FormalRewardBench: A Benchmark for Formal Theorem Proving Reward Models

Zeynel A. Ulusan[1,2] **Burak S. Akbudak**[3*] **Can S. Erer**[3*] **Gözde Gül Şahin**[1,4]
[1]Koç University, Department of Computer Science and Engineering
[2]Codeway Studios
[3]Boğaziçi University, Department of Computer Engineering
[4]Friedrich-Alexander-Universität Erlangen-Nürnberg, Intelligent Language Systems
https://gglab-ku.github.io/

Abstract

Recent neural theorem provers use reinforcement learning with verifiable rewards (RLVR), where proof assistants provide binary correctness signals. While verifiable rewards are cheap and scalable without reward hacking issues, they suffer from sparse credit assignment: models receive no learning signal from difficult problems where partial progress goes unrewarded. This motivates learned reward models that can evaluate proof quality beyond binary verification. However, comparing reward models is challenging since it typically requires expensive RL training ablations. To address this, we introduce **FormalRewardBench**, the first benchmark for evaluating reward models in formal theorem proving with Lean 4. Our benchmark consists of 250 preference pairs where correct proofs are paired with incorrect variants generated through five expert curated error injection strategies: forced mistakes, minimal single-point variations, verbose incorrect proofs, natural language justification, and Python code injection. We evaluate frontier LLMs (e.g., Claude Opus 4.5), judge LLMs (e.g., CompassJudger-1-14B), general-purpose LLMs (e.g., Qwen2.5-72B-Instruct), and specialized theorem proving models (e.g., DeepSeek-Prover-V2-7B). Our results reveal that frontier LLMs achieve the highest performance (59.8%) while specialized theorem provers perform the worst (24.4%), suggesting that theorem proving ability does not transfer to proof evaluation. We provide further insights on various error injection mechanisms, highlighting the challenging nature of most injection mechanisms. We release **FormalRewardBench** publicly to encourage more research on developing reward models in formal mathematics.

1 Introduction

Formal theorem proving provides machine-verifiable guarantees for mathematical claims, with applications in software verification and formalizing research mathematics. Large language models have achieved remarkable results on this task, including gold-medal-level performance at the International Mathematical Olympiad (Hubert et al., 2026) and solving challenging problems from Putnam competitions. Recent systems like DeepSeek-Prover (Xin et al., 2025; Ren et al., 2025) and Gödel-Prover (Lin et al., 2025) leverage neural language models to generate formal proofs in proof assistants such as Lean 4, achieving strong results on challenging benchmarks like MiniF2F (Zheng et al., 2022) and PutnamBench (Tsoukalas et al., 2024).

Much of this progress comes from reinforcement learning with verifiable rewards (RLVR), where the proof assistant's type checker provides binary correctness signals. Unlike learned reward models that can suffer from overoptimization and reward hacking, verifiable rewards are cheap, scalable, and perfectly accurate. However, this binary feedback creates a

*Equal contribution.

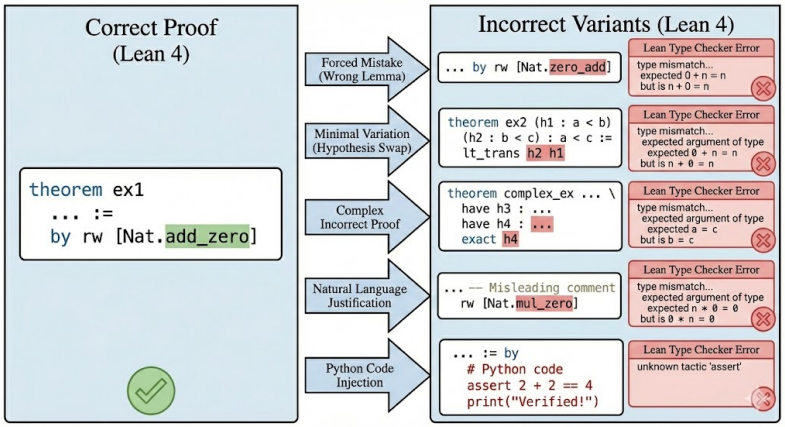

Figure 1: Overview of the error injection pipeline employed in FormalRewardBench. A formally verified correct Lean 4 proof (left) is transformed into incorrect variants (right) using five distinct strategies: Forced Mistakes, Minimal Single-Point Variations, Verbose Incorrect Proofs, Natural Language Justification, and Python Code Injection. While the generated variants remain syntactically valid and appear plausible to language models, they fail formal verification (type checking) as shown by the error messages.

fundamental limitation: **sparse credit assignment**. A proof attempt that makes substantial progress but fails at the final step receives the same zero reward as a completely wrong approach. Models cannot learn from difficult problems where they are on the right track but cannot complete the proof. This is a well-known challenge in reinforcement learning, where sparse rewards limit effective credit assignment (Sutton & Barto, 2018).This sparsity problem motivates learned reward models that can evaluate proof quality beyond binary verification. While methods such as DPO (Rafailov et al., 2023) and GRPO (Xin et al., 2025) bypass explicit reward models, theorem proving benefits from richer reward signals for both training and inference-time proof selection (Fan et al., 2026; Snell et al., 2025). However, comparing such reward models typically requires expensive RL training ablations.

