



Formally Solving Answer-Construction Problems in Lean

Jialiang Sun¹, Yuzhi Tang¹, Ao Li¹, Chris Maddison¹, and Kuldeep Meel^{1,2}

¹ University of Toronto, Toronto, Canada

jialiang.sun@mail.utoronto.ca, meel@cs.toronto.edu

² Georgia Institute of Technology, Atlanta, USA

Abstract

Large language models (LLMs) have achieved remarkable progress in formal mathematical reasoning. Mathematical competition problems fall into two broad types: theorem-proving problems ask for a proof of a fully specified statement, whereas answer-construction problems ask the solver to construct an answer object and prove that it satisfies the stated specification. Existing mathematical reasoning engines mainly target theorem-proving problems, yet answer-construction problems remain less studied. This setting is challenging because model capabilities are misaligned, with general LLMs better suited to answer construction and prover LLMs better suited to proof generation, and because Lean proof checking alone does not rule out inadmissible circular witnesses. To close this gap, we introduce *Enumerate-Conjecture-Prove* (ECP), a neuro-symbolic framework for solving answer-construction problems in Lean. ECP uses general LLMs to perform bounded enumeration and construct candidate answers, and invokes prover LLMs to produce machine-checked proofs. ECP introduces admissibility checking to ensure that each answer is canonical and does not involve a circular argument. On answer-construction problems from PutnamBench and autoformalized MathArena, ECP formally solves 17/346 PutnamBench instances and 18/75 MathArena instances with admissible answers and proofs at Pass@32. Under aligned proof-sampling and repair settings, ECP outperforms direct LLM baselines on MathArena at Pass@32 and on PutnamBench at Pass@4. Our code is available at <https://github.com/sunjia72/ecp-lpar>.

1 Introduction

Large language models (LLMs) have achieved remarkable progress in mathematical reasoning, with broad applications in math education, verified code generation [13, 39], and research-level math discovery [22, 29, 30, 32]. On one hand, *general LLMs*, such as models in the GPT-5 family, provide powerful human-like *informal math* capabilities. On the other hand, there is a growing community effort around large-scale formalized mathematics frameworks such as Lean Mathlib [19, 21], which provide the ecosystem and training corpus for *prover LLMs* that generate proofs in a machine-verifiable format. State-of-the-art prover LLMs can match and even outperform top human experts in formally solving high-school and undergraduate math competition problems [1, 5, 11].

We focus on math competition problems in Lean, which broadly fall into two types. *Theorem-proving* problems have the form $\forall X. P(X) \rightarrow Q(X)$: the task is to prove a fully specified target

statement. *Answer-construction* problems have the form $\exists \Psi. \forall X. P(X) \rightarrow Q(X, \Psi(X))$: the answer object Ψ is not supplied and must be synthesized together with a proof of correctness. Once Ψ is known, an answer-construction problem reduces to the theorem-proving task of proving the substituted statement. Figure 1 illustrates this distinction using AIME II (2026) Problem 8: the original problem asks for an answer and a proof, while the reduced version substitutes the ground-truth answer and asks only for a proof.

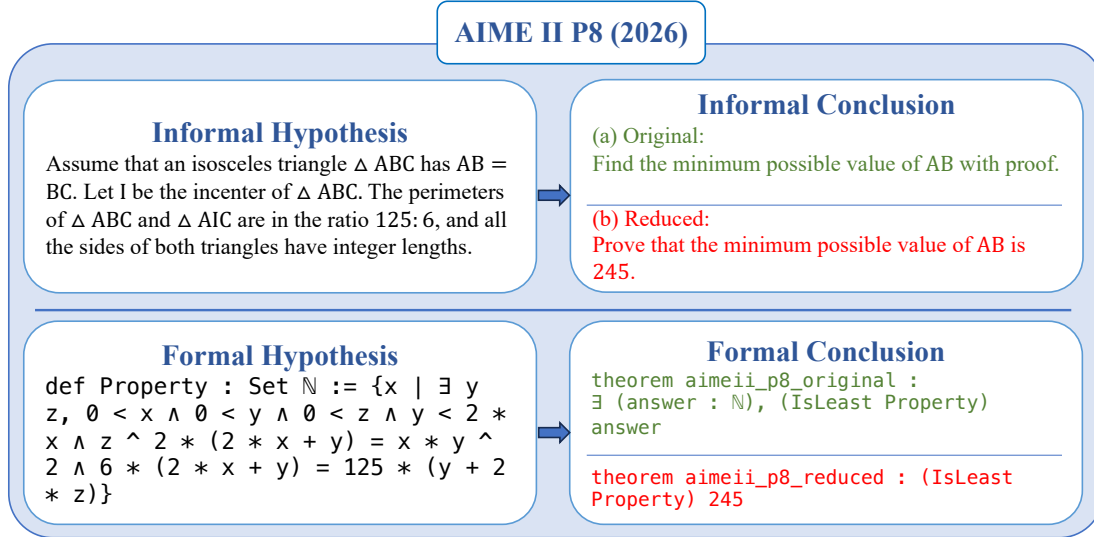


Figure 1: Example American Invitational Mathematics Examination (AIME) II Problem 8 (2026). **Top:** Informal problem statement in natural language, where the original task requires constructing an answer and proving it satisfies the statement, while the reduced task asks for a proof after the ground-truth answer has been substituted. **Bottom:** Formalized statement in Lean 4.

Existing methods and benchmarks in Lean mainly focus on theorem-proving problems. Popular benchmarks like ProofNet [2], MiniF2F [44], and FormalMath [43] are formalized from informal source problems, and answer-construction problems in the original informal sources are reduced to theorem-proving problems by substituting the ground-truth answers. Benchmarks like PutnamBench [33] include answer-construction problems, but these problems are reduced to theorem-proving versions in practice. In fact, according to the PutnamBench leaderboard for the Lean proof assistant¹, there are 33 “with-answer” (theorem-proving) evaluations, where the ground-truth answer is already substituted into the theorem statement, but only 2 “no-answer” (answer-construction) evaluations, where the answer needs to be constructed. Our method was ranked first in “no-answer” evaluation at the time of submission in June 2026. Current prover LLMs [15, 26, 34, 36, 37] focus on theorem-proving problems rather than answer-construction problems, and pure symbolic search is expensive for various domains [8, 10], with real quantifier elimination doubly exponential in the worst case [9].

The difficulty of formally solving answer-construction problems is twofold: (i) *admissibility checking for constructed answers* and (ii) *misaligned capabilities between general and prover LLMs*. For (i), the issue is as follows: in an answer-construction problem, we require the constructed answer to have the target format, such as “a natural-number numeral” (e.g. 1) or “a real number with arithmetic expressions” (e.g. $\frac{5}{2}$). We call such answers admissible

¹Source: <https://trishullab.github.io/PutnamBench/leaderboard.html>.

answers. However, Lean does not support checking whether a constructed answer is in the target format or simply “cheats” by circularly constructing a non-canonical answer that trivializes the proof. We show two proofs with different constructed answers in Figure 2: the AIME problem expects the answer to be a natural-number numeral, and both displayed proofs are accepted by the Lean compiler. The first proof’s answer is 245 and is clearly admissible. However, the second proof’s answer `sInf Property` essentially says “the minimum possible value of AB is the minimum possible value of AB ”. Although the second proof passes the Lean check, it is clearly inadmissible and trivializes the proof via a circular argument. In fact, as shown in our evaluation in Section 5, simply using a general or prover LLM to solve answer-construction problems can produce many spurious “successful” trivial proofs that contain inadmissible answers. For (ii), the issue is as follows: general LLMs are effective at constructing answers due to their

