ByteDance | Seed

Solving Formal Math Problems by Decomposition and Iterative Reflection

Yichi Zhou[1†], **Jianqiu Zhao**[1], **Yongxin Zhang**[1], **Bohan Wang**[2,*], **Siran Wang**[1], **Luoxin Chen**[1], **Jiahui Wang**[1], **Haowei Chen**[1], **Allan Jie**[1], **Xinbo Zhang**[1], **Haocheng Wang**[1], **Luong Trung**[1], **Rong Ye**[1], **Phan Nhat Hoang**[1], **Huishuai Zhang**[3], **Peng Sun**[1,†], **Hang Li**[1,†]

[1]ByteDance Seed, [2]University of science and technology of China, [3]Peking University

[*]Work done at ByteDance Seed, [†]Corresponding authors

Abstract

General-purpose Large Language Models (LLMs) have achieved remarkable success in intelligence, performing comparably to human experts on complex reasoning tasks such as coding and mathematical reasoning. However, generating formal proofs in specialized languages like Lean 4 remains a significant challenge for these models, limiting their application in complex theorem proving and automated verification. Current approaches typically require specializing models through fine-tuning on dedicated formal corpora, incurring high costs for data collection and training. In this work, we introduce **Delta Prover**, an agent-based framework that orchestrates the interaction between a general-purpose LLM and the Lean 4 proof environment. Delta Prover leverages the reflection and reasoning capabilities of general-purpose LLMs to interactively construct formal proofs in Lean 4, circumventing the need for model specialization. At its core, the agent integrates two novel, interdependent components: an algorithmic framework for reflective decomposition and iterative proof repair, and a custom Domain-Specific Language (DSL) built upon Lean 4 for streamlined subproblem management. **Delta Prover achieves a state-of-the-art 95.9% success rate on the miniF2F-test benchmark, surpassing all existing approaches, including those requiring model specialization.** Furthermore, Delta Prover exhibits a significantly stronger test-time scaling law compared to standard Best-of-N proof strategies. Crucially, our findings demonstrate that general-purpose LLMs, when guided by an effective agentic structure, possess substantial untapped theorem-proving capabilities. This presents a computationally efficient alternative to specialized models for robust automated reasoning in formal environments.

Date: July 20, 2025

Correspondence: Yichi Zhou at zhouyichi.123@bytedance.com, Peng Sun at wanhesong@bytedance.com, Hang Li at lihang.lh@bytedance.com

Project Page: https://github.com/ByteDance-Seed/lean4-agent

1 Introduction

Recent large language models (LLMs) have demonstrated remarkable progress in intellectually demanding tasks, such as solving math word problems [14], code generation [5], and planning [27]. Central to this

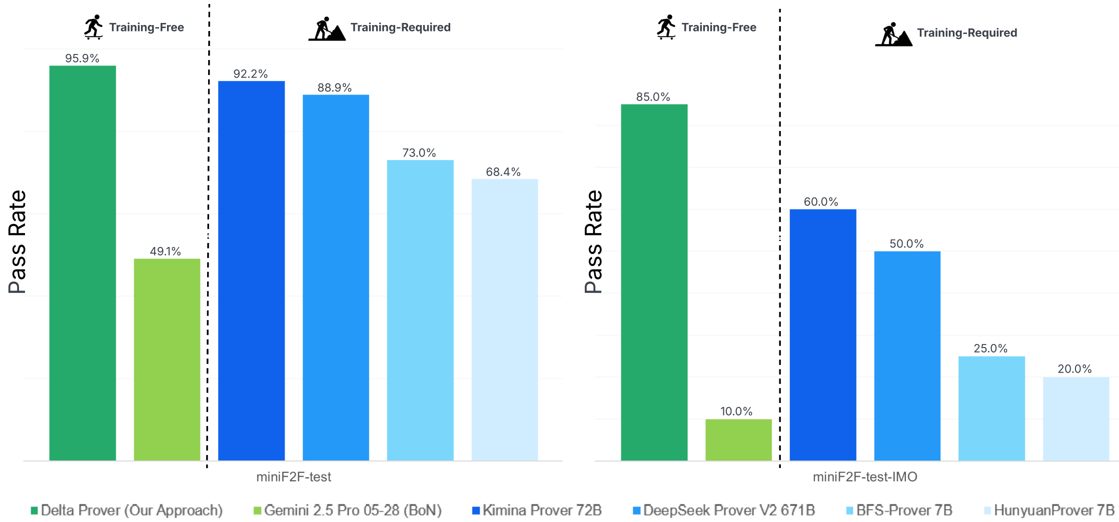


Figure 1 Performance comparison between Delta Prover (our approach) and existing works. Our approach achieves the state-of-the-art performance over the miniF2F-test and the miniF2F-test-IMO benchmarks without additional data collection and fine-tuning, using the stock Gemini 2.5 Pro 05-28 model as a backbone. "Training-Free" and "Training-Required" stand for whether the model needs to be additionally fine-tuned over formal datasets.

advancement is the emergence of reasoning and reflection capabilities within these models. These capabilities enable LLMs to iteratively refine their approach, moving closer to the correct solution by building upon their previous thought trajectories [3, 7, 29].

Proving complex mathematical theorems, which involves long chains of mathematical computation and logical deduction, is widely regarded as a pinnacle of human intellect [2, 28]. Consequently, this task has been widely adopted to benchmark the reasoning ability of LLMs [23, 58]. A common approach for LLMs tackling this task is to emulate human reasoning by generating proofs in natural language. However, verifying the correctness of proofs generated in this manner poses a significant challenge – identifying logical fallacies or subtle errors in natural language mathematical arguments can be as difficult as constructing the proof itself [20]. This verification difficulty is exacerbated by the prevalence of outcome-based reward modeling (ORM) in training state-of-the-art LLMs [3, 29]. ORM necessitates reliable verification not just for final benchmarking, but extensively throughout the training process to generate reward signals, demanding significantly more verification effort compared to simple benchmarking runs.

Formal theorem proving systems, such as Lean [26], Isabelle [13], and Coq [37], have been leveraged to address the aforementioned verification challenge. Specifically, these systems enable mathematicians to write proofs in a formal language. Corresponding proof assistants or kernels can then automatically verify the correctness of these formal proofs, often by checking if the associated code compiles or executes without error. Such an approach renders the verification process automatic and aligns well with the outcome-based reward modeling methods mentioned above, offering a scalable solution to the verification bottleneck.

However, validating proofs generated by LLMs using formal environments also presents challenges. The primary difficulty lies in effectively teaching LLMs to generate code in these formal languages. As languages specifically designed for proof verification, formal languages often exhibit syntactic structures and paradigms that are significantly different from common programming languages like C++ or Python, making them effectively 'out-of-domain' for models primarily trained on general code. Furthermore, the distribution of formal language data within large code corpora is typically sparse. For example, LEAN-GitHub [48], currently the largest Lean 4 codebase, contains only 0.13B tokens, representing a minuscule fraction 0.1% of the typical code corpora [11, 59], which contain around hundreds of billions of tokens. Consequently, general-purpose LLMs, such as DeepSeek R1 [3], Gemini 2.5 Pro [7], and OpenAI o3 [29], often exhibit limited proficiency in formal languages due to this data scarcity [41].

Moreover, developing specialized models via fine-tuning presents its own hurdles. Acquiring high-quality labeled data for formal theorem proving is expensive and challenging, as it requires annotators with rare expertise in both advanced mathematics and the specific formal language, and the volume of data needed can be substantial. This is compounded by the significant computational cost associated with training or fine-tuning large-scale models [33, 41].

Our Contribution. In contrast to approaches demanding extensive data labeling or task-specific fine-tuning, we explore a novel framework that leverages general-purpose LLMs [3, 7, 29] and a formal proof assistant environment using Lean 4. We propose an agent-based framework, Delta Prover[1], designed to harness the inherent reasoning and reflection capabilities of LLMs through effective orchestration, thereby eliminating the need for bespoke training data or model adaptation.

Delta Prover employs an LLM-in-the-loop agent that actively participates in the theorem-proving process. By iteratively interacting with the Lean 4 environment, this agent transforms the LLM from a passive generator into a dynamic partner in formal reasoning. The agent has two primary components: a novel algorithmic framework and a specialized Lean 4 syntax. The syntax provides the scaffolding for the entire process, managing task decomposition, storing the state of sub-problems, and assembling solved steps into a final, verified proof. The framework, in turn, is built upon two core mechanisms: (a) **Reflective Decomposition**: This leverages the LLM's capacity for high-level strategy to break down complex theorems into simpler sub-problems. Crucially, the agent can reflect on failed decompositions to revise its approach. (b) **Iterative Proof Repair**: This creates a tight feedback loop where errors from the Lean 4 verifier directly guide the LLM's next attempt, a process augmented by automated theorem retrieval to find relevant lemmas. Together, these elements enable the agent to systematically tackle complex proofs, learn from its mistakes, and produce fully machine-verifiable results in Lean 4.

Building on the techniques previously described, **we empirically prove that general-purpose LLMs, when employed within a prover agent, can achieve remarkable theorem-proving prowess**, effectively challenging the established dominance of bespoke models in the field. Specifically:

- As demonstrated in Figure 1, Delta Prover achieves a 95.9% success rate on the miniF2F-test benchmark, establishing a new state-of-the-art that surpasses existing provers, including those specifically fine-tuned. Notably, on the more challenging IMO problems within miniF2F-test, Delta Prover attains an 85% success rate, exceeding the previous state-of-the-art [33].
- Our ablation studies reveal that Delta Prover possesses significantly stronger test-time scaling compared to the widely applied best-of-N sampling strategy.

This agent-centric methodology not only unlocks more potent and adaptable automated reasoning capabilities but also markedly curtails the effort involved in data collection and fine-tuning bespoke models. Furthermore, our findings offer valuable insights into harnessing LLM reflection and reasoning to construct highly capable agents.

2 Approach

General-purpose Large Language Models (LLMs) demonstrate impressive capabilities when faced with natural language mathematical problems. However, their direct application to formal theorem proving faces two significant drawbacks that limit their effectiveness: first, formal proofs demand complete correctness, a stringent requirement often unmet as the probability of LLM-generated errors tends to grow exponentially with the proof length; second, current approaches often deploy LLMs to sample proof steps or entire proofs in parallel [33, 41], but the (conditionally) independent nature of these samples limits effective scaling and efficient discovery of a complete, valid proof.