We address this by introducing **FormalRewardBench**, a benchmark for directly evaluating reward models on formal theorem proving. Similar to how RewardBench (Lambert et al., 2025) enables reward model comparison for RLHF without training runs, FormalRewardBench enables comparison for formal reasoning without RL ablations. Our benchmark consists of preference pairs where correct proofs are paired with incorrect variants, allowing direct measurement of whether models can distinguish valid from invalid proofs. We build our benchmark from MiniF2F (Zheng et al., 2022), a dataset of 488 olympiad-level mathematical problems formalized in Lean 4, covering algebra, number theory, and combinatorics from competitions like AMC, AIME, and IMO. We generate incorrect proof variants through five error injection strategies designed by the authors, who have formal training in mathematics, and implemented via carefully crafted LLM prompts (see Figure 1 and §3.1.2). These strategies target different dimensions of proof correctness: (1) *minimal single-point variations* that make small but semantically impactful edits, (2) *natural language justification* that augments incorrect proofs with misleading explanatory comments, (3) *Python code injection* that replaces formal proof steps with executable Python code, (4) *forced LLM mistakes* such as applying the wrong proof rule or referencing the wrong assumption, and (5) *verbose incorrect proofs* with lengthy but fundamentally flawed reasoning. We evaluate four categories of models: frontier LLMs (e.g., Claude Opus 4.5, GPT-5.2), judge LLMs trained for preference evaluation (e.g., Skywork-Critic-Llama-3.1-70B, CompassJudger-1-14B-Instruct), general-purpose LLMs trained on math and code domains (e.g., Qwen2.5-72B-Instruct, DeepSeek-Coder-V2), and theorem proving specialized models (e.g., DeepSeek-Prover-V2-7B, Gödel-Prover-V2-8B), ranging from 7B to 72B parameters. Our results reveal that frontier LLMs achieve the highest performance (70.1% pointwise), judge models substantially out-

perform specialized theorem provers among open-weight models (52.8% vs 12.8%), and error categories vary widely in difficulty.

Our contributions are threefold: (1) **FormalRewardBench**, the first benchmark for evaluating reward models in formal theorem proving, consisting of 250 preference pairs; (2) **five expert-curated error injection strategies** that form a reusable methodology for generating realistic incorrect formal proofs, applicable to other proof assistants, training data augmentation, or curriculum design for theorem provers; and (3) **comprehensive evaluation** revealing that general reward modeling capabilities transfer more effectively than domain-specific training. Developing effective reward models for formal theorem proving could accelerate mathematical discovery and verification. Automated theorem provers with nuanced feedback could assist mathematicians in formalizing complex proofs and verify software and hardware systems more efficiently. However, overreliance on imperfect reward models carries risks: our work highlights current models' limitations, emphasizing the continued necessity of formal verification as the ultimate arbiter of correctness. FormalRewardBench is publicly available at FormalRewardbench.

2 Related Work

Reward Models for Reasoning. RLVR uses external verifiers (proof assistants, code executors, symbolic solvers) to provide binary correctness signals (Lambert, 2025). While scalable, this binary feedback provides sparse reward signals that limit learning from partial progress (Xin et al., 2025). Two approaches provide denser signals: LLM-as-Judge prompts models to compare responses directly (Zheng et al., 2023), and Generative Reward Models (GenRMs) produce natural language explanations before judgment (Mahan et al., 2025; Zhao et al., 2025). While these approaches have been extensively studied for informal reasoning tasks, their effectiveness in formal theorem proving remains unexplored. FormalRewardBench addresses this gap by providing the first benchmark to evaluate these reward modeling approaches on formal proofs.

Reward Model Benchmarks. RewardBench (Lambert et al., 2025) and its successor RewardBench 2 (Malik et al., 2026) evaluate reward models across chat, safety, and reasoning categories. ProcessBench (Zheng et al., 2024) and PRMBench (Song et al., 2025) evaluate process reward models for step-level mathematical reasoning. However, all operate on informal natural language outputs. Formal theorem proving requires syntactic precision, type correctness, and logical soundness (de Moura & Ullrich, 2021), where proofs that appear reasonable informally may contain subtle errors only apparent during formal verification. No existing benchmark evaluates reward models for formal proofs.

Neural Theorem Proving. DeepSeek-Prover (Xin et al., 2025; Ren et al., 2025), Gödel-Prover (Lin et al., 2025), and Kimina-Prover (Wang et al., 2025a) achieve strong results on MiniF2F (Zheng et al., 2022) (488 olympiad problems), ProofNet (Azerbayev et al., 2024) (undergraduate mathematics), and PutnamBench (Tsoukalas et al., 2024) (1697 competition problems). These benchmarks use pass@k metrics, capturing success but not proof quality or error severity. FormalRewardBench complements them by evaluating the ability to distinguish correct from incorrect proofs.