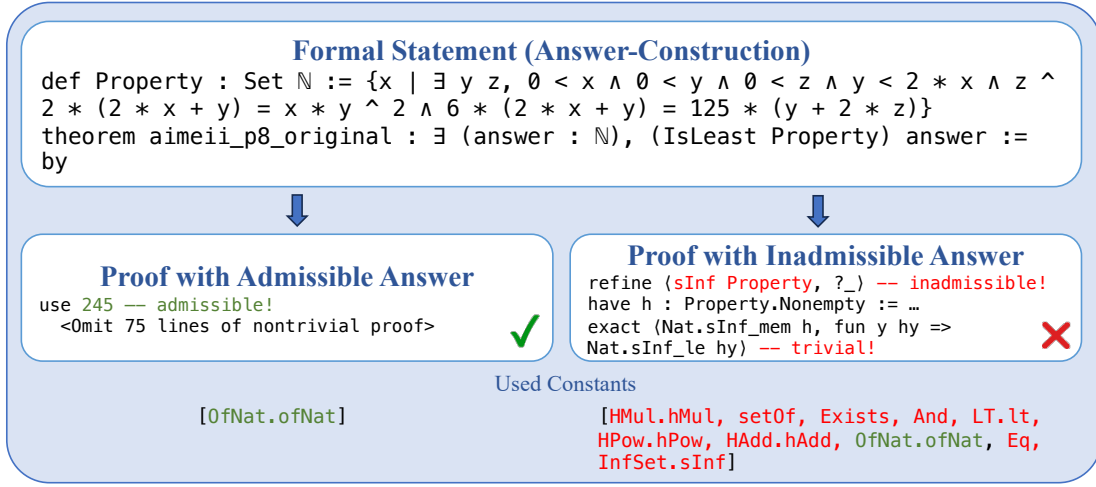


Figure 2: Two compiling Lean proofs for AIME II P8 (2026) with different constructed answers. **Left:** Proof with an admissible natural-number numeral. **Right:** Proof with an inadmissible circular answer. Complete proofs are available in the released artifact (Section 6).

strong informal reasoning capabilities and agentic tool calls, but are expensive; prover LLMs are fine-tuned on vast formal proof data and have higher formal proof-generation accuracy than general LLMs under aligned parameter sizes and cost budgets, but are mostly designed for pure theorem-proving tasks and are less reliable at answer construction.

To address the limitations of existing work, we propose Enumerate-Conjecture-Prove (ECP), an end-to-end framework for formally solving answer-construction problems. ECP first leverages general LLMs for enumeration and answer construction, then generates formal proofs via prover LLMs with answer refinement upon disproof. ECP verifies the solution through admissibility checking for the constructed answer and Lean compilation of the proof. Overall, ECP takes advantage of general and prover LLMs to correctly solve more answer-construction problems at lower cost, while ensuring the admissibility of constructed answers to avoid trivializing proofs. The ECP workflow is largely inspired by Pólya’s problem-solving methodology [24], which emphasizes exploration of simpler cases, pattern generalization, and proof. The answer refinement loop is also connected to work on counterexample-guided abstraction refinement [6, 7].

We organize the rest of the paper as follows. We first introduce preliminary notation and formulations of answer-construction problems in Section 2, followed by a discussion of related work in Section 3. We present our method, including problem preprocessing, answer construction, and proving, in Section 4. We then describe the benchmarks and present the evaluation results

in Section 5. Finally, we summarize ECP with an analysis of limitations and future work in Section 6.

2 Preliminaries

In this section, we introduce mathematical notation and basic Lean background, then describe helper functions and formulations of answer-construction problems that illustrate the ECP workflow.

2.1 Basic Notation

By convention, we denote by D the universe of discourse (domain) and by V the set of variables, each denoted by a lowercase letter with optional subscripts. Given sequences of variables $X = \langle x_1, \dots, x_n \rangle$ and $Y = \langle y_1, \dots, y_m \rangle$, and a predicate P over X and Y , we write $\forall X. \exists Y. P(X, Y)$ as shorthand for $\forall x_1. \dots \forall x_n. \exists y_1. \dots \exists y_m. P(x_1, \dots, x_n, y_1, \dots, y_m)$. When the variables are clear from the context, we write P as shorthand for $P(X, Y)$, and $\text{Vars}(P)$ for the set of variables occurring free in P . An assignment to a formula P is a mapping $\sigma : \text{Vars}(P) \rightarrow D$ that assigns each variable a concrete value in the domain; σ satisfies $P(X, Y)$ if P evaluates to True under σ . For a variable $y_i \in Y$ and a value $b \in D$, we write $P(X, Y)_{|y_i=b}$ for the formula obtained by substituting all free occurrences of y_i in P with b . For a tuple $Y' = \langle y'_1, \dots, y'_m \rangle$, we write $P(X, Y')$ for the simultaneous substitution of each y_i with y'_i in P . We say that a set is *extensionally defined* if it is an explicit finite enumeration of elements (e.g., $\{1, 2, 3\}$), or *intensionally defined* if it is specified by a predicate that characterizes the elements belonging to the set (e.g., $\{x \in \mathbb{R} \mid x > 0\}$).

2.2 Lean 4 and Dependent Type Theory

Lean 4 [21] is a proof assistant and functional programming language whose kernel implements a dependent type theory in the family of the Calculus of Inductive Constructions (CIC). Mathlib [19] is the largest Lean standard library of definitions, theorems, and proofs in cutting-edge formalized mathematics. In Lean, propositions are types and proofs are terms according to the Curry-Howard correspondence. A *type judgment* has the form $\Gamma \vdash t : T$, meaning that the term t has type T under a typing context Γ (a list of variable bindings). The language is based on a small core of term formers: variables, constants (definitions) from the Lean environment Σ , function application, λ -abstraction, dependent function types (also written as $\forall$ -types), and let-bindings. Example constants include `Real.pi` (denoting π) and functions such as `Real.sin` (denoting $\sin(\cdot)$) and `HAdd.hAdd` (denoting $+$). The Lean elaborator translates Lean code into this kernel representation by inserting *implicit* arguments, such as universe parameters and type arguments, and *instance* arguments, such as typeclass dictionaries. To align our presentation with standard dependent type theory, we describe Lean expressions using a compact core calculus according to LeanAuto's [25] λC type system with a countable number of non-cumulative universe levels. Let U_ℓ ($\ell \in \mathbb{N}$) denote the sorts, let V denote the set of variables, and let Σ denote the set of constants. The syntax of expressions is inductively defined as follows:

$$T ::= V \mid \Sigma \mid U_\ell \mid TT \mid \lambda(V : T).T \mid \forall(V : T).T \mid \text{let } (V : T) := T; T.$$

Here, TT denotes function application, $\lambda(V : T).T$ denotes λ -abstraction, $\forall(V : T).T$ denotes a dependent product type, and $\text{let } (V : T) := T; T$ denotes a let-binding. For example, `Real.sin x`

corresponds to TT with head constant `Real.sin` and variable x , and `HAdd.hAdd a b` corresponds to $((\text{HAdd.hAdd } a) \ b)$, i.e., nested applications using TT . Similarly, `fun x : Real => Real.sin x + 1` corresponds to $\lambda(x : \mathbb{R}). ((+) (\sin x)) \ 1$ after elaboration, where $+$ is a constant such as `HAdd.hAdd` with implicit and instance arguments inserted.

2.3 Helper Functions

We use the following helper functions for notational convenience in algorithm descriptions. Let $\perp$ denote the empty string. `LLM(prompt)` queries a language model with an input prompt string `prompt` and outputs a response string. `Python(code)` executes a Python program string `code` and returns its standard output and standard error. `Lean(code)` compiles a Lean program string `code` and returns `error_message`. When `code` is composed of a theorem statement and proof, `error_message =  $\perp$` indicates that the proof is correctly verified and does not contain `sorry`. We refer to *general LLMs* as models trained for broad tasks, such as GPT-5.4 [27] and Qwen3 [38], and to *prover LLMs* as models fine-tuned for formal theorem proving in Lean, such as Goedel-Prover-V2 [15].