To address these issues, we introduce an agent-driven Lean 4 prover, Reflection Agent Prover (Delta Prover), which operates in an iterative loop and seamlessly integrates a general-purpose LLM with the Lean 4 environment. As illustrated in Figure 2, our prover builds upon the emerged reflection and reasoning ability

[1] "Delta" stands for "**De**composition and iterative ref**l**ec**t**ion **a**gent"

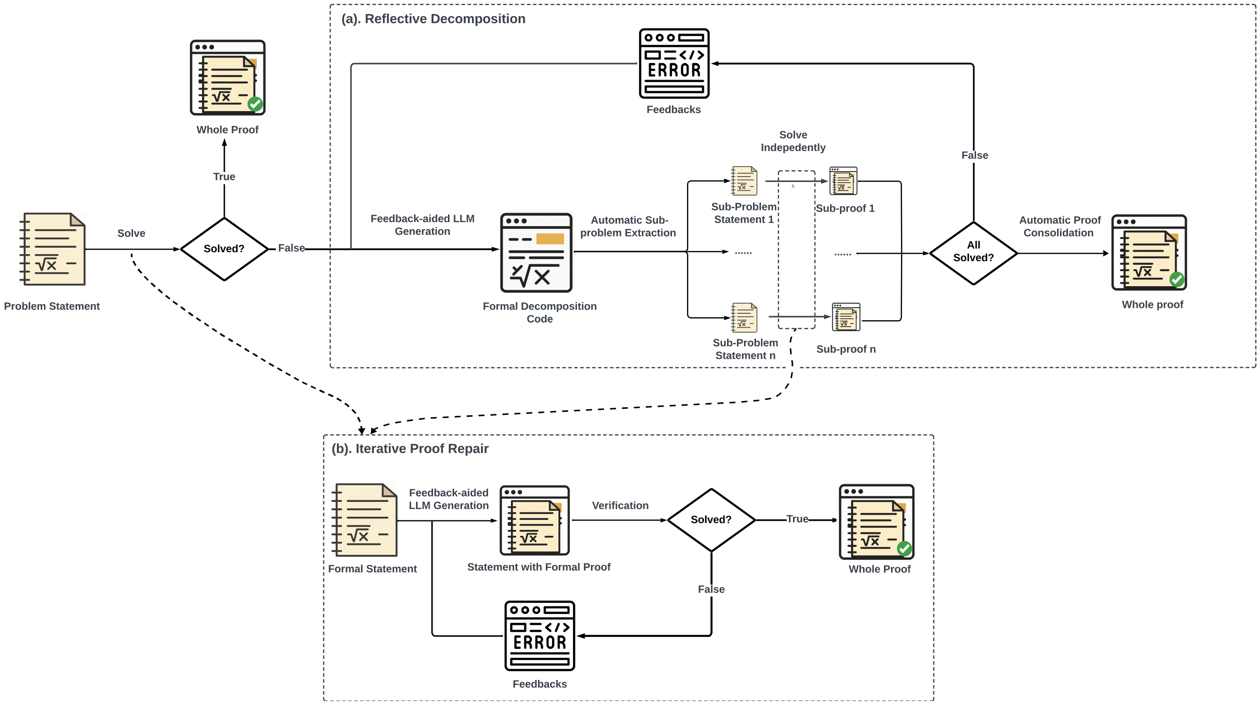


Figure 2 An overview of our agent-driven prover framework. The system features the interplay between (a). a Reflective Decomposition process and (b). an iterative proof repair strategy.

of recent general-purpose LLMs, and relies on two key pillars designed to overcome the aforementioned challenges:

- **Iterative Proof Repair:** The LLM repeatedly attempts formal proof steps, progressively refining its output based on textual feedback from the Lean 4 environment until a complete and verified proof is achieved. Should an attempt fail, Delta Prover provides the LLM with diagnostic information from Lean 4, alongside retrieved relevant tactics and theorems, to guide subsequent iterations.
- **Reflective Decomposition:** For complex problems, the LLM initially formulates a high-level proof sketch using a novel Domain-Specific Language (DSL, detailed below), guided by a natural language outline. Delta Prover then systematically decomposes this formal sketch into sub-problems via the DSL. Each sub-problem is addressed using the iterative proof repair mechanism. Should a sub-problem prove intractable, the LLM reflects on the decomposition, revising it or devising alternative proof strategies.

To support this framework, we leverage a **Decomposition via Extended Syntax** methodology. This involves a custom Domain-Specific Language (DSL) built atop Lean 4, enabling the LLM to create and manage subgoals annotated with proof placeholders. Once all subgoals are successfully proven, the DSL automatically integrates these intermediate proofs to construct the complete formal proof for the original goal.

The remainder of this section is organized as follows: First, we elaborate on Delta Prover's two main components: the Iterative Proof Repair loop and the Reflective Decomposition process, and describe how these approaches are deployed. We then detail the extended DSL facilitating goal management within Lean 4.

2.1 Algorithmic Framework

This section details the workings of Delta Prover. To facilitate clear descriptions in the subsequent text, we first establish the following notation. Let S denote a formal statement and s its corresponding natural language version. A formal proof of S is denoted by P. For the decomposition stage, p represents an informal proof (or proof plan), and D signifies a formal proof sketch expressed in our domain-specific language (DSL, introduced in Section 2.2).

2.1.1 Iterative Proof Repair

Current approaches [33, 41, 49] typically parallelize Large Language Model (LLM) rollouts, hoping that with a sufficient number of samples, independent exploration will eventually discover a valid solution. This strategy, however, often leads to isolated trajectories, fails to fully leverage the capacity for reflection and reasoning, and results in an unsatisfactory test-time scaling law as demonstrated by recent reasoning-focused models [3, 7, 29]. Recognizing this limitation, our framework incorporates an iterative error correction process as a core component, with the corresponding pseudocode provided in Algorithm 1. Specifically, given a formal statement S and its informal counterpart s, this process proceeds as follows:

Initial Solution Generation. The LLM is prompted with the formal statement S to generate an initial formal proof candidate, P^0. Recognizing that general-purpose LLMs may lack proficiency in Lean 4 syntax and tactics, we provide guidelines demonstrating:

- **Formatting Conventions:** LLMs often demonstrate inconsistent application of Lean 4's code formatting rules, such as the correct syntax following various tactics (e.g., `rcases`, `cases`). We provide explicit instructions on proper formatting.
- **Effective Tactics:** We provide the doc-string of powerful Mathlib tactics like `linarith`, `ring`, and `omega`. These tactics automate significant reasoning steps within Lean 4, and providing explicit examples helps the LLM understand their appropriate application scope.
- **Lean 4 Specification:** LLMs frequently generate Lean 3 code when prompted for Lean proofs if not explicitly instructed otherwise. We conjecture that this is due to the relatively recent community transition from Lean 3 to Lean 4, and thus many Lean 3 codes remain in the knowledge base of LLMs. This Lean 3 code is incompatible with Lean 4 verification. Therefore, we emphasize in the prompt that the generated proof must adhere strictly to Lean 4 syntax and libraries.

These guidelines aim to improve the quality and correctness of the initial proof P^0. If the statement-proof pair (S, P^0) successfully validates against the Lean 4 kernel, the process terminates successfully for this problem. Otherwise, we proceed to the correction phase.

Feedback Augmented Proof Repair. If a proof attempt P^i fails validation, the Lean 4 kernel provides error messages indicating the location and type of the first error, sometimes suggesting potential fixes. While useful, this message alone often lacks sufficient context for effective correction. Therefore, we augment the prompt for the next iteration $(i + 1)$ with additional information:

- **Failed Proof:** The incorrect proof P^i.
- **Message from Lean 4 kernel:** The tactic state and Lean 4 Error Message from Lean 4 kernel.
- **Retrieved Information:** We observe that LLMs may know the informal step but struggle to find the exact name and signature of a formal theorem or tactic from Mathlib4, which is a large and complex library. If the error involves an unknown or incorrect identifier, we retrieve relevant theorems/definitions based on the errored name and include them in the prompt.

This comprehensive prompt (please refer to the concrete template in Figure 8) is fed to the LLM to generate a revised proof, P^{i+1}. If (S, P^{i+1}) validates, the process succeeds. Otherwise, the fixup cycle repeats. If a valid proof is not found after a predetermined maximum number of iterations, this direct approach is deemed unsuccessful.

2.1.2 Reflective Decomposition

The iterative error correction mechanism effectively leverages the reflective capabilities of LLMs to derive correct proofs. However, as problem difficulty escalates, the complexity of corresponding proofs also increases, potentially leading to an exponential accumulation of errors throughout the reasoning process. Consequently, even with this error correction, LLMs may still require an excessive number of trials to converge on a correct proof.

Algorithm 1 Iterative Proof Repair

```
Require: Formal statement $S$; Informal statement $s$; LLM $\Phi$; Retrieval model $\Psi$; Number of rounds $m$; Number of iterative repair per round $n$, Lean 4 Kernel & DSL Environment $\mathcal{L}$
Ensure: A formal proof $P$ or Failure
Let $I_{fp}$ be the instruction for formal proof.
if $s$ is None then
    $s \leftarrow \Phi(S)$                                  ▷ Generate the informal statement $s$ if not provided
end if
for $i \leftarrow 0$ to $m - 1$ do                          ▷ Loop for $m$ rounds
    $P \leftarrow \Phi(S, s, I_{fp})$                        ▷ Generate initial proof candidate
    for $j \leftarrow 0$ to $n - 1$ do                      ▷ Loop for $n$ iterative repairs
        $kernelOutput \leftarrow \mathcal{L}(S, P)$          ▷ Verify $P$ using Lean 4 kernel
        if $kernelOutput$ contains no error then
            return $P$                                       ▷ Successfully verified proof found
        else
            $retrievedInfo \leftarrow \Psi(S, s, P, kernelOutput)$      ▷ Retrieve theorems/definitions
            $P \leftarrow \Phi(S, s, I_{fp}, kernelOutput, retrievedInfo)$   ▷ Repair proof based on error and retrieved information
        end if
    end for
end for
return Failure                    ▷ No verifiable proof found after $m$ rounds and $n$ repairs per round
```

A typical approach to resolve this challenge is Draft, Sketch, and Prove (DSP) introduced in [15], which breaks down the original problem into easier sub-problems, and solves sub-problems instead. While this approach appears straightforward and standard, its implementation presents several challenges:

- **Management of sub-problems.** DSP uses the formal language Isabella, while current Lean 4 lack robust support for managing decomposed sub-problems. Specifically, there is no convenient way to simultaneously: (i) store decomposed sub-problems; (ii) extract them as formal statements; and (iii) integrate their individual proofs back into a unified whole proof. For instance, using the `have` tactic is inadequate for (iii) [33], while listing sub-problems as lemmas falls short for (ii).
- **Proper decomposition.** LLMs typically can not precisely estimate to which extent of difficulty the sub-problem can be solved, and if the decomposition is not proper, some decomposed sub-problems will again become the proof bottleneck.

To address the first challenge, sub-problem management, we introduce a Domain-Specific Language (DSL) for problem decomposition, detailed in Section 2.2. For now, it suffices to note that this DSL enhances sub-problem management. Our framework design primarily tackles the second challenge: proper decomposition. This process involves: (1) generating an informal proof plan and translating it into a formal sketch using our DSL; (2) extracting sub-problems via the DSL and attempting to solve each independently with the Iterative Proof Repair loop; and (3) if any sub-problems remain unsolved, prompting the LLM to regenerate the formal sketch, informed by the list of unsolved sub-problems. These steps are illustrated below, with corresponding pseudocode in Algorithm 2.

Initial Formal Sketch Construction. We first construct the informal proof plan p by prompting the LLM with the statements (S, s) and specific guidance to generate p in a step-by-step format, conducive to later translation into a formal sketch (please refer to Figure 7 for the prompt template).

Given the statements (S, s) and the informal proof plan p, the LLM is then tasked with generating a corresponding formal proof sketch D in our custom Domain-Specific Language (DSL) (detailed in Section 2.2). Our DSL is intentionally designed so that the structure of the formal proof sketch D closely mirrors that of the informal plan p. Consequently, this generation process is analogous to the auto-formalization task [46]. To guide the LLM towards accurate formalization, we incorporate the following elements into the prompt (see a concrete template in Figure 6):

- **DSL Format Guidelines:** Since apparently our DSL is not part of standard LLM pre-training data, directly prompting for code generation in our DSL would likely lead to errors. We mitigate this by providing in-context examples that illustrate the syntax and usage of each DSL construct, with particular emphasis on the custom tactics introduced in Section 2.2.
- **Auto-formalization Pitfalls:** The translation from informal mathematical language to formal statements can harbor subtle traps, such as implicit assumptions or incorrect quantifier scope. To address these, the prompt explicitly highlights common pitfalls and includes examples of correct formalization.