3 Methodology

Our benchmark targets Lean 4, a proof assistant based on dependent type theory where proofs are constructed via tactics and verified by a type checker (see Appendix A and Appendix B for details on Lean 4 and reward modeling formulations). We create a benchmark for evaluating reward models on formal theorem proving that goes beyond binary outcome evaluation. The core design philosophy is controlled difficulty through synthetic error injection. We begin with formally verified correct proofs and systematically generate incorrect variants that satisfy three criteria:

- **Syntactic validity:** The generated incorrect proof must parse as valid Lean code.

- **Semantic plausibility:** Errors should not be trivially obvious (e.g., undefined variables, basic syntax errors).
- **Realistic error patterns:** Mistakes should reflect actual failures of language models on the task.

This approach allows us to create a challenging benchmark that measures genuine formal proof understanding rather than surface-level pattern matching.

3.1 Dataset Construction

We build our benchmark from MiniF2F (Zheng et al., 2022), a dataset of 488 olympiad-level mathematical problems formalized in Lean 4. Problems span algebra, number theory, and combinatorics from competitions including AMC, AIME, and IMO. Data sources vary by strategy: For S1 (Minimal Change), we use correct proofs from DeepSeek-Prover-V2-671B and prompt an LLM to introduce minimal errors. For S2 (NL Justification), we randomly sample incorrect proofs from DeepSeek and Gödel model families, then augment them with misleading comments. For S3, S4, and S5 we prompt LLMs to generate incorrect proofs directly from theorem statements.

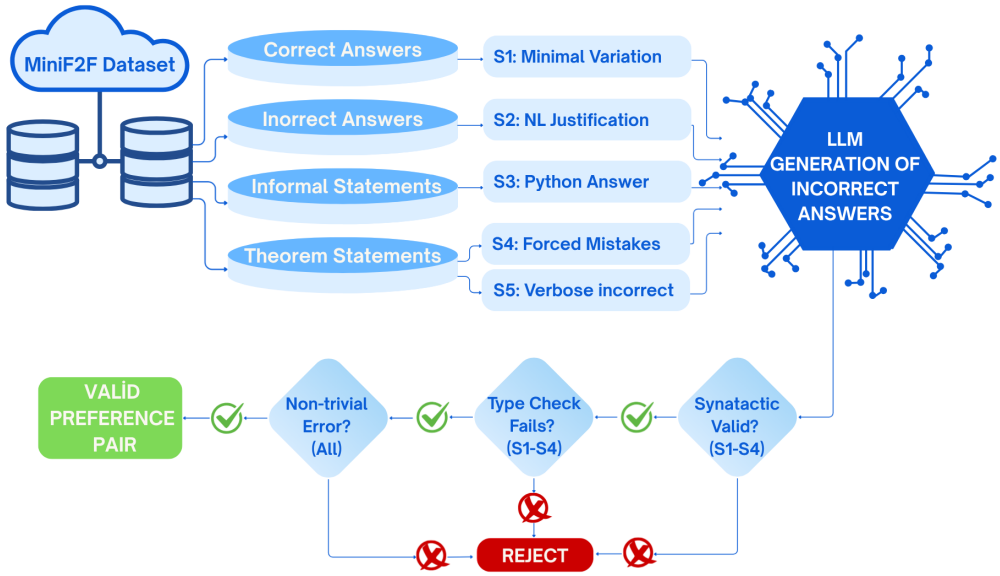

Figure 2: FormalRewardBench generation pipeline. Top: Data flows from MiniF2F through five strategies to LLM generation. Each strategy uses different inputs: correct proofs (S1), incorrect proofs (S2), informal statements (S3), or formal statements (S4, S5). Bottom: Quality control filters. S1–S4 require syntactic validity and type check failure. All strategies filter trivial errors. S3 bypasses Lean validation as Python code.

3.1.1 Generation Pipeline and Quality Control

Figure 2 illustrates our data generation pipeline. We source correct proofs from DeepSeek-Prover-V2-671B (Ren et al., 2025), the strongest open-source theorem prover at the time of data collection, which provides verified solutions for a substantial subset of MiniF2F problems. Incorrect proofs are sampled from the DeepSeek-Prover and Gödel-Prover (Lin et al., 2025) model families, as their failed proof attempts represent realistic LLM failure modes on these problems.

Each error injection strategy uses different source data: S1 (Minimal Variation) takes correct proofs and introduces minimal errors; S2 (NL Justification) takes incorrect proofs and adds misleading comments; S3 (Python Answer) takes informal statements and generates Python

code; S4 (Forced Mistakes) and S5 (verbose incorrect) take formal theorem statements and generate incorrect proofs directly.

