

A Additional implementation details

Algorithm 1 presents a high-level pseudo code of the Apollo framework. We highlight that the framework is built-upon the recursive function calls up to a maximum recursion depth given by parameter r . **LLM** refers to a proof generator model that takes in a proof statement and outputs a formal proof attempt. **LeanCompiler** refers to Lean REPL that verifies the proof and outputs **True** or **False**. In the case of **False**, it also provides a detailed list of errors, their locations and types along with the error messages generated by the Lean compiler. Other functions in the pseudo code (**SyntaxRefiner**, **Sorrifier**, **AutoSolver**, **ProofAssembler**) represent the sub-modules described in the main text. To give deeper insight into Apollo’s design, we now outline the rationale and responsibilities of each core module.

A.1 SyntaxRefiner

The **SyntaxRefiner** class applies a set of rule-based corrections to eliminate syntax errors in LLM-generated Lean code. During model experimentation, we collected a corpus of common mistakes, such as stray Lean 3 keywords (**from**, **begin**...**end**), misplaced tactics, or missing delimiters, and encoded each repair as a regular-expression rule. For example:

- Convert [**from by**] to [**:= by**].
- Replace Lean 3 blocks (**begin**...**end**) with the corresponding Lean 4 structure.
- Ensure that commands like **rw** use correctly placed square brackets to avoid compilation errors.

In more complex cases, multiple regex patterns and replacements are composed to restore valid Lean 4 syntax without altering the logical content of the proof sketch.

A.2 Sorrifier

The **Sorrifier** module uses a feedback loop between Apollo and the Lean REPL as follows:

1. Apollo sends the current proof candidate to REPL.
2. The REPL returns a list of error messages and their source locations.
3. Apollo parses the proof into a syntax tree, navigates to each failing sub-lemma, and attempts to fix it or replace it with a **sorry** placeholder.
4. The updated proof is re-submitted to REPL, and this cycle repeats until no error remain.

If the REPL indicates the original theorem statement itself is malformed (e.g. due to LLM hallucination), Apollo abandons the current proof and requests a fresh generation from LLM. One example of malformed formal statement is if LLM introduces a variable in a hypothesis that was not previously defined. Such case would trigger a compilation error in the formal statement; therefore, when parsing the proof, Apollo will catch this error and request a fresh proof from LLM. To guarantee consistent variable types during lemma extraction and proof assembly, Apollo sets the following Lean options to true:

```
set_option pp.instanceTypes true
set_option pp.numericTypes true
set_option pp.coercions.types true
set_option pp.letVarTypes true
set_option pp.structureInstanceTypes true
set_option pp.instanceTypes true
set_option pp.mvars.withType true
set_option pp.coercions true
set_option pp.funBinderTypes true
set_option pp.piBinderTypes true
```

so that the compiler reports explicit types (e.g. \mathbb{Z} vs. \mathbb{R}). This fail-safe prevents trivialization of theorems and ensures correct typing in the final assembled proof.

795 A.3 AutoSolver

796 The **AutoSolver** module takes the “sorrified” proof and applies a sequence of tactics to close as many
797 goals as possible:

- 798 1. **Hint-based tactics.** It first invokes `hint`, which proposes candidate tactics. We filter out
799 any suggestions that merely progress the goal and retain only those that fully discharge it.
- 800 2. **Built-in tactics.** If `hint` fails, AutoSolver systematically tries a suite of powerful Lean
801 tactics, such as `nlinarith`, `norm_num`, `norm_cast`, `ring_nf`, `simp`, `simp_all`, and others,
802 often combining them with `try` to catch failures without aborting.
- 803 3. **Recursive fallback.** Any goals that remain unsolved are reported back to Apollo. Apollo
804 queries the REPL for the exact goal statements, spawns corresponding sub-lemmas, and
805 then invokes the full repair loop (LLM generation, SyntaxRefiner, Sorrifier, and AutoSolver)
806 on each sub-lemma.

807 This tiered strategy maximizes automation while ensuring that challenging subgoals are handled via
808 targeted recursion.

Algorithm 1 Apollo Pseudocode for Formal Theorem Proving

Require: problem statement ps , current recursion depth $r_{current}$, maximum depth r

```

1: if  $r_{current}$  is not initialized then
2:    $r_{current} \leftarrow 0$ 
3: end if
4: if  $r_{current} > r$  then
5:   return sorry ▷ If recursion depth reached, return sorry
6: end if
7:  $proof \leftarrow \text{LLM}(ps)$  ▷ Generate proof from LLM
8:  $proof \leftarrow \text{SyntaxRefiner}(proof)$  ▷ Refine the proof syntax
9:  $proof \leftarrow \text{Sorrifier}(proof)$  ▷ Patch failing sub-lemmas
10:  $proof \leftarrow \text{AutoSolver}(proof)$  ▷ Try built-in Lean solvers
11: if  $\text{LeanCompiler}(proof) == \text{True}$  then
12:   return  $proof$ 
13: end if
14:  $r_{current} \leftarrow r_{current} + 1$ 
15:  $n \leftarrow \text{CountSorries}(proof)$  ▷ Number of 'sorry' placeholders
16:  $sub\_proofs \leftarrow []$ 
17: for  $i = 1$  to  $n$  do
18:    $ps_{goal} \leftarrow \text{GetTheoremGoal}(proof, i)$ 
19:    $sub\_ps \leftarrow \text{TransformGoal}(ps_{goal})$ 
20:    $sub\_proofs[i] \leftarrow \text{Apollo}(sub\_ps, r_{current}, r)$ 
21: end for
22:  $repaired\_proof \leftarrow \text{ProofAssembler}(sub\_proofs)$ 
23: return  $repaired\_proof$ 

