Benchmarking Testing in Automated Theorem Proving

Jongyoon Kim[1]* **Hojae Han**[2]* **Seung-won Hwang**[1]†
Interdisciplinary Program in Artificial Intelligence, Seoul National University[1]
Electronics and Telecommunications Research Institute[2]
{john.jongyoon.kim,seungwonh}@snu.ac.kr
{hojae.han}@etri.re.kr

Abstract

Recent advances in large language models (LLMs) have shown promise in formal theorem proving, yet evaluating semantic correctness remains challenging. Existing evaluations rely on indirect proxies such as lexical overlap with human-annotated proof, or expensive manual inspection. Inspired by the shift from lexical comparison to test-based evaluation in code generation, we propose T^2, a framework that evaluates the semantic correctness of formal theorems: a generated theorem is considered correct only if all dependent successor theorems compile successfully, analogous to integration testing. We construct a benchmark from 5 real-world Lean 4 repositories, comprising 2,206 problems paired with 41 successor theorems on average, automatically extracted without human effort. Experiments demonstrate that while state-of-the-art models achieve high compilation success, they perform significantly worse under our semantic metric. The best model, Claude-Sonnet-4.5, achieves only 38.9% Testing Accuracy on the full set, given both natural language proof and successor theorems as context, revealing a critical gap in current theorem generation capabilities.[1]

1 Introduction

With the rapid advancement of Large Language Models (LLMs), there has been growing interest in applying these models to Automated Theorem Proving (ATP), where Natural Language (NL) problems are formalized as theorems in proof assistants such as Lean, with approaches ranging from fine-tuned provers (Wang et al., 2025a; Xin et al., 2024) to in-context learning agents (Thakur et al., 2024; Kumarappan et al., 2025; Varambally et al., 2026). Recent benchmarks demonstrate that LLMs can generate valid formal proofs spanning high school olympiad- to undergraduate-level mathematics (Zheng et al., 2022; Azerbayev et al., 2023; Yu et al., 2025), where correctness is verified by checking whether the generated theorem compiles, confirming its syntactic and logical validity.

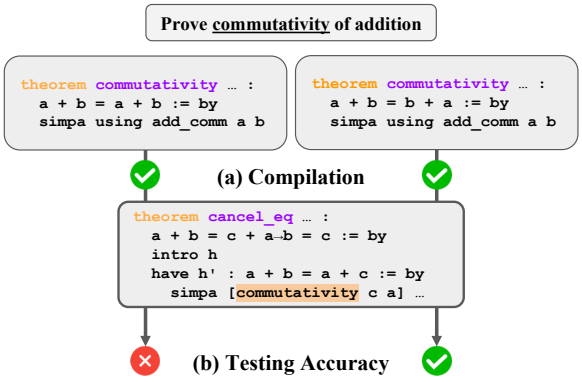


Figure 1: Two candidate theorems for proving commutativity of addition: one correct ($a + b = b + a$) and one tautology ($a + b = a + b$). (a) Under compilation-based evaluation, both candidates pass, as each is logically valid in isolation. (b) Under testing-based evaluation, the tautology is exposed: the successor theorem (`cancel_eq`) fails when it depends on the incorrect candidate, while it compiles successfully with the correct one.

However, compilation alone does not guarantee *semantic* correctness. As illustrated in Figure 1-(a), an NL specification may ask to prove commutativity ($a + b = b + a$), while the generated theorem is a tautology ($a + b = a + b$). As this tautology compiles, which only verifies its logical validity, the intended meaning may be lost.

Existing benchmarks have recognized this limitation and proposed various proxy metrics to assess semantic correctness, as summarized in Table 1. These approaches evaluate the generated theorem by comparing with a ground-truth proof reference, using BLEU (Zheng et al., 2022), or equivalence checking using linguistic entailment such as natu-

* Both authors contributed equally to this research.
† Corresponding Author
[1]Implementation and dataset are available at: https://github.com/ldilab/T2.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track), pages 2241–2260
July 2-7, 2026

Benchmark	Eval. Method	Annot.-Free	Semantic
MiniF2F (Zheng et al., 2022)	—	✗	✗
FormalMATH (Yu et al., 2025)	—	✗	✗
PutnamBench (Tsoukalas et al., 2024)	—	✗	✗
MiniF2F-v2 (Ospanov et al., 2025)	Human	✗	✓
ProofNet (Azerbayev et al., 2023)	Lexical	✓	✗
Lean-Workbook (Ying et al., 2025)	Lexical	✓	✗
Con-NF (Liu et al., 2025)	Prover	✓	✗
ProofNetVerif (Poiroux et al., 2025)	Prover	✓	✗
T^2 (Ours)	**Testing**	✓	✓

Table 1: Comparison of semantic correctness evaluation across formal theorem benchmarks. All benchmarks verify logical validity via compilation. **Annot.-Free**: operates without human annotation or evaluation. **Semantic**: directly verifies semantic correctness.

ral language inference (Ying et al., 2025), or using a compiler built-in prover such as BEq (Liu et al., 2025). Alternatively, humans may investigate the generated theorem without any reference annotation (Ospanov et al., 2025; Tsoukalas et al., 2024; Yu et al., 2025). Desirably, this evaluation should be automatic, requiring neither human evaluation nor reference annotation, while directly verifying semantic correctness.

Towards this goal, we take inspiration from integrated testing in software engineering, where a component's correctness is verified, not in isolation, but by executing the modules that depend on it. Drawing on the *Curry-Howard correspondence* (Curry, 1934), where proofs correspond to programs, we observe that a successor theorem that uses a generated lemma is analogous to a module that calls a function. That is, if the lemma is semantically incorrect, the successor theorem cannot be correct. Based on this insight, we propose evaluating a generated theorem by whether its successor theorems, those that invoke or build upon it, compile successfully, as illustrated in Figure 1-(b). This approach is fully automatic, requires no external references, and directly verifies semantic correctness through successor compilation.

Building on this idea, we introduce **Testing Accuracy (TA)**, the first testing-based metric for semantic evaluation of formal theorems, and **T**heorem **T**esting ($\mathbf{T}^2$), a large-scale Lean benchmark designed to evaluate the generated theorem by testing it with successor theorems. T^2 comprises 2,206 theorems curated from real-world Lean repositories, each paired with successor theorems that serve as integration test cases. We further identify a Hard subset of 389 problems that requires the model to generate both an auxiliary proposition and a proof of the target theorem.

Across experiments with 18 open and closed-source models, we observe a substantial gap between compilation and testing accuracy: models that appear successful under traditional measures often fail semantically. Our key findings are:

- **High false positive rate of existing metrics**: Up to 93.1% of compilable theorems are semantically incorrect, and BLEU fails to distinguish correct from incorrect outputs.
- **Overfits to compilation**: Specialized proving models overfit to syntax and score lower on semantic correctness than similarly sized general-purpose models.
- **Dependency structure enables 1k+ test cases**: TA becomes more discriminative as successor coverage grows, with the majority of problems verified at depth 7 and an average of 1.6k successor theorems per problem at that depth.
- **Successor context improves generation**: Providing successor theorems as context consistently improves semantic correctness, while NL proofs alone do not.