For error generation, we use Claude Opus 4.5 as the prompting model across all five strategies. We chose a frontier general-purpose LLM rather than a theorem proving specialist to avoid biasing the generated errors toward any single prover's failure distribution, ensuring broader coverage of possible error patterns. For each theorem and strategy, we generate 3 candidates. Generated proofs pass through three quality filters: (1) *syntactic validation* (S1–S4) verifies that the proof parses in Lean 4; (2) *semantic validation* (S1–S4) confirms that the proof fails Lean's type checker; and (3) a *triviality filter* rejects proofs with superficial errors (e.g., "unexpected token") while accepting semantic errors (e.g., "type mismatch"). S3 (Python Answer) bypasses Lean validation since Python code does not parse as Lean. We randomly sample 50 validated pairs from each strategy, yielding 250 preference pairs total. The choice of 50 pairs per strategy is justified by our stability analysis (Appendix H), which shows that model accuracy stabilizes around this threshold.

Quality Assurance. Our benchmark employs two-layer quality assurance. First, *automatic verification*: for S1–S4, every correct proof passes Lean's type checker and every incorrect proof fails it, providing objective, deterministic correctness labels unlike informal benchmarks that rely on subjective human judgment. Second, *manual inspection*: the authors reviewed a random sample of 50 pairs (10 per strategy) to verify that (i) incorrect proofs are syntactically plausible, (ii) errors are non-trivial and require genuine proof understanding to detect, and (iii) error patterns are realistic.

3.1.2 Error Injection Strategies

Since we source data from actual LLM outputs and use LLMs to generate error variants, our benchmark naturally captures realistic LLM failure modes. All strategies are implemented using carefully designed prompts; details in Appendix D. We provide concrete examples for all five strategies in Appendix C.

Strategy 1: Minimal Single-Point Variations. This strategy creates incorrect proofs by making minimal, surgical single-point modifications with maximal semantic impact. Valid variations include changing a single variable, introducing type mismatches (e.g., replacing `Int` with `Nat`), or using the wrong hypothesis among similar ones (e.g., swapping `h1` and `h2`). Identifying which modifications are semantically meaningful rather than causing trivial syntax errors requires strong reasoning, code comprehension, and instruction-following capabilities, which motivates the use of frontier LLM (Claude Opus 4.5) over rule-based approaches.

Strategy 2: Natural Language Justification. LLM judges are known to exhibit verbosity bias, systematically favoring responses with more detailed explanations regardless of correctness (Zheng et al., 2023; Saito et al., 2023; Ye et al., 2025). Exploiting this, we augment incorrect proofs with natural language comments that justify the flawed reasoning as if it were correct. This tests whether reward models rely on surface-level explanations or genuinely verify proof correctness.

Strategy 3: Python Code Injection. LLMs exhibit a well-documented preference for Python due to its dominance in pretraining corpora (Twist et al., 2025). This strategy exploits this bias by replacing Lean proof steps with Python code that may computationally solve the problem correctly. While the code itself may be valid, using Python when a formal Lean proof is required is fundamentally wrong. This tests task adherence.

Strategy 4: Forced LLM Mistakes. This strategy generates incorrect proofs containing common formal proof errors likely made by language models. To identify common error patterns, we analyzed correct and incorrect proof pairs generated by theorem proving models (DeepSeek-Prover and Gödel-Prover families), prompting Claude to categorize the mistakes (prompt in Appendix D). This analysis revealed frequent error patterns such as incorrect application of tactics, wrong hypothesis selection, and incorrect lemma instantiation.

Strategy 5: Verbose Incorrect Proofs. Large language models are known to produce unnecessarily long reasoning chains (Hassid et al., 2026). This strategy generates unnecessarily

long proofs with many proof steps and intermediate lemmas. We explicitly prompt the LLM to produce sophisticated-looking proofs with multiple techniques and lengthy reasoning chains, while ensuring the final proof is incorrect. These proofs test whether reward models can track correctness over extended reasoning or are deceived by superficial verbosity.

3.2 Evaluation Protocol

We evaluate reward models in both pointwise and pairwise settings. Given a Lean theorem statement T and two candidate proofs P_{correct} and $P_{\text{incorrect}}$, a reward model R is evaluated on its ability to correctly identify P_{correct} as superior to $P_{\text{incorrect}}$. We evaluate models in two settings based on their scoring mechanism:

In **pointwise evaluation**, the model scores each proof independently; a sample is correct if $r(T, P_{\text{correct}}) > r(T, P_{\text{incorrect}})$. In **pairwise evaluation**, the model directly compares both proofs and outputs a preference judgment. Accuracy is the mean correctness over all N examples. For pairwise methods, we mitigate position bias by requiring consistent predictions across both orderings:

$$\text{Correct}^{(i)}_{\text{cons}} = \text{Correct}^{(i)} \wedge \text{Correct}^{(i)}_{\text{swapped}} \tag{1}$$