The final output of this step is the formal proof sketch D.

Sub-problem Extraction and Solving. Following the methodology detailed in Section 2.2 and illustrated in the "Subproblem extraction and proving" segment of Figure 3, our Domain-Specific Language (DSL) employs its `show ... by` tactic to parse the initial proof draft D. This tactic automatically extracts formal statements for all required sub-proofs. The outcome is either a collection of sub-problems, $S_1, \ldots, S_n$, or the classification of D as an illegal formal draft. Valid sub-problems are then passed to the Iterative Proof Repair loop.

Iterative Decomposition Repair. Should any sub-problems prove unsolvable by the Iterative Proof Repair loop, these failures are logged. This feedback, encompassing the list of unsolved sub-problems, directs the LLM to regenerate the proof sketch D with an improved decomposition strategy (e.g., a more appropriate "grid"). This repair loop continues until all sub-problems are solved or a maximum number of decomposition attempts is exhausted.

2.1.3 The Overall Framework

Building upon the previously detailed Iterative Proof Repair loop (Algorithm 1) and Reflective Decomposition approach (Algorithm 2), we now describe their integration into our final framework, Delta Prover.

When presented with a problem, defined by a statement pair (S, s), Delta Prover initially attempts a direct solution using through Iterative Proof Repair. If this direct approach is unsuccessful, the framework then employs Reflective Decomposition, which returns either `Failure` or a list of (sub-problem, proof) pairs. In the former case, Delta Prover terminates and returns **Failure**. In the latter, Delta Prover performs **Automatic Proof Consolidation** (illustrated in the "Proof consolidation" section of Figure 3). During this consolidation stage, the derived sub-proofs P_i are substituted back into the DSL-based sketch. Subsequently, necessary concluding commands (e.g., a `conclude` tactic), as dictated by the DSL's structure, are appended to produce the complete formal proof P for the original statement S.

The pseudocode for this comprehensive framework is provided in Algorithm 3.

2.2 Decomposition via Extended Syntax

The previous section introduced our use of a Domain-Specific Language (DSL) to address challenges in managing sub-problems within Lean 4. We noted that while native Lean 4 syntax is powerful for low-level tactics, it is less suited for high-level proof sketching, isolating subproblem contexts, and automatically reintegrating subproofs. This section delves into the details of our DSL.

Our DSL is implemented by leveraging Lean 4's metaprogramming facilities. Specifically, we introduce an additional monad layer, `PlayM`, built upon `TacticM`, to record and manage intermediate proof states. These saved states are subsequently converted into complete and valid Lean 4 proof scripts using Lean 4's built-in delaborator.

To facilitate more effective decomposition, we have implemented four specialized tactics in PlayM. Figure 3 illustrates syntaxes of these tactics and how these tactics are employed to systematically split a theorem into subproblems and subsequently reassemble the solutions into a coherent proof. In a nutshell, our DSL provides four key capabilities:

Algorithm 2 `Reflective Decomposition`

```
Require: Formal statement $S$; Informal statement $s$; LLM $\Phi$; Retrieval model $\Psi$; Lean 4 Kernel & DSL Environment $\mathcal{L}$; Max decomposition attempts $l_{max}$; Sub-problem solving rounds $\tilde{m}$; Sub-problem repair iterations $\tilde{n}$
Ensure: A list of (sub-problem, proof) pairs: $((S'_1, p'_1), \ldots, (S'_k, p'_k))$, or Failure
Let $I_{ip}$ be the instruction template for informal proof generation
Let $I_{fs}$ be the instruction template for formal sketch generation
$p_{informal} \leftarrow \Phi(S, s, I_{ip})$    ▷ Generate initial informal proof sketch
$D \leftarrow \Phi(S, s, p_{informal}, I_{fs})$    ▷ Generate initial formal sketch from $p_{informal}$
for $i \leftarrow 1$ to $l_{max}$ do
    $kernelOutput \leftarrow \mathcal{L}(D)$    ▷ Verify formal sketch $D$; $kernelOutput$ contains errors or sub-problems
    if $kernelOutput$ indicates a kernel-level error then
        $E \leftarrow kernelOutput$    ▷ Feedback $E$ is the set of kernel errors
    else
        $(S'_1, \ldots, S'_k) \leftarrow kernelOutput$    ▷ Extract sub-problems
        $UnsolvedSubproblems \leftarrow [\ ]$
        $SolvedProofs \leftarrow [\ ]$
        for $j \leftarrow 1$ to $k$ do    ▷ Attempt to solve each sub-problem
            $P'_j \leftarrow$ Iterative Proof Repair$(S'_j, S, \texttt{None}, \Phi, \Psi, \mathcal{L}, \tilde{m}, \tilde{n})$    ▷ $S$ is overall context, None for initial sub-proof attempt
            if $P'_j =$ Failure then
                Add $S'_j$ to $UnsolvedSubproblems$
            else
                Add $(S'_j, P'_j)$ to $SolvedProofs$
            end if
        end for
        if $UnsolvedSubproblems$ is empty then
            return $SolvedProofs$    ▷ All sub-problems solved
        else
            $E \leftarrow UnsolvedSubproblems$    ▷ Feedback $E$ is the list of unsolved sub-problems
        end if
    end if
    if $i < l_{max}$ then    ▷ Refine $D$ if not the last attempt
        $D \leftarrow \Phi(S, s, D, E)$    ▷ Regenerate formal sketch $D$ using feedback $E$
    end if
end for
return Failure    ▷ Maximum decomposition attempts reached or all failed
```

Algorithm 3 The Framework of Delta Prover

```
Require: Formal statement $S$; Informal statement $s$; LLM $\Phi$; Retrieval model $\Psi$; Lean 4 Kernel & DSL
    Environment $\mathcal{L}$; Maximum decomposition attempts $l_{\max}$; Rounds for direct proof attempt $m$; Iterative
    repairs per round (for direct proof) $n$; Rounds for sub-problem proofs $\tilde{m}$; Iterative repairs per round (for
    sub-problems) $\tilde{n}$
Ensure: A formal proof $P$ for $S$, or Failure
$P \leftarrow$ Iterative Proof Repair$(S, \text{None}, \Phi, \Psi, \mathcal{L}, m, n)$        ▷ Attempt direct proof of $S$
if $P$ is not Failure then
    return $P$
end if
                                          ▷ Direct proof failed; attempt Reflective Decomposition
$decompResult \leftarrow$ Reflective Decomposition$(S, s, \Phi, \Psi, \mathcal{L}, l_{\max}, \tilde{m}, \tilde{n})$    ▷ $decompResult$ is a list of
    (sub-problem, proof) pairs or Failure
if $decompResult$ is not Failure then                              ▷ Automatic Proof Consolidation Stage
    $P_{\text{consolidated}} \leftarrow \mathcal{L}(decompResult)$          ▷ Consolidate sub-proofs with the DSL sketch via $\mathcal{L}$
    if $P_{\text{consolidated}}$ is not Failure then
        return $P_{\text{consolidated}}$
    end if
end if
return Failure                                                       ▷ All approaches failed
```

1. Tactic **Suppose** for Hypothesis Introduction: we use this tactic to introduce new hypotheses into the proof context. It allows the assumption of arbitrary Lean 4 types as premises without the need to explicitly specify their values.

2. Tactic **Define** for Arbitrary Expression Construction: The define tactic enables the introduction of arbitrary Lean 4 expressions, with playm automatically inferring their types. For example, one can use define to declare a new function from integer to integer or a Finset over natural numbers.

3. Tactic **ShowBy** for Subgoal Creation: The ShowBy tactic provides an interface between PlayM and TacticM for leveraging well-developed tactics from Mathlib. It allows us to explicitly pose subproblems. It functions similarly to the native Lean 4 showby tactic, with the key difference being that our implementation records the proofs generated within it. These recorded proofs are then utilized later for assembling a complete, integrated proof.

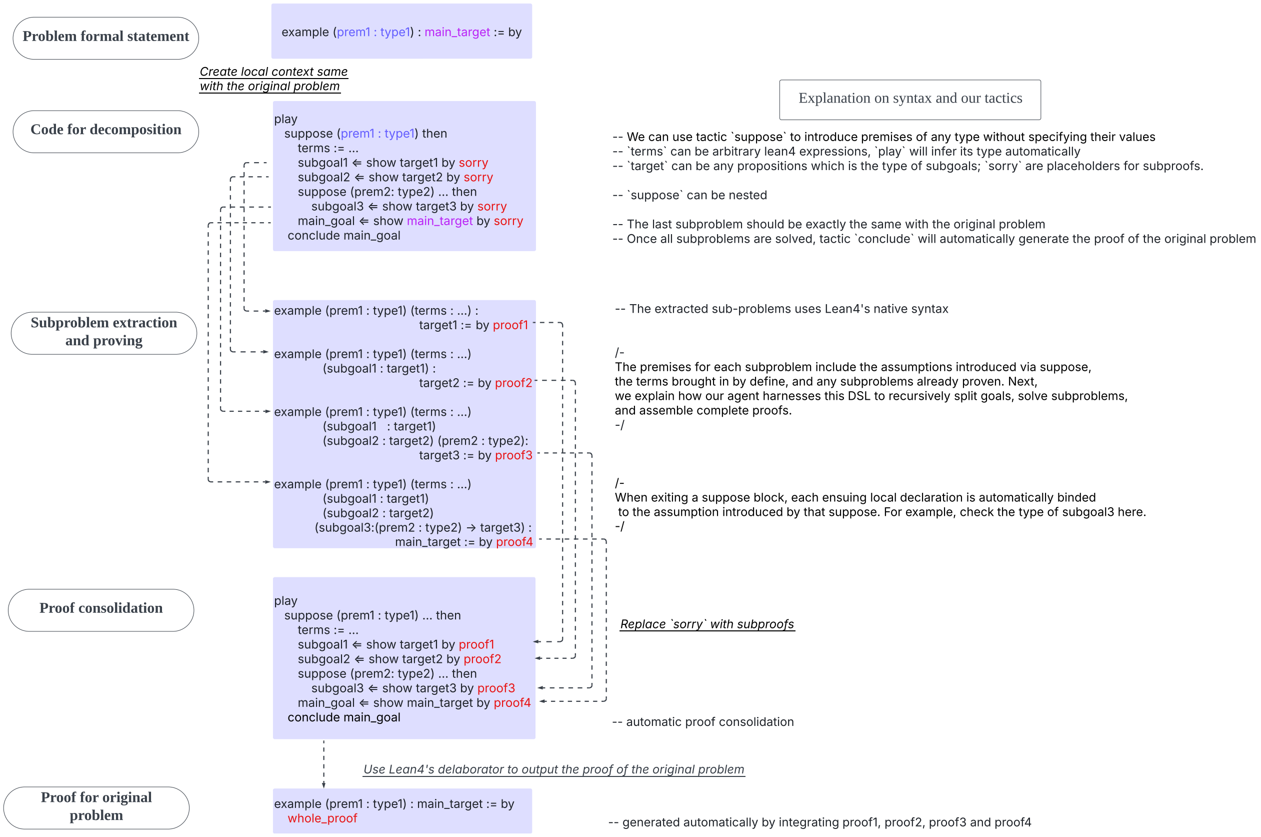
4. Tactic **Conclude** for Proof Consolidation: The conclude tactic is responsible for proof consolidation: within PlayM, every subproblem's proof steps and dependency graph are recorded, and conclude tactic leverages these saved proofs—together with Lean 4's delaborator to emit a fully coherent Lean 4 proof for the original problem.

