PROOFFLOW: A DEPENDENCY GRAPH APPROACH TO FAITHFUL PROOF AUTOFORMALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

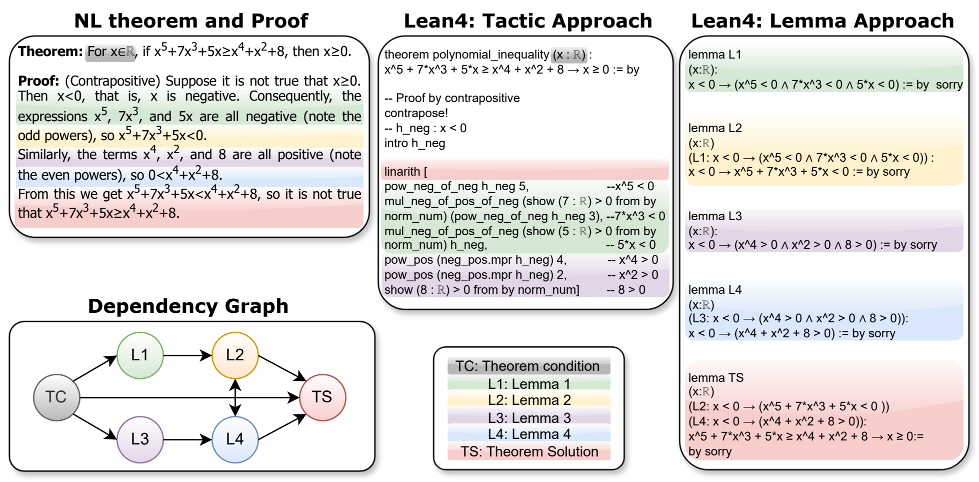
Proof autoformalization, the task of translating natural language theorems and proofs into machine-verifiable code, is a critical step for integrating large language models into rigorous mathematical workflows. Current approaches focus on producing executable code, but they frequently fail to preserve the semantic meaning and logical structure of the original human-written argument. To address this, we introduce PROOFFLOW, a novel pipeline that treats structural fidelity as a primary objective. PROOFFLOW first constructs a directed acyclic graph (DAG) to map the logical dependencies between proof steps. Then, it employs a novel lemma-based approach to systematically formalize each step as an intermediate lemma, preserving the logical structure of the original argument. To facilitate evaluation, we present a new benchmark of 184 undergraduate-level problems, manually annotated with step-by-step solutions and logical dependency graphs, and introduce PROOFSCORE, a new composite metric to evaluate syntactic correctness, semantic faithfulness, and structural fidelity. Experimental results show our pipeline sets a new state-of-the-art for autoformalization, achieving a PROOFSCORE of 0.545, substantially exceeding baselines like full-proof formalization (0.279), which processes the entire proof at once, and step-proof formalization (0.046), which handles each step independently. Our pipeline, benchmark, and score metric are open-sourced to encourage further progress at `https://anonymous.4open.science/r/ProofFlow-351E`.

1 INTRODUCTION

The effort to automate mathematical reasoning is advancing on two key fronts. Recent advances in large language models (LLMs) have greatly enhanced their ability to solve mathematical problems (Liang et al., 2025). Meanwhile, symbolic engines such as Lean (Moura & Ullrich, 2021) and Isabelle (Hales et al., 2017) provide machine-verifiable frameworks that enforce strict logical correctness. LLM-based automated theorem provers (ATPs), such as Goedel-Prover (Lin et al., 2025) and Kimina-Prover (Wang et al., 2025a), generate formal proofs for problems that are written in a symbolic language, which are then checked by the symbolic engine to ensure logical correctness.

This paper studies automated proof formalization: the task of faithfully translating the natural language theorem and proof into a machine-verifiable formal representation. This task is distinct from the aforementioned ATPs, where the goal is to discover a proof from an already formalized problem. Here, the goal is to translate an existing proof, a crucial step for verification in real-world mathematical workflows. After composing a proof, a mathematician may wish to formalize it to verify its correctness, uncover missing assumptions, or fill logical gaps. Given the steep learning curve of formal languages like Lean, a system that can automate this translation is highly desirable. The manual effort these automated proof formalizers seeks to automate is immense, as demonstrated by landmark efforts such as the Flyspeck project, which took over 20 years to formalize the Kepler conjecture (Hales et al., 2017); the Blue-Diamond project for the Polynomial Freiman-Ruzsa conjecture (Tao, 2023); and the ongoing formalization of Fermat's Last Theorem (Buzzard & Taylor, 2025), among other projects (Math Inc., 2025; Scholze, 2021; Gonthier et al., 2013).

A key challenge in autoformalization is the fundamental mismatch between the flexible and high-level nature of human language and the rigid, low-level syntax of formal systems. To bridge this gap, ATPs typically generate "tactics" for formal proof assistants like Lean 4. However, this tactic-



Theorem: For $x \in \mathbb{R}$, if $x^5+7x^3+5x \geq x^4+x^2+8$, then $x \geq 0$.

Proof: (Contrapositive) Suppose it is not true that $x \geq 0$. Then $x<0$, that is, x is negative. Consequently, the expressions x^5, $7x^3$, and $5x$ are all negative (note the odd powers), so $x^5+7x^3+5x<0$. Similarly, the terms x^4, x^2, and 8 are all positive (note the even powers), so $0<x^4+x^2+8$. From this we get $x^5+7x^3+5x<x^4+x^2+8$, so it is not true that $x^5+7x^3+5x \geq x^4+x^2+8$.

```
theorem polynomial_inequality (x : ℝ) :
x^5 + 7*x^3 + 5*x ≥ x^4 + x^2 + 8 → x ≥ 0 := by

-- Proof by contrapositive
contrapose!
-- h_neg : x < 0
intro h_neg

linarith [
pow_neg_of_neg h_neg 5,                              --x^5 < 0
mul_neg_of_pos_of_neg (show (7 : ℝ) > 0 from by
norm_num) (pow_neg_of_neg h_neg 3),  --7*x^3 < 0
mul_neg_of_pos_of_neg (show (5 : ℝ) > 0 from by
norm_num) h_neg,                                      -- 5*x < 0
pow_pos (neg_pos.mpr h_neg) 4,                -- x^4 > 0
pow_pos (neg_pos.mpr h_neg) 2,                -- x^2 > 0
show (8 : ℝ) > 0 from by norm_num]          -- 8 > 0
```

```
lemma L1
(x:ℝ):
x < 0 → (x^5 < 0 ∧ 7*x^3 < 0 ∧ 5*x < 0) := by  sorry

lemma L2
(x:ℝ)
(L1: x < 0 → (x^5 < 0 ∧ 7*x^3 < 0 ∧ 5*x < 0)) :
x < 0 → x^5 + 7*x^3 + 5*x < 0 := by sorry

lemma L3
(x:ℝ):
x < 0 → (x^4 > 0 ∧ x^2 > 0 ∧ 8 > 0) := by sorry

lemma L4
(x:ℝ)
(L3: x < 0 → (x^4 > 0 ∧ x^2 > 0 ∧ 8 > 0)):
x < 0 → (x^4 + x^2 + 8 > 0) := by sorry

lemma TS
(x:ℝ)
(L2: x < 0 → (x^5 + 7*x^3 + 5*x < 0 ))
(L4: x < 0 → (x^4 + x^2 + 8 > 0)):
x^5 + 7*x^3 + 5*x ≥ x^4 + x^2 + 8 → x ≥ 0:=
by sorry
```

TC: Theorem condition
L1: Lemma 1
L2: Lemma 2
L3: Lemma 3
L4: Lemma 4
TS: Theorem Solution

Figure 1: Comparison of our PROOFFLOW "Lemma Approach" and the common "Tactic Approach" in formalizing a natural language (NL) theorem and its proof into Lean 4 code. The Lemma Approach directly mirrors the sequence of steps and their dependencies in the NL proof. By contrast, the Tactic Approach produces tactics that fail to adhere to the structure of the initial NL proof.

based approach often fails to preserve the logical structure of the original human argument. For instance, as shown in Figure 1, under "Lean 4: Tactic Approach", the generated tactics do not follow the sequence of steps in the natural language proof. Furthermore, a single tactic, *linarith*, consolidates three distinct steps from the original proof (TS, L2, and L4). This disparity creates two major issues. First, not all natural language mathematical expressions can be directly translated into low-level tactics, which can cause the formalization process to fail. Second, even if an automated system generates a verifiable proof, it may take shortcuts or skip intermediate steps, arriving at the correct conclusion without mirroring the explicit, step-by-step reasoning of the original proof. This makes it difficult to verify that the formal proof truly captures the human's intended logic.

To address these issues and ensure faithful autoformalization, we propose a novel approach that avoids direct translation into limited, low-level formal tactics. Instead, we deconstruct the natural language proof into a sequence of structured, high-level lemmas, as illustrated by the "Lemma Approach" in Fig. 1. The key advantage of this method is that it enables us to explicitly follow the sequence of steps in the original proof with low friction[1]. This stands in sharp contrast to the conventional tactic-based approach, which is often convoluted and hinders mirroring the human proof structure. Second, our lemma-based approach preserves the logical structure of the natural language proof by explicitly defining dependencies. As shown in Figure 1, the final step, lemma TS (the theorem's solution), depends only on lemma L2 and lemma L4. This explicit dependency management is crucial in enforcing structurally faithful formalization, which is often not achieved in a standard tactic-based workflow. For instance, a common tactic like "have" in Lean 4 feeds the entire preceding context to each new step, not just the specific premises required by the original proof logic. In contrast, our approach of explicitly defining how lemmas depend on one another prevents the system from incorrectly using unintended dependencies, a common failure in tactic-based proving. These two innovations, high-level lemma generation and explicit dependency tracking, form the core of our new autoformalization pipeline, PROOFFLOW.

Another primary challenge is defining what constitutes a "faithful" formalization. In existing work, researchers either focus on syntactic correctness (Hu et al., 2025), i.e., no compilation errors, or use a simple BLEU score for semantic measurement (Poiroux et al., 2024; Wu et al., 2022), while ignoring the structural fidelity of the proof. To properly evaluate proof formalizations, we propose viewing a proof not as a monolithic block of text, but as a structured sequence of theorem condi-

[1]While we currently use the placeholder "by sorry" for unproven parts, our ultimate goal is to generate Lean tactics to prove each lemma and the final solution.

tions, definitions, and lemmas that form a logical progression toward the final theorem solution or solutions (see Figure 1). Based on this, we introduce a new and more comprehensive proof autoformalization scoring metric (PROOFSCORE), and we also address the lack of an advanced benchmark by providing a new university-level dataset tailored for this task (PROOFFLOWBENCH).