4 Experimental Setup

We evaluate four categories of models using nucleus sampling (temperature= 1.0, top-$p = 0.75$) with no additional fine-tuning. Each model receives the same prompt template per evaluation setting (pointwise or pairwise), and no model-specific prompt engineering is applied. **Frontier LLMs** are large-scale proprietary models accessed via API, including Claude Opus 4.5, Claude Sonnet 4.5, GPT-5.2, GPT-4.1, GPT-4o, GPT-5.1, and Gemini 2.5 Flash; we select these models to represent the strongest available general reasoning capabilities across major providers. **Judge LLMs** are trained explicitly on preference data and achieve strong performance on RewardBench (Lambert et al., 2025), including Selene-1-70B, LMUnit-72B (Saad-Falcon et al., 2025), CompassJudger-1-7B and -14B (Cao et al., 2024), Skywork-Critic-70B and -8B (Wang et al., 2025b), and RISE-Judge-7B (Yu et al., 2025); we select top-performing models from RewardBench's reasoning category to cover a range of model sizes. **General-Purpose LLMs** are strong instruction-following models trained on math and code domains, including Qwen2.5-72B-Instruct (Qwen et al., 2025), Qwen2.5-Coder-32B-Instruct (Hui et al., 2024), DeepSeek-Coder-V2-Lite (Guo et al., 2024), and Qwen2.5-Math-7B-Instruct (Yang et al., 2024); we select these as representatives of open-weight models with strong mathematical reasoning. **Theorem Proving Specialized Models** are specifically trained for formal proof generation in Lean, including DeepSeek-Prover-V2-7B (Ren et al., 2025), DeepSeek-Prover-V1.5-SFT, DeepSeek-Prover-V1.5-RL (Xin et al., 2025), Gödel-Prover-V2-32B, Gödel-Prover-V2-8B (Lin et al., 2026), and Gödel-Prover-SFT (Xin et al., 2025); we select the most widely used Lean-specialized models to represent this category.Frontier LLMs are proprietary models accessed via API. For open-weight models, we cite the corresponding technical reports where available.

5 Results and Discussion

Table 1 presents model performance across both evaluation settings. The results reveal a clear hierarchy: frontier LLMs dominate (Claude Opus 4.5: 70.1% pointwise, 59.8% pairwise), followed by judge LLMs, general-purpose LLMs, and finally theorem proving specialized models.

The poor performance of specialized provers is counterintuitive. DeepSeek-Prover and Gödel-Prover families achieve state-of-the-art pass@k on MiniF2F, yet struggle at evaluating proofs. We attribute this to a generation–evaluation gap. These models are trained to produce correct proofs via SFT and RLVR, but are not trained to detect errors or learn from incorrect proofs. As a result, they can generate valid proofs without developing the ability to assess correctness. This is consistent with prior findings that models can produce

Table 1: Overall performance on FormalRewardBench. Left: models ranked by pointwise accuracy. Right: models ranked by pairwise accuracy (position-consistent). Models appear in both columns; ranking differences highlight that the two settings measure complementary capabilities. Results grouped by model family can be found in AppendixE

Ranked by Pointwise

Model	Acc.
Claude Opus 4.5[F]	**70.1**
Claude Sonnet 4.5[F]	62.0
Con-J-Qwen2-7B[J]	52.8
Gemini 2.5 Flash[F]	50.9
Claude Sonnet 4[F]	49.8
GPT-5.2[F]	48.9
Selene-1-70B[J]	46.8
GPT-4.1[F]	44.0
GPT-4o[F]	44.0
CompassJudger-7B[J]	43.6
GPT-5.1[F]	41.8
LMUnit-72B[J]	41.2
CompassJudger-14B[J]	40.3
Qwen2.5-72B-Inst.[G]	39.8
Skywork-Critic-70B[J]	39.2
DS-Coder-V2-Lite[G]	38.3
Gödel-V2-32B[S]	36.4
Gödel-SFT[S]	36.4
Qwen2.5-Coder-32B[G]	36.4
RISE-Judge-7B[J]	18.9
DS-Prover-V2-7B[S]	13.7
Prover-V1.5-SFT[S]	12.6
Prover-V1.5-RL[S]	11.7
Qwen2.5-Math-7B[G]	8.9
Skywork-Critic-8B[J]	0.0
Gödel-V2-8B[S]	0.0

Ranked by Pairwise

Model	Acc.
Claude Opus 4.5[F]	**59.8**
Claude Sonnet 4.5[F]	45.7
Selene-1-70B[J]	44.4
Con-J-Qwen2-7B[J]	42.8
GPT-5.2[F]	39.7
Qwen2.5-Coder-32B[G]	37.6
LMUnit-72B[J]	36.8
CompassJudger-14B[J]	35.2
Claude Sonnet 4[F]	32.4
GPT-4.1[F]	32.1
RISE-Judge-7B[J]	32.0
CompassJudger-7B[J]	30.8
GPT-5.1[F]	30.2
DS-Coder-V2-Lite[G]	28.0
Qwen2.5-72B-Inst.[G]	27.6
Skywork-Critic-70B[J]	25.6
Gemini 2.5 Flash[F]	25.2
Gödel-V2-32B[S]	24.4
GPT-4o[F]	23.1
Skywork-Critic-8B[J]	22.0
DS-Prover-V2-7B[S]	9.4
Prover-V1.5-RL[S]	9.2
Prover-V1.5-SFT[S]	6.4
Gödel-SFT[S]	0.8
Gödel-V2-8B[S]	0.4
Qwen2.5-Math-7B[G]	0.0

