supplies the divisibility.
- The expression $n(n^2 + 5)$ is therefore arranged so that parity comes from consecutive factors, while divisibility by 3 comes from the three residue classes.

[Reasoning Path]

To prove divisibility by 6, first examine the parity of the expression. The number 5 is odd, so $n^2 + 5$ has opposite parity from n^2, and n^2 has the same parity as n. Therefore n and $n^2 + 5$ always have opposite parity. One of the two factors is even, which proves that their product is divisible by 2.

For divisibility by 3, classify n according to its residue modulo 3. This is a particularly natural choice because the quadratic term n^2 has only two possible values modulo 3: 0 for multiples of 3, and 1 otherwise.

If $n \equiv 0 \pmod 3$, then n itself is divisible by 3, so the whole product is divisible by 3.

If $n \equiv 1$ or $2 \pmod 3$, then $n^2 \equiv 1 \pmod 3$. Hence

$$n^2 + 5 \equiv 1 + 5 = 6 \equiv 0 \pmod 3.$$

In these cases the second factor is divisible by 3. Thus, in every residue class, one of the factors contributes a factor of 3.

We have now shown that 2 divides the product and that 3 divides the product. Since these divisors are coprime, their product 6 divides the product.

[Key Derivation]

For parity:

$$n^2 \equiv n \pmod 2,$$

so

$$n^2 + 5 \equiv n + 1 \pmod 2.$$

Therefore

$$n(n^2 + 5) \equiv n(n + 1) \equiv 0 \pmod 2.$$

For divisibility by 3:

$$n \equiv 0 \pmod 3 \implies 3 \mid n,$$

while

$$n \equiv \pm 1 \pmod 3 \implies n^2 \equiv 1 \pmod 3 \implies n^2 + 5 \equiv 6 \equiv 0 \pmod 3.$$

Repair example: first segment, written for the first attempt (continued)

Hence

$$3 \mid n(n^2+5).$$

Combining these:

$$2 \mid n(n^2+5), \qquad 3 \mid n(n^2+5),$$

and therefore

$$6 \mid n(n^2+5).$$

[Proof Type]

Direct modular-arithmetic proof, using divisibility by the coprime factors 2 and 3.

[Formal Anchors]

The formal proof begins by reducing divisibility by 6 to a remainder calculation:

```
refine Int.dvd_of_emod_eq_zero
```

The mathematical content of the resulting goal is precisely that

$$n(n^2+5) \equiv 0 \pmod 6,$$

which follows from the separate congruence arguments modulo 2 and modulo 3.

[Final Solution]

Let

$$N = n(n^2+5).$$

We show that $2 \mid N$ and $3 \mid N$.

Modulo 2, we have $n^2 \equiv n$, and $5 \equiv 1$, so

$$n^2+5 \equiv n+1 \pmod 2.$$

Thus

$$N \equiv n(n+1) \equiv 0 \pmod 2,$$

because one of two consecutive integers is even.

Now consider modulo 3. If $3 \mid n$, then certainly $3 \mid N$. Otherwise $n \equiv \pm 1 \pmod 3$, so $n^2 \equiv 1 \pmod 3$. Consequently,

$$n^2+5 \equiv 1+5 = 6 \equiv 0 \pmod 3,$$

and again $3 \mid N$.

Therefore N is divisible by both 2 and 3. Since 2 and 3 are coprime, $6 \mid N$. Hence

$$\boxed{6 \mid n(n^2+5)}.$$

Repair example: second attempt (verified)

```
import Mathlib

/- Prove that for any integer $n$ the number $n (n^2 + 5)$ is divisible by $6$. -/
theorem number_theory_47061 (n : ℤ) : 6 ∣ n * (n ^ 2 + 5) := by
  apply Int.dvd_of_emod_eq_zero
  have h : n % 6 = 0 ∨ n % 6 = 1 ∨ n % 6 = 2 ∨ n % 6 = 3 ∨ n % 6 = 4 ∨ n % 6 = 5 :=
 by omega
  rcases h with (h | h | h | h | h | h) <;> simp [h, pow_two, Int.mul_emod, Int.
add_emod]
```

Repair example: repair segment

[Error Analysis]

The lemma application was rejected because `Int.dvd_of_emod_eq_zero` is an implication: it needs a proof that the target expression has remainder 0 modulo 6. Using `refine` without supplying that premise

Repair example: repair segment (continued)

asked Lean to treat the lemma itself as a completed proof of divisibility, rather than to apply it and create its remainder goal.