2.4 Answer-Construction Formulations

We model the *answer-construction* problem as a *functional-synthesis* problem: synthesize a mathematical object (an answer function) satisfying given specifications, together with a machine-checkable proof of correctness. Let D be the universe of discourse, and let $V = X \cup Y$ be the variables, including *context variables* $X = \langle x_1, \dots, x_n \rangle$ and *answer variables* $Y = \langle y_1, \dots, y_m \rangle$. The *hypothesis predicate* $P(X)$ denotes the constraint for context variables, and the *conclusion predicate* $Q(X, Y)$ denotes the constraint for both context and answer variables. Given any X satisfying $P(X)$, the answer function $\Psi(X) = \langle \psi_1(X), \dots, \psi_m(X) \rangle$ must map each such X to either a witness Y satisfying $Q(X, Y)$ (*find-any problem*) or the set of all witnesses Y satisfying $Q(X, Y)$ (*find-all problem*). In particular, a find-any problem is written as $\exists \Psi. \forall X. P(X) \rightarrow Q(X, \Psi(X))$, and a find-all problem is written as $\exists \Psi. \forall X. P(X) \rightarrow \forall Y (Y \in \Psi(X) \leftrightarrow Q(X, Y))$. The admissible vocabulary A is the set of allowed constants that can appear in the expression of constructed answers, and the answer Ψ is said to be *admissible* if all constants it uses are in A . Overall, each answer-construction problem is specified by the 5-tuple $T = (D, V, P, Q, A)$. When the context is clear, we use T to denote the formalized Lean statement for the answer-construction task, and T_Ψ to denote the statement after the $\exists$ is eliminated by substituting the explicit Ψ . We say that a proof successfully solves the answer-construction problem T if it compiles without error in Lean, does not contain placeholders like `sorry`, `admit`, and the constructed answer in the proof is admissible.

Example 1. The AIME II P8 example in Figure 1 is a find-any problem. In fact, it does not have any free context variables, so $X := \emptyset$ and $P := \text{True}$. The only answer variable is $Y = \langle y_1 \rangle$ in domain $D = \mathbb{N}$, which is essentially the minimum possible value of the edge length AB in the problem. Notice that this problem can equivalently be stated as “*find the least positive integer x such that there exist positive integers y, z with $y < 2x$, $z^2(2x + y) = xy^2$, and $6(2x + y) = 125(y + 2z)$* ” by translating the geometric information into algebraic form. Thus, $Q(y_1) := \left(y_1 = \min\{y \in \mathbb{N} : \exists z_1, z_2 \in \mathbb{N}, (y, z_1, z_2 > 0) \wedge 2y > z_1 \wedge z_2^2(2y + z_1) = yz_1^2 \wedge 6(2y + z_1) = 125(z_1 + 2z_2)\} \right)$. The formalized statement is obtained from this information. Since the AIME competition requires the synthesized Ψ to be a canonical natural number without arithmetic operations, the admissible vocabulary is $A = [\text{OfNat.ofNat}]$.

3 Related Work

Prover LLMs in Lean. Interactive theorem provers (ITPs) such as Isabelle [23] and Lean [21] support human-friendly environments for proof development. Lean supports built-in proof automation tactics such as LeanSMT [20] and aesop [14]. Recent prover LLMs combine these formal environments with large-scale proof search and training on a vast formal corpus. Open-weight prover LLMs such as Goedel-Prover [15], DeepSeek-Prover [26, 36, 37], InternLM-Prover [42], and Kimina-Prover [34] substantially improve Lean proof success on standard Lean theorem-proving benchmarks such as miniF2F and PutnamBench. Inference-time techniques like Seed-Prover [5] and Aristotle [1] maintain lemma pools of proved conjectures to optimize the efficacy of prover LLMs and achieve silver- to gold-medal-level results in IMO. These systems are primarily evaluated on theorem-proving (non-answer-construction) problems with specified conclusions, whereas our answer-construction setting focuses on tasks with existentially quantified answer variables.

Answer-Construction Systems. Several systems address answer-construction problems. PyEuclid [12] combines an AlphaGeometry-style deduction engine [31] with algebraic computation to solve Euclidean geometry calculation problems with proofs in a geometry DSL. FPS [18] studies general answer-construction problems in Lean 4 through a search-based algorithm, and reports solving 1 PutnamBench problem in the answer-construction format. High-performance formal reasoning engines have also shown strong results on Olympiad-level problems. AlphaProof [11] solved three out of six IMO 2024 problems in Lean 4 and one more problem in the AlphaGeometry DSL, contributing to silver-medal performance. It also leverages general LLMs for answer-construction problems to sample hundreds of candidate answers and attempt proofs on the substituted Lean statements. However, AlphaProof’s model weights and implementation are closed-source. Moreover, its training and inference pipelines are substantially more expensive: in fact, AlphaProof reports around 180,000 TPU-days of Google v6e for data processing and reinforcement learning, several orders of magnitude more compute than our experiments use.

ECP combines bounded Python enumeration for answer construction with admissibility checking, processed answer-construction benchmarks, and an openly released implementation.

Formal Benchmarks. General formal benchmarks such as MiniF2F [44], FIMO [16], Lean-WorkBook [41], ProofNet [2], and FormalMath [43], as well as domain-specific benchmarks like LeanGeo [28] (geometry), MO-INT [35] (inequality), and FormalML [40] (machine learning theory), are constructed by formalizing raw informal mathematics from math textbooks and past competition problems. These benchmarks focus on formal theorem-proving problems, and the ground-truth answers are substituted into the statements during formalization if the raw informal statements are answer-construction problems. PutnamBench [33] (Putnam competition) and CombiBench [17] (combinatorics) include some answer-construction problems. However, most methods still evaluate on the reduced theorem-proving versions of problems, and there is no formal mechanism to detect the admissibility of candidate constructed answers.

4 Methods

In this section, we describe the main workflow of ECP. Prior to problem solving, we first preprocess the raw answer-construction problems into Lean formalizations with admissible vocabularies according to the template in Section 2.4. Then ECP solves the answer-construction problem as follows. (i) During the Enumerate and Conjecture stages, the general LLM constructs

an answer by first performing bounded Python enumeration and then generalizing to the broader case to conjecture an admissible answer in Lean. (ii) During the Prove stage, the constructed answer is substituted into the theorem statement and the prover LLM generates the proof with automation tactics. ECP returns to the Enumerate and Conjecture stage if the substituted statement can be disproved. An overview of ECP is shown in Figure 3. We explain each stage in detail in the subsequent subsections.

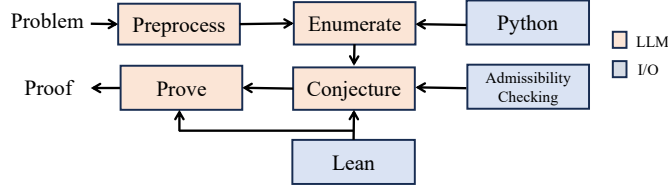


Figure 3: An overview of the ECP workflow.

4.1 Preprocess Stage

Autoformalization. During the Preprocess stage, the raw informal problem statement in natural language is converted into a formal problem with an admissible vocabulary according to the problem formulation template in Section 2.4. Specifically, we prompt a general LLM with the raw informal problem and instructions for producing the problem formulation according to the template provided in the released artifact (Section 6), and ask it for a compilable formalization. Each formalization is validated using Lean compilation for syntax correctness and manual inspection for equivalence to its informal statement. Failed formalizations are sent to the LLM for repair. Using this approach, we autoformalized the 75 problems in the 2026 MathArena Final Answer Competitions [3], consisting of 30 AIME problems, 33 HMMT (Harvard-MIT Mathematics Tournament) problems, and 12 Apex problems. In addition, out of 672 existing formalized problems in PutnamBench [33], we found that 346 are answer-construction problems and processed them into our problem formulation.

Admissible Vocabularies. We label the admissible vocabularies for each problem using rule-based heuristics according to each problem’s source and answer type. In particular, we design three levels of vocabulary breadth for different problem sources: (i) basic numbers without arithmetic operations (for AIME problems in MathArena); (ii) expressions with elementary arithmetic operations such as $+$, $-$, $\times$, and $\div$ (for HMMT and Apex problems in MathArena); (iii) expressions with university-level constants (for PutnamBench problems). We also infer vocabularies from answer types: if the expected answer has tuple-, set-, or function-related constructor constants, detected by occurrences of $\times$, **Set**, or $\rightarrow$ in the answer type, we include additional vocabulary accordingly.