Drawing on our new framework, we make the following key contributions:

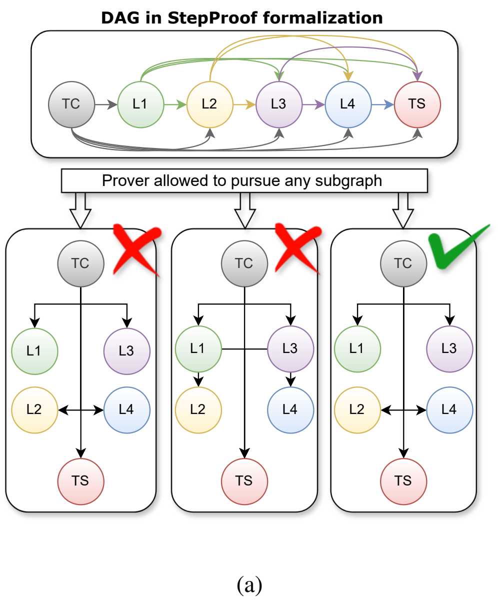
- PROOFFLOW: In Section 3, we propose a novel pipeline for translating natural language theorems and proofs into structured and formal Lean code, ensuring the preservation of the proof's logical structure. When a formalization step fails, the pipeline identifies the error source, be it in the formalization, the tactic completion process, or the initial NL statement, thereby alerting mathematicians of potential flaws in their original proof.
- PROOFSCORE: Section 4 introduces a new and comprehensive scoring method to evaluate the quality of autoformalized proofs. This metric is the first to explicitly measure syntactic correctness, semantic faithfulness, and structural fidelity, providing a more complete assessment than existing methods.
- PROOFFLOWBENCH: In Section 5, we present a new, manually curated benchmark dataset for proof autoformalization, containing a collection of 184 undergraduate level problems.
- Comparative Study: Section 6 presents an empirical study using state-of-the-art models to evaluate our structure-aware pipeline, PROOFFLOW, against alternative strategies. The results show that PROOFFLOW has significantly higher proof autoformalization quality.

2 BACKGROUND AND RELATED WORK

Proof assistants and automatic theorem proving: Proof assistants like Isabelle (Paulson, 1994), Lean 4 (Moura & Ullrich, 2021), and Coq (Barras et al., 1997) are software environments for developing and verifying mathematical proofs. Proofs constructed within these systems are what we call "formal" proofs, distinguishing them from informal or natural language proofs written in standard mathematics (e.g., in LaTeX). The user's workflow involves interactively applying tactics, which are small programs that perform logical inferences, to solve the theorem's goals (Jiang et al., 2024). Despite their power, a steep learning curve and the significant manual effort required by their rigid syntaxes have hindered widespread adoption (Zhou et al., 2024). Recently, Large Language Models (LLMs) have emerged as powerful automated theorem provers (ATPs) capable of generating complete formal proofs from already formalized theorem statements (Shang et al., 2025; Lin et al., 2025; Wang et al., 2025b; Ren et al., 2025b). Proof Agents have further advanced this ability (Chen et al., 2025; Zhou et al., 2025; Baba et al., 2025). In the context of Figure 1, the task of an ATP is to automatically replace the placeholder command "sorry" with formal tactics to complete the proof.

Autoformalization with LLMs: For a fully automatic theorem-proving system, mathematical problems originating in natural language must first be translated into a formal language, a process known as autoformalization. Historically, autoformalization efforts have primarily focused on translating theorem statements, and not the natural language proof (Huang et al., 2025; Wu et al., 2025; Liu et al., 2025; Yu et al., 2025b; Poiroux et al., 2024; Pathak, 2024; Patel et al., 2023), often to support the training of automated provers (Lin et al., 2025; Wang et al., 2025b). A different approach uses informal proof sketches to guide an LLM's search for a formal proof (Cao et al., 2025; Zhou et al., 2024; Jiang et al., 2023). In these methods, the natural language proof sketch is not the target of formalization itself. Rather, it serves as a high-level guide, often interleaved as comments within the formal code to steer the generation process. This guidance technique is also employed by recent ATPs like DeepseekProver-V2 (Ren et al., 2025b) and Goedel Prover V2 (Lin et al., 2025).

Proof Autoformalization: In the literature of proof autoformalization, most existing attempts directly translate entire proofs using LLMs (Gao et al., 2024; Lu et al., 2024; Cunningham et al., 2023). This approach, however, remains highly challenging due to frequent syntactic errors and it often produces outputs that are not semantically trustworthy. Step-level formalization, which involves solving the autoformalization step-by-step based on the proof's logical steps, has been explored by Hu et al. (2025) in Isabelle. However, this approach suffers from two limitations. First, they treated all previous proof steps as valid premises for the current step, a significant simplification that overlooks the proper logical structure. Second, their evaluations focused solely on syntactic correctness while overlooking semantic consistency.


DAG in StepProof formalization
TC
L1
L2
L3
L4
TS
Prover allowed to pursue any subgraph
(a)

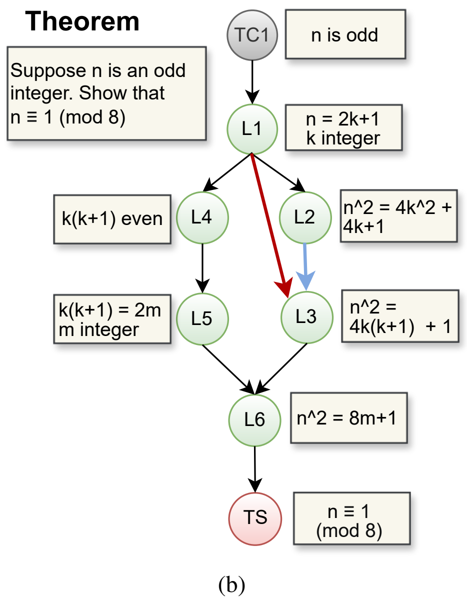


Figure 2: Comparing structural fidelity in automated proof generation. (a) A scenario for the problem in Figure 1, where the dependency graph fails to adhere to the structure of the original proof. (b) This problem was intentionally formalized without enforcing a DAG. This approach sacrificed structural fidelity, by reusing lemma L1 to prove L3, thereby rendering L2 redundant.

3 Proof Autoformalization

Although most prior work formalizes entire proofs at once, a notable exception is the "Step Proof" method (Hu et al., 2025). This approach assumes each proof step depends on all preceding steps, a simplification that creates an unfaithful dependency structure and can lead to an ATP taking a "shortcut." This shortcut may involve using only the initial theorem conditions or an incorrect subset of lemmas to construct a valid proof that does not follow the logic of the original NL proof, as depicted in Figure 2a. A clear example found in ProofFlowBench is illustrated in Figure 2b. In the natural language proof, step L3 directly uses the outcome of L2, which is faithfully followed by our DAG-enforcing pipeline. By contrast, an ablated version of our pipeline which lacks the mechanism to enforce the correct dependencies (noDAG) reuses the outcome of L1 to prove L3, despite having proven L2. Thus, it not only dissipates step L2 but also disregards the structure of the original proof. More details and examples are found in Appendix A.5.

If the sole objective is to find any correct formal proof for a theorem, which is often the sole goal of ATPs, these behaviors are not problematic. However, for proof autoformalization, they represent a critical failure of faithfulness. Our goal is to ensure that each step is proven only taking into account the correct set of previous lemmas and theorem conditions specified by the original proof's logic. Enforcing this correct dependency structure offers significant practical advantages as well. It improves efficiency by constraining the autoformalizer's search space, which can lead to fewer spent tokens and faster verification.

3.1 New workflow for proof autoformalization

As shown in Figure 3, our method enforces a correct dependency graph through a three-stage pipeline, leveraging LLMs at each step to bridge the gap between natural language (NL) and formal proof code. The first stage, Graph Builder, constructs a dependency directed acyclic graph (DAG) from the original proof. Next, the Formalizer uses an LLM to translate each proof step into formal Lean code. Finally, the Tactic Completer fills in the necessary tactics to complete the Lean proof. The specific LLM models used at each stage are detailed in Section 6.

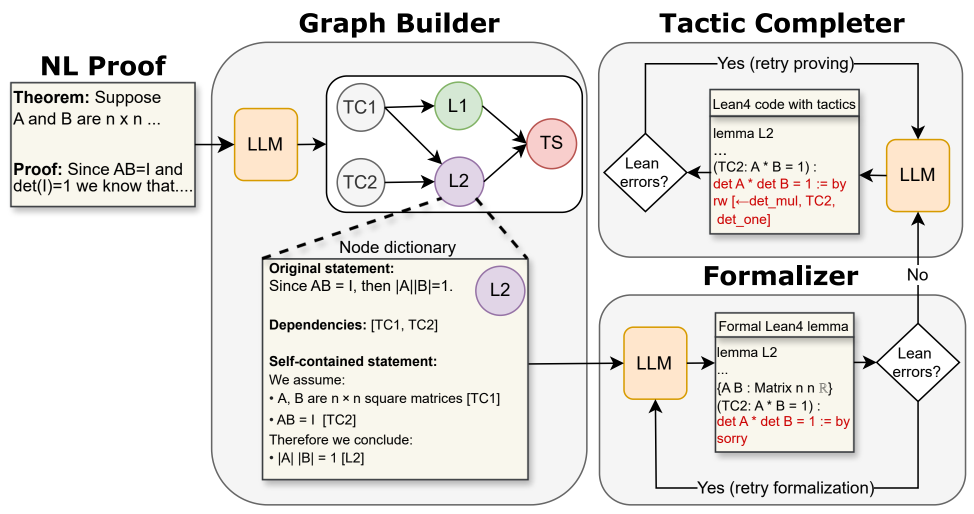

NL Proof
Theorem: Suppose A and B are n x n ...
Proof: Since AB=I and det(I)=1 we know that....
Graph Builder
LLM
TC1
TC2
L1
L2
TS
Node dictionary
Original statement: Since AB = I, then |A||B|=1.
Dependencies: [TC1, TC2]
Self-contained statement: We assume: • A, B are n × n square matrices [TC1] • AB = I [TC2] Therefore we conclude: • |A| |B| = 1 [L2]
Tactic Completer
Yes (retry proving)
Lean4 code with tactics
lemma L2 ... (TC2: A * B = 1) : det A * det B = 1 := by rw [←det_mul, TC2, det_one]
Lean errors?
No
Formalizer
Formal Lean4 lemma
lemma L2 ... {A B : Matrix n n ℝ} (TC2: A * B = 1) : det A * det B = 1 := by sorry
Yes (retry formalization)

Figure 3: Our proof autoformalization pipeline with three parts: (1) Graph builder; (2) Lemma Formalizer; and (3) Tactic Completer. Lean errors are verified by the Lean 4 compiler.