2 Related Work

Formal Theorem Proving Benchmarks Recent benchmarks have driven progress in neural automated theorem proving. MiniF2F (Zheng et al., 2022), ProofNet (Azerbayev et al., 2023), and PutnamBench (Tsoukalas et al., 2024) evaluate models on high school Olympiad- to undergraduate-level problems, while Lean-Workbook (Ying et al., 2025) and FormalMATH (Yu et al., 2025) provide large-scale training corpora. However, as summarized in Table 1, these benchmarks primarily verify logical validity via compilation, leaving semantic correctness either unaddressed or assessed through unreliable proxies such as BLEU, natural-language-inference-based back-translation, or prover-based equivalence checking, all of which require ground-truth references or external models and provide only indirect estimates of semantic correctness. Manual inspection (Ospanov et al., 2025) offers a reliable evaluation but lacks scalability. Moreover, since these benchmarks consist of isolated, self-contained problems, they are far removed from real-world mathematical proof development, where theorems are deeply interconnected through dependency structures.

Test-Based Evaluation in Code Generation In code generation, test-based evaluation is standard practice. Benchmarks such as APPS (Hendrycks et al., 2021), HumanEval (Chen et al., 2021), and MBPP (Austin et al., 2021) evaluate generated code by executing it against test cases rather than comparing it lexically to a reference solution. This paradigm directly measures functional correctness and is both automatic and reference-free. Despite the well-established connection between proofs and programs via the Curry-Howard correspondence (Curry, 1934), test-based evaluation has not been applied to formal theorem generation before our work.

3 Testing-Based Semantic Evaluation

Evaluating semantic correctness, whether a generated theorem captures the intended logical meaning of an NL theorem, is a central challenge in formal theorem generation.

Given an NL theorem t_{nl}, the task is to generate a formal theorem $t_{\text{fl}} = (s_{\text{fl}}, p_{\text{fl}})$ comprising a statement s_{fl} and a proof p_{fl}, such that t_{fl} is both logically valid and semantically faithful to t_{nl}. Compilation verifies **logical validity** ($s_{\text{fl}} \vdash p_{\text{fl}}$), but not **semantic correctness** ($s_{\text{fl}} \vdash p_{\text{fl}} \wedge t_{\text{fl}} \equiv [\![t_{\text{nl}}]\!]$). Proving strict logical equivalence ($t_{\text{fl}} \equiv t_{\text{GT}}$) is generally intractable for complex theories, and it is hard to find the ground-truth of the target theorem t_{GT}. Existing proxies that rely on surface-level similarity or approximate entailment fail to capture the intended mathematical semantics, as stated in Section 2.

To address this, we adopt an observational approach: rather than proving equivalence directly, we verify whether t_{fl} behaves equivalently to t_{GT} in all observed contexts.

3.1 Integration Testing for Formal Theorems

In software engineering, a function may be compiled yet fail to meet its specification. To catch such failures, integration testing validates that a component, when combined with its dependents, produces correct behavior (Ammann and Offutt, 2017). If a modified component causes dependent modules to fail, it signals a semantic violation.

We adopt this principle for formal theorems by viewing each theorem as a node in a dependency graph (Prasad et al., 2024; Cabral et al., 2026) rather than in isolation. Let t_{GT} be the **ground-truth of the target theorem** corresponding to the **generated candidate theorem** t_{fl}. We denote the theorems t_{GT} depends on as **predecessors** $\mathcal{T}_{\text{pred}} = \{t^{(1)}_{\text{pred}}, \ldots, t^{(m)}_{\text{pred}}\}$, and the theorems that transitively depend on t_{GT} as **successors** $\mathcal{T}_{\text{succ}} = \{t^{(1)}_{\text{succ}}, \ldots, t^{(n)}_{\text{succ}}\}$, which serve as the test cases. Figure 2a shows a concrete Lean source view of these three types of theorem, where **InfiniteAdeleRing** serves as a predecessor, **algebraMap_apply** is the target, and **mixedEmbedding_eq_algebraMap_comp** is one of the successors that invokes the target. Figure 2b gives the corresponding dependency view, where predecessors flow into t_{GT}, and successors flow out of it. We consider t_{fl} semantically correct if substituting it

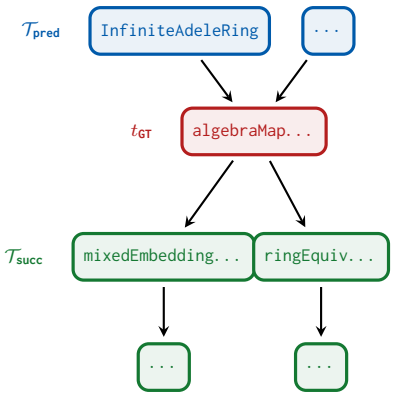
```
-- Predecessors 𝒯pred
-- Predecessor Theorems: required to state tGT
def InfiniteAdeleRing := (v : InfinitePlace K) -> v.
    completion
...

-- Target tGT
-- Replaced by the generated tfl during evaluation
@[simp] theorem algebraMap_apply (x : K) :
    algebraMap K (InfiniteAdeleRing K) x v = x := rfl

-- Successors 𝒯succ
-- Successor Theorems test suite: depend on tGT
theorem mixedEmbedding_eq_algebraMap_comp {x : K} :
    mixedEmbedding K x = ringEquiv_mixedSpace K (
     algebraMap K (InfiniteAdeleRing K) x) := by
  ext v ⟨i⟩ simp only [ringEquiv_mixedSpace_apply,
    algebraMap_apply, ...]
  ...

theorem ringEquiv_mixedSpace_apply (x : InfiniteAdeleRing K
    ) :
    ringEquiv_mixedSpace K x = ... := rfl
```

(a) Lean source code



(b) Dependency graph

Figure 2: An example problem of T^2. The target t_{GT} (**algebraMap_apply**) has predecessors $\mathcal{T}_{\text{pred}}$, (e.g. definition of **InfiniteAdeleRing**), and successors $\mathcal{T}_{\text{succ}}$ that invoke it (e.g. **mixedEmbedding_eq_algebraMap_comp**).

for t_{GT} leaves every theorem in $\mathcal{T}_{\text{succ}}$ compilable.[2]

3.2 Cut Elimination as Theorem Testing

We ground this intuition formally in the Curry-Howard correspondence, which identifies proofs as programs and logical rules as computational operations. Specifically, we identify integration testing with the logical principle of *Cut Elimination*.

The *Cut Rule* in sequent calculus formalizes the composition of proofs. For instance, proving $X \to W$ by chaining intermediate theorems Y and Z, given a shared set of background assumption Γ, corresponds to integration testing across multiple modules:

$$\frac{\Gamma \vdash X \to Y \quad \Gamma \vdash Y \to Z \quad \Gamma \vdash Z \to W}{\Gamma \vdash X \to W} \tag{1}$$

where $X \to Y$ corresponds to t_{fl}, and the remaining chain represents $\mathcal{T}_{\text{succ}}$. In proof theory, the use of a lemma introduces a cut, and cut elimination removes the auxiliary lemma to produce a direct proof, which computationally corresponds to a function call. By the cut elimination theorem, if t_{fl} is semantically correct, this chain of cuts can be eliminated, yielding a valid proof of $X \to W$. In other words, since $X \to W$ holds via the chain $X \to Y \to Z \to W$, if t_{fl} breaks the compilation of $\mathcal{T}_{\text{succ}}$, then we treat t_{fl} as semantically incorrect. Successful compilation across the chain provides evidence of semantic correctness, though not a logical guarantee. Crucially, this process is not limited to immediate callers, so it can recursively verify t_{fl} across the entire chain of dependent proofs.