```

809 B An example of Apollo proof repair procedure

810 Figure 6 shows a detailed run of the Apollo framework on `mathd_algebra_332`. First, the LLM
811 generates an initial proof sketch, which fails to type-check in Lean. Apollo then applies its `Syntax`
812 `Refiner` to correct simple errors (e.g. changing “from by” to “:= by”) and to strip out all comment
813 blocks. Next, the `Sorrifier` replaces each failing sub-lemma with a `sorry` statement (six in this
814 example). The `AutoSolver` module then attempts to discharge these sub-lemmas: it resolves four
815 of them, leaving lemmas #5 and #6 unsolved. Apollo recurses on those two: each is passed again
816 through LLM, `Syntax Refiner`, `Sorrifier`, and `AutoSolver`. After this single recursive iteration,
817 all goals are closed. Finally, Apollo reassembles the full proof and returns it to the user.

818 This example illustrates Apollo’s repair logic. In this case the proof was fixed in one iteration; more
819 complex theorems may require deeper recursive repair.

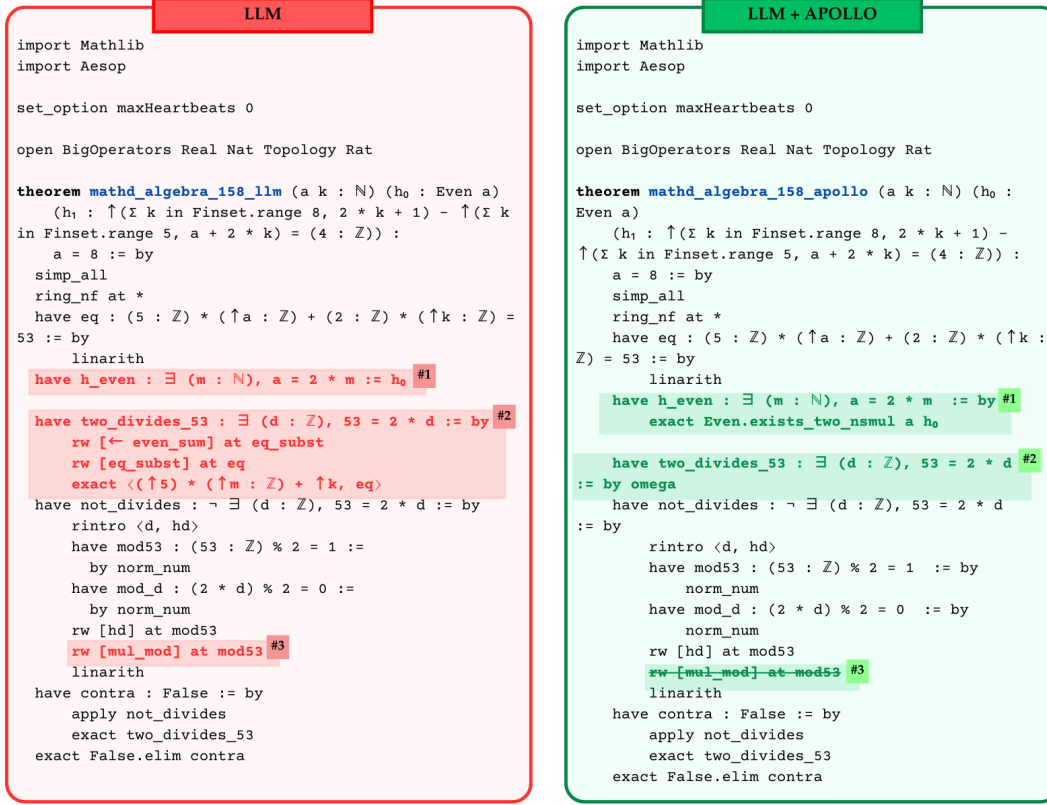


Figure 7: Proof attempt of `mathd_algebra_158` produced by the base model (o4-mini) versus after Apollo’s targeted repair. The left shows the original end-to-end LLM output, which does not compile in Lean; the right shows the corrected, repaired proof assembled by Apollo that closes all of the goals.

C Comparison between base proofs generated from LLMs and Apollo-repaired counterparts

In this section, we present and discuss examples of initial proof attempts alongside their Apollo-repaired counterparts.