[F]Frontier [J]Judge [G]General-Purpose [S]Specialized

plausible hypotheses but fail to verify or use them (Qiu et al., 2024). Judge models, trained on preference data where distinguishing good from bad responses is the objective, transfer this ability to formal proofs despite lacking Lean-specific training.

Model size alone does not determine performance, the 7B Con-J-Qwen2 (52.8% pointwise) outperforms several 70B+ models, suggesting training objective matters more than scale.

5.1 Interpreting pointwise vs pairwise discrepancies.

Table 1 ranks models separately by pointwise and pairwise accuracy, expressing notable ranking differences. This is expected: the two settings measure different capabilities. Pointwise evaluation tests whether a model can assign meaningful absolute quality scores to proofs independently, while pairwise evaluation tests whether it can directly compare two proofs and consistently identify the better one regardless of presentation order. Models with strong internal scoring but high position bias (e.g., Gödel-Prover-SFT: 36.4% pointwise, 0.8% pairwise) score well pointwise but collapse under our consistency requirement. Conversely, models that struggle with absolute scoring but benefit from direct comparison (e.g., RISE-Judge: 18.9% pointwise, 32.0% pairwise) perform better pairwise. We recommend evaluating on both settings, as each reveals complementary strengths and weaknesses.

5.2 Performance by Error Category

Table 2 reveals a clear difficulty gradient across error strategies. S3 (Python Injection) is easiest. Most frontier and judge models achieve 94–100%, as detection requires only language identification rather than proof understanding. Yet specialized provers struggle even here, with most scoring below 50%. S5 (Verbose Incorrect) and S4 (Forced Mistakes) are hardest. S5 proofs use sophisticated structure to camouflage flawed reasoning, while S4 errors require tracking proof state across tactic applications. Even the best model achieves only 60% on S5 and 50% on S4. Judge LLMs show a sharply bimodal pattern: near perfect on S3 but collapsing to 0–2% on S5, revealing that preference training develops surface level discrimination but not deep formal reasoning. Frontier LLMs exhibit more balanced profiles across categories.

Table 2: Pairwise accuracy (%) by error strategy.

Model	S1 (Minimal)	S2 (NL)	S3 (Python)	S4 (Forced)	S5 (Verbose)
Frontier LLMs					
Claude Opus 4.5	**46**	**52**	72	**50**	**60**
Claude Sonnet 4.5	42	32	56	42	42
GPT-5.2	26	40	38	46	36
Judge LLMs					
Selene-1-70B	44	44	96	18	20
LMUnit-72B	32	24	94	12	22
CompassJudger-14B	28	34	**100**	12	2
RISE-Judge-7B	16	40	96	8	0
Skywork-Critic-70B	22	10	80	6	10
General-Purpose LLMs					
Qwen2.5-Coder-32B	32	44	94	14	4
DS-Coder-V2-Lite	22	12	72	22	12
Specialized					
Gödel-V2-32B	8	28	46	18	22
DS-Prover-V2-7B	16	24	48	20	10
Prover-V1.5-RL	2	6	16	12	10

5.3 Position Bias

We observe substantial position bias across all model categories in our pairwise evaluation. Most models systematically prefer whichever proof is presented first, regardless of correctness. This preference is particularly pronounced among specialized theorem provers and weaker judge models, while a few models exhibit the reverse pattern, favoring the second proof.

Table 3 reports normal (correct-first), reversed (correct-second), consistency, and agreement metrics. Claude Opus 4.5 shows moderate bias (85.0% vs 69.7%) while maintaining the highest consistency (70.4%). In contrast, DeepSeek-Prover-V2-7B exhibits extreme bias (94.9% vs 9.8%), with consistency of only 9.9%. This also explains why its pointwise score (13.7%) exceeds its pairwise score (9.4%), since pointwise evaluation is immune to position bias.

RISE-Judge-Qwen2.5-7B and GPT-4o show the opposite pattern, preferring the second proof. Models with high agreement but low accuracy, such as Qwen2.5-Math-7B (98.4% agreement, 0% accuracy), consistently select the wrong proof. The most desirable profile, high agreement with high accuracy, is best approximated by Claude Opus 4.5 (64.9%, 59.8%) and Selene-1-70B (78.4%, 44.4%).Our position-consistent accuracy metric accounts for this by requiring correct predictions in both orderings, which is why pairwise scores are consistently lower than pointwise scores in Table 1.