1. Graph Builder: This step parses the natural language (NL) theorem and its proof to construct a dependency graph with LLM. Formally, this graph is a DAG, $G = (V, E)$, where V is the set of all nodes and $E \subseteq V \times V$ is the set of directed edges. The nodes are partitioned into disjoint sets: $V = V_{TC} \cup V_D \cup V_L \cup V_{TS}$, representing Theorem Conditions, Definitions, Lemmas, and Theorem Solutions, respectively. Each edge $(u, v) \in E$ signifies that node u is a prerequisite for proving the statement of node v. From the theorem statement, we extract nodes for theorem conditions and theorem solutions. From the proof statement, we extract nodes for lemmas and extra definitions. Each node is assigned its original NL statement, its dependencies, and a self-contained NL statement that provides a complete description of the current proof step (see Figure 3). To ensure the graph's validity, the system checks for forward references and cycles and verifies that every node, except the theorem solutions, has an outgoing edge. If the check fails, we task the LLM to improve the graph.

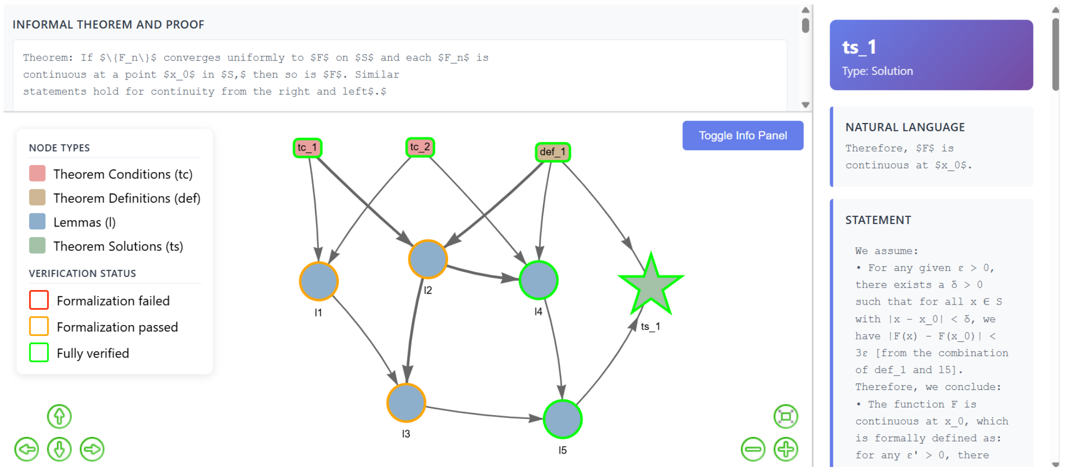
2. Formalizer: For each node in the graph, the LLM formalizes its self-contained NL statement into Lean 4 code. This process is iterative: generated errors are fed back to the LLM for correction. At this stage, each lemma is finalized with the "by sorry" placeholder, as tactics are not yet applied.

3. Tactic Completer: The final step completes the proofs for the lemmas by replacing the "by sorry" placeholders in the formalized Lean 4 code with the appropriate Lean 4 tactics.

To streamline the entire workflow illustrated in Figure 3, we developed a user-friendly Python package for PROOFFLOW. This package automates the complete process, allowing users to provide an informal theorem and proof with just a few lines of code. The package then autonomously executes the full workflow. It also integrates PROOFSCORE evaluation and our PROOFFLOWBENCH benchmark for comprehensive assessment. The package is available [2] at `https://anonymous.4open.science/r/ProofFlow-351E`, with all the LLM prompts used in this project included in the repository.

The final output of our package is an interactive diagram that visualizes the proof graph, as seen in Figure 4. Users can click on each node to get detailed information about its results. The contours of each node provide an at-a-glance summary of the outcomes, indicating whether its formalization and solving steps were successful. This allows users to immediately assess the progress of the autoformalization and pinpoint the locations in the proof graph where manual effort is needed.

[2] A public GitHub repository containing the package will be made available upon paper acceptance.



```
Theorem: If $\{F_n\}$ converges uniformly to $F$ on $S$ and each $F_n$ is
continuous at a point $x_0$ in $S,$ then so is $F$. Similar
statements hold for continuity from the right and left$.$
```

```
Therefore, $F$ is
continuous at $x_0$.
```

```
We assume:
• For any given ε > 0,
there exists a δ > 0
such that for all x ∈ S
with |x - x_0| < δ, we
have |F(x) - F(x_0)| <
3ε [from the combination
of def_1 and l5].
Therefore, we conclude:
• The function F is
continuous at x_0, which
is formally defined as:
for any ε' > 0, there
```

Figure 4: An example of the interactive visualization generated by PROOFFLOW. Node contours signify the outcome of each step: Red for a formalization error, orange for formalized statement without Lean 4 tactics, and green for formalized statement with Lean 4 tactics.

4 SCORING PROOF AUTOFORMALIZATION AND ERROR DETECTION

As discussed in Section 1, a faithful autoformalization must satisfy three key properties: (1) Structural Fidelity, which ensures the proof's dependency graph is preserved; (2) Syntactic Correctness, which ensures the output is verifiable code without compilation errors; and (3) Semantic Faithfulness, which ensures each formalized statement accurately preserves the precise mathematical meaning of its original natural language statement.

4.1 PROOFSCORE

To evaluate the effectiveness of our PROOFFLOW pipeline, we introduce PROOFSCORE, a single unified score that synthesizes these three criteria.

Structural Fidelity is evaluated by checking whether the dependencies for a given node are valid. For a node v_i, we check if its estimated dependencies, $D_{\text{est}}(v_i) = \{u \in V_{\text{est}} \mid (u, v_i) \in E_{\text{est}}\}$, match the dependencies in a ground truth graph. We permit several valid graphs ($\mathcal{G}_{\text{true}}$) because the level of granularity can vary. For example, the calculation $1 + 13 + 5 = 14 + 5 = 19$ could be broken down into either one or two steps, depending on the user's granularity preference.

Syntactic Correctness is denoted by $c_i \in \{0, 1\}$, where $c_i = 1$ if the formalization of node v_i is free of Lean 4 compilation errors at the end of the Tactic Completer step and $c_i = 0$ otherwise.

Semantic Faithfulness is assessed by adapting the "LeanScore" metric from Yu et al. (2025a), which was originally designed to evaluate the semantic faithfulness of a theorem statement formalization. A detailed description is given in Appendix A.2. This metric provides a faithfulness score, $f_i \in [0, 1]$, for each node v_i of the dependency graph, and this score measures the semantic equivalence between the input NL statement and its corresponding formalized Lean 4 lemma. The higher the score, the more faithful the Lean 4 lemma devotes to the input NL statement.

The final unified PROOFSCORE, for a proof with n steps is computed as:

$$\text{PROOFSCORE} = \frac{1}{n} \sum_{i}^{n} f_i \, c_i \, I[D_{\text{est}}(v_i) \in \mathcal{D}_{\text{true}}(v_i)],$$

where ProofScore $\in [0, 1]$, and $\mathcal{D}_{\text{true}}(v_i)$ is the set of valid dependencies for node step v_i. When the GraphBuilder is configured to enforce the DAG structure, structural fidelity is guaranteed for all nodes. Otherwise, we use the Claude-Sonnet-4 LLM with a specialized prompt to verify structural fidelity. This same LLM model is also used to check for semantic faithfulness, utilizing prompts available within the PROOFFLOW package. Additional details about PROOFSCORE, including ex-

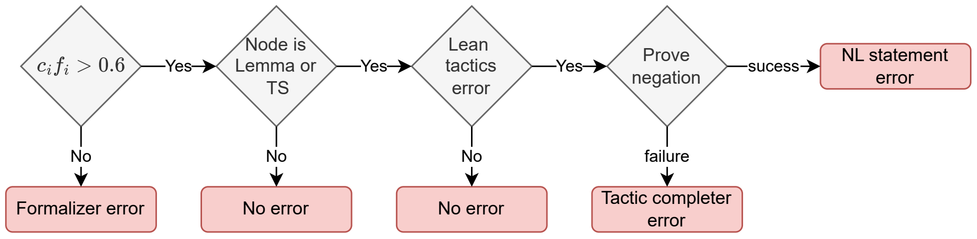


Figure 5: Flowchart illustrating the error detection mechanism. This is applied lemma by lemma and f_i is the semantic faithfulness of node v_i and $c_i = 0$ if there are syntactic errors.

amples of natural language proofs paired with their Lean code and respective scores, are provided in Appendix A.2.

4.2 Error analysis

ProofFlow also has an error detection system operating at each proof step. This process (Figure 5) determines the error source, which can be the formalizer, tactic completer, or original natural language proof. First, a proof node's semantic faithfulness score is compared against a 0.6 threshold, following Yu et al. (2025a). If it passes, the system checks if it's a provable statement (lemma or theorem solution). Otherwise, the process for that node ends. For provable statements, the tactic completer attempts to apply Lean 4 tactics. Success means the proof step is complete with no errors. If the tactics fail, the LLM tries to prove the negation of the statement. If the negation can be proven, then the original natural language statement is deemed imprecise. Otherwise, the error is considered to be with the tactic completer LLM, which is incapable of either proving or disproving the step.

5 ProofFlowBench

Existing mathematical benchmarks are often limited to pure calculation and are designed to test problem solving accuracy. Previous natural language proof datasets are also not self-contained, often referencing external sources for non-standard theorems (Welleck et al., 2021). Furthermore, they focus on specific topics and sometimes also including calculation problems instead of proofs (Sheng et al., 2025). To address these limitations, we introduce a new benchmark, ProofFlowBench, to specifically evaluate automated proof formalization pipelines. The benchmark, provided as part of the ProofFlow package, consists of 184 undergraduate-level mathematics theorems and proofs in natural language from six key areas: number theory and algebra (27), real analysis (42), inequality (36), probability and set theory (31), complex analysis (25), and sequences and series (23). To construct this dataset, we adapted 63 problems from the NaturalProofs benchmark (Welleck et al., 2021) and 36 problems from the IneqMath benchmark (Sheng et al., 2025).

The benchmark also contains the natural language proofs divided into proof steps, and the respective dependency graphs (DAGs), which can be used to evaluate structural fidelity. On average, each problem's graph consists of 8.4 total nodes (more statistics provided in Appendix A.3). The proof graphs have been manually validated. However, we emphasize that for the same proof different proof graphs are possible depending on the desired level of detail of each proof node.

6 Comparison study

To evaluate our ProofFlow pipeline, we conduct a comparison study with three main objectives. First, we assess the effectiveness of formalizing proofs using our high-level, lemma-based approach (ProofFlow) versus low-level tactics. This compares the "Lemma Formalization" (ours) with the "Tactic Formalization" (existing), as depicted in Figure 1. Secondly, to examine the role of explicit dependency management in ensuring structural fidelity to the original proof, we compare the DAG-enforcing version of ProofFlow with a variant where this is relaxed (noDAG). Finally, we conduct the error analysis of Section 4.2, to detect the source of errors in the ProofFlow pipeline variants.