Figures 7, 8, 9 illustrate cases where Apollo repairs the proof without invoking the LLM again. In each example, the LLM struggles to generate the precise tactics needed to close certain goals; however, we find that regenerating the entire proof is unnecessary. Instead, built-in Lean solvers can discharge those subgoals directly. Moreover, in Figure 7, problematic block #3 is resolved simply by removing it, since the surrounding proof context is correct. Thus, omitting problematic lines can sometimes yield a valid proof.

In Figure 10, one goal is expanded via an LLM query, but the model injects extra tactics that trigger a `no goals to be solved` error. Apollo then repairs the first block using a combination of LLM generation and AutoSolver, while the second block is removed entirely.

Figure 11 illustrates a case where the model fails to apply `linarith` to discharge the goal. We observe that, when uncertain, the model often resorts to broad tactics in hopes of closing the goal. Here, the goal is too complex for `linarith`, so Apollo leverages both the LLM and AutoSolver to guide the proof to completion.

Figure 12 illustrates an example of proof that required a recursion depth $r = 5$ to repair the initial proof attempt. We observe that LLM’s proof structure is correct; however, in many cases it over-relies on built-in solvers and rewrites. However, since the goals are too complex, this approach leads to a total of nine error blocks. Repairing those blocks requires aid from both AutoSolver module of

841 Apollo and LLM itself. We show that if LLM grasps the correct approach, then Apollo is able to
 842 repair fine grained logic errors to produce a correct proof.

843 Figure I3 illustrates a case where the LLM produces an incomplete proof sketch. The first half of
 844 the proof is correct, but the model fails to discharge the remaining goal and instead uses `all_goals`
 845 `positivity`. Apollo detects this error and performs two additional recursive repair iterations to
 846 complete the proof.

847 Figure I4 illustrates a case where the base model takes a lazy approach by trying to close all goals
 848 with `linarith`. In contrast, Apollo’s repaired proof performs additional setup and discharges each
 849 goal by first proving the necessary auxiliary hypotheses. This example shows that, although the model
 850 often understands the high-level strategy, it lacks the fine-grained tactics (and compiler feedback)
 851 needed to close subgoals. Apollo decomposes the proof, identifies the failure points, and applies
 852 targeted repairs to generate a fully verified solution.

853 Figure I5 shows another scenario in which Apollo successfully closes the first two blocks with
 854 `linarith`, but the final block requires a deeper reasoning chain. Apollo augments the proof by
 855 introducing and proving the missing lemmas, which leads to a correct solution with a series of
 856 rewrites.

857 Figure I6 presents an example of a very large repair. As in Figure I2, Apollo requires $r = 5$ to fully
 858 fix the proof. The final proof length increases from 100 lines to 216, which means Apollo applied
 859 roughly $\times 2.2$ more tactics to close the goal. The repair relied on combined effort from AutoSolver
 860 and the LLM to close all goals. We show that, even for large, complex proofs, Apollo can repair each
 861 failing sub-block and generate a correct solution.

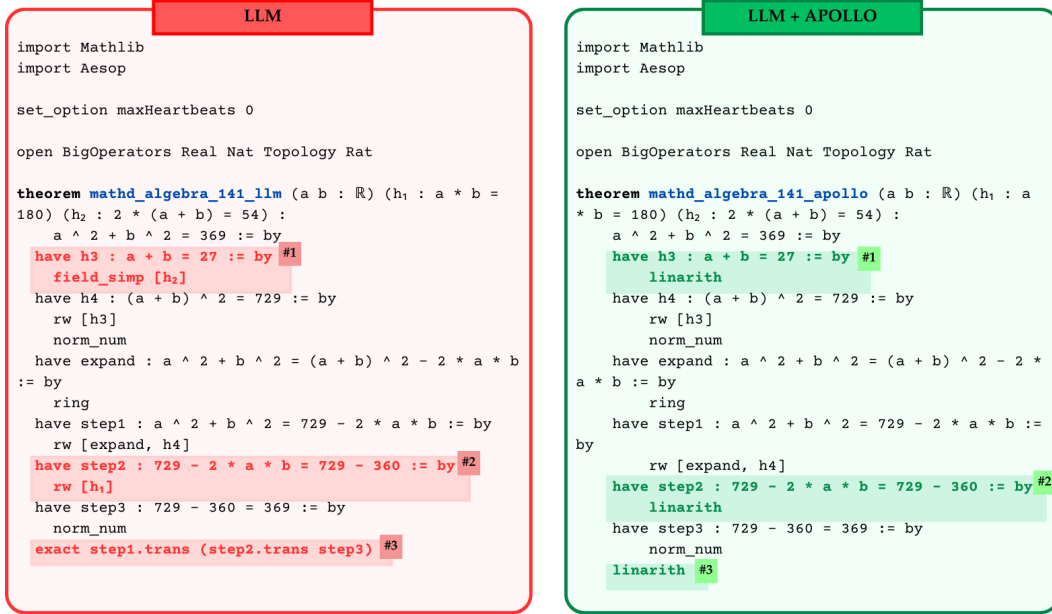


Figure 8: Proof attempt of `mathd_algebra_141` produced by the base model (o4-mini) versus after Apollo’s targeted repair. The left shows the original end-to-end LLM output, which does not compile in Lean; the right shows the corrected, repaired proof assembled by Apollo that closes all of the goals.