Table 3: Position bias analysis. Normal: correct proof first. Reversed: correct proof second. Cons.: consistency of correct predictions. Agree.: same answer in both orderings.

Model	Normal	Reversed	Cons.	Agree.
Frontier LLMs				
Claude Opus 4.5	85.0	69.7	70.4	64.9
Claude Sonnet 4.5	72.2	62.8	63.3	56.4
GPT-5.2	69.7	58.6	57.1	51.3
GPT-4.1	61.1	53.4	52.5	49.6
Gemini 2.5 Flash	48.3	44.4	52.2	57.7
GPT-4o	45.7	53.4	43.2	47.0
Judge LLMs				
Selene-1-70B	50.0	60.4	73.5	78.4
LMUnit-72B	66.4	40.8	55.4	66.4
CompassJudger-14B	55.6	38.8	63.3	76.0
RISE-Judge-7B	33.2	58.4	54.8	72.4
Theorem Proving Specialized				
Gödel-V2-32B	37.2	34.4	65.6	77.2
Prover-V2-7B	**94.9**	9.8	*9.9*	14.1
Prover-V1.5-RL	36.0	20.4	25.6	62.0
Gödel-V2-8B	10.8	4.8	3.7	85.2

5.4 Why Does Proof Generation Not Transfer to Evaluation?

We hypothesize that the generation–evaluation gap could be due to three factors. First, *training distribution mismatch*: provers are trained on correct proofs only and may not learn to identify errors, while judge models are explicitly trained on preference pairs. Second, *different cognitive demands*: generating a valid tactic sequence may differ from detecting where reasoning breaks down, analogous to how writing code does not imply bug finding ability. Third, *breadth of training*: judge models may develop transferable evaluation heuristics from diverse preference data that apply to formal proofs even without Lean-specific training.

6 Conclusion and Future Work

We introduced FormalRewardBench, the first benchmark for evaluating reward models in formal theorem proving, consisting of 250 preference pairs across five error injection strategies in Lean 4. Our evaluation reveals three findings: (1) judge models substantially outperform specialized provers in distinguishing correct from incorrect proofs, exposing a generation–evaluation asymmetry; (2) most models perform at or below random baseline; and (3) error categories exhibit a clear difficulty gradient, reflecting the varying levels of semantic complexity intentionally targeted by our error injection strategies.

Future work should focus on developing reward models that provide reliable dense feedback for failed proof attempts, training theorem provers to identify and critique errors in addition to generating proofs, and exploring hybrid approaches that combine learned reward models with formal verification. Another direction is process-level evaluation for step-by-step supervision, as well as extending the benchmark to other proof assistants such as Coq and Isabelle. Finally, it remains an open question whether improvements on this benchmark translate to downstream gains in RLVR training. FormalRewardBench is publicly available at FormalRewardbench.

Limitations

Our benchmark relies primarily on automatic verification, and only a subset of examples (50 pairs) were manually inspected. While this ensures scalability, some generated examples may contain unintended artifacts or inconsistencies.We focus on Lean 4, and although the methodology could be extended, we do not evaluate transfer to other proof assistants such

as Coq or Isabelle.Our five error injection strategies do not cover all possible failure modes (e.g., incorrect induction schemes or deeper semantic errors), which may limit coverage of certain reasoning patterns.Finally, we evaluate single-turn preference judgments and do not consider process-level or step-by-step evaluation.

Acknowledgments

This work has been supported by the Scientific and Technological Research Council of Türkiye (TÜBİTAK) as part of the project "Automatic Learning of Procedural Language from Natural Language Instructions for Intelligent Assistance" with the number 121C132. We gratefully acknowledge KUIS AI Lab for providing computational support. We also thank TÜBİTAK ULAKBİM High Performance and Grid Computing Center (TRUBA) and the MareNostrum supercomputer at the Barcelona Supercomputing Center for providing access to computational resources used in this work.

References

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward William Ayers, and Dragomir Radev. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics, 2024. URL https://openreview.net/forum?id=Zix86UbMGh.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

Maosong Cao, Alexander Lam, Haodong Duan, Hongwei Liu, Songyang Zhang, and Kai Chen. Compassjudger-1: All-in-one judge model helps model evaluation and evolution, 2024. URL https://arxiv.org/abs/2410.16256.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. https://leanprover.github.io/, 2021. Accessed: 2024.