6.1 Experimental Settings

ProofFlow: We consider the ProofFlow variant that explicitly enforces the correct dependency graph as our main method, referred to as ProofFlow DAG. To evaluate the role of this dependency enforcement, as an ablation study, we also consider a ProofFlow noDAG version, where such mechanism is relaxed. That is, in ProofFlow noDAG, all previous lemmas and premises are provided when formalizing each step, similarly to how dependency is handled by most prior work.

Existing methods: We compare our pipeline against existing tactic-based formalization methods: (1) "Full Proof" autoformalization (Lu et al., 2024), which calls the LLM once to formalize the entire theorem and proof using Lean 4 tactics, and (2) "Step Proof" autoformalization (Hu et al., 2025), which formalizes one proof step at a time into Lean 4 tactics. We note that Step Proof utilizes the same Graph Builder LLM and prompt as ProofFlow to decomponse the initial NL proof into steps.

Thinking modes: All methods were evaluated in both thinking and non-thinking modes to provide a more comprehensive assessment. For existing methods, Goedel-Prover-V2-32B and Gemini-Flash-2.5 (Gemini Team, 2025) are used for the thinking and non-thinking modes. For our ProofFlow pipeline, the Graph Builder always uses Gemini-2.5-Pro; the thinking mode utilizes Goedel-Formalizer-2-32B and Goedel-Prover-V2-32B (Lin et al., 2025) as the Formalizer and the Tactic Completer, while non-thinking mode utilizes Gemini-2.5-Flash and DeepSeekProver V2 671B (Ren et al., 2025a) for the two components. All LLM models were given appropriate system prompts.

Evaluation metrics: Our main evaluation metric in ProofScore, supplemented by measures of syntactic correctness, both at step-level and proof-level. Step-level syntactic correctness of the generated Lean 4 code is measured at two key stages: after the Formalizer step (Formalizer accuracy) and after the Tactic Completer step (Tactic accuracy). We also report proof-level "correct syntax", whether all of the formalizer and solver steps are syntactically valid. All syntactic correctness checks are verified by LeanServer (Santos et al., 2025) with Lean version v4.15.0. We also adapted ProofScore (Sect. 4), so it could evaluate the Full Proof and Step Proof pipelines (see Appendix A.2.2 for details). Finally, we report the elapsed time and the number of generated tokens. Since our experiments use both local and API-based model deployments, inference speeds vary. Therefore, the number of generated tokens is a more meaningful indicator of the effort required.

Data and reproducibility: We utilized the problems in the ProofFlowBench benchmark for this comparison study. Reproducible code is provided as part of the ProofFlow package.

6.2 Empirical Results

The results evaluated at a Pass@5 rate are presented in Table 1. This Pass@5 setting allows for up to five self-correction retries for each stage in the pipelines. If a generated step produces syntactically incorrect code, the model is re-prompted with the error and its previous output, giving it an opportunity to fix the mistake. The results for Pass@1 and 3 are show in Appendix A.4.

Existing Methods Comparison: The results clearly demonstrate that our pipeline outperforms existing methods (Full Proof and Step Proof), achieving the highest ProofScore (0.545). The Full Proof method calls the LLM once for the entire proof formalization and therefore we only report proof-level results. This approach achieves a comparatively high syntax passing rates (0.571 for the thinking mode), however, the ProofScore was comparatively low, with the most frequent errors being the misuse of unknown theorem names and tactic failures. The Step Proof method exhibits the lowest proof-level syntax passing rates (0.005 for the non-thinking mode and 0.119 for the thinking mode) because the LLM struggles to maintain correct and consistent indentation in Lean 4 across steps. Also, once an error appears in one step, subsequent steps cannot achieve syntactic correctness. The poor syntax passing rates of the Step Proof approach results in correspondingly low ProofScore values, given that this metric incorporates syntactic correctness as a component.

Ablation Study: The ProofScore evaluation metric establishes the DAG version of ProofFlow as the top-performing method for proof autoformalization. This is most evident in the "thinking" mode, where the DAG variant achieves a ProofScore of 0.545, compared to the noDAG's 0.417, due to improved structural fidelity and syntactic correctness. While the DAG also outperforms the noDAG variant for all syntax metrics in the thinking mode, we observed a higher syntax passing rate for the noDAG in the non thinking setting. This is because the noDAG variant provides all previ-

Table 1: Performance metrics under the Pass@5 setting on our 184-problem benchmark. Step-level averages are computed over all individual steps, while proof-level averages are computed per proof. Entries marked with "/" indicate not applicable.

Pipeline	Think mode	Step-Level Form. accuracy	Step-Level Tactic accuracy	Proof-Level Proof Score	Proof-Level Correct syntax	Proof-Level Time (mins)	Proof-Level Output tokens (k)
PROOFFLOW DAG	No	0.751	0.358	0.355	0.027	8.8	22.4
	Yes	**0.939**	**0.742**	**0.545**	0.375	31.8	94.2
PROOFFLOW noDAG	No	0.807	0.391	0.347	0.049	12.3	25.8
	Yes	0.936	0.681	0.417	0.353	32.0	98.5
FULL PROOF	No	/	/	0.021	0.027	0.8	10.5
	Yes	/	/	0.279	**0.571**	3.8	15.1
STEP PROOF	No	/	0.068	0.046	0.005	0.2	1.2
	Yes	/	0.129	0.029	0.119	10.6	32.9

ous steps as known conditions to the LLM, and this extra information can lead to higher syntactic passing rates for weaker models, but often results in a logically inconsistent formal proof structure, as shown by the lower PROOFSCORE averages. This finding underscores the critical importance of maintaining structural fidelity.

Computational efficiency: Appendix A.7 provides a detailed breakdown of computational costs across pipeline components. Our analysis reveals that the Tactic Completer stage is the primary computational bottleneck, consuming 81–89% of total execution time across all configurations. A straightforward solution to address this bottleneck, enabled by our DAG-based pipeline design, is parallelization. Our architecture naturally supports parallel execution at multiple levels: proof nodes in different branches can be formalized concurrently, and once formalized, tactic completion can proceed independently across nodes. In principle, our DAG design enables extensive parallelization that could substantially reduce wall-clock time. However, we did not implement parallelization in our experiments due to limited GPU resources, which restricted us to sequential processing.

6.3 ERROR ANALYSIS

The error analysis, shown in Table 2, identifies the Formalizer as the primary source of failure, accounting for 32% to 47% of all autoformalization outcomes, depending on the pipeline configuration. These failures are predominantly semantic, stemming from discrepancies between the natural language and the formalized Lean 4 code, despite the high syntactic correctness (Table 1). Our most robust configuration, which uses the PROOFFLOW DAG architecture in thinking mode, has 53.3% error-free proof steps and clearly outperforms the 42.8% rate of the baseline (noDAG) version. This confirms that while the DAG architecture improves performance, improving semantic preservation during the formalization stage remains a challenge for future work.

The PROOFFLOW's error detection process goes beyond simple checks, being able, in some instances, to identify a spectrum of flaws in the initial natural language proof. Examples are provided in Appendix A.6. These detected issues range from blatant algebraic errors to subtle ambiguities and unstated assumptions. By flagging these mistakes, pipelines like PROOFFLOW can provide feedback that not only helps correct errors but also strengthens the overall rigor and clarity of the proof.

7 DISCUSSION

Our results demonstrate the importance of preserving a proof's structural fidelity. The PROOFFLOW lemma-based approach significantly outperforms both monolithic ("FULL PROOF") and sequential ("STEP PROOF") methods, which falter by tackling excessive complexity at once or by imposing a

Table 2: Breakdown of step-level outcomes for different pipeline configurations. "None" indicates the percentage of total steps completed successfully, while other columns show the percentage of steps that failed due to a specific error source.

Pipeline	**Think**	**Total Steps**	**Error Source (%)**			
			None	**Formalizer**	**Tactic**	**NL Statement**
PROOFFLOW DAG	No	1735	33.7	46.5	19.6	0.2
PROOFFLOW DAG	Yes	1737	**53.3**	38.9	5.6	2.2
PROOFFLOW noDAG	No	1751	32.8	45.6	21.4	0.2
PROOFFLOW noDAG	Yes	1755	42.8	47.0	7.6	2.6

linear structure unfaithful to the proof logic. Our method succeeds by using a lemma-based structure that explicitly models the proof's dependency graph. This approach deconstructs the problem into manageable, logically-constrained steps. It guides the LLM along the author's intended path, preventing logical "shortcuts" that undermine other approaches. However, performance hinges on the Formalizer step, where poor semantic preservation remains the primary bottleneck.

A limitation of our work was the exclusive focus on undergraduate-level proofs, omitting research-level problems. However, this scope reflects the current capabilities of the field rather than intrinsic limitations of the pipeline architecture we propose. Testing our system on more challenging research-level problems did reveal the expected performance drop, consistent with observations across the literature (e.g., (Jiang et al., 2025)). Critically, our pipeline's core contributions—the DAG-based lemma decomposition and structural preservation—are not the primary limiting factors. The main bottlenecks lie in the underlying formalizer and tactic-completer LLMs, which currently struggle with research-level concepts, and in MathLib's incomplete coverage of specialized mathematical domains. As these foundational components mature and the formal mathematics ecosystem grows, we anticipate that our pipeline's performance will naturally improve, providing a scalable framework that is designed to grow more capable alongside these broader advances.

Our modular three-stage pipeline also enables the curation of high-quality training data for both end-to-end proof autoformalization models and individual Formalizer and Tactic Completer LLMs—a key direction for future work. Our error detection mechanisms can filter low-quality autoformalizations at different pipeline stages, creating curated datasets that support multiple training paradigms: supervised fine-tuning on the complete chain-of-thought reasoning process and reinforcement learning using detected errors as negative examples.

Ultimately, our work demonstrates that proof autoformalization is an achievable goal: we achieved 37.5% proof-level syntactic accuracy on undergraduate problems, a significant leap from existing approaches that only achieved 6.10% accuracy on simpler middle-school mathematics (Hu et al., 2025). More importantly, we establish that for these tools to be truly useful to mathematicians, they must faithfully represent the structure and logic of the human-created arguments they seek to formalize. Addressing the current bottlenecks could lead to practical tools that function as a proof "auto-correct" for mathematicians, pinpointing errors and ambiguities directly in natural language and in real time.

ETHICS STATEMENT

This research adheres to the ICLR Code of Ethics and does not involve human subjects, privacy concerns, or ethical issues. The study uses only publicly available mathematical data, with no personal or sensitive data involved.