Our extended DSL streamlines the workflow of Delta Prover. However, its specialized nature means general-purpose LLMs may lack familiarity and struggle to generate appropriate code. To overcome this, we meticulously designed prompts to guide the LLM in using the DSL effectively for both problem decomposition and solving, as elaborated in Section 2.1.2.

3 Related Work

Specialized LLMs for Formal Theorem Proving. Developing specialized models for formal theorem proving by training on large, proof-focused datasets has been a major research direction. Early efforts [1, 10, 12, 24, 32, 35, 43–45, 52] explored diverse tasks and formalisms, but were often hampered by limitations in model scale, architecture, and data availability, restricting their performance or scope, sometimes only addressing sub-tasks like premise selection.

A turning point came with GPT-f [30], which demonstrated the power of generative pretraining and model scaling. Subsequent work built upon this, introducing key techniques and resources: the challenging miniF2F


Problem formal statement
Create local context same with the original problem
Code for decomposition
Explanation on syntax and our tactics
Subproblem extraction and proving
Replace `sorry` with subproofs
Proof consolidation
Use Lean4's delaborator to output the proof of the original problem
Proof for original problem

```
example (prem1 : type1) : main_target := by
```

```
play
  suppose (prem1 : type1) then
    terms := ...
    subgoal1 ⇐ show target1 by sorry
    subgoal2 ⇐ show target2 by sorry
    suppose (prem2: type2) ... then
      subgoal3 ⇐ show target3 by sorry
    main_goal ⇐ show main_target by sorry
  conclude main_goal
```

```
-- We can use tactic `suppose` to introduce premises of any type without specifying their values
-- `terms` can be arbitrary lean4 expressions, `play` will infer its type automatically
-- `target` can be any propositions which is the type of subgoals; `sorry` are placeholders for subproofs.

-- `suppose` can be nested

-- The last subproblem should be exactly the same with the original problem
-- Once all subproblems are solved, tactic `conclude` will automatically generate the proof of the original problem
```

```
example (prem1 : type1) (terms : ...) :
                          target1 := by proof1

example (prem1 : type1) (terms : ...)
          (subgoal1 : target1) :
                          target2 := by proof2

example (prem1 : type1) (terms : ...)
          (subgoal1   : target1)
          (subgoal2 : target2) (prem2 : type2):
                          target3 := by proof3

example (prem1 : type1) (terms : ...)
          (subgoal1 : target1)
          (subgoal2 : target2)
        (subgoal3:(prem2 : type2) → target3) :
                     main_target := by proof4
```

```
-- The extracted sub-problems uses Lean4's native syntax

/-
The premises for each subproblem include the assumptions introduced via suppose,
the terms brought in by define, and any subproblems already proven. Next,
we explain how our agent harnesses this DSL to recursively split goals, solve subproblems,
and assemble complete proofs.
-/

/-
When exiting a suppose block, each ensuing local declaration is automatically binded
 to the assumption introduced by that suppose. For example, check the type of subgoal3 here.
-/
```

```
play
  suppose (prem1 : type1) ... then
    terms := ...
    subgoal1 ⇐ show target1 by proof1
    subgoal2 ⇐ show target2 by proof2
    suppose (prem2: type2) ... then
      subgoal3 ⇐ show target3 by proof3
    main_goal ⇐ show main_target by proof4
  conclude main_goal
```

```
-- automatic proof consolidation
```

```
example (prem1 : type1) : main_target := by
  whole_proof
```

```
-- generated automatically by integrating proof1, proof2, proof3 and proof4
```

Figure 3 A demonstration of our Domain Specific Language (DSL), built upon Lean 4, and its role in managing decomposed sub-problems. Specifically, the DSL facilitates: maintaining the current state of sub-problem decomposition, extracting sub-problems into formal statements, and consolidating the proofs of sub-problems into the proof of the original problem.

benchmark [58], data augmentation via kernel information extraction [8], and expert iteration for continuous improvement [31]. The integration of tree search methods [18] provided another performance boost. These combined strategies significantly improved results, reaching milestones like 40.6% pass@1 accuracy on miniF2F-test. Building on these foundations (large-scale pretraining, expert iteration, tree search) and leveraging progress in base models, numerous recent provers have emerged [6, 16, 17, 19, 22, 39, 46, 47, 49, 50, 54]. Additional enhancements include integrating dedicated premise retrieval and reranking models [25, 53] and deploying problem generation models to enable self-play learning loops [4]. Furthermore, inspired by demonstrations of enhanced reasoning via explicit thinking steps [14], some systems now incorporate natural language reasoning alongside formal proof generation [21, 33, 41, 42].

Overall, the development of specialized, fine-tuned models represents the current state-of-the-art in automated formal theorem proving. However, achieving these results typically demands significant resources for data curation and training. This work diverges from that framework by employing general-purpose LLMs as prover agents, without specialized training. We now review prior work adopting this alternative approach.

General-Purpose LLMs for Formal Theorem Proving. Recently, the rapidly advancing capabilities of general-purpose Large Language Models (LLMs) have inspired a complementary line of research: using these powerful models in agentic roles to perform complex tasks without specialized fine-tuning. In the realm of theorem proving, DSP (Draft, Sketch, and Prove) [15] stands out as a prominent early example. DSP treats a general-purpose LLM as a black-box reasoning engine, prompting it to first decompose an informal proof into subproblems, and then formalize and solve these subproblems within a proof assistant like Isabelle. This approach yielded strong performance without requiring any model fine-tuning. Building on this, subsequent work introduced further refinements: COPRA [36] and Lyra [57] iteratively augment the prompt using execution feedback; [55, 56] improved the quality of the initial informal proof decomposition to better guide formal sketch generation; LEGO-Prover [38] proposed combining DSP with a growing library of solved subproblems; and [40] applied the DSP strategy recursively, expanding the proof as a tree structure.

Our work aligns with and draws inspiration from this established line of research. The iterative proof repair loop, for instance, is conceptually akin to CORPA. We advance this by leveraging an LLM's capacity for generating entire proofs, contrasting with CORPA's incremental execution. Similarly, our reflective decomposition process extends DSP by uniquely employing the LLM's reflective capabilities to iteratively derive more effective decomposition strategies.

4 Experiments

In this section, we provide a thorough evaluation of our prover agent. We will start with introduction of basic experiment settings, and then assess the performance of our prover agent over popular benchmarks. We end this section by demonstrating the effect of the key components of our prover through ablation studies.

4.1 Basic Experiment Setup

General-Purpose LLM. We employ Gemini 2.5 Pro 05-28 [7] as the general-purpose LLM in our agent. We choose the sampling parameters as `temperature` $= 1$, agreeing with common usage.

Benchmarks. We use the following benchmarks for evaluation:

- **MiniF2F-test.** MiniF2F-test is the test split of the MiniF2F benchmark [58]. This dataset contains 244 problems collected from mathematics competitions and the MATH dataset [9] with various difficulties. We use its Lean 4 version [51] and fix errors in the statements similar to [33, 41]. We also consider miniF2F-test-IMO, which consists of all IMO problems within miniF2F-test, and thus works as a more challenging benchmark.

4.2 Benchmark Performance

Table 1 presents a performance comparison of our prover agent against state-of-the-art provers on the MiniF2F-test benchmark. Our agent establishes a new highest accuracy on this benchmark, outperforming strong baselines including Kimina-Prover 72B [41, 42] and DeepSeek-Prover-V2 671B [33]. It is noteworthy that these competing methods require substantial data collection and fine-tuning efforts.

Due to the sequential nature of components like proof repair in our approach, the sample budget reported in Table 1 is determined as follows: once Delta Prover solves a problem, we record the number of API calls (or equivalent computational budget) used for that successful attempt. The final sample budget is then the **maximum** of these values across all solved problems. These per-problem statistics also enable us to illustrate Delta Prover's test-time scaling law, as plotted in Figure 4.

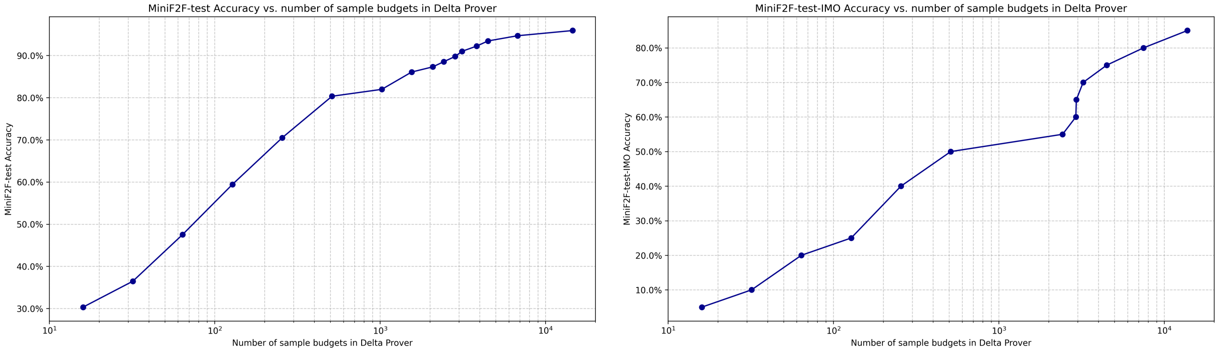


Figure 4 The test-time scaling law of Delta Prover over the miniF2F-test and the miniF2F-test-IMO benchmarks.

Table 1 Comparison with state-of-the-art approaches on the miniF2F-test dataset. The sample budgets and accuracy of state-of-the-art methods are found in the corresponding papers.

Method	Training-free	Sample budget	Accuracy
State-of-the-art Methods			
Goedel-Prover-SFT [22]	✖	25600	64.7%
STP [4]	✖	25600	67.6%
HunyuanProver 7B [19]	✖	1920000	68.4%
BFS-Prover 7B [50]	✖	2457600	70.8%
DeepSeek-Prover-V2 671B [33]	✖	8192	88.9%
Kimina-Prover 72B [42]	✖	42000	92.2%
Gemini 2.5 Pro 05-28 (BoN)	✔	16384	49.1%
Delta Prover (Our Approach)	✔	16384	95.9%

4.3 Ablation Studies

As detailed in Section 2.1, our prover agent employs two primary components: Proof Generation via Iterative Proof Repair and Reflective Decomposition. Given the strong performance reported previously, it is natural to investigate the extent to which each component contributes to the final results. This section presents ablation studies designed to quantify these individual contributions.

Effect of Iterative Proof Repair. We first isolate the impact of Iterative Proof Repair (Algorithm 1) by using it exclusively for proof construction. For these experiments, we utilize Gemini 2.5 Pro 05-28 [7] as the backbone large language model. Specifically, we conduct the following two experiments:

- **Optimal Hyperparameter Configuration.** We fix the total budget as 1024 and vary the number of iterative repairs per round n and the number of rounds m to see what is the optimal hyperparameter choice under a prefixed budget. The result can be seen in the left figure of Figure 5. Given that a higher n allows the model more opportunities to fix errors within a proof trajectory, while a higher m explores more distinct initial ideas, one might anticipate a trade-off between m and n. However, our results reveal a consistent trend within the tested budget range: accuracy generally improves as the number of repairs n increases for a fixed total budget $m \times n$. This finding, while perhaps counter-intuitive to the exploration-exploitation trade-off, highlights the potent self-correction capabilities inherent in modern large language models.