After labeling, we run a validation test to examine whether each problem’s labeled admissible vocabulary covers the constants actually used in the ground-truth answer. We found that all 75/75 MathArena problems and 301/346 (87.0%) PutnamBench problems passed validation. We discovered that the remaining 45/346 (13.0%) PutnamBench problems have complicated answer formats and manually adjusted their vocabularies as follows.

These problems are mostly find-all tasks whose answers require quantified descriptions, sums or products, special constants, or intensional sets. Table 1 summarizes the adjustments. The categories overlap, since a problem may require more than one adjustment. In particular, quantified and intensional descriptions allow answers to specify a set by a predicate rather than

by explicitly listing its elements. After these adjustments, all 346 PutnamBench ground-truth answers passed vocabulary validation. The complete vocabularies and problem lists are available in the released artifact.

Answer format	Count	Adjustment
Quantified descriptions	25	Enable quantifiers in candidate answers
Sums or products	3	Include <code>Finset.sum</code> , <code>Finset.prod</code> , etc.
Special constants	5	Add matrix and measure-theoretic constants
Intensional sets	20	Use intensional set vocabulary

Table 1: Overlapping vocabulary adjustments for 45 PutnamBench problems.

4.2 Admissibility Checking

In addition to semantic correctness, answer-construction problems in Lean require that the synthesized answer be admissible, meaning that it is composed of constants in the admissible vocabulary. For example, in the AIME II P8 example in Figure 1, answers like 1 or 245 are admissible, but answers like `sInf Property` are inadmissible. To check admissibility, it is sufficient to extract the constants used by the answer Ψ following Lean’s expression syntax from Section 2.2. Let Σ be the set of global constants, let $A \subseteq \Sigma$ be the admissible vocabulary, and let $A_{\text{admin}} \subseteq \Sigma$ denote the set of *administrative* constants that arise from the Lean compiler runtime, including type formers, typeclass instances, instance-producing constants, and compiler-generated private constants. We ignore constants in A_{admin} during admissibility checking.

The check recursively collects the constants from each expression and combines the results over its subexpressions. For applications, it visits the head and explicit arguments; implicit and instance arguments are omitted. Binder types and bodies, let-bound values, and projections are handled by the corresponding cases below.

For an application expression t , we use spine decomposition: $t = h \ a_1 \ a_2 \ \cdots \ a_n$, where h is the *head* and $a_1, \dots, a_n$ are the arguments. We define $\text{ExplicitArgs}(t)$ using Lean’s elaborated function-argument metadata for explicit binders. In the projection case, π_i denotes the corresponding field projection constant. The constants used by the expression t , $\text{UsedConstants}(t)$, are inductively defined as follows:

$$\text{UsedConstants}(t) = \begin{cases} \{c\} & t = c, \ c \in \Sigma \setminus A_{\text{admin}} \\ \bigcup_{t' \in \{A, b\}} \text{UsedConstants}(t') & t = \lambda(x : A). b \\ \bigcup_{t' \in \{A, B\}} \text{UsedConstants}(t') & t = \forall(x : A). B \\ \bigcup_{t' \in \{A, u, v\}} \text{UsedConstants}(t') & t = \text{let } (x : A) := u; v \\ \text{UsedConstants}(t') \cup \{\pi_i\} & t = \pi_i(t') \\ \bigcup_{t' \in \{h\} \cup \text{ExplicitArgs}(t)} \text{UsedConstants}(t') & t = h \ a_1 \ \cdots \ a_n \\ \emptyset & \text{otherwise} \end{cases} \quad (1)$$

We define admissibility of the constructed answer Ψ with respect to A , $\text{Admissible}(A, \Psi)$, as $\text{UsedConstants}(\Psi) \subseteq A$. Existential and universal quantifiers require slightly different handling.

First, $\exists$ can be treated as the constant `Exists` because the expression $\exists(x : A). B$ is elaborated as `Exists (fun x: A => B)` in Lean. This expression is handled by the application case $t =$

$h a_1 \cdots a_n$, with `Exists` as the head h , together with the lambda-abstraction case $t = \lambda(x : A).b$ for `(fun x: A => B)`.

Second, if $\forall$ is not allowed, then the expression t is inadmissible whenever the case $t = \forall(x : A).B$ is executed during `UsedConstants(t)`.

Answer Extraction for Baseline Proofs. For direct LLM baselines, the answer is contained in the generated proof rather than supplied separately. We inspect the compiled proof term to extract the witness of its outermost existential quantifier. The extractor reduces the proof at its root and traverses metadata, let-bindings, and lambda-abstractions until it finds an application of `Exists.intro`, from which it takes the witness. The same admissibility check is then applied to this witness. A proof is not counted as successful if witness extraction fails or the extracted answer is inadmissible. The extraction code is included in the released artifact.

4.3 Enumerate and Conjecture Stage

During the Enumerate and Conjecture stage, ECP is given the Lean answer-construction problem statement T and the expected answer type. It first writes Python programs for bounded enumeration (Enumerate stage), then constructs an answer that generalizes from enumeration with mathematical reasoning (Conjecture stage), as described in Algorithm 1.

LLMs with Tool Calling. General LLMs, such as the commercial model GPT-5.4 and the open-source local model Qwen3, support reasoning with tool calls for code generation and execution. In particular, the LLM encloses tool calls (code content) inside delimiters `<tool_call>...</tool_call>` under the OpenAI function-calling JSON schema. The compiler backend executes the tool call and adds stdout and stderr to the rollout response using delimiters `<tool_response>...</tool_response>` immediately after `</tool_call>`. For fast and parallel code execution, we deploy the SandboxFusion [4] server as the code interpreter backend, and we restrict the tool interface to Python calls. Let `ParseToolCall` denote the helper function that extracts the tool call from a response.

Enumerate Stage. Initially, the prompt is built from the system instruction template in the released artifact and the input examples. ECP first queries a general LLM to write one or more Python programs to directly solve for the answer or search for tuples (X, Y) satisfying $P(X) \wedge Q(X, Y)$ over bounded and tractable values, as shown in Lines 2–7 in Algorithm 1. Each Python program is executed by SandboxFusion, and each program and its output are added back to the context. There may be multiple rounds of tool calls, depending on whether the enumeration results have provided sufficient evidence for answer construction. When the LLM no longer generates tool calls or hits `max_enumerate_rounds`, it proceeds to the Conjecture stage.

Conjecture Stage. During the Conjecture stage, the LLM sees the Python programs and execution results produced by the Enumerate stage, performs mathematical reasoning toward the answer construction by generalizing beyond bounded enumerations, and proposes the explicit final answer Ψ inside the delimiter `<<<...>>>` for extraction (denoted by `ExtractAnswer(response)`). ECP then runs Lean compilation and an admissibility check for the constructed answer, and returns the answer for repair if it fails either test.