The proposed PROOFFLOW pipeline, PROOFSCORE metric, code, and datasets are designed to assist mathematicians in theorem formalization, aiming to advance mathematical research and AI for mathematics. This work poses no foreseeable harmful applications and contributes positively to the scientific community by enhancing automated theorem proving capabilities.

REPRODUCIBILITY STATEMENT

To ensure the reproducibility of this work, we have made all necessary materials openly available. The complete implementation of the methodology (PROOFFLOW pipeline, PROOFSCORE metric), all experimental data, LLM prompts and comparison study code are publicly accessible through our package repository: `https://anonymous.4open.science/r/ProofFlow-351E`. The provided GitHub repository includes detailed instructions and documentation for replicating all experimental results presented in this paper.

REFERENCES

Kaito Baba, Chaoran Liu, Shuhei Kurita, and Akiyoshi Sannai. Prover agent: An agent-based framework for formal mathematical proofs. *arXiv preprint arXiv:2506.19923*, 2025.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Héctor Herbelin, Gérard Mohring, Amokrane Saïbi, and Benjamin Werner. The coq proof assistant reference manual: Version 6.1. Technical Report RT-0203, INRIA, 1997.

Kevin Buzzard and Richard Taylor. A lean proof of fermat's last theorem. Technical report, Technical report, Imperial College of London, 2024. https . . . , 2025.

Chenrui Cao, Liangcheng Song, Zenan Li, Xinyi Le, Xian Zhang, Hui Xue, and Fan Yang. Reviving dsp for advanced theorem proving in the era of reasoning models. *arXiv preprint arXiv:2506.11487*, 2025.

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.

Garett Cunningham, Razvan C Bunescu, and David Juedes. Towards autoformalization of mathematics and code correctness: Experiments with elementary proofs. *arXiv preprint arXiv:2301.02195*, 2023.

Guoxiong Gao, Yutong Wang, Jiedong Jiang, Qi Gao, Zihan Qin, Tianyi Xu, and Bin Dong. Herald: A natural language annotated lean 4 dataset. *arXiv preprint arXiv:2410.10878*, 2024.

Google Gemini Team. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025. URL `https://arxiv.org/abs/2507.06261`.

Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the Odd Order Theorem. *Lecture Notes in Computer Science*, 8332:163–179, 2013.

Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017.

Xiaolin Hu, Qinghua Zhou, Bogdan Grechuk, and Ivan Y. Tyukin. Stepproof: Step-by-step verification of natural language mathematical proofs, 2025. URL `https://arxiv.org/abs/2506.10558`.

Yanxing Huang, Xinling Jin, Sijie Liang, Peng Li, and Yang Liu. Formarl: Enhancing autoformalization with no labeled data. *arXiv preprint arXiv:2508.18914*, 2025.

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, 2023. URL `https://arxiv.org/abs/2210.12283`.

Albert Q. Jiang, Wenda Li, Mateja Jamnik, Sean B. Holden, and Lawrence C. Paulson. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL `https://openreview.net/forum?id=Vp0Zz0w2V1`.

Jiedong Jiang, Wanyi He, Yuefeng Wang, Guoxiong Gao, Yongle Hu, Jingting Wang, Nailing Guan, Peihao Wu, Chunbo Dai, Liang Xiao, and Bin Dong. Fate: A formal benchmark series for frontier algebra of multiple difficulty levels, 2025. URL `https://arxiv.org/abs/2511.02872`.

Zhenwen Liang, Linfeng Song, Yang Li, Tao Yang, Feng Zhang, Haitao Mi, and Dong Yu. Towards solving more challenging imo problems via decoupled reasoning and proving. *arXiv preprint arXiv:2507.06804*, 2025.

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, 2025. URL `https://arxiv.org/abs/2508.03613`.

Xiaoyang Liu, Kangjie Bao, Jiashuo Zhang, Yunqi Liu, Yuntian Liu, Yu Chen, Yang Jiao, and Tao Luo. Atlas: Autoformalizing theorems through lifting, augmentation, and synthesis of data. *arXiv preprint arXiv:2502.05567*, 2025.

Jianqiao Lu, Yingjia Wan, Zhengying Liu, Yinya Huang, Jing Xiong, Chengwu Liu, Jianhao Shen, Hui Jin, Jipeng Zhang, Haiming Wang, et al. Process-driven autoformalization in lean 4. *arXiv preprint arXiv:2406.01940*, 2024.

Math Inc. The strong prime number theorem. `https://github.com/math-inc/strongpnt`, 2025. Accessed: 2025-09-11.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pp. 625–635. Springer, 2021.

Nilay Patel, Rahul Saha, and Jeffrey Flanigan. A new approach towards autoformalization. *arXiv preprint arXiv:2310.07957*, 2023.

Shashank Pathak. Gflean: An autoformalisation framework for lean via gf. *arXiv preprint arXiv:2404.01234*, 2024.

Lawrence C. Paulson. Isabelle: A generic theorem prover. In Alan Bundy (ed.), *Proceedings of the 12th International Conference on Automated Deduction (CADE-12)*, volume 814 of *Lecture Notes in Computer Science*, pp. 37–41. Springer, 1994.

Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. Improving autoformalization using type checking. *arXiv preprint arXiv:2406.07222*, 2024.

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, 2025a. URL `https://arxiv.org/abs/2504.21801`.

ZZ Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia 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*, 2025b.

Marco Dos Santos, Haiming Wang, Hugues de Saxcé, Ran Wang, Mantas Baksys, Mert Unsal, Junqi Liu, Zhengying Liu, and Jia Li. Kimina lean server: Technical report, 2025. URL `https://arxiv.org/abs/2504.21230`.

Peter Scholze. The Liquid Tensor Experiment. Xena Blog, 2021. Available at: `https://xenaproject.wordpress.com/2021/05/18/the-liquid-tensor-experiment/`.

Shijie Shang, Ruosi Wan, Yue Peng, Yutong Wu, Xiong-hui Chen, Jie Yan, and Xiangyu Zhang. Stepfun-prover preview: Let's think and verify step by step. *arXiv preprint arXiv:2507.20199*, 2025.

Jiayi Sheng, Luna Lyu, Jikai Jin, Tony Xia, Alex Gu, James Zou, and Pan Lu. Solving inequality proofs with large language models. *arXiv preprint arXiv:2506.07927*, 2025.

Michio Sugeno. Theory of fuzzy integrals and its applications. *Doctoral Thesis, Tokyo Institute of Technology*, 1974.

Terence Tao. Formalizing the proof of PFR in Lean4 using blueprint: a short tour. What's new Blog, 2023. Available at: https://terrytao.wordpress.com/2023/11/18/formalizing-the-proof-of-pfr-in-lean4-using-blueprint-a-short-tour/.

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. 2025a. URL http://arxiv.org/abs/2504.11354.

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*, 2025b.

Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. Naturalproofs: Mathematical theorem proving in natural language, 2021. URL https://arxiv.org/abs/2104.01112.

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.

Yutong Wu, Di Huang, Ruosi Wan, Yue Peng, Shijie Shang, Chenrui Cao, Lei Qi, Rui Zhang, Zidong Du, Jie Yan, et al. Stepfun-formalizer: Unlocking the autoformalization potential of llms through knowledge-reasoning fusion. *arXiv preprint arXiv:2508.04440*, 2025.

Xuejun Yu, Jianyuan Zhong, Zijin Feng, Pengyi Zhai, Roozbeh Yousefzadeh, Wei Chong Ng, Haoxiong Liu, Ziyi Shou, Jing Xiong, Yudong Zhou, et al. Mathesis: Towards formal theorem proving from natural languages. *arXiv preprint arXiv:2506.07047*, 2025a.

Zhouliang Yu, Ruotian Peng, Keyi Ding, Yizhe Li, Zhongyuan Peng, Minghao Liu, Yifan Zhang, Zheng Yuan, Huajian Xin, Wenhao Huang, et al. Formalmath: Benchmarking formal mathematical reasoning of large language models. *arXiv preprint arXiv:2505.02735*, 2025b.

Jin Peng Zhou, Charles Staats, Wenda Li, Christian Szegedy, Kilian Q. Weinberger, and Yuhuai Wu. Don't trust: Verify – grounding llm quantitative reasoning with autoformalization, 2024. URL https://arxiv.org/abs/2403.18120.

Yichi Zhou, Jianqiu Zhao, Yongxin Zhang, Bohan Wang, Siran Wang, Luoxin Chen, Jiahui Wang, Haowei Chen, Allan Jie, Xinbo Zhang, et al. Solving formal math problems by decomposition and iterative reflection. *arXiv preprint arXiv:2507.15225*, 2025.

A APPENDIX

A.1 LLM USAGE STATEMENT

In this work, Large Language Models (LLMs) were employed in specific components of our research pipeline and experimental framework. The primary applications of LLMs include:

1. **PROOFFLOW Pipeline Implementation**: LLMs were utilized as core components within our proposed ProofFlow pipeline as Graph Builder, Formalizer and Tactic Completer.

2. **Existing Methods Comparison**: During the experimental evaluation, LLMs were employed to perform inference with existing methods for comparative analysis against our proposed approach.

3. **PROOFSCORE Computation**: LLMs were integrated into the calculation process of our proposed PROOFSCORE metric for quantitative assessment of formalization quality.

4. **Benchmark Dataset Construction**: The proof graphs included in our benchmark dataset were initially generated by LLMs, followed by human verification.

LLMs served as computational tools in these specific applications and for grammatical correction of an early draft. All research ideation, methodological design, analysis, and interpretation of results were conducted exclusively by the human authors, who take full responsibility for the content of this paper, including any LLM-assisted components.

A.2 ADDITIONAL DETAILS ON PROOFSCORE

A.2.1 EVALUATION OF SEMANTIC EQUIVALENCE OF PROOFSCORE

In this section, we detail how to obtain the semantic equivalence score f_i of each node, as proposed in Section 4. Note that by default, the score f_i is 0 if the syntactic check at the formalizer step is not passed.

In particular, we assess the semantic equivalence between the self-contained natural language statement nls_i and Lean code lc_i of each node, which are the input and output of the Formalizer step of PROOFFLOW (Sect. 3.1). The process is as follows:

1. First, we prompt LLM to break down both nls_i and lc_i into components.
2. We employ an LLM-as-a-judge to evaluate the semantic equivalence between each formalized component, lc_i, and its natural language counterpart, nls_i. The LLM is provided with few-shot examples illustrating specific error types and assigns one of the following evaluations to each component:
   - *Perfectly match*: The component lc_i fully captures the reasoning of nls_i. It may include additional constraints or conditions that were implicit in the natural language but are necessary for formal rigor.
   - *Minor inconsistency*: The component lc_i correctly represents the core logic of nls_i but may feature slight structural reordering or other small deviations.
   - *Major inconsistency*: The component lc_i either omits a key part of the logic from nls_i or introduces entirely different reasoning.
3. A Fuzzy measurement score is then computed for each component based on these evaluations, following the method in Yu et al. (2025a). This score is effectively a weighted average of the counts for "Perfectly match" and "Minor inconsistency," with zero tolerance for any component rated as a "Major inconsistency."
4. The final score, f_i, is calculated by aggregating the Fuzzy measurements using a Sugeno Integral (Sugeno, 1974).