- **Comparison with Best-of-N (BoN) Sampling.** We further consider two extreme settings, i.e., $m = 1$ and $n = 1$. When the number of repairs $n = 1$, Algorithm 1 effectively reduces to the standard Best-of-N (BoN) sampling strategy with a total sample budget of m. Our experimental setup thus facilitates a direct comparison between Iterative Proof Repair and BoN. As shown in the right figure of Figure 5, Iterative Proof Repair consistently outperforms BoN sampling, and this performance advantage becomes increasingly pronounced as the total computational budget $(m \times n)$ increases, demonstrating a stronger test-time scaling law of Iterative Proof Repair.

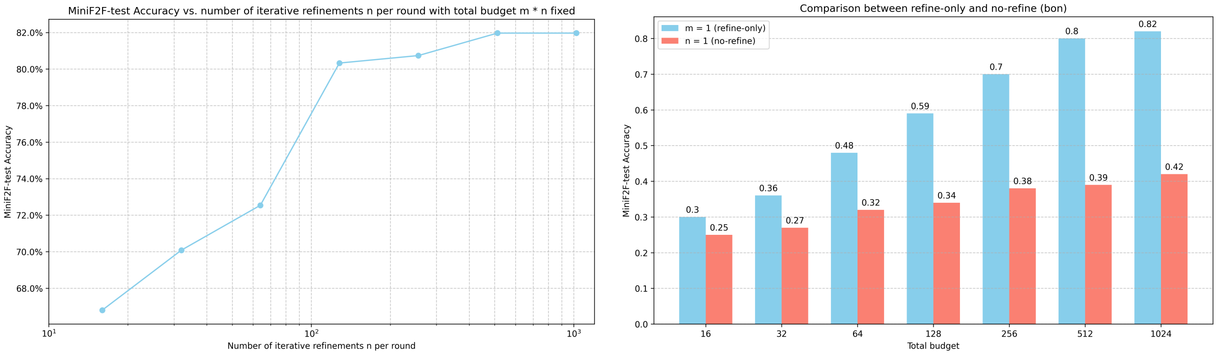


Figure 5 Ablation study on iterative error correction. Left: We keep the total budget fixed (as 1024), and observe the performance by varying the number of iterative repairs per round (i.e., n) and the number of rounds (i.e., m). Right: We compare the performance for the repair-only setting (i.e., $m = 1$) and no-repair setting (i.e., $n = 1$).

Effect of Reflective Decomposition. Our second experiment investigates the efficacy of Reflective Decomposition, particularly its role in guiding proof sketch generation. We selected the IMO 2019 Problem 1 (`imo_2019_p1`) from the MiniF2F test set. This problem, stated below, has proven challenging for automated theorem provers, remaining unsolved by prior methods, including the recent DeepSeek-Prover-V2 [33]. This makes it an excellent benchmark for our approach.

```
theorem imo_2019_p1 (f : ℤ → ℤ) :
  ((∀ a b, f (2 * a) + (2 * f b) = f (f (a + b))) ↔ (∀ z, f z = 0 \/ ∃ c, ∀ z, f z = 2 *
  ↪  z + c)) := by sorry
```

To evaluate the impact of decomposition, we compared two strategies for solving this problem: (I) a baseline approach using pure iterative proof correction, and (II) Delta Prover, which incorporated iterative proof correction together with Reflective Decomposition.

The baseline approach (pure iterative proof correction) failed to find a solution after 1024 API calls. In contrast, Delta Prover successfully solved the problem. The initial Reflective Decomposition round broke the problem into 83 sub-problems. Each sub-problem required, on average, 4 API calls to be solved. Consequently, the entire problem was solved using approximately $83 \times 4 = 332$ API calls. This is significantly fewer than the baseline and well under an initial budget of 400 API calls, clearly demonstrating the power and efficiency of Reflective Decomposition. The formal sketch found by our method is included in Appendix B.

5 Conclusion and Future Work

In conclusion, while the reasoning capabilities of large language models (LLMs) have shown immense promise, their application to formal theorem proving has been significantly challenged by the difficulty of mastering specialized formal languages and the substantial costs associated with bespoke model training. This work introduced Delta Prover, an agent-based framework designed to overcome these hurdles. By leveraging the inherent reasoning and reflection capabilities of general-purpose LLMs within an interactive Lean 4 environment, Delta Prover successfully orchestrates complex proof construction through reflective decomposition and iterative repair, eliminating the need for task-specific fine-tuning or extensive labeled data. Our findings demonstrate that this approach not only achieves state-of-the-art performance on rigorous benchmarks like miniF2F-test, surpassing even specialized models, but also offers a more scalable and resource-efficient pathway. This agent-centric methodology significantly advances automated theorem proving and offers broader insights into harnessing the sophisticated cognitive capacities of LLMs for complex problem-solving.

For future work, we aim to further enhance performance through several promising directions:

- Exploring more elaborate test-time techniques, such as evolutionary algorithms (e.g., FunSearch [34], AlphaEvolve [2]), to move beyond predefined workflows towards dynamic proof search optimization.
- Implementing reinforcement learning (RL) based on the agentic workflow proposed in this paper, enabling the agent to continuously refine its proof strategies.

References

[1] Kshitij Bansal, Christian Szegedy, Markus N Rabe, Sarah M Loos, and Viktor Toman. Learning to reason in large theories without imitation. arXiv preprint arXiv:1905.10501, 2019.

[2] Deepmind. Ai achieves silver-medal standard solving international mathematical olympiad problems, 2024. URL https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/.

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

[4] Kefan Dong and Tengyu Ma. Stp: Self-play llm theorem provers with iterative conjecturing and proving. arXiv e-prints, pages arXiv–2502, 2025.

[5] Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, et al. Competitive programming with large reasoning models. arXiv preprint arXiv:2502.06807, 2025.

[6] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 1229–1241, 2023.

[7] Google DeepMind. Gemini 2.5 pro. https://deepmind.google/technologies/gemini/pro/, 2025.

[8] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. arXiv preprint arXiv:2102.06203, 2021.

[9] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. NeurIPS, 2021.

[10] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. Gamepad: A learning environment for theorem proving. arXiv preprint arXiv:1806.00608, 2018.

[11] Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, et al. Opencoder: The open cookbook for top-tier code large language models. arXiv preprint arXiv:2411.04905, 2024.

[12] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. Deepmath-deep sequence models for premise selection. Advances in neural information processing systems, 29, 2016.

[13] Isabelle Project. Isabelle. https://isabelle.in.tum.de. Accessed: 2024-03-18.

[14] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. arXiv preprint arXiv:2412.16720, 2024.

[15] Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. arXiv preprint arXiv:2210.12283, 2022.

[16] Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. Lisa: Language models of isabelle proofs. In 6th Conference on Artificial Intelligence and Theorem Proving, pages 378–392, 2021.

[17] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers. Advances in Neural Information Processing Systems, 35:8360–8373, 2022.

[18] Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. Advances in neural information processing systems, 35:26337–26349, 2022.

[19] Yang Li, Dong Du, Linfeng Song, Chen Li, Weikang Wang, Tao Yang, and Haitao Mi. Hunyuanprover: A scalable data synthesis framework and guided tree search for automated theorem proving. arXiv preprint arXiv:2412.20735, 2024.

[20] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. arXiv preprint arXiv:2305.20050, 2023.

[21] Haohan Lin, Zhiqing Sun, Sean Welleck, and Yiming Yang. Lean-star: Learning to interleave thinking and proving. arXiv preprint arXiv:2407.10040, 2024.

[22] Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, et al. Goedel-prover: A frontier model for open-source automated theorem proving. arXiv preprint arXiv:2502.07640, 2025.

[23] 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, 2024.

[24] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. arXiv preprint arXiv:1701.06972, 2017.

[25] Maciej Mikuła, Szymon Tworkowski, Szymon Antoniak, Bartosz Piotrowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. Magnushammer: A transformer-based approach to premise selection. arXiv preprint arXiv:2303.04488, 2023.

[26] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28, pages 625–635. Springer, 2021.

[27] OpenAI. Introducing deep research. https://openai.com/index/introducing-deep-research/.

[28] OpenAI. Solving (some) formal math olympiad problems, 2022. URL https://openai.com/index/formal-math/.

[29] OpenAI. Introducing openai o3 and o4-mini, 2025. URL https://openai.com/index/introducing-o3-and-o4-mini/.

[30] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. arXiv preprint arXiv:2009.03393, 2020.

[31] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. In The Eleventh International Conference on Learning Representations, 2023.

[32] Markus N Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. Mathematical reasoning via self-supervised skip-tree training. arXiv preprint arXiv:2006.04757, 2020.

[33] 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. 2025.

[34] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. Nature, 625(7995):468–475, 2024.

[35] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. arXiv preprint arXiv:1802.03685, 2018.

[36] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving. arXiv preprint arXiv:2310.04353, 2023.

[37] The Coq Development Team. The Coq proof assistant. https://coq.inria.fr/. Accessed: 2024-02-29.

[38] Haiming Wang, Huajian Xin, Chuanyang Zheng, Lin Li, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, et al. Lego-prover: Neural theorem proving with growing libraries. arXiv preprint arXiv:2310.00656, 2023.

[39] Haiming Wang, Ye Yuan, Zhengying Liu, Jianhao Shen, Yichun Yin, Jing Xiong, Enze Xie, Han Shi, Yujun Li, Lin Li, et al. Dt-solver: Automated theorem proving with dynamic-tree sampling guided by proof-level value function. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 12632–12646, 2023.

[40] Haiming Wang, Huajian Xin, Zhengying Liu, Wenda Li, Yinya Huang, Jianqiao Lu, Zhicheng Yang, Jing Tang, Jian Yin, Zhenguo Li, et al. Proving theorems recursively. arXiv preprint arXiv:2405.14414, 2024.

[41] 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. 2025. URL http://arxiv.org/abs/2504.11354.

[42] 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: Kimina-prover: Applying test-time rl search on large formal reasoning models. 2025. URL https://huggingface.co/blog/AI-MO/kimina-prover.

[43] Mingzhe Wang and Jia Deng. Learning to prove theorems by learning to generate theorems. Advances in neural information processing systems, 33:18146–18157, 2020.

[44] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. Advances in neural information processing systems, 30, 2017.

[45] Daniel Whalen. Holophrasm: a neural automated theorem prover for higher-order logic. arXiv preprint arXiv:1608.02644, 2016.

[46] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. Advances in Neural Information Processing Systems, 35: 32353–32368, 2022.

[47] Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Jiayu Wang, Dahua Lin, and Kai Chen. Internlm2. 5-stepprover: Advancing automated theorem proving via expert iteration on large-scale lean problems. arXiv preprint arXiv:2410.15700, 2024.

[48] Zijian Wu, Jiayu Wang, Dahua Lin, and Kai Chen. Lean-github: Compiling github lean repositories for a versatile lean prover. arXiv preprint arXiv:2407.17227, 2024.

[49] 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.

[50] Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Kai Shen. Bfs-prover: Scalable best-first tree search for llm-based automatic theorem proving. arXiv preprint arXiv:2502.03438, 2025.

[51] Kaiyu Yang. miniF2F-lean4: A collection of formal-to-formal statements and proofs in Lean 4. https://github.com/yangky11/miniF2F-lean4, 2023.

[52] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In International Conference on Machine Learning, pages 6984–6994. PMLR, 2019.

[53] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. Advances in Neural Information Processing Systems, 36:21573–21612, 2023.