An explicit example of LLM traces during answer construction for AIME II P8 (2026) is shown in Figure 4. During the Enumerate stage, the LLM first uses SymPy to solve for a key

Algorithm 1 ConstructAnswer(T , FailedAnswers = $\perp$)

```

1: context  $\leftarrow$  BuildPrompt( $T$ , FailedAnswers)
2: for  $i \leftarrow 1$  to max_enumerate_rounds do
3:   response  $\leftarrow$  LLM(context)
4:   code  $\leftarrow$  ParseToolCall(response)
5:   if code =  $\perp$  then
6:     break
7:   context  $\leftarrow$  context + code + Python(code)
8: for  $i \leftarrow 1$  to max_conjecture_rounds do
9:   response  $\leftarrow$  LLM(context)
10:   $\Psi \leftarrow$  ExtractAnswer(response)
11:  error_message  $\leftarrow$  Lean( $\Psi$ )
12:  is_admissible  $\leftarrow$  Admissible( $A$ ,  $\Psi$ )
13:  if error_message =  $\perp$  and is_admissible then
14:    return  $\Psi$ 
15:  if error_message  $\neq \perp$  then
16:    context  $\leftarrow$  context +  $\Psi$  + error_message
17:  if not is_admissible then
18:    context  $\leftarrow$  context +  $\Psi$  + "inadmissible"
19: return  $\perp$ 

```

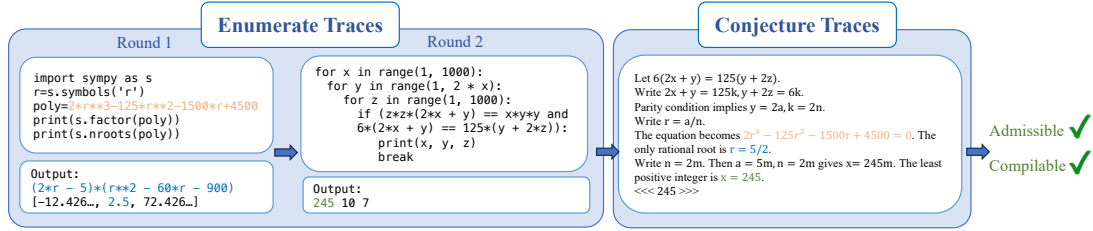


Figure 4: LLM traces for solving AIME II P8 (2026) during the Enumerate (**Left**) and Conjecture (**Right**) stages. The Conjecture trace is manually compacted due to the page limit.

equation with rational root $r = \frac{5}{2}$ and writes a brute-force enumeration to print the smallest valid answer. In the Conjecture stage, it leverages the key enumerated results, performs mathematical reasoning, and concludes that the answer is 245, which compiles in Lean and is admissible.

Additional Example: Putnam 1990 A1. As an additional example beyond AIME II P8, we consider Putnam 1990 A1, whose answer is a pair of functions. Let $T_0 = 2$, $T_1 = 3$, and $T_2 = 6$, and for $n \geq 3$, let

$$T_n = (n + 4)T_{n-1} - 4nT_{n-2} + (4n - 8)T_{n-3}$$

The problem asks for a formula $T_n = A_n + B_n$, where $\{A_n\}$ and $\{B_n\}$ are well-known sequences, together with a proof. Its Lean answer type is $(\mathbb{N} \rightarrow \mathbb{Z}) \times (\mathbb{N} \rightarrow \mathbb{Z})$. The preprocessing mechanism therefore includes tuple constructors together with the advanced numeric vocabulary used for Putnam problems.

The Lean 4 statement is shown below:

Lean 4 Statement (Proof Omitted)

```

theorem putnam_1990_a1 :  $\exists$  (putnam_1990_a1_solution :  $(\mathbb{N} \rightarrow \mathbb{Z}) \times (\mathbb{N} \rightarrow \mathbb{Z})$ ),  $\forall$  (T :  $\mathbb{N} \rightarrow \mathbb{Z}$ ),  $((T(0 : \mathbb{N}) = (2 : \mathbb{Z}) \wedge T(1 : \mathbb{N}) = (3 : \mathbb{Z}) \wedge T(2 : \mathbb{N}) = (6 : \mathbb{Z})) \wedge (\forall$ 
 $(n : \mathbb{N}), T(n + (3 : \mathbb{N})) = ((\uparrow n : \mathbb{Z}) + (7 : \mathbb{Z})) * T(n + (2 : \mathbb{N})) - (4 : \mathbb{Z}) * ((\uparrow n$ 
 $: \mathbb{Z}) + (3 : \mathbb{Z})) * T(n + (1 : \mathbb{N})) + ((4 : \mathbb{Z}) * (\uparrow n : \mathbb{Z}) + (4 : \mathbb{Z})) * T n)) \rightarrow (T =$ 
putnam_1990_a1_solution.1 + putnam_1990_a1_solution.2)

```

During the Enumerate stage, ECP computes the initial terms using the following Python program:

Python

```

def seq(N):
    T=[0]*(N+1)
    T[0]=2; T[1]=3; T[2]=6
    for n in range(N-2):
        T[n+3]=(n+7)*T[n+2]-4*(n+3)*T[n+1]+(4*n+4)*T[n]
    return T
T=seq(15)
print(T)

```

The output begins 2, 3, 6, 14, 40, 152, 784, 5168. In the Conjecture stage, the model observes that these values match $n! + 2^n$ and constructs the following Lean answer:

Lean

```

(fun n :  $\mathbb{N}$  => (Nat.factorial n :  $\mathbb{Z}$ ), fun n :  $\mathbb{N}$  =>  $(2 : \mathbb{Z}) \sim n$ )

```

This answer generalizes beyond the enumerated terms. The initial values agree, and both $A_n = n!$ and $B_n = 2^n$ satisfy the recurrence. After substituting the answer, the Prove stage verifies $T_n = n! + 2^n$ for all n by strong induction, using the three initial values and the recurrence in the inductive step. The complete trace and Lean proof are provided in the released artifact.

4.4 Prove Stage

After the Conjecture stage, the constructed answer is substituted into the theorem statement to form T_Ψ using the Lean tactic `use  $\Psi$` . The goal of the Prove stage is to find a proof of T_Ψ . It first applies Lean’s automation engines, denoted by `Auto`, by sequentially applying the automation tactics `native_decide`, `simp`, `aesop` [14], `ring`, and `norm_num`. If automation fails, it invokes a prover LLM with a repair loop that incorporates error messages from unsuccessful attempts, as shown in Algorithm 2. In practice, this procedure runs for multiple parallel independent attempts and succeeds if any attempt outputs a successful proof.

In ECP, `Prove` is invoked for both T_Ψ and its negation $\neg T_\Psi$. A successful `Prove( $T_\Psi$ )` concludes the ECP loop; if `Prove( $\neg T_\Psi$ )` succeeds, ECP adds Ψ to the failed answer collection and returns to the Enumerate and Conjecture stage for answer reconstruction. The full proof of the AIME II P8 example in Figure 1 is available in the released artifact.

4.5 Equivalence Testing for Answers

In addition to end-to-end proof generation, this approach naturally applies to equivalence checking between the ground-truth answer $\Psi_{\text{ground_truth}}$ and a constructed answer Ψ by proving the Lean goal $\Psi = \Psi_{\text{ground_truth}}$, when the problem has a find-all type or a find-any type

Algorithm 2 $\text{Prove}(T_\Psi)$

```

1: for  $j \leftarrow 1$  to parallel_proof_samples do
2:   feedback  $\leftarrow \perp$ 
3:   error_message  $\leftarrow \text{Lean}(T_\Psi + \text{automation})$ 
4:   if error_message  $= \perp$  then
5:     return automation
6:   for  $i \leftarrow 1$  to max_prove_rounds do
7:     proof  $\leftarrow \text{LLM}(T_\Psi + \text{feedback})$ 
8:     error_message  $\leftarrow \text{Lean}(\text{proof})$ 
9:     if error_message  $= \perp$  then
10:      return proof
11:   feedback  $\leftarrow \text{proof} + \text{error\_message}$ 
12: return  $\perp$ 

```

Algorithm 3 $\text{ECP}(T)$

```

1: FailedAnswers  $\leftarrow \perp$ 
2: for  $r \leftarrow 1$  to max_ecp_rounds do
3:    $\Psi \leftarrow \text{ConstructAnswer}(T, \text{FailedAnswers})$ 
4:   if  $\Psi = \perp$  then
5:     return  $\perp$ 
6:   proof  $\leftarrow \text{Prove}(T_\Psi)$ 
7:   disproof  $\leftarrow \text{Prove}(\neg T_\Psi)$ 
8:   if proof  $\neq \perp$  then
9:     return use  $\Psi$ ; + proof
10:  if disproof  $\neq \perp$  then
11:    FailedAnswers  $\leftarrow \text{FailedAnswers} + \Psi$ 
12:    continue
13:  break
14: return  $\perp$ 

```

with a unique answer. This test uses the fixed tactic portfolio and prover LLMs described in Section 4.4, making the check type-aware and certified whenever it succeeds. It can also prove equivalences that may be missed by script-based symbolic checkers, such as MathArena’s evaluation protocol [3], or by traditional LLM judges.