In brief, the score f_i of a proof step is equal to 1 if the Lean code fully captures the self-contained natural language statement with all necessary conditions and f_i is equal to 0 if at least one component of the Lean code contains major flaws.

A.2.2 ProofScore for Full Proof and Step Proof

Our evaluation score, ProofScore, is the average of faithfulness evaluations of individual steps (nodes in the dependency graph) of our ProofFlow method. To create a fair comparison with Full Proof and Step Proof, we adapted this evaluation metric as follows. In all cases, any syntactically incorrect Lean code was automatically assigned a score of 0.

- For Full Proof: Since this method generates a single, complete proof for each problem, we applied the scoring process (detailed in Appendix A.2.1) to the entire proof. The final score in Tables 1 and 6 is the average score across all problems. Note that if the proof fails Lean syntactic check, the score is 0 by default.
- For Step Proof: This method generates code for each individual step. Therefore, we applied the scoring process to each step independently. The final score in the tables is the average across all individual steps from every problem in the benchmark. If any step fails Lean syntactic check, the score is 0 by default.

The results of this comparative analysis are presented in Tables 1 and 6.

A.2.3 Reliability and Consistency of ProofScore

As Claude is not involved in any module of the pipeline (Graph builder, Formalizer and Tactic Completer), we employ Claude-Sonnet-4 to evaluate the semantic faithfulness in ProofScore.

To further validate the reliability of ProofScore, we compare it with 2 variants (using Gemini-2.5-Flash and DeepSeek-V3 in lieu of Claude) and merely Claude-as-a-Judge. While the variants of ProofScore evaluate faithfulness on a component basis and aggregate using Sugeno Integral (see Sect. 4), for Claude-as-a-Judge, we prompt it to evaluate directly whether the Lean code is semantically equivalent to the natural language proof step.

We employ these methods to evaluate a sample of 200 natural language proof steps from our benchmark and check their formalization faithfullness evaluations against the results obtained by a human expert. For human evaluations, the ground-truth label is “PASS” if the Lean code is semantically equivalent to natural language and “FAIL” otherwise. Consistently with Section 6, for variants of ProofScore, a proof score that is larger or equal to 0.6 means the scorer method predicts “PASS”, and the scorer predicts “FAIL” otherwise. On the other hand, Claude-as-a-Judge predicts “PASS”/“FAIL” directly. We compute the F1 scores of these predictions versus the ground-truth labels.

The results are summarized in Table 3. While the variants of ProofScore obtain close performances, they all achieve higher scores in judging semantic equivalence than Claude-as-a-Judge. These results suggest that ProofScore is reliable in judging semantic faithfulness and this reliability is consistent across different Large Language Models.

Table 3: Comparison of F1 scores of variants ProofScore by Claude, Gemini and DeepSeek and Claude-as-a-Judge against human expert evaluations. All variants of ProofScore achieve a higher level of correctness in judging semantic equivalence than Claude, demonstrating the reliability and consistency of ProofScore.

Method	**Claude-as-a-Judge**	**ProofScore Claude (paper)**	**ProofScore Gemini**	**ProofScore DeepSeek**
F1-score	0.83	0.91	0.88	0.9

A.2.4 Examples of ProofScore results

To illustrate how ProofScore behaves for each proof node, we provide several examples in Tables 4 and 5. Each example shows a natural language proof step paired with its formalized Lean code and the corresponding ProofScore. Note that we display each proof step’s individual contribution to the overall ProofScore from Section 4—that is, the product of terms within the summation (each term in the sum).

In short, any Lean code is assigned a score of 0 if either the code is syntactically incorrect or the tactics are structurally unfaithful to the proof step or there is at least one component that is significantly inconsistent with the natural language. By contrast, score 1 means that all components of the Lean code match perfectly with their counterparts in the natural language proof step.

Natural Language	Lean code	Score and Explanation
From $n = 2k+1$ [l1] and $n^2 = 4k^2+4k+1$ [l2], we conclude: $n^2 = 4k(k+1)+1$ [l3]. (Example highlighted in Figure 2(b)).	`lemma l3` `(n : Z)` `(tc_1 : n % 2 = 1)` `(l1 : $\exists$ k : Z, n = 2*k + 1)` `(l2 : n^2 = 4 * (Classical.choose l1)^2 + 4 * (Classical.choose l1) + 1) :` `n^2 = 4 * (Classical.choose l1) * ((Classical.choose l1) + 1) + 1 := by` `have h : n = 2 * (Classical.choose l1) + 1 := by` `have h1 : n = 2 * (Classical.choose l1) + 1 := by` `exact Classical.choose_spec l1` `exact h1` `have h3 : n ^ 2 = 4 * (Classical.choose l1) ^ 2 + 4 * (Classical.choose l1) + 1 := by` `have h4 : n ^ 2 = 4 * (Classical.choose l1) * ((Classical.choose l1) + 1) + 1 := by` `have h5 : 4 * (Classical.choose l1) ^ 2 + 4 * (Classical.choose l1) + 1 = 4 * (Classical.choose l1) * ((Classical.choose l1) + 1) + 1 := by` `ring_nf` `linarith [h3, h5]` `exact h4`	Score 0: structurally unfaithful. The code to prove L3 only utilizes header by L1 while ignoring L2, structurally unfaithful to the input NL proof.
We have: $(x-2)^2+(y-1)^2 = 5$ and $(x-4)^2+(y-1)^2 = 5$. So, $x = 3$.	`theorem l3` `(x y : R)` `(l1 : (x - 2)^2 + (y - 1)^2 = 5)` `(l2 : (x - 4)^2 + (y - 1)^2 = 5) :` `x = 3 = by sorry`	Score 0: wrong syntax. It should be ":= by sorry" but "= by sorry" was used. As the code fails Lean check (syntax error), the PROOFSCORE is 0.

Table 4: Examples of varying PROOFSCORE with natural language proof step and Lean code.

A.3 BENCHMARK DATASET STATISTICS

The distribution of the number of nodes per proof in the PROOFFLOWBENCH benchmark is shown in Figure 6. On average, the proofs have 2 theorem conditions, 0.6 definitions, 4.4 lemmas, and 1.2 theorem solutions.

Natural Language	Lean code	Explanation
Because $(a+b-c), (b+c-a), (c+a-b) > 0$ and we have: $a^2 \geq (a+c-b)(a+b-c), b^2 \geq (b+c-a)(b+a-c)$, and $c^2 \geq (c+b-a)(c+a-b)$. Thus: $a^2b^2c^2 \geq (a+c-b)(a+b-c)(b+c-a)(b+a-c)(c+b-a)(c+a-b)$.	`theorem l6 (a b c : R) (tc_1 : a > 0 ^ b > 0 ^ c > 0 ^ a + b > c ^ b + c > a ^ c + a > b) (l4 : (a + b - c > 0) ^ (b + c - a > 0) ^ (c + a - b > 0)) (l5 : a^2 >= (a + c - b) * (a + b - c) ^ b^2 >= (b + c - a) * (b + a - c) ^ c^2 >= (c + b - a) & (c + a - b)) : a^2 * b^2 * c^2 > (a + c - b) * (a + b - c) * (b + c - a) * (b + a - c) * (c + b - a) * (c + a - b) := by sorry`	Score 0: major inconsistency. The Lean code is syntactically correct but contains a major inconsistency at the proposition. While NL has $\geq$, the proposition in Lean has $>$. This results in Proof Score 0.
As the complex conjugate of z is $\bar{z} = x - iy$, we conclude: $(2-i)\bar{z} = (2x-y)-i(x+2y)$.	`theorem l3 (x y : E) (z : C) ( h : z = x + y * Complex.I) : (2 - Complex.I) * star z = (2 * x - y) - Complex.I * (2 * y + x) := by sorry`	Score 0.67: minor inconsistency. The Lean code is syntactically correct and equivalent to the proof. The right-hand side of the proposition contains a slight variable ordering: $(x+2y)$ vs. $(2 * y + x)$ in Lean. However, this change is minor and still preserves the semantic meaning of the proof step.
Because $(a+b-c), (b+c-a), (c+a-b) > 0$ and we have: $a^2 \geq (a+c-b)(a+b-c), b^2 \geq (b+c-a)(b+a-c)$, and $c^2 \geq (c+b-a)(c+a-b)$. Thus: $a^2b^2c^2 \geq (a+c-b)(a+b-c)(b+c-a)(b+a-c)(c+b-a)(c+a-b)$.	`theorem l6 (a b c : R) (tc_1 : a > 0 ^ b > 0 ^ c > 0 ^ a + b > c ^ b + c > a ^ c + a > b) (l4 : (a + b - c > 0) ^ (b + c - a > 0) ^ (c + a - b > 0)) (l5 : a^2 >= (a + c - b) * (a + b - c) ^ b^2 >= (b + c - a) * (b + a - c) ^ c^2 >= (c + b - a) & (c + a - b)) : a^2 * b^2 * c^2 >= (a + c - b) * (a + b - c) * (b + c - a) * (b + a - c) * (c + b - a) * (c + a - b) := by sorry`	Score 1: perfectly match. The Lean code is syntactically correct and semantically equivalent to the proof step.

Table 5: Examples of varying ProofScores with natural language proof step and Lean code.

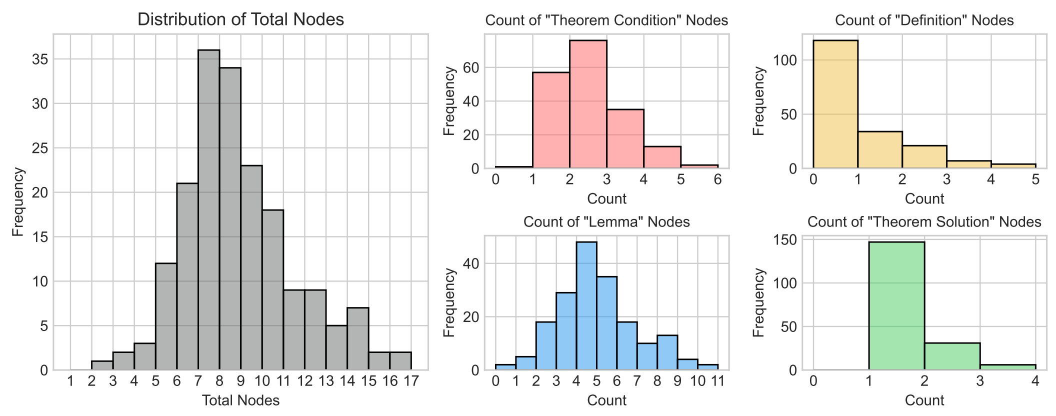


Figure 6: The distribution of the total number of nodes per problem (left) and the frequency of each node type (right).