[54] Jingyuan Zhang, Qi Wang, Xingguang Ji, Yahui Liu, Yang Yue, Fuzheng Zhang, Di Zhang, Guorui Zhou, and Kun Gai. Leanabell-prover: Posttraining scaling in formal reasoning. arXiv preprint arXiv:2504.06122, 2025.

[55] Xueliang Zhao, Wenda Li, and Lingpeng Kong. Decomposing the enigma: Subgoal-based demonstration learning for formal theorem proving. arXiv preprint arXiv:2305.16366, 2023.

[56] Xueliang Zhao, Lin Zheng, Haige Bo, Changran Hu, Urmish Thakker, and Lingpeng Kong. Subgoalxl: Subgoal-based expert learning for theorem proving. arXiv preprint arXiv:2408.11172, 2024.

[57] Chuanyang Zheng, Haiming Wang, Enze Xie, Zhengying Liu, Jiankai Sun, Huajian Xin, Jianhao Shen, Zhenguo Li, and Yu Li. Lyra: Orchestrating dual correction in automated theorem proving. arXiv preprint arXiv:2309.15806, 2023.

[58] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. arXiv preprint arXiv:2109.00110, 2021.

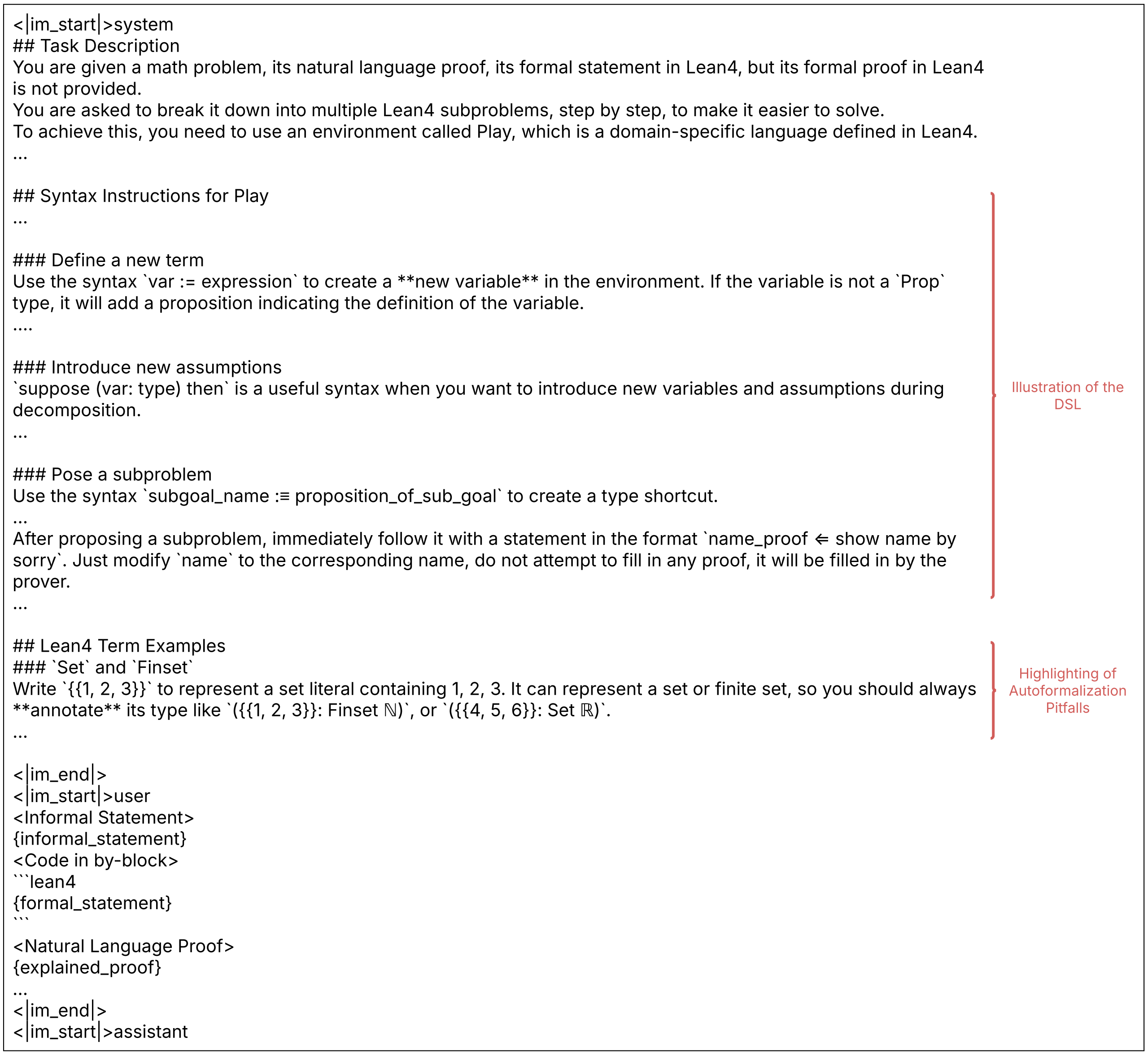
[59] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. arXiv preprint arXiv:2406.11931, 2024.

Appendix

A Prompt Templates

In this section, we provide the concrete templates for the prompt introduced in Section 2.1. Specifically, there are three types of prompts employed in our prover agent, described as follows:

- **Prompt for Sub-problem Decomposition.** As outlined in Section 2.1.2, the prompt is composed of the illustration of the DSL and the highlighting of autoformalization pitfalls. We provide the concrete template in Figure 6.
- **Prompt for Informal Proof Generation.** As outlined in Section 2.1.2, the prompt is composed of getting refined solutions and scoring solutions at a multi-dimensional fine-grained level. We provide the concrete template in Figure 7.
- **Prompt for Formal Proof Generation.** As outlined in Section 2.1.1, the prompt consists of formatting conventions, Lean 4 specification, effective tactics, and fixup information. The corresponding template can be seen in Figure 8.

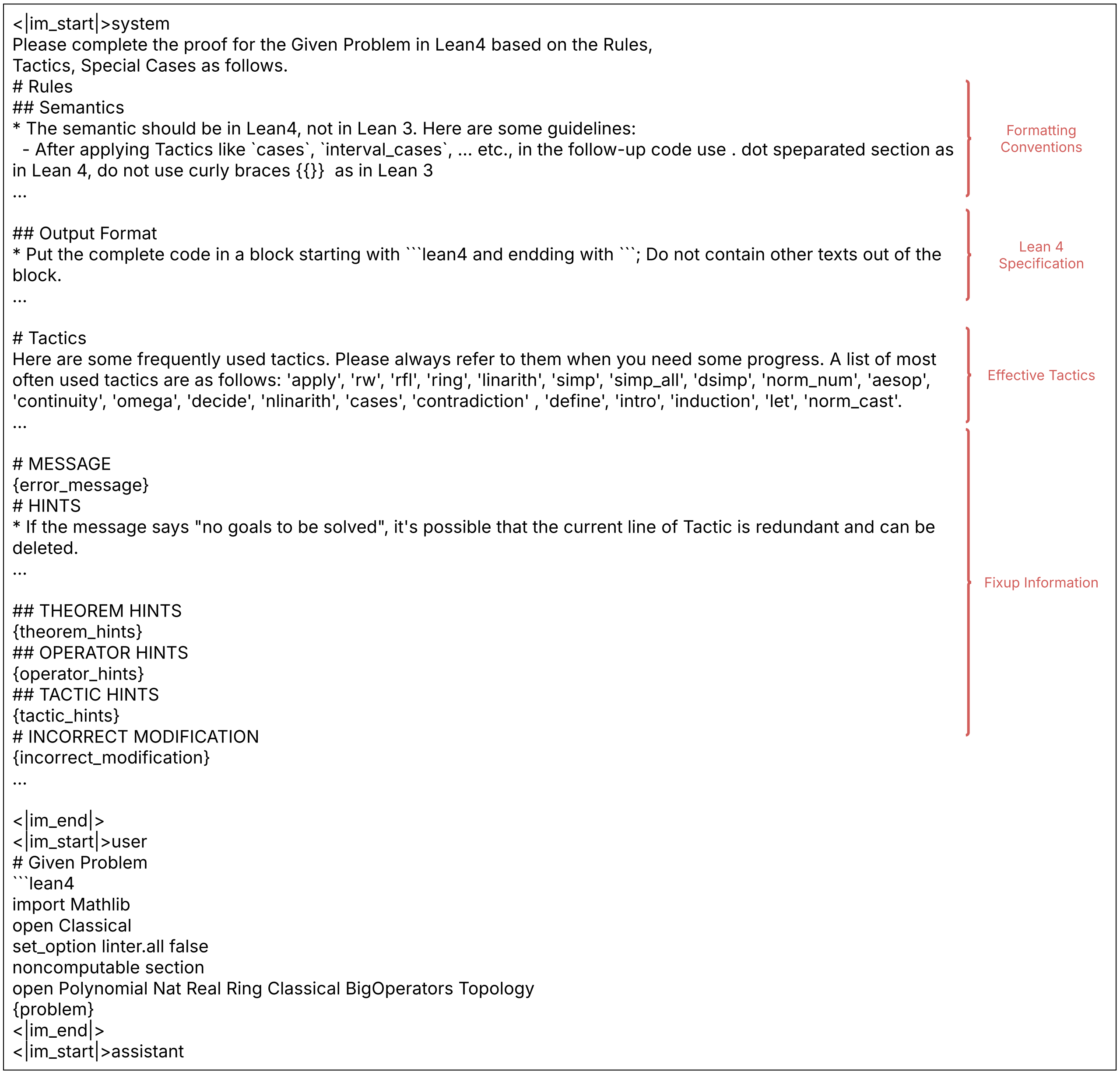


~~~
<|im_start|>system
## Task Description
You are given a math problem, its natural language proof, its formal statement in Lean4, but its formal proof in Lean4
is not provided.
You are asked to break it down into multiple Lean4 subproblems, step by step, to make it easier to solve.
To achieve this, you need to use an environment called Play, which is a domain-specific language defined in Lean4.
...

## Syntax Instructions for Play
...

### Define a new term
Use the syntax `var := expression` to create a **new variable** in the environment. If the variable is not a `Prop`
type, it will add a proposition indicating the definition of the variable.
....

### Introduce new assumptions
`suppose (var: type) then` is a useful syntax when you want to introduce new variables and assumptions during
decomposition.
...

### Pose a subproblem
Use the syntax `subgoal_name :≡ proposition_of_sub_goal` to create a type shortcut.
...
After proposing a subproblem, immediately follow it with a statement in the format `name_proof ⇐ show name by
sorry`. Just modify `name` to the corresponding name, do not attempt to fill in any proof, it will be filled in by the
prover.
...

## Lean4 Term Examples
### `Set` and `Finset`
Write `{{1, 2, 3}}` to represent a set literal containing 1, 2, 3. It can represent a set or finite set, so you should always
**annotate** its type like `({{1, 2, 3}}: Finset ℕ)`, or `({{4, 5, 6}}: Set ℝ)`.
...

<|im_end|>
<|im_start|>user
<Informal Statement>
{informal_statement}
<Code in by-block>
```lean4
{formal_statement}
```
<Natural Language Proof>
{explained_proof}
...
<|im_end|>
<|im_start|>assistant
~~~