So the right entry point is `apply Int.dvd_of_emod_eq_zero`. This converts the divisibility statement into the concrete modular claim

$$n(n^2 + 5) \bmod 6 = 0.$$

[Revised Reasoning]

Rather than separately assemble the factors 2 and 3, we can work directly modulo 6. Every integer has one of the six remainders

$$0, 1, 2, 3, 4, 5$$

when divided by 6. The `omega` tactic can establish this exhaustive classification for `n % 6`.

For each residue, substitute the corresponding value of `n % 6` into the remainder of the expression. Modular reduction respects multiplication and addition, and rewriting n^2 as `n * n` makes the arithmetic normalization straightforward. In every one of the six cases, the remainder of

$$n(n^2 + 5)$$

is 0 modulo 6.

This directly supplies the premise required by `Int.dvd_of_emod_eq_zero`, so the desired divisibility follows.

[Updated Proof Plan]

1. Apply `Int.dvd_of_emod_eq_zero` to reduce the goal to remainder 0 modulo 6.
2. Use `omega` to split into the six possible values of `n % 6`.
3. Simplify each case using modular addition and multiplication rules, with `pow_two` exposing the square as a product.

F.5 Reasoning SFT prompts and targets

In the informal-reasoning stage (Section 4.3), the prompt is the base prompt of Appendix F.1 with a longer instruction, and the target places the informal solution before the proof. The two instructions are, for the standard prover and the world-model prover respectively,

```
Replace every sorry statement with an appropriate proof. Before writing any Lean
code, give an informal solution with the sections [Problem Restatement], [Useful
Observations], [Reasoning Path], [Key Derivation], [Proof Type], [Formal Anchors]
and [Final Solution]. Then provide the complete solution in the lean4 code block. If
 a previous attempt failed, first give [Error Analysis], [Revised Reasoning] and [
Updated Proof Plan], then provide the corrected solution.
```

```
Replace every sorry statement with an appropriate proof. Before writing any Lean
code, give an informal solution with the sections [Problem Restatement], [Useful
Observations], [Reasoning Path], [Key Derivation], [Proof Type], [Formal Anchors]
and [Final Solution]. Then provide the complete solution in the lean4 code block,
annotated with Lean 4 compiler feedback blocks in the form `/- <feedback> ... </
feedback> -/` at the appropriate locations. If a previous attempt failed, first give
[Error Analysis], [Revised Reasoning] and [Updated Proof Plan], then provide the
corrected solution, annotated with Lean 4 compiler feedback blocks in the form `/- <
feedback> ... </feedback> -/` at the appropriate locations.
```

Below, the reasoning sections are typeset as in Appendix F.4; all other text, including every lead-in line and code block, is reproduced verbatim, and each target ends with <|endoftext|> (omitted here).

Direct example. For a NuminaMath-LEAN problem, the standard prover's prompt is

```
Solve the following problem with Lean 4 code and explanatory comments:
```

````
```lean4
import Mathlib

/- Which of the following expressions is equal to 5 ?
(A) $(2 \times 3)^{2}$
(D) $3^{2}-2^{2}$
(B) $3+2^{2}$
(E) $(3+2)^{2}$
(C) $2^{3}-1$ -/
theorem algebra_281960 : 3^2 - 2^2 = 5 := by sorry
```

Replace every sorry statement with an appropriate proof. Before writing any Lean
code, give an informal solution with the sections [Problem Restatement], [Useful
Observations], [Reasoning Path], [Key Derivation], [Proof Type], [Formal Anchors]
and [Final Solution]. Then provide the complete solution in the lean4 code block. If
 a previous attempt failed, first give [Error Analysis], [Revised Reasoning] and [
Updated Proof Plan], then provide the corrected solution.
````

and its target is

Standard prover: direct target

[Problem Restatement]

Show that $3^2 - 2^2 = 5$.