A.4 Comparison results

This section contains extra comparison results. We show in Table 6 the extended version of Table 1, which includes Pass@1, 3, and 5 rates.

A.5 Comparison Examples

In this section, we present several examples of proof autoformalization to illustrate how different pipelines perform when formalizing natural language proofs. The examples below, selected from our comparative study in Section 6, show how ProofFlow DAG maintains a high degree of fidelity to the input proof. In contrast, other approaches, such as ProofFlow noDAG, Full Proof, and Step Proof, often fail to either adhere to the flow of the natural proof or even generate valid Lean 4 code.

To evaluate structural fidelity, we analyzed the dependencies for each syntactically correct proof step. By inspecting the Lean tactics, we identified exactly which previously proven steps and theorem conditions the solver utilized. A step was deemed structurally faithful if this set of dependencies precisely matched the logic of the original natural language proof. If the dependencies differed in any way, the step was marked as unfaithful.

For the purposes of illustration, the figures presented in this section are based on the actual dependency graph of the input natural language proof, and annotations will be provided in the case the the logical flow of the generated proof graph deviates from this input dependency graph.

A.5.1 Example: ProofFlow DAG is superior in structure fidelity

This example corresponds to entry "dummy_6" in the ProofFlowBench benchmark.

Theorem

If n is an odd integer, then $n^2 \equiv 1 \pmod 8$.

Proof

Since n is odd, we can write $n = 2k + 1$ for some integer k. Then $n^2 = (2k + 1)^2 = 4k^2 + 4k + 1$. We can factor this as $n^2 = 4k(k + 1) + 1$. Now, either k is even or k is odd. If k is even, then $k + 1$ is odd, and if k is odd, then $k + 1$ is even. In either case, $k(k + 1)$ is even, so $k(k + 1) = 2m$ for some integer m. Therefore $n^2 = 4(2m) + 1 = 8m + 1$, which means $n^2 \equiv 1 \pmod 8$.

Table 6: Performance metrics for all pipelines, evaluated at Pass@1, 3, and 5 rates on our 184-problem benchmark. Entries marked with "/" indicate not applicable.

			Step-Level		Proof-Level			
Pipeline	**Think mode**	**Pass**	**Form. accuracy**	**Tactic accuracy**	**Proof Score**	**Correct syntax**	**Time (mins)**	**Output tokens (k)**
PROOFFLOW DAG	No	1	0.644	0.252	0.320	0.016	3.3	8.1
		3	0.722	0.323	0.347	0.027	6.3	15.7
		5	0.751	0.358	0.355	0.027	8.8	22.4
	Yes	1	0.844	0.629	0.508	0.245	19.3	55.7
		3	0.925	0.723	0.541	0.348	27.0	78.0
		5	**0.939**	**0.742**	**0.545**	0.375	31.8	94.2
PROOFFLOW noDAG	No	1	0.697	0.225	0.312	0.022	4.8	9.8
		3	0.791	0.328	0.344	0.038	9.0	18.4
		5	0.807	0.391	0.347	0.049	12.3	25.8
	Yes	1	0.860	0.573	0.397	0.217	18.9	56.0
		3	0.921	0.662	0.414	0.332	27.0	80.0
		5	0.936	0.681	0.417	0.353	32.0	98.5
FULL PROOF	No	1	/	/	0	0	0.2	2.7
		3	/	/	0.011	0.011	0.5	6.6
		5	/	/	0.021	0.027	0.8	10.5
	Yes	1	/	/	0.156	0.309	0.9	3.5
		3	/	/	0.216	0.467	2.5	9.5
		5	/	/	0.279	**0.571**	3.8	15.1
STEP PROOF	No	1	/	0.054	0	0	0.03	0.1
		3	/	0.061	0	0	0.05	0.3
		5	/	0.068	0.046	0.005	0.2	1.2
	Yes	1	/	0.123	0.023	0.065	1.3	4.9
		3	/	0.127	0.028	0.092	3.7	12.9
		5	/	0.129	0.029	0.119	10.6	32.9

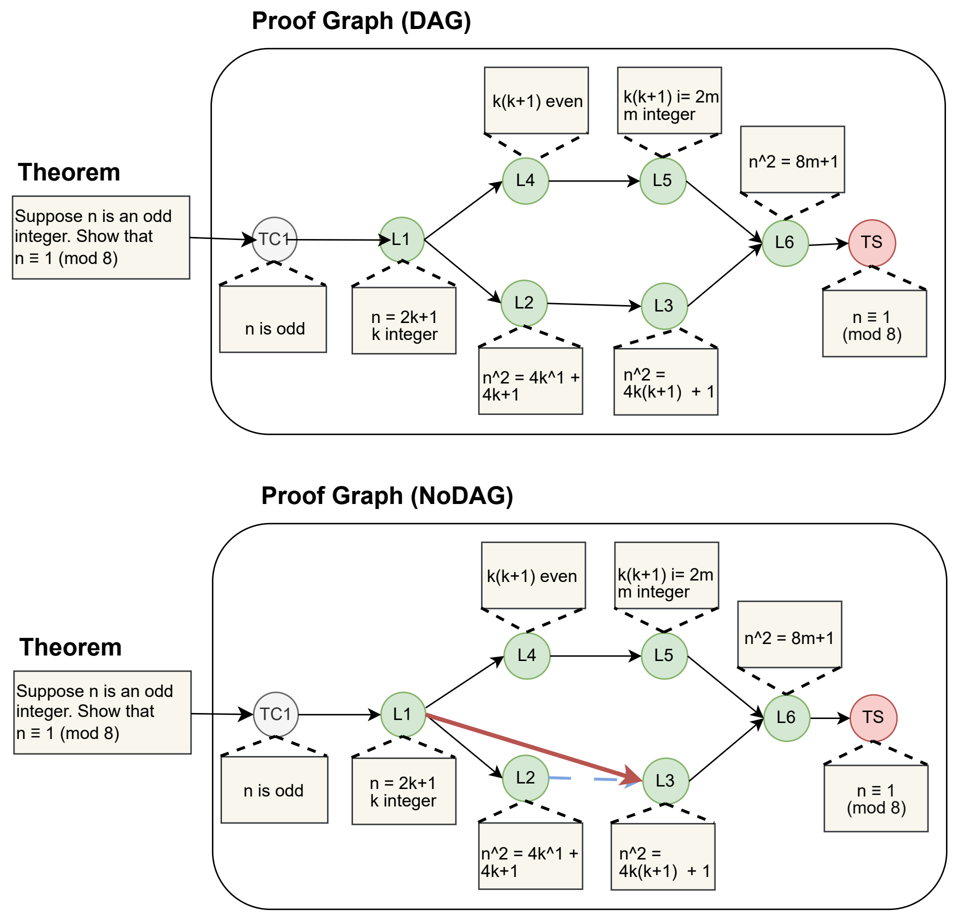


Figure 7: Comparison of proof structures generated by PROOFFLOW DAG (top) and PROOFFLOW noDAG (bottom) with respect to the original natural language proof. The red arrow indicates a dependency in PROOFFLOW noDAG not in the natural language proof. The blue dashed arrow indicates a dependence in the natural language proof but not included in PROOFFLOW noDAG. The PROOFFLOW DAG formal proof faithfully follows the dependency of the original proof. However, the one by PROOFFLOW noDAG proves step L2 then reuses L1 to prove L3, which is structurally unfaithful to the original proof.

PROOFFLOW DAG: The proof produced by PROOFFLOW DAG follows correctly the structure of the natural language proof and generates syntatically correct Lean code (see Figure 7).

PROOFFLOW noDAG: PROOFFLOW noDAG fails to adhere to the dependency structure of the natural language proof. In particular, as illustrated in Fig. 7, the transition from step L2 to L3 was inherent in the natural language proof but neglected in noDAG. The proof code for step L3 utilized step L1 to reprove the result of step L2. In other words, step L2 was made redundant.

FULL PROOF: FULL PROOF failed to generate syntactically correct Lean code (the first syntax error is "unknown identifier").

STEP PROOF: STEP PROOF failed to generate syntactically correct Lean code for steps L4, L5, L6, and TS (the first syntax error was "type mismatch").

A.5.2 EXAMPLE: PROOFFLOWDAG IS SUPERIOR IN PROVER ACCURACY

This example corresponds to entry "dummy_7" in the PROOFFLOWBENCH benchmark.

Theorem

If $P(A) = 0.6$ and $P(B) = 0.7$, then $P(A \cap B) \geq 0.3$.

Proof

We know that $P(A \cup B) = P(A) + P(B) - P(A \cap B)$. Since $P(A \cup B) \leq 1$, we have $P(A) + P(B) - P(A \cap B) \leq 1$. Substituting the given values: $0.6 + 0.7 - P(A \cap B) \leq 1$, which gives $1.3 - P(A \cap B) \leq 1$. Therefore $P(A \cap B) \geq 0.3$.

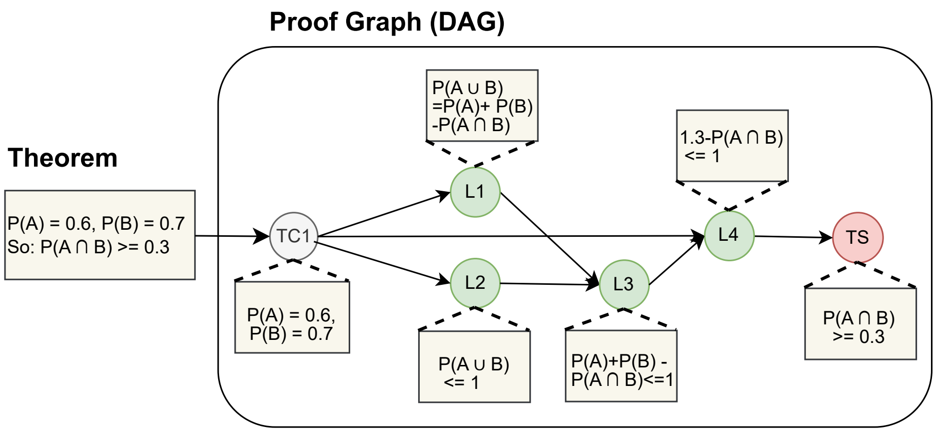


Figure 8: The structure of the proof produced by ProofFlow DAG faithfully follows the dependency graph of the input natural language proof.