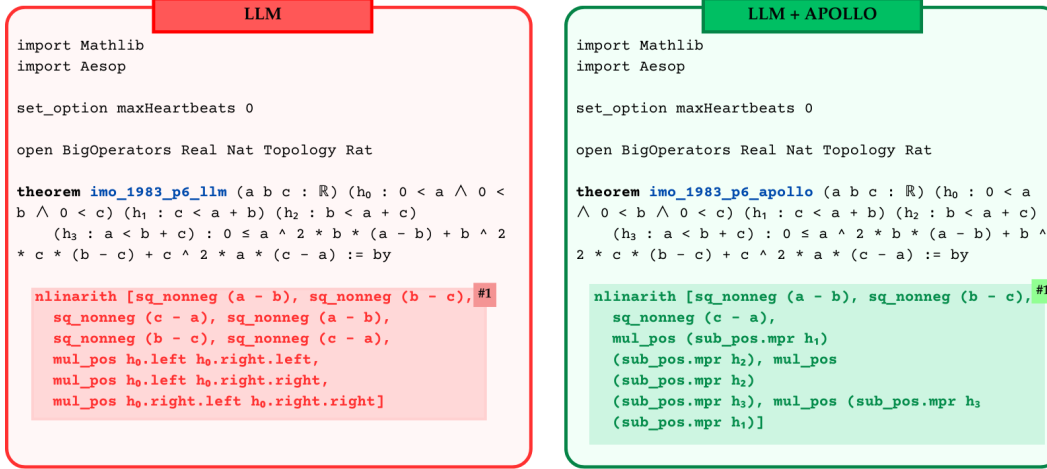


Figure 9: Proof attempt of imo_1983_p6 produced by the base model (Kimina-Prover-Preview-Distill-7B) versus after Apollo's targeted repair. The left shows the original end-to-end LLM output, which does not compile in Lean; the right shows the corrected, repaired proof assembled by Apollo that closes all of the goals.

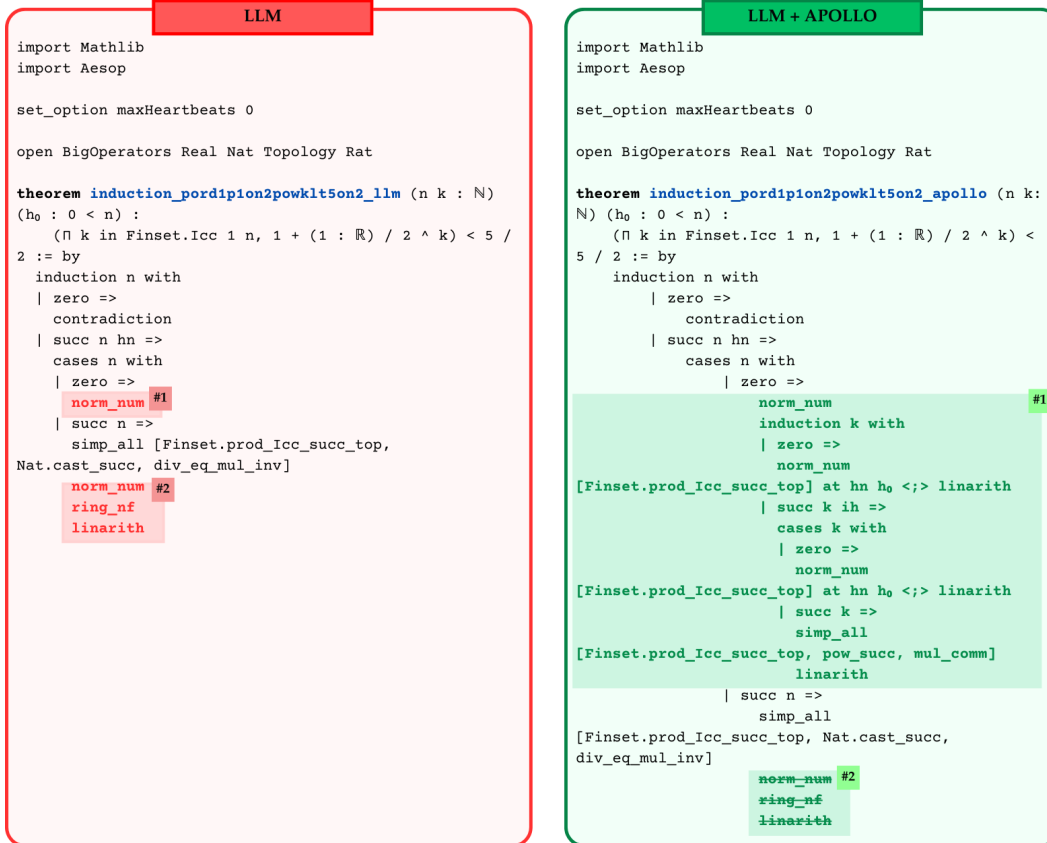


Figure 10: Proof attempt of induction_pord1p1on2powklt5on2 produced by the base model (Goedel-Prover-SFT) versus after Apollo's targeted repair. The left shows the original end-to-end LLM output, which does not compile in Lean; the right shows the corrected, repaired proof assembled by Apollo that closes all of the goals.