[Useful Observations]

- The statement is a direct numerical identity.
- We only need to evaluate the powers 3^2 and 2^2, then subtract.
- Since $3^2 = 9$ and $2^2 = 4$, the left-hand side becomes $9 - 4$, which is 5.

[Reasoning Path]

The expression $3^2 - 2^2$ involves only concrete integers, so the natural proof is simply computation. First square each number:

$$3^2 = 3 \cdot 3 = 9, \qquad 2^2 = 2 \cdot 2 = 4.$$

Substituting these values into the original expression gives

$$3^2 - 2^2 = 9 - 4.$$

Finally,

$$9 - 4 = 5.$$

Thus the claimed equality holds.

[Key Derivation]

$$3^2 - 2^2 = 9 - 4 = 5.$$

[Proof Type]

Direct proof by numerical evaluation.

[Formal Anchors]

The Lean proof uses numerical normalization:

```
norm_num
```

Standard prover: direct target (continued)

This checks the arithmetic identity by evaluating the powers and subtraction.

[Final Solution]

We compute directly:

$$3^2 = 9$$

and

$$2^2 = 4.$$

Therefore,

$$3^2 - 2^2 = 9 - 4 = 5.$$

Hence,

$$3^2 - 2^2 = 5.$$

````
The complete typechecked Lean 4 proof is:

```lean4
import Mathlib

theorem algebra_281960 : 3^2 - 2^2 = 5 := by
  norm_num
```
````

The world-model prover's prompt carries the second instruction, and its target repeats the same reasoning sections and then continues with the annotated proof:

````
Here is the complete Lean 4 proof annotated with Lean 4 compiler feedback blocks:

```lean4
import Mathlib

theorem algebra_281960 : 3^2 - 2^2 = 5 := by
  /- <feedback>
  -- proof state:
  ⊢ 3 ^ 2 - 2 ^ 2 = 5
  </feedback> -/
  norm_num
```
````

Repair example. For a Goedel-Pset-v1 problem whose first attempt fails, the standard prover's prompt continues, after the instruction, with the reasoning segment written for that attempt and the attempt itself, annotated by Lean:

````
Solve the following problem with Lean 4 code and explanatory comments:

```lean4
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat


 /-Given points A $(x,1,2)$ and B $(2,3,4)$, and the distance $|AB|=2\sqrt{6}$, find
 the value of the real number $x$.-/
theorem lean_workbook_plus_3769 (x : ℝ) (h₁ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24) : x
= 6 ∨ x = -2   := by sorry
```
````

```
Replace every sorry statement with an appropriate proof. Before writing any Lean
code, give an informal solution with the sections [Problem Restatement], [Useful
Observations], [Reasoning Path], [Key Derivation], [Proof Type], [Formal Anchors]
and [Final Solution]. Then provide the complete solution in the lean4 code block. If
 a previous attempt failed, first give [Error Analysis], [Revised Reasoning] and [
Updated Proof Plan], then provide the corrected solution.
```

Standard prover: repair prompt, reasoning segment of the failed attempt

[Problem Restatement]

The hypothesis is

$$(2-x)^2+2^2+2^2=24.$$

The stated conclusion claims that $x=6$ or $x=-2$.

[Useful Observations]

Since $2^2+2^2=8$, the hypothesis simplifies to

$$(2-x)^2=16.$$

Thus the relevant square is 16, not 12.

[Reasoning Path]

From $(2-x)^2=16$, taking the two possible square roots gives

$$2-x=4 \quad \text{or} \quad 2-x=-4.$$

Solving these linear equations yields $x=-2$ or $x=6$, exactly the desired disjunction.

[Key Derivation]

$$\begin{gathered}(2-x)^2+4+4=24\\(2-x)^2=16\\2-x=\pm 4\\x=-2 \quad \text{or} \quad x=6.\end{gathered}$$

[Proof Type]

Direct proof by simplifying the equation and splitting the two square-root possibilities.