ProofFlow DAG: ProofFlow DAG generates step-by-step proof faithfully following the logical transition of the natural language proof (see Fig. 8).

ProofFlow noDAG: The Lean code proof achieves very low prover accuracy. This is likely attributed to the fact that each step is given all previous steps, in contrast to only the necessary steps. As a consequence, the prover fails to produce concise and correct Lean code.

Full Proof: Full Proof failed to generate syntactically correct Lean code to prove this theorem (the first syntax error is “unknown identifier”).

Step Proof: Step Proof failed to generate syntactically correct Lean code to prove this theorem (the first syntax error is function type error).

A.5.3 Example: ProofFlow DAG is superior in proof efficiency

This example corresponds to entry “dummy_9” in the ProofFlowBench benchmark.

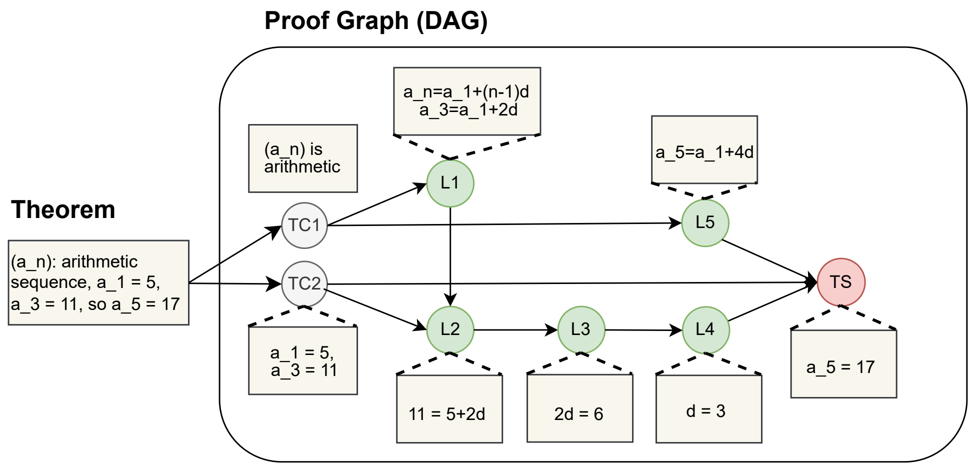
Theorem

If (a_n) is an arithmetic sequence with $a_1 = 5$ and $a_3 = 11$, then $a_5 = 17$.

Proof

Since (a_n) is arithmetic, there exists a common difference d such that $a_n = a_1 + (n-1)d$ for all n. From the given information, $a_3 = a_1 + 2d$. Substituting the values: $11 = 5 + 2d$, which gives us $2d = 6$, so $d = 3$. Now we can find $a_5 = a_1 + 4d = 5 + 4(3) = 5 + 12 = 17$.

ProofFlow DAG: ProofFlow DAG generates step-by-step proof faithfully following the logical transition of the natural language proof (Fig. 9).



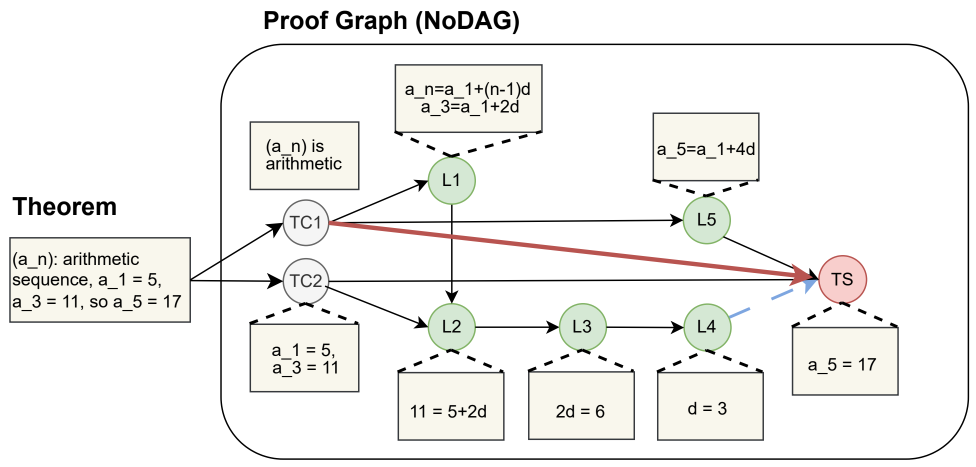


Figure 9: Comparison of proof structures generated by PROOFFLOW DAG (top) and PROOFFLOW noDAG (bottom) with respect to the original natural language proof. The red arrow indicates a dependency in PROOFFLOW noDAG not in the natural language proof. The blue dashed arrow indicates a dependence in the natural language proof but servered in PROOFFLOW noDAG. The one by PROOFFLOW DAG faithfully follows the dependency of the original proof. However, the one by PROOFFLOW noDAG proves the final step TS solely by the two given theorem conditions TC1 and TC2 and step L5. This is not only structurally unfaithful to the structure of the natural language proof but also inefficient as the efforts to prove all intermediate steps L1, L2, L3, L4 are squandered.

PROOFFLOW noDAG: The Lean code proof by PROOFFLOW noDAG is not only structurally unfaithful but also inefficient. As illustrated in Fig. 9, the final step TS is proven by the tactics stemming from theorem conditions TC1 and TC2 and intermediate step L5 (direct result of TC1) while neglecting all intermediate steps L1, L2, L3, L4. These steps are proven but not utilized for the final goal of the proof. As a consequence, the logical flow of the proof by PROOFFLOW noDAG fails to adhere to the structural of the natural language proof and squanders the resources used to prove L1, L2, L3, L4.

FULL PROOF: FULL PROOF failed to generate syntactically correct Lean code to prove this theorem (the first syntax error is function type error).

STEP PROOF: STEP PROOF failed to generate syntactically correct Lean code to prove this theorem (the first syntax error is function type error).

A.6 EXAMPLES OF ERRORS IN THE NATURAL LANGUAGE PROOF

The errors listed in Table 7 were identified using the thinking mode of our PROOFFLOW DAG pipeline at pass@5. These examples were specifically extracted from proof steps that our error detection pipeline (detailed in Section 4.2) flagged as an "NL statement error."

We have decided to keep these issues in the benchmark because they reflect the kinds of common and subtle ambiguities made by humans when writing proofs. For example, using ambiguous informal language like "we get [result x]" is a frequent occurrence in practice. By including these ambiguities, we ensure the benchmark remains a realistic representation of human-written proofs, rather than an overly artificial one.

Original Natural Language	Error Type	Comment
$3a^3-3a^2b-3ab^2+3b^3 = (a^2-b^2)(a-b)$	Incorrect Statement	An outright algebraic error. The left-hand side is exactly three times the right-hand side.
Let n be a natural number. If $n = 1$, ... If n is prime, we are done. If n is composite, then $n = ab$ ... By induction...	Logical Flaw	The proof sketch is intuitively correct but structurally flawed. It conflates the base case and the inductive step of a strong induction proof.
...we get $(r\frac{\sqrt{3}}{2})^2 + (r/2-1)^2 = 1$	Ambiguity	The phrase "we get" misleadingly suggests a general identity, when this is actually a conditional equation that only holds for the specific value $r = 1$.
From the condition $\mathrm{Arg}(z) = \pi/6$, we can write $z = r(\frac{\sqrt{3}}{2} + \frac{i}{2})$ for some $r > 0$.	Missing Assumption	The statement is incomplete because it relies on the unstated precondition that $z \neq 0$ for the $\mathrm{Arg}(z)$ function to be well-defined.
Hence, $(X \cdot Y)^2 \leq \|X\|^2\|Y\|^2$, because if not, then p would have two distinct real zeros...	Incomplete Argument	The reasoning, based on the discriminant of a quadratic, is incomplete as it fails for the case where $Y = 0$, where the polynomial degenerates.
Integrating... yields $v(x, y) = 2xy + 3y + g(x)$, where $g(x)$ is a function of x.	Incomplete Statement	The statement is true but insufficient. In the context of solving a PDE, it omits the crucial condition that the function of integration $g(x)$ must also be differentiable.

Table 7: Analysis of Flaws Identified in Natural Language Proof Steps

A.7 COMPUTATIONAL EFFICIENCY

We provide comprehensive diagnostics of the pipeline by examining each subcomponent in detail. Table 8 presents key metrics across different pipeline configurations.

Our analysis reveals that the current pipeline bottleneck, in terms of both time and token efficiency, lies primarily in the Tactic Completer stage. The Graph Builder demonstrates high reliability, succeeding on the first attempt in most cases (1.01–1.08 tries per problem). In contrast, the Formalizer typically requires 1–2 attempts depending on the thinking mode, while the Tactic Completer demands significantly more iterations, averaging 2–3 attempts in thinking mode and 3–4 attempts in non-thinking mode per node to generate valid Lean code. The temporal distribution further empha-

Table 8: Performance breakdown of ProofFlow pipeline components across different configurations. Metrics include average number of attempts per component, time distribution percentages, and token usage distribution.

Metric	**ProofFlow DAG**		**ProofFlow noDAG**	
Think Mode	No	Yes	No	Yes
Average Attempts				
Graph Builder (per problem)	1.08	1.08	1.01	1.01
Formalizer (per node)	2.20	1.39	1.94	1.37
Tactic Completer (per node)	4.04	2.39	3.69	2.67
Time Distribution (%)				
Graph Builder	13.8	3.9	10.1	3.3
Formalizer	4.9	9.8	4.7	7.9
Tactic Completer	81.2	86.3	85.3	88.8
Total Time (mins)	**8.8**	**31.8**	**12.3**	**32.0**
Token Distribution (%)				
Graph Builder	44.5	10.0	36.7	8.3
Formalizer	14.1	9.0	19.0	7.5
Tactic Completer	41.4	81.0	44.3	84.2
Total Tokens (k)	**22.4**	**94.2**	**25.8**	**98.5**

sizes this bottleneck: the Tactic Completer consumes 81–89% of the total execution time across all configurations, with the Graph Builder and Formalizer accounting for substantially smaller fractions.

Regarding token generation, we observe a difference between thinking and non-thinking modes. In thinking mode, the Tactic Completer dominates token usage (81–84%), primarily due to Goedel models generating substantial informal reasoning and extensive thinking before producing formal proofs. Conversely, in non-thinking mode, token distribution is more balanced, with both the Graph Builder and Tactic Completer each accounting for approximately 40% of tokens. This difference stems from the reasoning-heavy output of Gemini-2.5-Pro in the Graph Builder stage, contrasting with DeepSeek-Prover-V2, which was instructed to minimize preliminary reasoning.