3.3 Testing Accuracy

We apply the above principle to define our metric, TA. Let $\mathcal{D}$ be the distribution of problems in the benchmark. For a problem $P = (t_{\text{nl}}, \mathcal{T}_{\text{pred}}, \mathcal{T}_{\text{succ}}) \in \mathcal{D}$, let t_{fl} be the generated theorem. Here, $\mathcal{T}_{\text{pred}}$ corresponds to Γ in Equation (1), namely the predecessor theorems used by both t_{fl} and $\mathcal{T}_{\text{succ}}$. Each successor theorem $t^{(i)}_{\text{succ}}$ imposes a semantic constraint on t_{fl}. As the number and depth of these constraints increase, their conjunction increasingly approximates a guarantee of semantic correctness. We define TA as the expectation over problems:

$$\texttt{TA} = \mathop{\mathbb{E}}_{P \sim \mathcal{D}} \left[\bigwedge_{i=1}^{k} \texttt{compiles}(t^{(i)}_{\text{succ}} \mid t_{\text{fl}}) \right] \tag{2}$$

[2]As a theorem with a different name will fail compilation easily, we condition the theorem name in the context.

where $\texttt{compiles}(t^{(i)}_{\text{succ}} \mid t_{\text{fl}}) = 1$ if $t^{(i)}_{\text{succ}}$ successfully compiles when t_{GT} is replaced by t_{fl}, and 0 otherwise.

To ensure meaningful coverage, we restrict our benchmark to theorems with $|\mathcal{T}_{\text{succ}}| \geqslant 2$ (see Section 4). The resulting problems have dependency depth up to 7 and an average of 1.6k successor theorems, so each evaluation involves multiple semantic constraints.

4 Benchmark Construction

4.1 Extraction Pipeline

We employ a fully automated, dependency-aware extraction pipeline targeting public Lean 4 repositories.

Dependency Graph Construction We parse the Lean environment to build a global dependency graph $G = (V, E)$, where V is the set of theorems and lemmas and E represents usage relationships between them, with Lean-Dojo (Yang et al., 2023).

Target Selection. From G, we select target theorems t_{GT} that satisfy two criteria: (1) non-triviality: the depth, which is the distance from the target theorem to the successor theorems, is greater than 1 in G, excluding standalone theorems that existing benchmarks predominantly target, and (2) successor coverage: $|\mathcal{T}_{\text{succ}}| \geqslant 2$, ensuring that each target admits multiple semantic constraints for meaningful evaluation, reducing the possibility of false positives.

Context Extraction For each target t_{GT}, we extract the predecessor theorems $\mathcal{T}_{\text{pred}}$, consisting of the definitions and lemmas necessary to state and prove t_{GT}, and the successor theorems $\mathcal{T}_{\text{succ}}$, comprising all theorems that transitively depend on t_{GT}.

4.2 Natural Language Annotation

Since real-world repositories typically lack natural language descriptions, we generate t_{nl} for each target using a strong LLM (Claude Sonnet 4.5). This allows us to construct a problem instance $P_i = (t_{\text{nl}}, \mathcal{T}_{\text{pred}}(t_{\text{GT}}, t^{(i)}_{\text{succ}}), t^{(i)}_{\text{succ}})$ for each target t_{GT} and each successor theorem $t^{(i)}_{\text{succ}} \in \mathcal{T}_{\text{succ}}(t_{\text{GT}})$. To validate annotation quality, we manually verified 10 randomly sampled NL annotations and found no errors. Full details are provided in Appendix C.

4.3 Dataset Statistics

We first examined whether existing benchmarks could support testing-based evaluation. However, few problems in these benchmarks satisfy our non-triviality and successor coverage criteria, yielding only 12 problems from evaluation benchmarks and 110 from training benchmarks (see Table 6).

We therefore extract from 5 high-quality Lean 4 repositories listed on the Lean community page[3]. The resulting benchmark comprises 2,206 problems with an average of 41 successor theorems per target, providing a rigorous testbed for semantic correctness evaluation. To further challenge model capabilities, we construct T^2 Hard, a subset of 389 problems whose target declarations have a body of type Prop, requiring generation of both the proposition and its proof.

5 Experiment Setup

5.1 Models

We evaluate 18 models spanning general-purpose and domain-specialized systems, as summarized in Table 5. All models are evaluated in a zero-shot setting with temperature 0.6 and top_p 0.95. Prompts are provided in Appendix K.

5.2 Metrics

Our primary metric is TA: the generated theorem t_{fl} replaces the original theorem t_{GT} in the repository, and we recompile the entire dependency chain as a test suite. The generated theorem passes the test if and only if all successor theorems compile successfully. TA is the only metric that directly evaluates semantic correctness, and all main results are reported using it.

To contextualize our findings and demonstrate the limitations of existing evaluation methods, we additionally report two baseline metrics. Compilation accuracy verifies that the generated theorem compiles in isolation, corresponding to the standard pass criterion in existing ATP benchmarks.[4] **BLEU** measures lexical similarity to the ground-truth via n-gram overlap. We analyze the gap between Compilation Accuracy and TA to quantify how much compilation overestimates semantic correctness, and examine whether BLEU can serve as a reliable proxy for TA, in Section 6.3.

[3] https://leanprover-community.github.io/papers.html

[4] For repository-sourced problems, we append `#exit` after the generated theorem to prevent the Lean compiler from compiling successor theorems.

5.3 Evaluation Environment

Our benchmark comprises two categories of problems with different verification requirements: T^2 (including the T^2 Hard subset). T^2 is drawn from real-world Lean repositories that rely on project-specific dependencies and varying Lean versions. For these problems, we execute `lake build` within each repository's native environment to strictly adhere to its original configuration and enable full dependency checking. For the supplementary existing benchmarks (e.g., FormalMath, CombiBench), which consist of single theorem problems without successor theorems, we use `kimina-lean-server` (Santos et al., 2025) to verify compilation within a standardized Lean 4.25.0 environment. Details are provided in Appendix B.

6 Results

We evaluate state-of-the-art LLMs on the T^2 benchmark. We first present model performance and key results in Section 6.1. We then analyze the effect of few-shot prompting and iterative refinement with compiler feedback in Section 6.2, compare TA against existing metrics in Section 6.3, and study the impact of successor theorem coverage in Section 6.4.

6.1 Overview of Results

Table 2 presents compilation accuracy and TA across T^2 and T^2 Hard subsets, under the default setting (providing both NL proof and successor theorem). The best model (Claude-Sonnet-4.5) achieves 80.3% compilation accuracy on the Full set while achieving only 38.9% TA. Scaling model size yields modest improvements within model families (e.g., Llama-3.1 improves from 24.8% at 8B to 33.2% at 405B), but overall TA remains below 40% (Full) and 6% (Hard), indicating that the semantic gap is far from closed. We additionally evaluate on existing benchmarks, where the gap between compilation and TA is substantially smaller due to the limited number of successor theorems. Full results are provided in Appendix H.