5 Evaluation

In this section, we evaluate ECP against baseline methods for solving answer-construction problems across multiple math competition benchmarks in Lean. The evaluated benchmarks span difficulty levels from high school to university and include contamination-free problems beyond the evaluated models’ knowledge cutoffs. Our evaluation aims to answer the following research questions (RQs):

- RQ 1. Problem-Solving Accuracy.** Can ECP successfully solve more answer-construction problems in Lean than baseline methods?
- RQ 2. Answer-Only Accuracy.** Can ECP predict more correct answers without proof for answer-construction problems than baseline methods?
- RQ 3. Cost Efficiency.** Can ECP solve formal answer-construction problems at a lower cost than baseline methods?

RQ 4. Ablation Studies. What are the contributions of each component in ECP?

Dataset Setup. We evaluate on two benchmarks: PutnamBench [33] and MathArena [3]. PutnamBench originally includes 672 Putnam Mathematical Competition problems from 1962–2025 that are manually formalized in Lean 4 by human experts. We extract the subset of 346 instances that are answer-construction problems. The MathArena Final Answer Competitions benchmark² [3] contains 75 problems from the AIME I, AIME II, HMMT (February), and Apex competitions in 2026, all of which are contamination-free for the evaluated models. Since the raw MathArena problems are originally written in natural language, we preprocess them into the target formulation of answer-construction problems following Section 4.1.

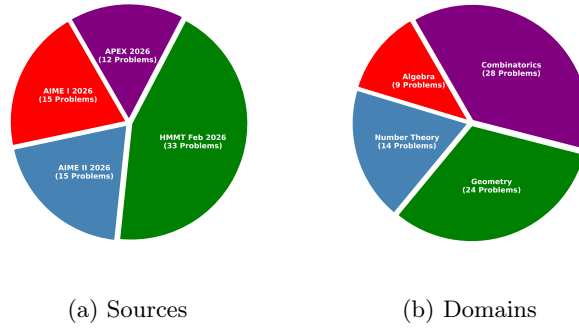


Figure 6: Problem sources and domains for MathArena 2026.

Hyperparameter Setup. We evaluate the general LLMs GPT-5.4 and Qwen3-32B, and the prover LLM Goedel-Prover-V2-32B. For the ECP setup, we set the Enumerate, Conjecture, and ECP round limits to 10, 5, and 2, respectively. For the Prove stage, we use 4 and 32 parallel proof samples and a maximum of 3 prove rounds. Lean compilation is conducted on 64 AMD EPYC 9655 CPU cores in parallel with a 120-second timeout. The Qwen3 and Goedel-Prover-V2-32B models are deployed on 4 H100 SXM (80GB) GPUs in parallel. The GPT-5.4 model is queried via the OpenAI API platform. The default model setup of ECP uses GPT-5.4 for the Enumerate and Conjecture stages, and Goedel-Prover-V2-32B for the Prove stage.

Metric Setup. For **RQ 1**, we evaluate ECP’s and the baseline models’ accuracy for answer-construction problem solving. A proof is said to successfully solve problem T if it compiles in Lean without error, does not contain `sorry`, and the constructed answer in the proof is admissible, as described in Section 2. The verifier rejects reported `sorry` placeholders and incomplete-proof warnings. The released successful proof scripts contain no additional `axiom` declarations. Recall that `parallel_proof_samples` in ECP’s Prove is aligned to the $\text{Pass}@k$ accuracy metric, which generates k independent parallel outputs and succeeds if any proof is successful. We also compare baseline LLMs GPT-5.4 and Goedel-Prover-V2-32B without ECP under the same $\text{Pass}@k$ and proof-correction attempt settings, meaning that they have the same `parallel_proof_samples`, `max_prove_rounds`, and access to the proof automation portfolio. This alignment applies to the Prove stage; **RQ 3** reports the costs of the full workflows. For **RQ 2**, we focus on answer

²Source: https://matharena.ai/?comp=overall-final_answer_comps&view=problem

prediction without proof. That is, we only evaluate the number of constructed answers that are provably equivalent to the ground-truth answers of each dataset. For **RQ 3**, we report the API costs and GPU hours of the methods. The OpenAI API rates of GPT-5.4 are 2.5 USD per 1M input tokens and 15.0 USD per 1M output tokens. Our GPUs are on computing clusters, but we assume the H100 GPU hourly rate is 2.0 USD³ to normalize costs. For **RQ 4**, we analyze the effects of the Enumerate stage, answer reconstruction, proof repair, and admissibility checking.

5.1 RQ 1: Problem-Solving Accuracy

ECP successfully solves more instances than baseline LLMs on both the MathArena and PutnamBench benchmarks, as shown in Table 2. In particular, on MathArena, the strongest LLM-only baseline GPT-5.4 solves 13/75 problems at Pass@32, while ECP with GPT-5.4 answer construction and Goedel-Prover-V2-32B solves 18/75. On PutnamBench, due to the high cost (projected at approximately 1400 USD) of GPT-5.4 baselines with Pass@32, we omit this setup and compare ECP with the GPT-5.4 baseline under Pass@4: ECP and GPT-5.4 solve 9/346 and 8/346 problems, respectively. At Pass@32, ECP solves 17/346 problems, with no corresponding GPT-5.4 result at this setting. The complete lists of solved instances are available in the released artifact (Section 6). Overall, ECP gives the strongest end-to-end accuracy among the evaluated methods on both benchmarks, with the GPT-5.4 comparison at Pass@32 limited to MathArena.

Method	Pass@ <i>k</i>	High School (MathArena 2026)					Undergraduate
		AIME I (15)	AIME II (15)	HMMT (33)	Apex (12)	Total (75)	PutnamBench (346)
LLM Baselines							
Goedel-Prover-V2-32B	4	3 (20.0%)	1 (6.7%)	2 (6.1%)	0 (0.0%)	6 (8.0%)	6 (1.7%)
	32	4 (26.7%)	1 (6.7%)	2 (6.1%)	0 (0.0%)	7 (9.3%)	7 (2.0%)
GPT-5.4	4	2 (13.3%)	2 (13.3%)	3 (9.1%)	0 (0.0%)	7 (9.3%)	8 (2.3%)
	32	6 (40.0%)	2 (13.3%)	5 (15.2%)	0 (0.0%)	13 (17.3%)	-
ECP (Ours)							
GPT-5.4+Goedel-Prover-V2-8B	4	6 (40.0%)	4 (26.7%)	6 (18.2%)	0 (0.0%)	16 (21.3%)	2 (0.6%)
	32	6 (40.0%)	4 (26.7%)	7 (21.2%)	0 (0.0%)	17 (22.7%)	7 (2.0%)
GPT-5.4+Goedel-Prover-V2-32B	4	6 (40.0%)	4 (26.7%)	7 (21.2%)	0 (0.0%)	17 (22.7%)	9 (2.6%)
	32	6 (40.0%)	5 (33.3%)	7 (21.2%)	0 (0.0%)	18 (24.0%)	17 (4.9%)

Table 2: Solved instances by Pass@*k* setup across MathArena subcategories and PutnamBench.

5.2 RQ 2: Answer-Only Accuracy

ECP can predict more correct answers than baseline methods, resulting in fewer proof failures due to incorrectly constructed answers. Table 3 focuses only on answer construction without considering proofs. For GPT-5.4, ECP improves MathArena correct answer construction from 58/75 to 68/75 and PutnamBench answer construction from 290/346 to 303/346. Similar gains appear for the smaller open-source model Qwen3-32B, which improves from 35/75 to 39/75 on MathArena and from 201/346 to 217/346 on PutnamBench. We also record statistics for Enumerate-stage tool calls for ECP equipped with GPT-5.4. In particular, more than 50% of the instances use at least two tool calls. The number of tool calls increases with problem difficulty. For AIME problems, the easiest exam in MathArena, ECP uses about 2.5 tool calls on average, while for Apex problems, this rises to 8.2 tool calls. Overall, ECP enables solving more answer-construction problems by correctly predicting more answers, largely due to bounded enumeration.