Figure 6 The prompt template for sub-problem decomposition.

```
<|im_start|>system
# Task Description
As a mathematician specializing in formal-to-informal proof translation and explanation, your task is to construct a detailed, step-by-step informal mathematical proof in natural language. This proof should correspond to the provided Lean 4 statement and its natural language problem description. Your proof should be precise, with clear logical coherence. Each step should be justified, building upon previous steps or established mathematical principles.
# Input Specifications
You will be provided with:
## 1. Lean 4 Code:
...

## 2. Natural Language Problem Statement:
...

# Proof Construction Requirements
## 1. Mathematical Precision
...

## 2. Step Detailing
...

## 3. Logical Completeness
...

# Response Format
Your output should be the natural language proof, structured as follows:
- Natural Language Proof
**Step 1:** [Describe the first logical step of the proof. Explain the reasoning clearly. For example: "We start by considering the definition of 'even number' as applied to `n`..." or "Our goal is to show [some intermediate statement]. We begin by..."]
**Step 2:** [Describe the second logical step, building on Step 1 if appropriate. For example: "From Step 1, and applying the property that the sum of two even numbers is even, we deduce..." or "Now, we consider the case where [condition for case 1]..."]
...
**Step N:** [The final step leading directly to the conclusion. For example: "...which simplifies to `LHS = RHS`. This shows that the original equality holds under the given assumptions."]

<|im_end|>
<|im_start|>user
- Problem:
{informal_statement}
- Formal Statement:
{formal_statement}

<|im_end|>
<|im_start|>assistant
```

Figure 7 The prompt template for natural language sketch.



~~~
<|im_start|>system
Please complete the proof for the Given Problem in Lean4 based on the Rules,
Tactics, Special Cases as follows.
# Rules
## Semantics
* The semantic should be in Lean4, not in Lean 3. Here are some guidelines:
  - After applying Tactics like `cases`, `interval_cases`, ... etc., in the follow-up code use . dot speparated section as
in Lean 4, do not use curly braces {{}}  as in Lean 3
...

## Output Format
* Put the complete code in a block starting with ```lean4 and endding with ```; Do not contain other texts out of the
block.
...

# Tactics
Here are some frequently used tactics. Please always refer to them when you need some progress. A list of most
often used tactics are as follows: 'apply', 'rw', 'rfl', 'ring', 'linarith', 'simp', 'simp_all', 'dsimp', 'norm_num', 'aesop',
'continuity', 'omega', 'decide', 'nlinarith', 'cases', 'contradiction' , 'define', 'intro', 'induction', 'let', 'norm_cast'.
...

# MESSAGE
{error_message}
# HINTS
* If the message says "no goals to be solved", it's possible that the current line of Tactic is redundant and can be
deleted.
...

## THEOREM HINTS
{theorem_hints}
## OPERATOR HINTS
{operator_hints}
## TACTIC HINTS
{tactic_hints}
# INCORRECT MODIFICATION
{incorrect_modification}
...

<|im_end|>
<|im_start|>user
# Given Problem
```lean4
import Mathlib
open Classical
set_option linter.all false
noncomputable section
open Polynomial Nat Real Ring Classical BigOperators Topology
{problem}
<|im_end|>
<|im_start|>assistant
~~~

Figure 8 The prompt template for proof repair.

B Representative Examples

This section presents representative examples illustrating the key components of Delta Prover.

The first example demonstrates our retrieval mechanism. As shown in the accompanying Lean code, the LLM initially generates an incorrect theorem name, `AntitoneOn.sum_le_integral_Icc`. Our retrieval mechanism then intervenes, using this erroneous suggestion to retrieve the correct theorem name (`AntitoneOn.sum_le_integral_Ico`) and its signature. This information is subsequently provided as a hint to the LLM. This case highlights how the synergy between the LLM's initial proposals and our retrieval system facilitates the discovery and application of advanced mathematical tools, which can be challenging for humans to find independently.

```
theorem algebra_sum1onsqrt2to1onsqrt10000lt198 :
  Σ k in (Finset.Icc (2 : ℕ) 10000), (1 / Real.sqrt k) < 198 := by
    /- ...(proof omitted above)... -/
    have h_shifted_sum_le_integral : (Σ i in Finset.Ico a b, f (↑(i + 1) : ℝ)) ≤ ∫ (x :
    ↪  ℝ) in (a : ℝ)..(b : ℝ), f x := by
        /- Attempt 1 (Failed) -/
        -- The original attempt used the wrong theorem name
        ↪  `Finset.AntitoneOn.sum_le_integral_Icc`.
        apply AntitoneOn.sum_le_integral_Icc hab_nat hf_antitone

        /- Attempt 2 (Succeeded) -/
        -- Based on the wrong name, our retrieval mechanism successfully finds the
        ↪  correct theorem `AntitoneOn.sum_le_integral_Ico`
        apply AntitoneOn.sum_le_integral_Ico hab_nat hf_antitone
    /- ...(proof omitted below)... -/
```

Our subsequent example showcases a more intricate scenario of iterative proof correction, requiring the LLM to reflect twice to reach the correct solution. The LLM's first attempt involves using the theorem `mul_div_cancel_left`, but this results in an error. Following this, the LLM reflects on the failure and proposes an alternative: the `simp` tactic. When Lean reports another error, signifying that `simp` alone cannot complete the arithmetic simplification, the LLM again adapts. It recognizes this further limitation and explicitly calls `norm_num` to finalize the proof.

```lean
theorem mathd_algebra_141
  (a b : ℝ)
  (h₁ : (a * b)=180)
  (h₂ : 2 * (a + b)=54) :
  (a^2 + b^2) = 369 := by
  /- ...(omitted above)... -/
  have h_sum : a + b = 27 := by
    have h₂_div_two : (2 * (a + b)) / 2 = 54 / 2 := by rw [h₂]

    /- Attempt 1 (Failed) -/
    -- Simplify both sides of the equation.
    rw [mul_div_cancel_left (a + b) two_ne_zero] at h₂_div_two
    rw [div_by_two (54 : ℝ)] at h₂_div_two
    -- Use norm_num to simplify the right side.
    norm_num at h₂_div_two

    /- Attempt 2 (Failed) -/
    -- Simplify both sides of the equation using simp.
    -- The original code used mul_div_cancel_left with incorrect arguments and a
    ↪  non-standard div_by_two.
    -- Using simp is sufficient here to simplify (2 * (a + b)) / 2 to a + b and 54 / 2
    ↪  to 27.
    simp at h₂_div_two

    /- Attempt 3 (Succeeded)-/
    -- Simplify the expression (2 * (a + b)) / 2 on the left side.
    -- According to the error message, simp simplifies the left side but not the right.
    simp at h₂_div_two
    -- Simplify the numerical expression 54 / 2 on the right side using norm_num.
    -- The error message indicates that simp did not simplify the numerical fraction
    ↪  54/2 to 27.
    -- We explicitly use norm_num to perform this numerical simplification.
    norm_num at h₂_div_two

    exact h₂_div_two
  /- ...(omitted below)... -/
```

The next example showcases the LLM's decomposition of the challenging IMO 2019 P1 problem, as referenced in the main text. Specifically, the LLM breaks down the problem into a remarkably fine-grained set of 83 subproblems, thereby facilitating the subsequent solving process.

```
variable (f : ℤ → ℤ)
play
  -- P1: (∀ a b, f (2 * a) + (2 * f b) = f (f (a + b)))
  -- P2: (∀ z, f z = 0 ∨ ∃ c, ∀ z, f z = 2 * z + c)
  -- We need to prove P1 → P2 and P2 → P1

  -- Part 1: P1 → P2
  suppose (h_P1 : ∀ a b, f (2 * a) + (2 * f b) = f (f (a + b))) then
    constant_c := f 0

    -- Step 1 (NL): For arbitrary b ∈ ℤ, set a = 0 in h_P1. Yields c + 2f(b) = f(f(b))
    ↪  (Eq1)
    subprob_eq1_intermediate :≡ ∀ b, f (2 * 0) + 2 * f b = f (f (0 + b))
    subprob_eq1_intermediate_proof ⇐ show subprob_eq1_intermediate by sorry
    subprob_eq1 :≡ ∀ b, constant_c + 2 * f b = f (f b)
    subprob_eq1_proof ⇐ show subprob_eq1 by sorry

    -- Step 2 (NL): For arbitrary a ∈ ℤ, set b = 0 in h_P1. Yields f(2a) + 2c = f(f(a))
    ↪  (Eq2)
    subprob_eq2_intermediate :≡ ∀ a, f (2 * a) + 2 * f 0 = f (f (a + 0))
    subprob_eq2_intermediate_proof ⇐ show subprob_eq2_intermediate by sorry
    subprob_eq2 :≡ ∀ a, f (2 * a) + 2 * constant_c = f (f a)
    subprob_eq2_proof ⇐ show subprob_eq2 by sorry

    -- Step 3 (NL): Equate expressions for f(f(a)). From Eq1, f(f a) = c + 2f(a).
    ↪  Substitute into Eq2.
    subprob_ffa_expr_from_eq1 :≡ ∀ a, f (f a) = constant_c + 2 * f a
    subprob_ffa_expr_from_eq1_proof ⇐ show subprob_ffa_expr_from_eq1 by sorry
    subprob_eq3_intermediate :≡ ∀ a, f (2 * a) + 2 * constant_c = constant_c + 2 * f a
    subprob_eq3_intermediate_proof ⇐ show subprob_eq3_intermediate by sorry
    -- Simplify to get f(2a) = 2f(a) - c (Eq3)
    subprob_eq3 :≡ ∀ a, f (2 * a) = 2 * f a - constant_c
    subprob_eq3_proof ⇐ show subprob_eq3 by sorry

    -- Step 4 (NL): Obtaining a recurrence for f. Set b = 1 in h_P1: f(2a) + 2f(1) =
    ↪  f(f(a+1)).
    subprob_P_a_1 :≡ ∀ a, f (2 * a) + 2 * f 1 = f (f (a + 1))
    subprob_P_a_1_proof ⇐ show subprob_P_a_1 by sorry
    -- Use Eq1 (b := a+1): f(f(a+1)) = c + 2f(a+1). So, f(2a) + 2f(1) = c + 2f(a+1).
    subprob_ffap1_expr_from_eq1 :≡ ∀ a, f (f (a+1)) = constant_c + 2 * f (a+1)
    subprob_ffap1_expr_from_eq1_proof ⇐ show subprob_ffap1_expr_from_eq1 by sorry
    subprob_eq4_intermediate1 :≡ ∀ a, f (2 * a) + 2 * f 1 = constant_c + 2 * f (a + 1)
    subprob_eq4_intermediate1_proof ⇐ show subprob_eq4_intermediate1 by sorry
    -- Substitute Eq3 for f(2a): (2f(a) - c) + 2f(1) = c + 2f(a+1).
    subprob_eq4_intermediate2 :≡ ∀ a, (2 * f a - constant_c) + 2 * f 1 = constant_c + 2 *
    ↪  f (a + 1)
    subprob_eq4_intermediate2_proof ⇐ show subprob_eq4_intermediate2 by sorry
    -- Rearrange: f(a+1) = f(a) + f(1) - c (Eq4)
    subprob_eq4 :≡ ∀ a, f (a + 1) = f a + f 1 - constant_c
    subprob_eq4_proof ⇐ show subprob_eq4 by sorry
```