We further investigate the impact of input context on TA. Table 3 presents the full ablation on the T^2 dataset. We find that both NL proofs and successor theorems must be provided together to yield consistent improvements, that is, neither alone offers meaningful benefit. For instance, Claude-Sonnet-4.5 improves by 4.9% when given both NL proofs and successor theorems, yet providing suc-

Model	T^2		T^2 Hard	
	Compile	TA	Compile	TA
Closed-Source LLMs				
Claude-3.7-Sonnet	75.1	36.8	34.3	2.6
Claude-4-Sonnet	81.1	36.0	47.5	4.5
Claude-Sonnet-4.5	80.3	38.9	46.0	4.5
GPT-4o-mini	59.1	36.7	12.2	1.5
GPT-5	85.7	37.7	68.3	3.4
GPT-5-mini	86.0	37.9	66.6	4.1
GPT-5-nano	88.7	36.6	75.6	1.5
Open-Source LLMs				
DeepSeek-R1	70.5	36.3	25.1	4.1
GPT-OSS-120B	68.2	32.3	40.0	4.5
Llama-3.1-405B	63.4	33.2	27.8	2.6
Llama-3.1-70B	65.4	37.0	20.1	2.4
Llama-3.1-8B	36.9	24.8	6.4	1.3
Math Specialized Models				
DeepSeek-Prover-v2-7B	62.2	30.0	35.5	3.2
Goedel-Formalizer-v2-8B	38.7	23.6	22.1	2.4
Goedel-Prover-v2-32B	74.6	31.0	54.6	3.2
Goedel-Prover-v2-8B	46.1	24.5	37.9	2.6
Kimina-Autoformalizer-7B	21.9	20.0	4.3	1.5
Kimina-Prover-Distill-8B	28.0	23.4	5.1	1.7

Table 2: Compilation accuracy and TA (%) on T^2 and T^2 Hard.

Model	NL ✓		NL ✗	
	ST ✗	ST ✓	ST ✗	ST ✓
Closed-Source LLMs				
Claude-3.7-Sonnet	29.4	36.8	29.8	32.4
Claude-4-Sonnet	27.7	36.0	30.0	32.3
Claude-Sonnet-4.5	33.0	38.9	34.0	32.9
GPT-4o-mini	33.8	36.7	26.2	26.7
GPT-5	36.9	37.7	29.1	28.7
GPT-5-mini	35.2	37.9	29.6	30.9
GPT-5-nano	33.4	36.6	30.8	31.1
Open-Source LLMs				
DeepSeek-R1	31.0	36.3	28.9	29.1
GPT-OSS-120B	26.7	32.3	28.1	27.4
Llama-3.1-405B	30.8	33.2	29.7	30.8
Llama-3.1-70B	28.9	37.0	28.5	29.1
Llama-3.1-8B	22.6	24.8	21.6	21.8
Math Specialized Models				
DeepSeek-Prover-v2-7B	32.0	30.0	28.2	26.1
Goedel-Formalizer-v2-8B	23.3	23.6	24.3	23.6
Goedel-Prover-v2-32B	30.6	31.0	29.4	31.3
Goedel-Prover-v2-8B	25.4	24.5	24.7	24.2
Kimina-Autoformalizer-7B	20.2	20.0	20.0	19.9
Kimina-Prover-Distill-8B	21.3	23.4	20.7	22.2

Table 3: Impact of input context on TA (%) on T^2 full dataset. We compare two axes: whether an NL proof is provided (✓) and whether a successor theorem (ST) is provided (✓) or withheld (✗).

cessor theorems alone without NL proofs actually decreases TA. This suggests that the successor theorem helps the model understand the required semantic constraints, while NL proofs provide the necessary reasoning context to satisfy them. The ablation results on T^2 Hard subset are provided in Appendix F.

6.2 Extended Experiments: Few-shot and Iterative Refinement

To test whether other prompting strategies can improve semantic correctness, we evaluate 2-shot prompting and iterative refinement on the T^2 Hard subset.

Few-shot prompting We test whether the low performance on T^2 Hard comes from the models struggling to understand the task format, or from a gap in their ability to solve it. We evaluate the four top-performing models with 2-shot prompting. We sample two examples from the Herald (Gao et al., 2025) training set that Claude Sonnet 4.5 solves, and prepend them as in-context examples. Table 4 compares TA between the zero-shot and 2-shot settings. We see no consistent gain across the four models, and in some cases, the performance drops. This shows that the difficulty of T^2 does not come from understanding what the task asks for. The harder part is producing a formalization that matches the natural-language annotation in meaning, which a few examples cannot teach.

Model	Zero-shot TA (%)	2-shot TA (%)
Claude-4-Sonnet	4.5	4.3
Claude-3.7-Sonnet	2.6	2.8
GPT-5	3.4	2.6
GPT-5-mini	4.1	2.8

Table 4: Zero-shot vs. 2-shot TA on T^2 Hard.

Iterative refinement We also evaluate Hilbert (Varambally et al., 2026), a method that takes feedback from the Lean compiler and uses it to refine the generated theorems over multiple rounds. Even with this feedback loop, Hilbert reaches 5.0% TA on T^2 Hard, compared to 3.2% from the zero-shot baseline (DeepSeek-Prover-v2-7B). The gain is small and does not reduce the semantic gap.

6.3 Comparison with Existing Metrics

Compilation rate As shown in Table 2, there is a significant divergence between compilation success and semantic correctness across all models. The precision of compilation as a predictor of

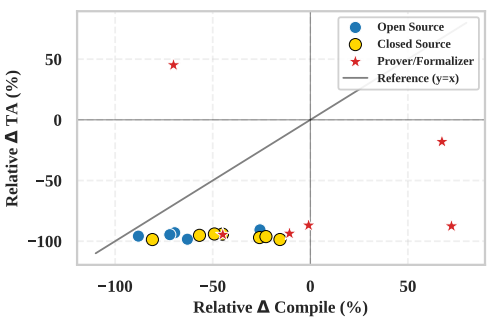


Figure 3: Relative performance drop from existing benchmarks to T^2 Hard, in compilation accuracy (x-axis) vs. TA (y-axis). Prover/Formalizer models (red stars) in the lower-right quadrant, where compilation gains do not translate to TA improvements.

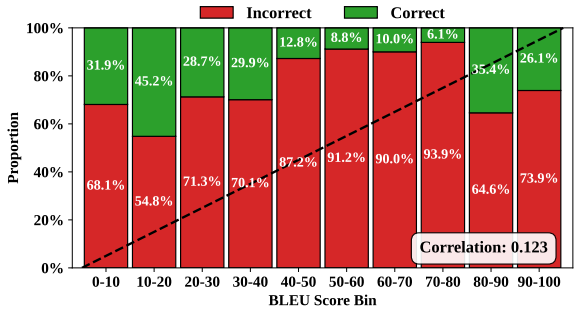
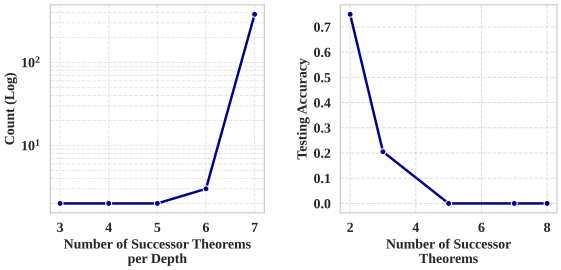


Figure 4: Proportion of testing-verified (green) and testing-failed (red) samples across BLEU score bins, aggregated over all models. High BLEU scores do not guarantee semantic correctness.

semantic correctness is only 6.89%. For instance, GPT-5 achieves **68.3%** compilation accuracy but only **3.4%** TA. This disparity persists regardless of model scale or specialization, confirming that compilation metrics fail to capture semantic correctness. Notably, as shown in Figure 3, specialized proving models report compilation gains over general-purpose models, but these gains do not translate to TA improvements, suggesting that domain-specific fine-tuning improves syntactic fluency without improving semantic correctness.