³Source: Vast AI’s H100 pricing: <https://vast.ai/pricing/gpu/H100-SXM>

Method	Setting	High School (MathArena 2026)					Undergraduate
		AIME I (15)	AIME II (15)	HMMT (33)	Apex (12)	Total (75)	PutnamBench (346)
GPT-5.4	ECP	14	15	32	7	68	303
	Baseline	13	15	26	4	58	290
Qwen3-32B	ECP	12	12	14	1	39	217
	Baseline	12	9	13	1	35	201

Table 3: Answer-only accuracy by model and setting.

5.3 RQ 3: Cost Efficiency

Method	Dataset	Pass@ <i>k</i>	GPT-5.4 (USD)	Goedel-Prover (USD / GPU hours)	Total
GPT-5.4	MathArena	32	341.2	0.0	341.2
ECP	MathArena	32	8.6	102.0 (51.0 GPU hours)	110.6
GPT-5.4	PutnamBench	4	178.8	0.0	178.8
GPT-5.4 (Projected)	PutnamBench	32	1430.4	0.0	1430.4
ECP	PutnamBench	32	24.5	450.0 (225.0 GPU hours)	474.5

Table 4: Total cost of GPT-5.4 and Goedel-Prover-V2-32B models for ECP and GPT-5.4 baselines. For the GPT-5.4 baseline on PutnamBench (Pass@32), the cost is projected from its Pass@4 experiment result.

ECP operates at a lower cost than LLM-only baselines while achieving better results. GPT-5.4 is much more expensive than locally deployable models like Goedel-Prover-V2-32B. ECP assigns answer construction to GPT-5.4 and heavier proof generation with multiple samples to Goedel-Prover-V2-32B, taking advantage of the cost efficiency and proving performance of Goedel-Prover-V2-32B and the accurate answer construction of GPT-5.4. On MathArena at Pass@32, ECP solves 18 problems versus 13 for GPT-5.4, with a reported cost of 110.6 USD versus 341.2 USD, approximately one-third of the baseline cost. On PutnamBench (Pass@32), the GPT-5.4 baseline has a projected cost of approximately 1400 USD (estimated by multiplying its Pass@4 cost by 8), while ECP costs approximately 475 USD (which only includes 24.5 USD of OpenAI API cost) because it uses the cheaper locally deployable Goedel-Prover-V2-32B for proof generation. The PutnamBench Pass@32 comparison concerns projected cost only; GPT-5.4 accuracy at this setting was not evaluated.

5.4 RQ 4: Ablation Studies

Problem Solving. We perform ablation studies to evaluate the main design choices of ECP for answer-construction problem solving. Table 5 reports end-to-end ablations for the default ECP configuration when using a smaller prover model, disabling proof correction, and disabling proof automation. Each ablated case results in accuracy degradation in at least one benchmark. On the high-school MathArena benchmark, the main degradation comes from removing the proof automation portfolio, which causes 7 fewer solved instances, while on the more difficult undergraduate PutnamBench benchmark, switching to a weaker prover LLM causes 10 fewer solved instances. Overall, problem-solving performance depends on both the prover model selection and proof-generation techniques.

Answer Prediction. We focus on answer-only accuracy and ablate the effects of the Enumerate stage and answer reconstruction. Table 6 reports the answer-only results for the default ECP setup and the ablated cases of disabling the Enumerate stage and answer reconstruction upon disproof. The negative effect of disabling the Enumerate stage is significant: on each benchmark, there are at least 10 fewer solved instances. Answer reconstruction helps recover the answer

Variant	MathArena (75)	PutnamBench (346)
ECP (original)	18	17
- Smaller prover LLM (v1)	17 (-1)	7 (-10)
- Disable proof correction (v2)	16 (-2)	16 (-1)
- Disable proof automation (v3)	11 (-7)	17

Table 5: Ablation results for answer-construction problem solving. The red parentheses show decreases relative to ECP. Original ECP uses GPT-5.4 for the Enumerate and Conjecture stages, and Goedel-Prover-V2-32B for the Prove stage, including proof correction and automation. ECP v1 (smaller prover LLM) uses Goedel-Prover-V2-8B for the Prove stage; ECP v2 disables proof correction by setting `max_prove_rounds = 1`; ECP v3 disables the portfolio of proof automation tactics.

for 1 instance in MathArena. Therefore, the gains in answer-only accuracy mostly come from the Enumerate stage and its bounded enumeration results, while disproof-guided answer reconstruction has a smaller effect.

Variant	MathArena (75)	PutnamBench (346)
ECP (original)	68	303
- Disable Enumerate stage (v1)	58 (-10)	290 (-13)
- Disable reconstruction (v2)	67 (-1)	303

Table 6: Ablation results for answer prediction. The red parentheses show decreases relative to ECP. Original ECP uses GPT-5.4 for the Enumerate and Conjecture stages. ECP v1 disables the Enumerate stage; ECP v2 disables answer reconstruction upon a disproved substituted statement.

Admissibility Checking. We study the effects of admissibility checking in problem solving by comparing two evaluation metrics: (i) our standard metric that requires both Lean compilation of the proof and admissibility checking for the constructed answer; (ii) the naive metric that requires only Lean compilation.

Method	MathArena	PutnamBench
ECP	18+0	17+0
GPT-5.4 baseline	13+19	8+134
Goedel-Prover-V2-32B baseline	7+18	7+99

Table 7: Solved instances under two metrics: (i) the standard metric, enforcing both Lean compilation and admissibility checking; (ii) the naive metric, enforcing only Lean compilation. Each entry is $a + b$, where a denotes solved instances under the standard metric, while b denotes the number of spurious “solved” instances under the naive metric without enforcing admissibility checking.

Table 7 shows that direct LLM baselines for answer-construction problem solving are highly vulnerable to proofs with inadmissible answers: under the naive metric, GPT-5.4 and Goedel-Prover-V2-32B produce 19 and 18 additional spurious solves on MathArena, and 134 and 99 on PutnamBench, respectively. These proofs compile in Lean, but Lean compilation alone cannot detect witnesses that circularly restate the target property or otherwise fall outside the admissible vocabulary, as illustrated in Figure 2; complete examples are available in the released

artifact (Section 6). ECP avoids this failure mode by separating answer construction from proof generation, assigning them to specialized models, and checking $\text{Admissible}(A, \Psi)$ before proving the substituted statement T_Ψ . We manually inspected the instances counted under the standard metric and found that the constructed answers and their Lean proofs are nontrivial and do not rely on circular or trivial existential witnesses.

Overall, ECP improves both problem-solving accuracy and answer-only accuracy over direct LLM baselines while operating at lower cost. The ablations further show that bounded enumeration, prover LLM selection with proof correction and automation, and admissibility checking address distinct failure modes in solving answer-construction problems.

6 Conclusion

We introduced Enumerate-Conjecture-Prove (ECP), a neuro-symbolic framework for formally solving answer-construction problems in Lean. Enumerate searches bounded instances with Python tool calls, Conjecture generalizes the resulting traces into an explicit Lean answer, and Prove verifies the answer-substituted statement with automation and prover LLMs. Admissibility checking extracts the constants used by the constructed answer and rejects answers outside the problem-specific admissible vocabulary, preventing circular witnesses from being counted as solutions. Empirically, ECP improves both answer-only accuracy and end-to-end proof generation. It solves 17/346 PutnamBench answer-construction problems and 18/75 MathArena 2026 problems (including 6 AIME I problems, 5 AIME II problems, and 7 HMMT problems) end-to-end in Lean, outperforming the strongest evaluated direct proof-generation baselines at the reported budgets.

Limitations and Future Work. Currently, the admissible vocabulary is automatically labeled by heuristics that partially rely on implicit information about answer formats and the mathematical knowledge level required by the problem. Future work may extend this framework to broader mathematical domains beyond competitions and incorporate admissible-vocabulary labeling into the autoformalization procedure. Methodologically, ECP partially depends on bounded enumeration during the Enumerate stage, which can suffer from combinatorial explosion in difficult combinatorics problems. Designing specialized methods for Olympiad-level and research-level mathematical conjecture generation may be a promising direction for research.