```
-- Step 5 (NL): Solving recurrence f(a+1) = f(a) + K. Solution f(a) = c + a*(f(1)-c)
↪  (Eq5)
-- For a >= 0 (induction on n : ℕ for f(n))
P_pos := fun (n : ℤ) => f n = constant_c + n * (f 1 - constant_c)
subprob_eq5_pos_base :≡ P_pos 0
subprob_eq5_pos_base_proof ⇐ show subprob_eq5_pos_base by sorry
subprob_eq5_pos_step :≡ ∀ (k : ℕ), P_pos (k : ℤ) → P_pos ((k + 1) : ℤ)
subprob_eq5_pos_step_proof ⇐ show subprob_eq5_pos_step by sorry
subprob_eq5_pos :≡ ∀ (n : ℕ), P_pos (n : ℤ)
subprob_eq5_pos_proof ⇐ show subprob_eq5_pos by sorry
-- For a <= 0 (induction on n : ℕ for f(-n))
-- Need f(a-1) = f(a) - (f(1)-c). From Eq4: f(x) = f(x+1) - (f(1)-c). Let x = a-1.
subprob_eq4_for_neg :≡ ∀ a, f (a - 1) = f a - (f 1 - constant_c)
subprob_eq4_for_neg_proof ⇐ show subprob_eq4_for_neg by sorry
subprob_eq5_neg_base :≡ P_pos 0 -- Same as P_pos 0 for f(-(0 : ℤ))
subprob_eq5_neg_base_proof ⇐ show subprob_eq5_neg_base by sorry
subprob_eq5_neg_step :≡ ∀ (k : ℕ), P_pos (-(k : ℤ)) → P_pos (-((k + 1) : ℤ))
subprob_eq5_neg_step_proof ⇐ show subprob_eq5_neg_step by sorry
subprob_eq5_neg :≡ ∀ (n : ℕ), P_pos (-(n : ℤ))
subprob_eq5_neg_proof ⇐ show subprob_eq5_neg by sorry
-- Combine results for all integers a
subprob_eq5 :≡ ∀ a, f a = constant_c + a * (f 1 - constant_c)
subprob_eq5_proof ⇐ show subprob_eq5 by sorry

slope_m := f 1 - constant_c
subprob_eq5_m_form :≡ ∀ a, f a = slope_m * a + constant_c
subprob_eq5_m_form_proof ⇐ show subprob_eq5_m_form by sorry

-- Step 6 (NL): Determining slope m. Substitute f(x)=mx+c into Eq1: c + 2f(b) =
↪  f(f(b)).
-- LHS = 2mb + 3c
subprob_eq1_lhs_eval :≡ ∀ b, constant_c + 2 * (slope_m * b + constant_c) = 2 *
↪  slope_m * b + 3 * constant_c
subprob_eq1_lhs_eval_proof ⇐ show subprob_eq1_lhs_eval by sorry
subprob_eq1_lhs_subst :≡ ∀ b, constant_c + 2 * f b = 2 * slope_m * b + 3 * constant_c
subprob_eq1_lhs_subst_proof ⇐ show subprob_eq1_lhs_subst by sorry
-- RHS = m^2*b + mc + c
subprob_eq1_rhs_eval :≡ ∀ b, slope_m * (slope_m * b + constant_c) + constant_c =
↪  slope_m^2 * b + slope_m * constant_c + constant_c
subprob_eq1_rhs_eval_proof ⇐ show subprob_eq1_rhs_eval by sorry
subprob_eq1_rhs_subst :≡ ∀ b, f (f b) = slope_m^2 * b + slope_m * constant_c +
↪  constant_c
subprob_eq1_rhs_subst_proof ⇐ show subprob_eq1_rhs_subst by sorry
-- Equating LHS and RHS: 2mb + 3c = m^2*b + mc + c
subprob_poly_id :≡ ∀ b, 2 * slope_m * b + 3 * constant_c = slope_m^2 * b + slope_m *
↪  constant_c + constant_c
subprob_poly_id_proof ⇐ show subprob_poly_id by sorry

-- For this polynomial identity to hold for all b, coefficients must match.
-- Constant terms (set b=0): 3c = mc + c
subprob_const_terms_eq_intermediate :≡ 2 * slope_m * 0 + 3 * constant_c = slope_m^2 *
↪  0 + slope_m * constant_c + constant_c
subprob_const_terms_eq_intermediate_proof ⇐ show subprob_const_terms_eq_intermediate
↪  by sorry
subprob_const_terms_eq :≡ 3 * constant_c = slope_m * constant_c + constant_c
subprob_const_terms_eq_proof ⇐ show subprob_const_terms_eq by sorry
-- Coefficients of b (substitute b=1 and use const_terms_eq)
```

```
    subprob_coeff_b_eq_intermediate :≡ 2 * slope_m * 1 + 3 * constant_c = slope_m^2 * 1 +
    ↪  slope_m * constant_c + constant_c
    subprob_coeff_b_eq_intermediate_proof ⇐ show subprob_coeff_b_eq_intermediate by
    ↪  sorry
    subprob_coeff_b_eq :≡ 2 * slope_m = slope_m^2
    subprob_coeff_b_eq_proof ⇐ show subprob_coeff_b_eq by sorry

    -- From 2m = m^2 => m(m-2)=0. So m=0 or m=2.
    subprob_slope_m_factor :≡ slope_m * (slope_m - 2) = 0
    subprob_slope_m_factor_proof ⇐ show subprob_slope_m_factor by sorry
    subprob_slope_m_is_0_or_2 :≡ slope_m = 0 ∨ slope_m = 2
    subprob_slope_m_is_0_or_2_proof ⇐ show subprob_slope_m_is_0_or_2 by sorry

    -- Case (i): slope_m = 0
    suppose (h_slope_m_0 : slope_m = 0) then
      -- From const_terms_eq: 3c = 0*c + c => 3c = c => 2c = 0 => c = 0.
      subprob_case_m_0_implies_c_0_intermediate :≡ 3 * constant_c = 0 * constant_c +
      ↪  constant_c
      subprob_case_m_0_implies_c_0_intermediate_proof ⇐ show
      ↪  subprob_case_m_0_implies_c_0_intermediate by sorry
      subprob_case_m_0_implies_c_0 :≡ constant_c = 0
      subprob_case_m_0_implies_c_0_proof ⇐ show subprob_case_m_0_implies_c_0 by sorry
      -- If slope_m = 0 and constant_c = 0, then f(a) = 0*a + 0 = 0 for all a.
      subprob_sol_fx_eq_0 :≡ ∀ z, f z = 0
      subprob_sol_fx_eq_0_proof ⇐ show subprob_sol_fx_eq_0 by sorry
      -- This implies P2
      subprob_P2_from_m_0 :≡ (∀ z, f z = 0 ∨ ∃ c_val, ∀ z, f z = 2 * z + c_val)
      subprob_P2_from_m_0_proof ⇐ show subprob_P2_from_m_0 by sorry

    -- Case (ii): slope_m = 2
    suppose (h_slope_m_2 : slope_m = 2) then
      -- From const_terms_eq: 3c = 2c + c => 3c = 3c. Always true.
      subprob_case_m_2_holds_intermediate :≡ 3 * constant_c = 2 * constant_c + constant_c
      subprob_case_m_2_holds_intermediate_proof ⇐ show
      ↪  subprob_case_m_2_holds_intermediate by sorry
      subprob_case_m_2_holds :≡ constant_c = constant_c -- Or some trivial True statement
      subprob_case_m_2_holds_proof ⇐ show subprob_case_m_2_holds by sorry
      -- If slope_m = 2, then f(a) = 2a + constant_c for all a. (constant_c is f(0))
      subprob_sol_fx_eq_2x_plus_c :≡ ∀ z, f z = 2 * z + constant_c
      subprob_sol_fx_eq_2x_plus_c_proof ⇐ show subprob_sol_fx_eq_2x_plus_c by sorry
      -- This implies P2
      subprob_P2_from_m_2 :≡ (∀ z, f z = 0 ∨ ∃ c_val, ∀ z, f z = 2 * z + c_val)
      subprob_P2_from_m_2_proof ⇐ show subprob_P2_from_m_2 by sorry

    -- Combining cases for slope_m = 0 or slope_m = 2 using subprob_slope_m_is_0_or_2
    subprob_P1_implies_P2 :≡ (∀ z, f z = 0 ∨ ∃ c_val, ∀ z, f z = 2 * z + c_val)
    subprob_P1_implies_P2_proof ⇐ show subprob_P1_implies_P2 by sorry

  -- Part 2: P2 → P1
  suppose (h_P2 : (∀ z, f z = 0) ∨ (∃ c_val, ∀ z, f z = 2 * z + c_val)) then
    -- Case 1: f(z) = 0 for all z.
    suppose (h_fz_is_0 : ∀ z, f z = 0) then
      subprob_verify_lhs_is_0 :≡ ∀ a b, f (2 * a) + 2 * f b = 0 + 2 * 0
      subprob_verify_lhs_is_0_proof ⇐ show subprob_verify_lhs_is_0 by sorry
      subprob_verify_rhs_is_0 :≡ ∀ a b, f (f (a + b)) = f 0
      subprob_verify_rhs_is_0_proof ⇐ show subprob_verify_rhs_is_0 by sorry
      subprob_P1_holds_for_fz_0 :≡ ∀ a b, f (2 * a) + (2 * f b) = f (f (a + b))
      subprob_P1_holds_for_fz_0_proof ⇐ show subprob_P1_holds_for_fz_0 by sorry
    -- Case 2: ∃ c_val such that f(z) = 2z + c_val for all z.
    suppose (h_fz_is_2z_plus_c : ∃ c_val, ∀ z, f z = 2 * z + c_val) then
      ⟨c_0, hc0⟩ := h_fz_is_2z_plus_c -- hc0 is (∀ z, f z = 2 * z + c_0)

      subprob_verify_lhs_val :≡ ∀ a b, f (2 * a) + 2 * f b = (2 * (2 * a) + c_0) + 2 * (2
      ↪  * b + c_0)
      subprob_verify_lhs_val_proof ⇐ show subprob_verify_lhs_val by sorry
      subprob_verify_lhs_is_4ab3c :≡ ∀ a b, f (2 * a) + 2 * f b = 4 * (a + b) + 3 * c_0
      subprob_verify_lhs_is_4ab3c_proof ⇐ show subprob_verify_lhs_is_4ab3c by sorry

      subprob_verify_rhs_val1 :≡ ∀ a b, f (f (a + b)) = f (2 * (a+b) + c_0)
      subprob_verify_rhs_val1_proof ⇐ show subprob_verify_rhs_val1 by sorry
      subprob_verify_rhs_val2 :≡ ∀ a b, f (2 * (a+b) + c_0) = 2 * (2 * (a+b) + c_0) + c_0
      subprob_verify_rhs_val2_proof ⇐ show subprob_verify_rhs_val2 by sorry
      subprob_verify_rhs_is_4ab3c :≡ ∀ a b, f (f (a + b)) = 4 * (a + b) + 3 * c_0
      subprob_verify_rhs_is_4ab3c_proof ⇐ show subprob_verify_rhs_is_4ab3c by sorry

      subprob_P1_holds_for_fz_2zc :≡ ∀ a b, f (2 * a) + (2 * f b) = f (f (a + b))
      subprob_P1_holds_for_fz_2zc_proof ⇐ show subprob_P1_holds_for_fz_2zc by sorry

    -- Combine Case 1 and Case 2 using h_P2
    subprob_P2_implies_P1 :≡ ∀ a b, f (2 * a) + (2 * f b) = f (f (a + b))
    subprob_P2_implies_P1_proof ⇐ show subprob_P2_implies_P1 by sorry

  -- Final Goal (combining P1 → P2 and P2 → P1)
  subprob_final_goal :≡ ((∀ a b, f (2 * a) + (2 * f b) = f (f (a + b))) ↔ (∀ z, f z = 0
   ↪  ∨ ∃ c, ∀ z, f z = 2 * z + c))
  subprob_final_goal_proof ⇐ show subprob_final_goal by sorry
```