Lexical metric We find that lexical similarity is an unreliable proxy for semantic correctness. As shown in Figure 4, even in the highest BLEU score bins (90-100), over 70% of samples are semantically incorrect. Conversely, the lowest bins (0-10) still contain approximately 30% verified correct samples. This demonstrates that high BLEU scores do not guarantee semantic correctness, nor do low scores, confirming that lexical metrics cannot be utilized for semantic correctness.



(a) Depth distribution (b) TA by number of STs

Figure 5: (a) Distribution of successor theorem (ST) depth in T^2 Hard. Most problems (380) reach depth 7, with 1,600 STs on average at that depth. (b) TA drops sharply as STs grow, indicating that broader coverage yields stricter evaluation.

6.4 Analysis of Successor Theorem Coverage

We further examine how successor theorem coverage affects evaluation strictness. As shown in Figure 5a, the majority of problems in our benchmark have a dependency depth of 7, ensuring deep semantic verification by default. Figure 5b shows that TA decreases sharply as the number of successor theorems increases: with only 2 successor theorems, Claude-4-Sonnet achieves approximately 70% TA, dropping to near 0% when 5 or more must be satisfied. This confirms that each additional successor theorem imposes a meaningful semantic constraint, and that our benchmark's rich dependency structures provide a rigorous evaluation setting.

7 Conclusion

We introduced TA, the first testing-based metric for evaluating semantic correctness in automated theorem proving, and T^2, a large-scale Lean benchmark of 2,206 theorems with rich dependency structures. Our experiments across 18 models reveal that compilation-based evaluation fundamentally overestimates model capability, with up to 2× gap between compilation and TA on the full set and up to 50× on the hard set. We further showed that lexical metrics such as BLEU cannot reliably distinguish semantically correct theorems from incorrect ones, and that providing successor theorems as generation context consistently improves semantic correctness. These findings demonstrate that testing-based evaluation is essential for trustworthy assessment of formal theorem generation.

8 Limitations

The reliability of TA increases with the number and diversity of successor theorems, as each imposes an additional semantic constraint. However, for problems with few successor theorems, coverage may be insufficient to catch all semantic errors, and even with many dependents, full completeness cannot be guaranteed, analogous to the inherent limitation of software testing. TA is inherently inapplicable to standalone theorems without dependents, such as the most recently added theorems at the frontier of a repository that no other proof yet builds upon. Both our metric and benchmark are implemented exclusively for Lean 4. Moreover, T^2 is constructed from 5 Lean 4 repositories covering primarily research-level mathematics. The NL specifications t_{nl} are generated by a strong LLM rather than written by the original authors. While this enables scalable annotation, it may introduce noise that affects the autoformalization setting.

Acknowledgments

This work was supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by ICT R&D program of MSIT/IITP (2022-0-00995, Automated reliable source code generation from natural language descriptions), the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. RS-2024-00414981), and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) [NO.RS-2021-II211343, Artificial Intelligence Graduate School Program (Seoul National University)].

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, and 1 others. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*.

Paul Ammann and Jeff Offutt. 2017. *Introduction to software testing*. Cambridge University Press.

AI Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. *Claude-3 Model Card*, 1(1):4.

AI Anthropic. 2025a. Claude 3.7 sonnet system card. *Claude 3.7 Sonnet Model Card*.

AI Anthropic. 2025b. System card: Claude sonnet 4.5. *Claude Sonnet 4.5 Model Card*.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. 2023. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *Preprint*, arXiv:2302.12433.

Rafael Medeiros Cabral, Tuan Manh Do, Yu Xuejun, Wai Ming Tai, Zijin Feng, and SHEN XIN. 2026. Proofflow: A dependency graph approach to faithful proof autoformalization. In *The Fourteenth International Conference on Learning Representations*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

H. B. Curry. 1934. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590.

Guoxiong Gao, Yutong Wang, Jiedong Jiang, Qi Gao, Zihan Qin, Tianyi Xu, and Bin Dong. 2025. Herald: A natural language annotated lean 4 dataset. In *The Thirteenth International Conference on Learning Representations*.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Adarsh Kumarappan, Mo Tiwari, Peiyang Song, Robert Joseph George, Chaowei Xiao, and Anima Anandkumar. 2025. Leanagent: Lifelong learning for formal theorem proving. In *The Thirteenth International Conference on Learning Representations*.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, and 1 others. 2025. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*.

Qi Liu, Xinhao Zheng, Xudong Lu, Qinxiang Cao, and Junchi Yan. 2025. Rethinking and improving autoformalization: Towards a faithful metric and a dependency retrieval-based approach. In *International Conference on Learning Representations*.

OpenAI. 2024. GPT-4o mini: Advancing Cost-Efficient Intelligence.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. 2025. minif2f-lean revisited: Reviewing limitations and charting a path forward. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 39.

Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. 2025. Reliable evaluation and benchmarks for statement autoformalization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Vishesh Prasad, Brian Kim, and Nickvash Kani. 2024. Mathematical derivation graphs: A relation extraction task in stem manuscripts. *arXiv preprint arXiv:2410.21324*.

ZZ Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, and 1 others. 2025. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*.

Marco Dos Santos, Haiming Wang, Hugues de Saxcé, Ran Wang, Mantas Baksys, Mert Unsal, Junqi Liu, Zhengying Liu, and Jia Li. 2025. Kimina lean server: Technical report. *Preprint*, arXiv:2504.21230.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, and 1 others. 2025. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*.

Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. 2024. An in-context learning agent for formal theorem-proving. In *First Conference on Language Modeling*.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. 2024. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. 2026. Hilbert: Recursively building formal proofs with informal reasoning. In *The Fourteenth International Conference on Learning Representations*.

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, and 21 others. 2025a. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *Preprint*, arXiv:2504.11354.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, and 1 others. 2025b. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. 2024. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *Preprint*, arXiv:2405.14333.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Zheng Yuan, Dahua Lin, and Kai Chen. 2025. Lean workbook: A large-scale lean problem set formalized from natural language math problems. *Preprint*, arXiv:2406.03847.

Zhouliang Yu, Ruotian Peng, Keyi Ding, Yizhe Li, Zhongyuan Peng, Minghao Liu, Yifan Zhang, Zheng Yuan, Huajian Xin, Wenhao Huang, Yandong Wen, Ge Zhang, and Weiyang Liu. 2025. Formalmath: Benchmarking formal mathematical reasoning of large language models.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2022. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*.

A Model Details

Family	Models
Claude (Anthropic, 2025a, 2024, 2025b)	Claude-3.7-Sonnet Claude-4-Sonnet Claude-Sonnet-4.5
GPT (OpenAI, 2024; Singh et al., 2025)	GPT-4o-mini GPT-5 GPT-5-Mini GPT-5-Nano
GPT-OSS (Agarwal et al., 2025)	GPT-OSS-120B
Llama (Grattafiori et al., 2024)	Llama-3.1-8B Llama-3.1-70B Llama-3.1-405B
DeepSeek (Guo et al., 2025)	DeepSeek-R1
DeepSeek-Prover (Ren et al., 2025)	DeepSeek-Prover-v2-7B
Goedel-Prover (Lin et al., 2025)	Goedel-Prover-v2-8B Goedel-Prover-v2-32B Goedel-formalizer-v2-8B
Kimina (Wang et al., 2025b)	Kimina-Autoformalizer-7B Kimina-Prover-Distill-8B

Table 5: Models evaluated in our experiments.