[Formal Anchors]

The intermediate assertion shown in the supplied code,

```
have h₂ : (2 - x) ^ 2 = 12 := by
  nlinarith
```

does not follow from the hypothesis: simplification gives $(2-x)^2=16$. A valid formal proof needs the intermediate equality to be 16, followed by a factorization or `nlinarith` argument establishing the disjunction.

[Final Solution]

Because $2^2+2^2=8$, the hypothesis gives

$$(2-x)^2=24-8=16.$$

Hence $2-x=4$ or $2-x=-4$. In the first case $x=-2$; in the second case $x=6$. Therefore

$$x=6 \vee x=-2.$$

````
The complete typechecked Lean 4 proof is:

```lean4
import Mathlib
import Aesop
````
````

```
set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat



theorem lean_workbook_plus_3769 (x : ℝ) (h₁ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24) : x
= 6 ∨ x = -2   := by
  /- <feedback>
  -- proof state:
  x : ℝ
  h₁ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24
  ⊢ x = 6 ∨ x = -2
  </feedback> -/
  have h₂ : (2 - x) ^ 2 = 12 := by
    /- <feedback>
    -- proof state:
    ⊢ (2 - x) ^ 2 = 12
    </feedback> -/
    nlinarith
    /- <feedback>
    -- type: error, msg: linarith failed to find a contradiction
    case h2.h
    x : ℝ
    h₁ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24
    a† : 12 < (2 - x) ^ 2
    ⊢ False failed
    </feedback> -/
```

Its target gives the three repair sections and the corrected proof:

Standard prover: repair target

[Error Analysis]

The failed intermediate claim used 12 as the remaining square, but the two constant terms are $2^2 + 2^2 = 4 + 4 = 8$. Therefore the hypothesis implies

$$(2 - x)^2 = 24 - 8 = 16,$$

not 12. The arithmetic solver correctly could not prove the false equality: a square equal to 16 need not equal 12.

[Revised Reasoning]

Keeping the original equation available, first isolate the square as

$$(2 - x)^2 = 16.$$

To obtain the two possible linear cases formally, compare this with $4^2 = 16$. Equality of squares gives

$$2 - x = 4 \quad \text{or} \quad 2 - x = -4.$$

These cases resolve directly by linear arithmetic. From $2 - x = 4$, we get $x = -2$, which establishes the right branch of the desired disjunction. From $2 - x = -4$, we get $x = 6$, establishing the left branch.

[Updated Proof Plan]

1. Derive $(2 - x)^2 = 16$ from the distance equation.
2. Use equality of the squares $(2 - x)^2 = 4^2$ to split into $2 - x = 4$ or $2 - x = -4$.
3. Solve each linear case and place the resulting equality in the corresponding disjunct.

The corrected Lean 4 proof is:

```lean4
import Mathlib
import Aesop
```

```
set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat



theorem lean_workbook_plus_3769 (x : ℝ) (h₁ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24) : x
= 6 ∨ x = -2   := by
  have h₂ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24 := h₁
  have h₃ : (2 - x) ^ 2 = 16 := by linarith
  have h₄ : 2 - x = 4 ∨ 2 - x = -4 := by
    apply eq_or_eq_neg_of_sq_eq_sq
    linarith
  cases h₄ with
  | inl h₄ =>
    have h₅ : x = -2 := by linarith
    exact Or.inr h₅
  | inr h₄ =>
    have h₅ : x = 6 := by linarith
    exact Or.inl h₅
```

For the world-model prover, the prompt carries the second instruction and the same reasoning segment, but the failed attempt stops at its failing line:

```
Here is the complete Lean 4 proof annotated with Lean 4 compiler feedback blocks:

```lean4
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat



theorem lean_workbook_plus_3769 (x : ℝ) (h₁ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24) : x
= 6 ∨ x = -2   := by
  /- <feedback>
  -- proof state:
  x : ℝ
  h₁ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24
  ⊢ x = 6 ∨ x = -2
  </feedback> -/
  have h₂ : (2 - x) ^ 2 = 12 := by
    /- <feedback>
    -- proof state:
    ⊢ (2 - x) ^ 2 = 12
    </feedback> -/
    nlinarith
```

The target begins with the error block for that line, closes the code fence, repeats the same three repair sections, and ends with the corrected annotated proof:

```
    /- <feedback>
    -- type: error, msg: linarith failed to find a contradiction
    case h2.h
    x : ℝ
    h₁ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24
    a† : 12 < (2 - x) ^ 2
    ⊢ False failed
    </feedback> -/
```
```

World-model prover: repair target (continued)

[Error Analysis]

The failed intermediate claim used 12 as the remaining square, but the two constant terms are $2^2 + 2^2 = 4 + 4 = 8$. Therefore the hypothesis implies

$$(2 - x)^2 = 24 - 8 = 16,$$

not 12. The arithmetic solver correctly could not prove the false equality: a square equal to 16 need not equal 12.

[Revised Reasoning]

Keeping the original equation available, first isolate the square as

$$(2 - x)^2 = 16.$$

To obtain the two possible linear cases formally, compare this with $4^2 = 16$. Equality of squares gives

$$2 - x = 4 \quad \text{or} \quad 2 - x = -4.$$

These cases resolve directly by linear arithmetic. From $2 - x = 4$, we get $x = -2$, which establishes the right branch of the desired disjunction. From $2 - x = -4$, we get $x = 6$, establishing the left branch.

[Updated Proof Plan]

1. Derive $(2 - x)^2 = 16$ from the distance equation.
2. Use equality of the squares $(2 - x)^2 = 4^2$ to split into $2 - x = 4$ or $2 - x = -4$.
3. Solve each linear case and place the resulting equality in the corresponding disjunct.

````
Here is the corrected Lean 4 proof annotated with Lean 4 compiler feedback blocks:

```lean4
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat



theorem lean_workbook_plus_3769 (x : ℝ) (h₁ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24) : x
= 6 ∨ x = -2   := by
  /- <feedback>
  -- proof state:
  x : ℝ
  h₁ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24
  ⊢ x = 6 ∨ x = -2
  </feedback> -/
  have h₂ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24 := h₁
  /- <feedback>
  -- proof state:
  h₂ : (2 - x) ^ 2 + 2 ^ 2 + 2 ^ 2 = 24
  ⊢ x = 6 ∨ x = -2
  </feedback> -/
  have h₃ : (2 - x) ^ 2 = 16 := by linarith
  /- <feedback>
  -- proof state:
  h₃ : (2 - x) ^ 2 = 16
  ⊢ x = 6 ∨ x = -2
  </feedback> -/
  have h₄ : 2 - x = 4 ∨ 2 - x = -4 := by
    /- <feedback>
    -- proof state:
    ⊢ 2 - x = 4 ∨ 2 - x = -4
    </feedback> -/
    apply eq_or_eq_neg_of_sq_eq_sq
    /- <feedback>
````
````

```
    -- proof state:
    case a
    ⊢ (2 - x) ^ 2 = 4 ^ 2
    </feedback> -/
    linarith
  /- <feedback>
  -- proof state:
  h₄ : 2 - x = 4 ∨ 2 - x = -4
  ⊢ x = 6 ∨ x = -2
  </feedback> -/
  cases h₄ with
  | inl h₄ =>
    /- <feedback>
    -- proof state:
    h₄ : 2 - x = 4 (was 2 - x = 4 ∨ 2 - x = -4)
    case inl
    ⊢ x = 6 ∨ x = -2
    </feedback> -/
    have h₅ : x = -2 := by linarith
    /- <feedback>
    -- proof state:
    h₅ : x = -2
    case inl
    ⊢ x = 6 ∨ x = -2
    </feedback> -/
    exact Or.inr h₅
  | inr h₄ =>
    /- <feedback>
    -- proof state:
    h₄ : 2 - x = -4 (was 2 - x = 4)
    case inr
    ⊢ x = 6 ∨ x = -2
    </feedback> -/
    have h₅ : x = 6 := by linarith
    /- <feedback>
    -- proof state:
    h₅ : x = 6 (was x = -2)
    case inr
    ⊢ x = 6 ∨ x = -2
    </feedback> -/
    exact Or.inl h₅
```