Figure 11: Proof attempt of `mathd_algebra_293` produced by the base model (Goedel-Prover-SFT) versus after Apollo’s targeted repair. The left shows the original end-to-end LLM output, which does not compile in Lean; the right shows the corrected, repaired proof assembled by Apollo that closes all of the goals.

LLM	LLM+APOLLO
<pre> import Mathlib import Aesop set_option maxHeartbeats 0 open BigOperators Real Nat Topology Rat theorem mathd_algebra_184_llm (a b : NNReal) (h₀ : (0 : NNReal) < a ∧ (0 : NNReal) < b) (h₁ : a ^ (2 : ℕ) = (6 : NNReal) * b) (h₂ : a ^ (2 : ℕ) = (54 : NNReal) / b) : a = (3 : NNReal) * (NNReal.sqrt : NNReal → NNReal) (2 : NNReal) := by have h3 : (6 : NNReal) * b = (54 : NNReal) / b := by rw [←h₁, ←h₂] have hb : b = (3 : NNReal) := by have h4 : b ≠ 0 := by #1 linarith [h₄.right] have h5 : (6 : NNReal) * b ^ 2 = (54 : NNReal) := by by have h6 : b ≠ 0 := h4 field_simp at h3 nlinarith #2 have h7 : b ^ 2 = (9 : NNReal) := by #3 nlinarith have h8 : b = (3 : NNReal) := by have h9 : b > (0 : NNReal) := h₄.right have h10 : b ^ 2 - (9 : NNReal) = (0 : NNReal) := by nlinarith have h11 : (b - (3 : NNReal)) * (b + (3 : NNReal)) = (0 : NNReal) := by ring_nf at h10 linarith #5 cases' (mul_eq_zero.mp h11) with h12 h13 -- b - 3 = 0 #6 linarith -- b + 3 = 0, but b > 0, contradiction #7 linarith [h9] exact h8 have ha : a = (3 : NNReal) * (NNReal.sqrt : NNReal → NNReal) (2 : NNReal) := by rw [hb] at h₁ have h14 : a ^ 2 = (18 : NNReal) := by norm_num at h₁ linarith #8 have h15 : a = Real.sqrt (18 : NNReal) := by rw [←h14] field_simp [h₁.left] have h16 : Real.sqrt (18 : NNReal) = (3 : NNReal) * Real.sqrt (2 : NNReal) := by rw [Real.sqrt_eq_iff_sq_eq] < > norm_num < > ring_nf < > norm_num < > ring_nf < > norm_num rw [h15, h16] #9 exact ha </pre>	<pre> import Mathlib import Aesop set_option maxHeartbeats 0 open BigOperators Real Nat Topology Rat theorem mathd_algebra_184_apollo (a b : NNReal) (h₀ : (0 : NNReal) < a ∧ (0 : NNReal) < b) (h₁ : a ^ (2 : ℕ) = (6 : NNReal) * b) (h₂ : a ^ (2 : ℕ) = (54 : NNReal) / b) : a = (3 : NNReal) * (NNReal.sqrt : NNReal → NNReal) (2 : NNReal) := by have h3 : (6 : NNReal) * b = (54 : NNReal) / b := by rw [←h₁, ←h₂] have hb : b = (3 : NNReal) := by have h4 : b ≠ 0 := by #1 have hb : (0 : NNReal) < b := h₄.right exact ne_of_gt hb have h5 : (6 : NNReal) * b ^ 2 = (54 : NNReal) := by NNReal := by have h6 : b ≠ 0 := by exact h4 field_simp at h3 have h5 : (6 : NNReal) * b ^ (2 : ℕ) #2 = (54 : NNReal) := by have h6 : b ^ (2 : ℕ) = b * b := by simp [pow_two] rw [h6] try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith exact h3 have h7 : b ^ 2 = (9 : NNReal) := by have h6 : (6 : NNReal) * b ^ (2 : ℕ) = (54 : NNReal) := by gcongr have h7 : b ^ (2 : ℕ) = (9 : NNReal) #3 := by have h8 : (6 : NNReal) ≠ (0 : NNReal) := by norm_num have h9 : b ^ (2 : ℕ) = (54 : NNReal) / (6 : NNReal) := by field_simp [h8] try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith exact h6 rw [h9] norm_num try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith apply NNReal.eq norm_num exact h7 have h_real : (↑b : ℝ) ^ 2 = (3 : ℝ) ^ 2 := by rw [← NNReal.coe_pow, h7] norm_num have hb_real : (↑b : ℝ) = 3 := by simp_all only [ne_eq, NNReal.zero_le_coe, ofNat_nonneg, ofNat.ofNat_ne_zero, not_false_eq_true, pow_left_inj] have hb : b = 3 := by apply NNReal.eq exact hb_real have h8 : b = (3 : NNReal) := by have h9 : b > (0 : NNReal) := by exact pos_of_ne_zero h4 have h10 : b ^ 2 - (9 : NNReal) = (0 : NNReal) := by simp_all only [and_true, ne_eq, gt_iff_lt, tsub_self] have h11 : (b - (3 : NNReal)) * (b + (3 : NNReal)) = (0 : NNReal) := by ring_nf at h10 have hb : b = (3 : NNReal) := by have h7' : b ^ 2 = (9 : NNReal) := by simp using h7 have h9' : b > (0 : NNReal) := by gcongr have h11 : b = (3 : NNReal) := by have h12 : b ^ 2 - (9 : NNReal) = (0 : NNReal) := by gcongr have h13 : (b - (3 : NNReal)) * (b + (3 : NNReal)) = (0 : NNReal) := by ring_nf at h12 try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith ... </pre>