We evaluate 18 models across 8 model families, as listed in Table 5. **Closed-source models** are accessed via Snowflake AI Cortex API. Parameter counts for these models are not publicly disclosed, except for GPT-OSS-120B. **Open-source models** are served locally using vLLM. The Llama-3.1 family (8B, 70B, 405B) and DeepSeek-R1 are general-purpose models, while DeepSeek-Prover-v2-7B, Goedel-Prover-v2 (8B, 32B), Goedel-formalizer-v2-8B, Kimina-Autoformalizer-7B, and Kimina-Prover-Distill-8B are fine-tuned on formal mathematical corpora for theorem proving or autoformalization tasks. All models are evaluated in a zero-shot setting with temperature 0.6 and top_p 0.95. For each problem, we generate a single completion and evaluate it against the successor theorems as a test suite. Closed-source model evaluations were conducted between January and February 2026.

B Dataset Details

We constructed our T^2 benchmark by mining high-quality, real-world Lean 4 repositories. Table 7 lists the source projects included in our dataset, covering a diverse range of mathematical domains such as algebra, topology, and number theory. This diversity is crucial for evaluating the generalization capabilities of autoformalization models beyond standard educational problems. For evaluation, we employ a rigorous environment setup using `lake build` within each repository to ensure that all generated theorems are checked against the correct dependencies and definitions.

C NL Annotation Validation

To validate the quality of LLM-generated NL annotations, we randomly sampled 10 annotations from T^2 and manually verified them against the corresponding formal theorems. Each annotation was checked for (1) mathematical accuracy of the described statement, (2) completeness of hypotheses and conclusions, and (3) absence of hallucinated or missing conditions. All 10 annotations were judged correct, as summarized in Table 8.

For a qualitative example, consider the following annotation:

Formal Theorem

```
theorem EmptyShapeIn.rfl {S : Set
    Point} :
    EmptyShapeIn S S := by
  intro _ h
  simp at h
```

Generated NL Annotation

For any set of points S, the set S carves out an empty shape in itself. That is, the convex hull of S contains no points from S other than those already in S.

This NL description contains some redundancy (no points from S other than those already in S), but it correctly captures the semantics of the formal statement. Such minor stylistic issues do not affect the autoformalization task, as the logical content is faithfully conveyed.

D Evaluation Environment Details

All experiments are conducted on a server equipped with 2 NVIDIA A6000 or 1 NVIDIA 6000Pro GPUs, having 49 GB and 98 GB VRAM, respectively. Open-source models are served using vLLM with bfloat16 precision. For models exceeding single-GPU memory (e.g., Llama-3.1-405B, DeepSeek-R1), we use tensor parallelism across 2 GPUs for NVIDIA A6000.

For T^2 problems, we execute `lake build` within each repository's native environment. We set a timeout of 600 seconds per problem for both theorem generation and compilation verification. For Standalone benchmarks like supplementary existing

Overall Statistics				
Benchmark	**# Task**	**Test (Avg.)**	**Prompt (Avg.)**	**Solution (Avg.)**
		#	Char.	Char.
Existing Benchmarks				
Kimina (train)	5	2.00	1208.00	48.00
Numina (train)	105	6.53	1185.65	114.01
combibench (eval)	6	2.67	1229.83	70.33
formalmath (eval)	6	13.00	217.00	66.17
Total / Avg.	**122**	**6.48**	**1141.10**	**106.80**
T^2				
Directed-Topology	438	19.68	5489.52	172.28
Untangle	133	16.71	2413.51	727.90
WhitneyGraustein	4	2.00	177.75	22.00
adele-ring	67	4.52	3548.19	270.45
math2001	1564	50.12	1056.79	54.53
Total / Avg.	**2206**	**40.59**	**2092.78**	**125.00**
T^2 *(Hard)*				
Directed-Topology	6	3.00	5360.17	42.00
adele-ring	17	2.53	3239.24	337.76
math2001	366	6.97	1476.30	45.97
Total / Avg.	**389**	**6.71**	**1613.25**	**58.66**

Table 6: Dataset statistics for T^2 benchmark, including the number of tasks, average successor theorems (Test), and average prompt/solution lengths in characters. Untangle and WhitneyGraustein are absent in Hard set as they have no problem matching Hard set criteria.

Repository	**Github Link**	**Lean Version**	**Difficulty**
adele-ring	`smmercuri/adele-ring_locally-compact/tree/journal`	v4.10.0-rc2	Research
Directed-Topology	`Dominique-Lawson/Directed-Topology-Lean-4`	v4.6.0-rc1	Research
Untangle	`dignissimus/Untangle`	v4.6.1	Research
math2001	`hrmacbeth/math2001`	v4.3.0	Undergraduate
WhitneyGraustein	`MetalCreator666/WhitneyGraustein`	v4.10.0	Research

Table 7: Details of the Source Repositories used in T^2

benchmarks, we use `kimina-lean-server` within a standardized Lean 4.25.0 environment with a timeout of 120 seconds.

E Computation Cost

Table 9 summarizes the approximate computation cost of our experiments.

The dominant cost is closed-source API calls, as each of the 389 Hard problems (and 2,206 full problems) requires a separate generation call per model across 4 context settings. Compilation verification is comparatively inexpensive, as `lake build` runs on a CPU.

F Ablation Study: Input Context

Table 10 presents the full ablation results on the T^2 Hard subsets, varying whether a NL proof and/or successor theorem is provided as context.

On T^2 Hard (Table 10), providing successor theorems yields consistent improvements for most models. GPT-OSS-120B achieves the highest TA of 5.1% in the NL ✗, ST ✓setting. Notably, NL proofs show mixed or negligible effects: for several models (e.g., Claude-4-Sonnet, GPT-5-mini), adding NL proofs without a successor theorem does not improve or even decreases TA compared to the baseline (NL ✗, successor theorem ✗), while adding a successor theorem consistently helps regardless of whether NL is provided.

G Compilation vs. Testing Accuracy

Figure 6 visualizes the divergence between compilation accuracy and TA on the T^2 Hard subset. While compilation rates vary widely across models

Theorem Name	Repository	Verification
`isUniformizer_ne_zero`	adele-ring	Correct
`extensionEmbedding_of_comp_coe`	adele-ring	Correct
`isometry_extensionEmbedding_of_comp`	adele-ring	Correct
`algebraMap_injective`	adele-ring	Correct
`eq_pow_uniformizer_mul_unit`	adele-ring	Correct
`not_isUnit_iff_valuation_lt_one`	adele-ring	Correct
`bijective_extensionEmbedding_of_isReal`	adele-ring	Correct
`outputEnd`	Untangle	Correct
`Int.ModEq`	math2001	Correct
`parseExpression`	Untangle	Correct

Table 8: Manual verification of 10 randomly sampled NL annotations. All annotations were judged to be mathematically accurate.

Category	Time	Cost (USD)
Closed-source API calls	total 12 hrs	approx. 2500
Open-source inference	up to 48 GPU-hrs	–
Compilation verification	96 hrs	–
NL annotation	1 hr	approx. 100
Total	approx. 157 hrs	2600

Table 9: Approximate computation cost of experiments.

(4.3% to 75.6%) and generally improve with scale, TA remains consistently below 5% for nearly all models, visually confirming the persistent semantic gap discussed in Section 6.3.

H Detailed Experimental Results

Table 11 presents the full compilation and testing accuracy across three evaluation settings: existing benchmarks, T^2 (full), and T^2 Hard.

On existing benchmarks, the gap between compilation and testing accuracy is relatively small for most models. For instance, Claude-Sonnet-4.5 achieves 90.6% compilation and 88.1% testing accuracy. This is expected, as these benchmarks consist of isolated problems with a small number of successor theorems, limiting the discriminative power of TA.