Artifact Availability. We release the processed datasets, implementation, prompts, experiment logs, complete proofs, and lists of solved instances at <https://github.com/sunjia72/ecp-lpar>. The data are in `data/dataset/`, the implementation and prompts are in `src/`, and the experiment logs and proofs are in `output_artifact_eval/`.

Acknowledgements. We thank anonymous reviewers of LPAR conference for their constructive feedback. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) [RGPIN-2024-05956] and Ontario Graduate Scholarship program. Computations were performed on the Killarney Cluster, located at the University of Toronto and is managed by the Vector Institute and SciNet.

References

- [1] Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathis Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yuri Kudryashov, Alexander Meiburg, et al. Aristotle: IMO-Level Automated Theorem Proving. *arXiv preprint arXiv:2510.01346*, 2025.
- [2] Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics. *arXiv preprint arXiv:2302.12433*, 2023.
- [3] Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. MathArena: Evaluating LLMs on Uncontaminated Math Competitions, February 2025.
- [4] ByteDance. SandboxFusion. <https://github.com/bytedance/SandboxFusion>, 2024. GitHub repository.
- [5] Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. Seed-Prover: Deep and Broad Reasoning for Automated Theorem Proving. *arXiv preprint arXiv:2507.23726*, 2025.
- [6] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.
- [7] Edmund M Clarke, Anubhav Gupta, and Ofer Strichman. Sat-based counterexample-guided abstraction refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(7):1113–1123, 2004.
- [8] George E Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition: a synopsis. *ACM SIGSAM Bulletin*, 10(1):10–12, 1976.
- [9] James H Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1-2):29–35, 1988.
- [10] Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. Complexity-based prompting for multi-step reasoning, 2023.
- [11] Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, pages 1–3, 2025.
- [12] Zhaoyu Li, Hangrui Bi, Jialiang Sun, Zenan Li, Kaiyu Yang, and Xujie Si. Pyeuclid: A versatile formal plane geometry system in python. In *International Conference on Computer Aided Verification*, pages 405–420. Springer, 2025.
- [13] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. A survey on deep learning for theorem proving, 2024.
- [14] Jannis Limperg and Asta Halkjær From. Aesop: White-Box Best-First Proof Search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 253–266, 2023.

- [15] 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. In *The Fourteenth International Conference on Learning Representations*, 2026.
- [16] Chengwu Liu, Jianhao Shen, Huajian Xin, Zhengying Liu, Ye Yuan, Haiming Wang, Wei Ju, Chuanyang Zheng, Yichun Yin, Lin Li, et al. FIMO: A Challenge Formal Dataset for Automated Theorem Proving. *arXiv preprint arXiv:2309.04295*, 2023.
- [17] Junqi Liu, Xiaohan Lin, Jonas Bayer, Yael Dillies, Weijie Jiang, Xiaodan Liang, Roman Soletskyi, Haiming Wang, Yunzhou Xie, Beibei Xiong, Zhengfeng Yang, Jujian Zhang, Lihong Zhi, Jia Li, and Zhengying Liu. CombiBench: Benchmarking LLM Capability for Combinatorial Mathematics, 2025.
- [18] Qi Liu, Xinhao Zheng, Renqiu Xia, Xingzhi Qi, Qinxiang Cao, and Junchi Yan. Beyond theorem proving: Formulation, framework and benchmark for formal problem-solving. *arXiv preprint arXiv:2505.04528*, 2025.
- [19] The mathlib Community. The Lean Mathematical Library. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020.
- [20] Abdalrhman Mohamed. Lean-SMT: Tactics for Discharging Lean Goals into SMT Solvers, 2025.
- [21] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Proceedings of the International Conference on Automated Deduction*, 2021.
- [22] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.
- [23] Lawrence C Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994.
- [24] George Polya. How to solve it: A new aspect of mathematical method. In *How to solve it*. Princeton university press, 2014.
- [25] Yicheng Qian, Joshua Clune, Clark Barrett, and Jeremy Avigad. Lean-auto: An Interface Between Lean 4 and Automated Theorem Provers. In *International Conference on Computer Aided Verification*, pages 175–196. Springer, 2025.
- [26] ZZ Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanbiao Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, et al. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. *arXiv preprint arXiv:2504.21801*, 2025.
- [27] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. OpenAI GPT-5 System Card. *arXiv preprint arXiv:2601.03267*, 2025.
- [28] Chendong Song, Zihan Wang, Frederick Pu, Haiming Wang, Xiaohan Lin, Junqi Liu, Jia Li, and Zhengying Liu. LeanGeo: Formalizing Computational Geometry Problems in Lean. *arXiv preprint arXiv:2508.14644*, 2025.

- [29] Nat Sothanaphan. Resolution of Erdős Problem #728: A Writeup of Aristotle’s Lean Proof. *arXiv preprint arXiv:2601.07421*, 2026.
- [30] Anja Surina, Arun Suggala, George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Francisco JR Ruiz, Pushmeet Kohli, and Swarat Chaudhuri. An improved last-iterate convergence rate for anchored gradient descent ascent. *arXiv preprint arXiv:2604.03782*, 2026.
- [31] Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. Solving Olympiad Geometry without Human Demonstrations. *Nature*, 2024.
- [32] George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, Gergely Bérczi, Francisco JR Ruiz, Arun Suggala, Adam Zsolt Wagner, Eric Wieser, et al. Advancing mathematics research with ai-driven formal proof search. *arXiv preprint arXiv:2605.22763*, 2026.
- [33] George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. PutnamBench: Evaluating Neural Theorem-Provers on the Putnam Mathematical Competition. *arXiv preprint arXiv:2407.11214*, 2024.
- [34] Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.
- [35] Chenrui Wei, Mengzhou Sun, and Wei Wang. Proving olympiad algebraic inequalities without human demonstrations. *Advances in Neural Information Processing Systems*, 37:82811–82822, 2024.
- [36] Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-Prover: Advancing Theorem Proving in LLMs through Large-Scale Synthetic Data. *arXiv preprint arXiv:2405.14333*, 2024.
- [37] Huajian Xin, ZZ Ren, Junxiao Song, Zhihong Shao, Wanbiao Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiusi Du, et al. DeepSeek-Prover-V1.5: Harnessing Proof Assistant Feedback for Reinforcement Learning and Monte-Carlo Tree Search. *arXiv preprint arXiv:2408.08152*, 2024.
- [38] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388*, 2025.
- [39] Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075*, 2024.
- [40] Xiao-Wen Yang, Zihao Zhang, Jianuo Cao, Zhi Zhou, Zenan Li, Lan-Zhe Guo, Yuan Yao, Taolue Chen, Yu-Feng Li, and Xiaoxing Ma. Formalml: A benchmark for evaluating formal subgoal completion in machine learning theory, 2025.
- [41] Huaiyuan Ying, Zijian Wu, Yihan Geng, Jiayu Wang, Dahua Lin, and Kai Chen. Lean Workbook: A Large-Scale Lean Problem Set Formalized from Natural Language Math Problems. *arXiv preprint arXiv:2406.03847*, 2024.

- [42] Huaiyuan Ying, Shuo Zhang, Linyang Li, Zhejian Zhou, Yunfan Shao, Zhaoye Fei, Yichuan Ma, Jiawei Hong, Kuikun Liu, Ziyi Wang, et al. InternLM-Math: Open Math Large Language Models Toward Verifiable Reasoning. *arXiv preprint arXiv:2402.06332*, 2024.
- [43] Zhouliang Yu, Ruotian Peng, Keyi Ding, Yizhe Li, Zhongyuan Peng, Minghao Liu, Yifan Zhang, Zheng Yuan, Huajian Xin, Wenhao Huang, Yandong Wen, Ge Zhang, and Weiyang Liu. Formalmath: Benchmarking formal mathematical reasoning of large language models, 2025.
- [44] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. MiniF2F: A Cross-System Benchmark for Formal Olympiad-Level Mathematics. In *Proceedings of the International Conference on Learning Representations*, 2022.