Figure 12: Proof attempt of `mathd_algebra_184` produced by the base model (Kimina-Prover-Preview-Distill-7B) versus after Apollo’s targeted repair. The left shows the original end-to-end LLM output, which does not compile in Lean; the right shows the corrected, repaired proof assembled by Apollo that closes all of the goals.

LLM	LLM + APOLLO
<pre> import Mathlib import Aesop set_option maxHeartbeats 0 open BigOperators Real Nat Topology Rat theorem mathd_algebra_293_llm (x : NNReal) : Real.sqrt (60 * x) * Real.sqrt (12 * x) * Real.sqrt (63 * x) = 36 * x * Real.sqrt (35 * x) := by have h1 : Real.sqrt (60 * x) * Real.sqrt (12 * x) * Real.sqrt (63 * x) = Real.sqrt ((60 * x) * (12 * x) * (63 * x)) := by rw [← Real.sqrt_mul (by positivity), ← Real.sqrt_mul (by positivity)] rw [h1] have h2 : (60 * x) * (12 * x) * (63 * x) = (36 * x) ^ 2 * (35 * x) := by ring_nf < > simp [x_mul_x] < > linarith [show (0 : ℝ) ≤ x from by positivity] rw [h2] rw [Real.sqrt_mul (by positivity)] rw [Real.sqrt_sq (by simp [x])] all_goals positivity </pre>	<pre> import Mathlib import Aesop set_option maxHeartbeats 0 open BigOperators Real Nat Topology Rat theorem mathd_algebra_293_apollo (x : NNReal) : Real.sqrt (60 * x) * Real.sqrt (12 * x) * Real.sqrt (63 * x) = 36 * x * Real.sqrt (35 * x) := by have h1 : Real.sqrt (60 * x) * Real.sqrt (12 * x) * Real.sqrt (63 * x) = Real.sqrt ((60 * x) * (12 * x) * (63 * x)) := by rw [← Real.sqrt_mul (by positivity), ← Real.sqrt_mul (by positivity)] rw [h1] have h2 : (60 * x) * (12 * x) * (63 * x) = (36 * x) ^ 2 * (35 * x) := by ring_nf < > simp [x_mul_x] < > linarith [show (0 : ℝ) ≤ x from by positivity] try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith have h1 : √(60 : ℝ) = 2 * √15 := by rw [Real.sqrt_eq_iff_sq_eq] < > norm_num < > ring_nf < > norm_num have h2 : √(12 : ℝ) = 2 * √3 := by rw [Real.sqrt_eq_iff_sq_eq] < > norm_num < > ring_nf < > norm_num have h3 : √(63 : ℝ) = 3 * √7 := by rw [Real.sqrt_eq_iff_sq_eq] < > norm_num < > ring_nf < > norm_num have h4 : √(12 * 63 : ℝ) = √(12 * 63) = √(12 * 63) * (1 : ℝ) := by have h41 : √(12 * 63 : ℝ) = √(12 * 63) = √(12 * 63) * (1 : ℝ) := by simp [pow_succ] have h42 : √(12 * 63 : ℝ) = √(12 * 63) = √(12 * 63) := by rw [Real.sqrt_sq] exact NNReal.zero_le_coe rw [h41, h42] linarith rw [h1, h2', h3, h4] ring_nf < > simp [mul_assoc] try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith ... </pre>

Figure 13: Proof attempt of `mathd_algebra_293` produced by the base model (Kimina-Prover-Preview-Distill-7B) versus after Apollo’s targeted repair. The left shows the original end-to-end LLM output, which does not compile in Lean; the right shows the corrected, repaired proof assembled by Apollo that closes all of the goals.