On the full T^2 benchmark, a clear gap emerges. Models that compile at 60-80% drop to 24-39% testing accuracy, reflecting the stricter verification imposed by successor theorems in repository-level problems. Notably, testing accuracy across models converges to a narrow range (24-39%) despite wide variation in compilation rates, suggesting that most models can produce syntactically valid code for repository-level problems but struggle equally with semantic correctness.

On T^2 Hard, the gap becomes most pronounced. Compilation rates vary widely (4.3% to 75.6%), yet testing accuracy is uniformly low (1.3%-4.5%). GPT-5-nano achieves the highest compilation accuracy (75.6%) but only 1.5% TA, a 50× gap, while Claude-Sonnet-4.5 compiles at 46.0% but achieves the highest TA (4.5%). This rank reversal further demonstrates that compilation accuracy is a poor predictor of semantic correctness and that the Hard subset effectively isolates the challenge of genuine mathematical reasoning.

I Discussion

We now examine the scope and reliability of TA in light of the limitations above.

While coverage imposes an upper bound on what TA can detect, the risk of false negatives, where a logically correct but differently formulated theorem fails to compile against successor theorems, is minimal in our setup. The type declarations such as `theorem`, `lemma`, and `def`, and the exact name of the target are provided as context (Section 3.1, footnote 1), and successor theorems reference the target by this name. If a generated theorem is logically correct, it should type-check successfully against all successor theorems, leaving little room for false negatives.

The inapplicability of TA to standalone theorems is likewise bounded in practice: across our collected repositories, approximately 1.4% of theorems have no successor theorems. Existing methods, such as BLEU or manual inspection, can be applied to these few cases, while our contribution

Model	NL ✓		NL ✗	
	ST ✗	ST ✓	ST ✗	ST ✓
Closed-Source LLMs				
Claude-3.7-Sonnet	2.6	2.6	2.4	2.6
Claude-4-Sonnet	3.2	4.5	3.2	3.9
Claude-Sonnet-4.5	3.9	4.5	3.0	3.6
GPT-4o-mini	1.3	1.5	1.3	1.5
GPT-5	3.4	3.4	1.9	2.8
GPT-5-mini	3.6	4.1	2.1	2.6
GPT-5-nano	2.1	1.5	2.1	1.5
Open-Source LLMs				
DeepSeek-R1	3.0	4.1	2.8	3.0
GPT-OSS-120B	3.9	4.5	4.7	5.1
Llama-3.1-405B	2.6	2.6	2.4	2.8
Llama-3.1-70B	2.4	2.4	1.9	1.7
Llama-3.1-8B	1.5	1.3	1.5	1.5
Math Specialized Models				
DeepSeek-Prover-v2-7B	2.4	3.2	2.8	3.2
Goedel-Formalizer-v2-8B	1.5	2.4	1.9	2.1
Goedel-Prover-v2-32B	2.1	3.2	1.3	0.9
Goedel-Prover-v2-8B	1.7	2.6	2.4	3.0
Kimina-Autoformalizer-7B	1.3	1.5	1.3	1.5
Kimina-Prover-Distill-8B	2.6	1.7	1.7	2.6

Table 10: Impact of input context on TA (%) on T^2 Hard. We compare two axes: whether an NL proof is provided, and whether a successor theorem is provided (✓) or withheld (✗).

enables automatic semantic evaluation for the remaining 98.6%.

J Future Work

Our results suggest several promising directions for extending TA and T^2. A natural next step is to refine the binary scoring of TA into a graded metric that accounts for partial success in successor theorems compilation, providing a more nuanced signal for model development. Beyond Lean 4, TA is language-agnostic by design and can be extended to any proof assistant that provides a dependency parser and research-level repositories, such as Coq and Isabelle.[5] We chose Lean 4 for its actively growing ecosystem with diverse repositories, and leave extension to other proof assistants for future work. Within Lean 4 itself, expanding T^2 to additional repositories as the ecosystem grows would further improve domain coverage. We also plan to pursue a larger-scale verification of the natural language annotations in consultation with Lean mathematics experts, complementing our initial sample-based assessment.

[5]Coq provides `coq-dpdgraph` and the built-in `coqdep` for dependency parsing, with more than 45 research-level packages on GitHub. Isabelle provides a built-in session dependency system, and the Archive of Formal Proofs contains more than 935 entries.

K Prompts

Following standard conventions in code generation evaluation, we design structured prompts that provide clear task specifications and output format constraints. Similar to how SWE-Bench restricts models to bash-only interactions with explicit formatting requirements, our prompts enforce a strict output protocol: models must produce exactly one Lean 4 code block preceded by a reasoning section. The full prompt templates for informalization and autoformalization are provided in Figure 7 and Figure 8, respectively. We do not perform prompt sensitivity analysis (e.g., varying instruction phrasing or format constraints), as our goal is to benchmark model capabilities under a standardized evaluation protocol rather than optimize for prompt engineering. This is consistent with prior work in both code generation (HumanEval, APPS) and theorem proving (MiniF2F), where a fixed prompt template is used across all models to ensure fair comparison.

	Existing		T^2		T^2 (Hard)	
Model	**Compile**	**TA**	**Compile**	**TA**	**Compile**	**TA**
Closed-Source LLMs						
Claude-3.7-Sonnet	79.5	76.7	75.1	36.8	34.3	2.6
Claude-4-Sonnet	86.6	83.3	81.1	36.0	47.5	4.5
Claude-Sonnet-4.5	90.6	88.1	80.3	38.9	46.0	4.5
GPT-4o-mini	64.3	60.5	59.1	36.7	12.2	1.5
GPT-5	92.4	80.5	85.7	37.7	68.3	3.4
GPT-5-mini	86.3	80.8	86.0	37.9	66.6	4.1
GPT-5-nano	89.6	69.6	88.7	36.6	75.6	1.5
Open-Source LLMs						
DeepSeek-R1	82.3	79.0	70.5	36.3	25.1	4.1
GPT-OSS-120B	53.7	48.4	68.2	32.3	40.0	4.5
Llama-3.1-405B	75.4	72.9	63.4	33.2	27.8	2.6
Llama-3.1-70B	72.2	68.1	65.4	37.0	20.1	2.4
Llama-3.1-8B	54.4	45.8	36.9	24.8	6.4	1.3
Math Specialized Models						
DeepSeek-Prover-v2-7B	63.8	53.9	62.2	30.0	35.5	3.2
Goedel-Formalizer-v2-8B	22.0	19.2	38.7	23.6	22.1	2.4
Goedel-Prover-v2-32B	61.3	54.2	74.6	31.0	54.6	3.2
Goedel-Prover-v2-8B	22.3	18.7	46.1	24.5	37.9	2.6
Kimina-Autoformalizer-7B	14.2	1.0	21.9	20.0	4.3	1.5
Kimina-Prover-Distill-8B	3.0	2.0	28.0	23.4	5.1	1.7

Table 11: Full results across three evaluation settings: existing benchmarks, T^2 (full), and T^2 Hard. Results are grouped by model type.