LLM	LLM + APOLLO
<pre> import Mathlib import Aesop set_option maxHeartbeats 0 set_option pp.numericTypes true set_option pp.coercions.types true open BigOperators Real Nat Topology Rat theorem algebra_2varlineareq_fp3zeq11_3tfm1m5zeqn68_feqn10_zeq7 _llm (f z : C) (h₀ : f + (3 : C) * z = (11 : C)) (h₁ : (3 : C) * (f - 1) - (5 : C) * z = (-68 : C)) : f = (-10 : C) ^ z = (7 : C) := by -- Simplify the second equation using mul_sub have h₂ : 3 * f - 3 - 5 * z = -68 := by simp [mul_sub] using h₁ -- Combine like terms to isolate 3*f - 5*z have h₃ : 3 * f - 5 * z = -65 := by #1 linarith [h₂] -- Use the two equations to solve for z have z_eq : z = 7 := by #2 linarith [h₃, h₂] -- Substitute z back to solve for f have f_eq : f = -10 := by #3 linarith [h₃, z_eq] -- Conclude both equalities exact And.intro f_eq z_eq </pre>	<pre> import Mathlib import Aesop set_option maxHeartbeats 0 set_option pp.numericTypes true set_option pp.coercions.types true open BigOperators Real Nat Topology Rat theorem algebra_2varlineareq_fp3zeq11_3tfm1m5zeqn68_feqn10_zeq7 _apollo (f z : C) (h₀ : f + (3 : C) * z = (11 : C)) (h₁ : (3 : C) * (f - 1) - (5 : C) * z = (-68 : C)) : f = (-10 : C) ^ z = (7 : C) := by have h₂ : 3 * f - 3 - 5 * z = -68 := by simp [mul_sub] using h₁ have h₃ : 3 * f - 5 * z = -65 := by #1 try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith have h_expanded : (3 : C) * f - (3 : C) - (5 : C) * z = (-68 : C) := by gcongr have h_added : (3 : C) * f - (5 : C) * z = ((3 : C) * f - (3 : C) - (5 : C) * z) + (3 : C) := by try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith have h_add : f * (3 : C) - z * (5 : C) = (-68 : C) + (3 : C) := by have lhs : (-3 : C) + (f * (3 : C) - z * (5 : C)) + (3 : C) = f * (3 : C) - z * (5 : C) := by exact neg_add_cancel_comm (3 : C) (f * (3 : C) - z * (5 : C)) have rhs : (-68 : C) + (3 : C) = (-65 : C) + (3 : C) := by exact rfl rw [← lhs] at h₁ simp_all only have h_sum : (-68 : C) + (3 : C) = (-65 : C) := by try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith rw [h_add, h_sum] have h_subst : (3 : C) * f - (5 : C) * z = (-68 : C) + (3 : C) := by rw [h_expanded] at h_added exact h_added have h_simpl : (-68 : C) + (3 : C) = (-65 : C) := by norm_num exact Eq.trans h_subst h_simpl have z_eq : z = 7 := by try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith have f_eq_def : f = (11 : C) - (3 : C) * z := by exact eq_sub_of_add_eq h₀ have h3_sub : (3 : C) * ((11 : C) - (3 : C) * z) - (5 : C) * z = (-65 : C) := by rw [f_eq_def] at h₁ exact h₁ have h3_simplified : (3 : C) - (14 : C) * z = (-65 : C) := by norm_num try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith gcongr have isolate_z : (14 : C) * z = 98 := by try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith have eq1 : (30 : C) = ((-68 : C) + z * (14 : C)) := by rw [sub_eq_iff_eq_add] at h₁ gcongr have eq2 : (30 : C) + (68 : C) = ((-68 : C) + (68 : C)) + z * (14 : C) := by simp_all only [add_sub_cancel_right, neg_add_cancel_comm, neg_add_cancel, zero_add] ... have eq3 : (98 : C) = (0 : C) + z * (14 : C) := by norm_num at eq2 simp_all only [add_sub_cancel_right, zero_add] have eq4 : (0 : C) + z * (14 : C) = z * (14 : C) := by simp have eq_final : (98 : C) = z * (14 : C) := by simp_all only [add_sub_cancel_right, neg_add_cancel_comm, neg_add_cancel, zero_add] gcongr have z_sol : z = (7 : C) := by norm_num try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith have h_nonzero : (14 : C) ≠ 0 := by norm_num have z_inv : z = 98 * ((14 : C)⁻¹) := by exact (eq_mul_inv_iff_mul_eq h_nonzero).mpr isolate_z have z_div : z = 98 / (14 : C) := by exact z_inv have z_simpl : 98 / (14 : C) = (7 : C) := by norm_num rw [z_div, z_simpl] exact z_sol have f_eq : f = -10 := by try norm_cast ; try norm_num ; try simp_all ; try ring_nf at * ; try native_decide ; try linarith ; try nlinarith have h_f : f = (11 : C) - (21 : C) := by exact eq_sub_of_add_eq' h₀ have h_num : (11 : C) - (21 : C) = (-10 : C) := by norm_num rw [h_f, h_num] exact And.intro f_eq z_eq </pre>

Figure 14: Proof attempt of `algebra_2varlineareq_fp3zeq11_3tfm1m5zeqn68_feqn10_zeq7` produced by the base model (o4-mini) versus after Apollo’s targeted repair. The left shows the original end-to-end LLM output, which does not compile in Lean; the right shows the corrected, repaired proof assembled by Apollo that closes all of the goals.

LLM	LLM + APOLLO
<pre> import Mathlib import Aesop set_option maxHeartbeats 0 open BigOperators Real Nat Topology Rat theorem amc12a_2019_p12_llm (x y : ℝ) (h₀ : x ≠ 1 ∧ y ≠ 1) (h₁ : Real.log x / Real.log 2 = Real.log 16 / Real.log y) (h₂ : x * y = 64) : (Real.log (x / y) / Real.log 2) ^ 2 = 20 := by have x_nonzero : x ≠ 0 := by #1 rintro rfl simp at h₁ have y_nonzero : y ≠ 0 := by #2 rintro rfl simp at h₂ field_simp [Real.log_div, Real.log_mul, Real.log_inv, Real.log_pow] at h₁ ⊢ linarith #3 </pre>	<pre> import Mathlib import Aesop set_option maxHeartbeats 0 open BigOperators Real Nat Topology Rat theorem amc12a_2019_p12_apollo (x y : ℝ) (h₀ : x ≠ 1 ∧ y ≠ 1) (h₁ : Real.log x / Real.log 2 = Real.log 16 / Real.log y) (h₂ : x * y = 64) : (Real.log (x / y) / Real.log 2) ^ 2 = 20 := by have x_nonzero : x ≠ 0 := by #1 rintro rfl simp at h₁ linarith have y_nonzero : y ≠ 0 := by #2 rintro rfl simp at h₂ linarith field_simp [Real.log_div, Real.log_mul, Real.log_inv, Real.log_pow] at h₁ ⊢ ring_nf at h₁ ⊢ field_simp [Real.log_mul, Real.log_pow, Real.log_pow, mul_comm] at h₁ ⊢ let L := Real.log (2 : ℝ) let X := Real.log x let Y := Real.log y have hXY : X + Y = Real.log 64 := by rw [←Real.log_mul x_nonzero y_nonzero] rw [h₂] have hXY' : X + Y = 6 * L := by rw [hXY] have hL : 6 * Real.log (2 : ℝ) = Real.log (64) := by have h2_pos : (2 : ℝ) > 0 := by norm_num ... have H : Real.log (2^6) = 6 * Real.log (2 : ℝ) := by simp rw [←H] norm_num gcongr have h16 : Real.log (16 : ℝ) = 4 * L := by norm_num have h2_pos : (2 : ℝ) > 0 := by norm_num have H' : Real.log (2^4) = (4:ℝ) * Real.log (2 : ℝ) := by simp rw [←H'] norm_num rw [h16] at h₁ have hX : X = (4 * L^2) / Y := by have hL : Real.log (2 : ℝ) = L := rfl rw [hL] at h₁ have hL' : L * (4 * L) = 4 * L^2 := by linarith rw [hL'] at h₁ exact h₁ have hXY_prod : X * Y = 4 * L^2 := by rw [hX] field_simp [y_nonzero] ring_nf have Y_nonzero : Y ≠ 0 := by have y_log_nonzero : Real.log y ≠ 0 := by intro H have hy_exp : y = Real.exp (Real.log y) := by simp_all only [ne_eq, div_zero, log_eq_zero, false_or, log_neg_eq_log, log_one, exp_zero, mul_neg, mul_one, neg_neg, ofNat.one_ne_ofNat] rw [H, Real.exp_zero] at hy_exp have y_eq_one : y = 1 := hy_exp exact (h₀.right) y_eq_one exact y_log_nonzero simp_all only [ne_eq, isUnit_iff_ne_zero, not_false_eq_true, isUnit.mul_inv_cancel_right] have diff_eq : X^2 + Y^2 - 2 * X * Y = (X + Y)^2 - 4 * (X * Y) := by ring rw [hXY'] at diff_eq rw [hXY_prod] at diff_eq linarith </pre>

Figure 15: Proof attempt of amc12a_2019_p12 produced by the base model (Goedel-Prover-SFT) versus after Apollo’s targeted repair. The left shows the original end-to-end LLM output, which does not compile in Lean; the right shows the corrected, repaired proof assembled by Apollo that closes all of the goals.

Figure 16: Proof attempt of `mathd_algebra_289` produced by the base model (Kimina-Prover-Preview-Distill-7B) versus after Apollo’s targeted repair. The left shows the original end-to-end LLM output, which does not compile in Lean; the right shows the corrected, repaired proof assembled by Apollo that closes all of the goals. Import headers are the same as other presented examples, for visibility they were omitted in this diagram.