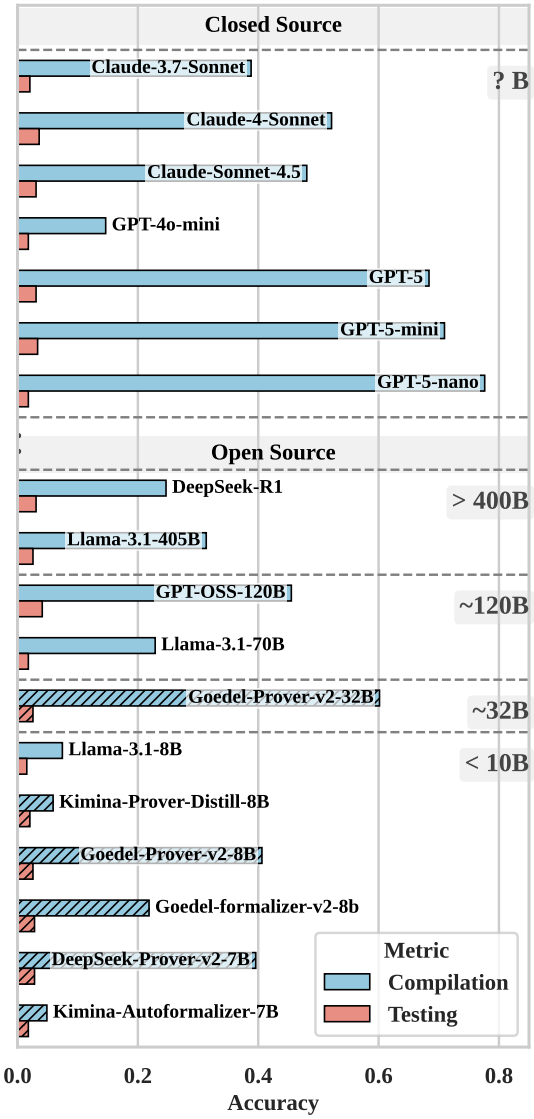


Figure 6: Compilation accuracy and testing accuracy on T^2 Hard. Models are grouped by source availability and parameter scale. Dashed bars indicate LLMs fine-tuned on formal mathematical reasoning tasks.

System Prompt

You are a helpful assistant that can interact multiple times with a computer shell to solve programming tasks. However, for this specific interaction, you are acting as an expert Lean 4 mathematician tasked with translating formal Lean 4 code into a natural language description or proof.
Your response must contain exactly ONE plaintext code block.
Include a THOUGHT section before your code where you explain your reasoning process. Format your response as shown in `<format_example>`.
`<format_example>`

Natural Language

```
THOUGHT: Your reasoning and analysis here
```

Natural Language

```
your_natural_language_description_here
```

`</format_example>`
Failure to follow these rules will cause your response to be rejected.

User Prompt

`<task_description>`
Your task is to provide a natural language description (that is informal proof or problem statement) for the specific Lean 4 declaration.
Target Code Name: `{{ target_code_name }}`
The declaration is part of a larger code sequence.
The Code Context is provided below.

- `Header`: Imports and open namespaces.
- `Code Before`: Code appearing before the target.
- `Target Code`: The specific code you need to describe/informalize.

Header:

Lean 4

```
{{ header }}
```

Code Before:

Lean 4

```
{{ before_target_code }}
```

Target Code:

Lean 4

```
{{ target_code }}
```

Code After:

Lean 4

```
{{ after_target_code }}
```

<instructions>
 # Task Instructions
Overview
You are an expert Lean 4 mathematician. You must provide a clear, concise, and mathematically accurate natural language description of the `Target Code`.

- If `Target Code` is a theorem/proof, provide an informal proof.
- If `Target Code` is a definition, provide a mathematical definition in latex.
- Use standard mathematical terminology (TeX/LaTeX style for formulas is preferred).
- You must not use the exact function name used in lean code. You need to describe in natural language.
- Do not start with which function name you are translating. Just start with the informal proof or problem statement.

Output Coding Rules
1. A **THOUGHT** section where you explain your reasoning:

- Analyze the `Target Code` to understand what it proves or defines.
- Consider the context from `Code Before` and `Code After` if relevant.
- Plan the natural language description.

2. A single plaintext code block with your description.
Format your responses like this:
<format_example>

Natural Language

```
THOUGHT: The target code defines a function \texttt{foo} that adds two numbers
    .
    I will describe it as "The function determines the sum of two integers."
```

Natural Language

```
The function determines the sum of two integers.
```

</format_example>
CRITICAL REQUIREMENTS:

- Your response SHOULD include a THOUGHT section.
- Your response MUST include EXACTLY ONE plaintext block.
- This block MUST contain the COMPLETELY NEW natural language description.
- Do NOT repeat the input code.

</instructions>

AI Response

Natural Language

```
{{ thoughts }}
```

Natural Language

```
{{ target_natural_language }}
```

Figure 7: Prompt template for informalizing Lean 4 code.

System Prompt

You are a helpful assistant that can interact multiple times with a computer shell to solve programming tasks. However, for this specific interaction, you are acting as an expert Lean 4 mathematician tasked with implementing a formal Lean 4 declaration (theorem or definition) based on the provided context.
Your response must contain exactly ONE lean code block.
Include a THOUGHT section before your code where you explain your reasoning process. Format your response as shown in `<format_example>`.
`<format_example>`

Natural Language

```
THOUGHT: Your reasoning and analysis here, explaining how you deduce the
    implementation from the context.
```

Lean 4

```
your_formal_lean_code_here
```

`</format_example>`
Failure to follow these rules will cause your response to be rejected.

User Prompt

`<task_description>`
Your task is to provide the full formal Lean 4 code for the specific declaration named `{{ target_code_name }}`. The declaration is part of a larger code sequence, and you must implement it to fit seamlessly into the provided context.
The Code Context is provided below.

- `Header`: Imports and open namespaces.
- `Code Before`: Code appearing before the target.
- `Target Code Name`: The name of the definition or theorem you need to implement.
- `Natural Language Description`: A description of the target code in natural language.
- `Code After`: Code appearing after the target.

Header:

Lean 4

```
{{ header }}
```

Code Before:

Lean 4

```
{{ before_target_code }}
```

Target Code Name:

Natural Language

```
{{target_code_name}}
```

Natural Language Description:

Natural Language

```
{{target_natural_language}}
```

Code After:

Lean 4

```
{{ after_target_code }}
```

<instructions>
 # Task Instructions
Overview
You are an expert Lean 4 mathematician. You must provide the complete and correct Lean 4 implementation for {{ target_code_name }}.

- If it is a theorem, provide the statement and the full proof (do not use sorry).
- If it is a definition, provide the full definition.
- Pay close attention to Code Before and Code After to infer the correct type signatures, variable names, and logical dependencies.
- Ensure your code compiles and integrates correctly with the surrounding context.

Output Coding Rules
1. A **THOUGHT** section where you explain your reasoning:

- Analyze the Code Before and Code After to deduce the purpose and signature of {{ target_code_name }}.
- Plan the implementation.

2. A single lean code block with your implementation.
Format your responses like this:
<format_example>

Natural Language

```
THOUGHT: The context shows a function \texttt{foo} is used to add two numbers.
I will implement \texttt{foo} as a definition taking two Nats and returning
    their sum.
```

Lean 4

```
def foo (n m : Nat) : Nat := n + m
```

</format_example>

CRITICAL REQUIREMENTS:

- Your response SHOULD include a THOUGHT section.
- Your response MUST include EXACTLY ONE lean block.
- This block MUST contain the COMPLETELY NEW implementation for {{ target_code_name }}.
- Do NOT repeat the Header or Code Before or Code After. Only output the code for {{ target_code_name }}.

</instructions>

AI Response

Natural Language

```
{{thoughts}}
```

Lean 4

```
{{ formal_proof }}
```

Figure 8: Prompt template for autoformalizing to Lean 4 code.