Zhiyuan Fan, Dadi Guo, Zhitao He, Elias Stengel-Eskin, Mohit Bansal, and Yi R. Fung. Proof-verifier: Enabling reinforcement learning from verifiable rewards for mathematical theorem proving, 2026. URL https://openreview.net/forum?id=FAe9Gts2Qd.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

Michael Hassid, Gabriel Synnaeve, Yossi Adi, and Roy Schwartz. Don't overthink it. preferring shorter thinking chains for improved LLM reasoning, 2026. URL https://openreview.net/forum?id=nhUlA8iMkD.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026. doi: 10.1038/s41586-025-09833-y. URL https://doi.org/10.1038/s41586-025-09833-y.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

Nathan Lambert. Reinforcement learning from human feedback, 2025. URL https://arxiv.org/abs/2504.12501.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, Lester James V. Miranda, Bill Yuchen Lin, Khyathi Raghavi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. Rewardbench: Evaluating reward models for language modeling. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, pp. 1755–1797. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.FINDINGS-NAACL.96. URL https://doi.org/10.18653/v1/2025.findings-naacl.96.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover: A frontier model for open-source automated theorem proving. *CoRR*, abs/2502.07640, 2025. doi: 10.48550/ARXIV.2502.07640. URL https://doi.org/10.48550/arXiv.2502.07640.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=j4C0nALrgK.

Dakota Mahan, Duy Van Phung, Alon Albalak, Rafael Rafailov, Chase Blagden, Nathan Lile, Louis Castricato, Jan-Philipp Fränken, and Chelsea Finn. Generative reward models, 2025. URL https://openreview.net/forum?id=MwU2SGLKpS.

Saumya Malik, Valentina Pyatkin, Sander Land, Jacob Morrison, Noah A. Smith, Hannaneh Hajishirzi, and Nathan Lambert. Rewardeval: Advancing reward model evaluation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=fb0G86Dewb.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

Linlu Qiu, Liwei Jiang, Ximing Lu, Melanie Sclar, Valentina Pyatkin, Chandra Bhagavatula, Bailin Wang, Yoon Kim, Yejin Choi, Nouha Dziri, and Xiang Ren. Phenomenal yet puzzling: Testing inductive reasoning capabilities of language models with hypothesis refinement. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=bNt7oajl2a.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=HPuSIXJaa9.

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. *CoRR*, abs/2504.21801, 2025. doi: 10.48550/ARXIV.2504.21801. URL https://doi.org/10.48550/arXiv.2504.21801.

Jon Saad-Falcon, Rajan Pathe Vivek, William Berrios, Nandita Shankar Naik, Matija Franklin, Bertie Vidgen, Amanpreet Singh, Douwe Kiela, and Shikib Mehri. LMUNIT: Fine-grained evaluation with natural language unit tests. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 3303–3324, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.176. URL https://aclanthology.org/2025.findings-emnlp.176/.

Keita Saito, Akifumi Wachi, Koki Wataoka, and Youhei Akimoto. Verbosity bias in preference labeling by large language models. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*, 2023. URL https://openreview.net/forum?id=magEgFpK1y.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4FWAwZtd2n.

Mingyang Song, Zhaochen Su, Xiaoye Qu, Jiawei Zhou, and Yu Cheng. Prmbench: A fine-grained and challenging benchmark for process-level reward models. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pp. 25299–25346. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.acl-long.1230/.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. Putnambench: A multilingual competition-mathematics benchmark for formal theorem-proving. In *AI for Math Workshop @ ICML 2024*, 2024. URL https://openreview.net/forum?id=vqW1VRFeVP.

Lukas Twist, Jie M. Zhang, Mark Harman, Don Syme, Joost Noppen, Helen Yannakoudakis, and Detlef Nauck. A study of llms' preferences for libraries and programming languages, 2025. URL https://arxiv.org/abs/2503.17181.

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 https://arxiv.org/abs/2504.11354.

PeiFeng Wang, Austin Xu, Yilun Zhou, Caiming Xiong, and Shafiq Joty. Direct judgement preference optimization. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 1979–2009, Suzhou, China, November 2025b. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.103. URL https://aclanthology.org/2025.emnlp-main.103/.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Haowei Zhang, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=I4YAIwrsXa.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement, 2024. URL https://arxiv.org/abs/2409.12122.

Jiayi Ye, Yanbo Wang, Yue Huang, Dongping Chen, Qihui Zhang, Nuno Moniz, Tian Gao, Werner Geyer, Chao Huang, Pin-Yu Chen, Nitesh V Chawla, and Xiangliang Zhang. Justice or prejudice? quantifying biases in LLM-as-a-judge. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=3GTtZFiajM.

Jiachen Yu, Shaoning Sun, Xiaohui Hu, Jiaxu Yan, Kaidong Yu, and Xuelong Li. Improve LLM-as-a-judge ability as a general ability. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 14099–14115, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.712. URL https://aclanthology.org/2025.emnlp-main.712/.

Jian Zhao, Runze Liu, Kaiyan Zhang, Zhimu Zhou, Junqi Gao, Dong Li, Jiafei Lyu, Zhouyi Qian, Biqing Qi, Xiu Li, and Bowen Zhou. Genprm: Scaling test-time compute of process reward models via generative reasoning. *CoRR*, abs/2504.00891, 2025. doi: 10.48550/ARXIV.2504.00891. URL https://doi.org/10.48550/arXiv.2504.00891.

Chujie Zheng, Zhenru Zhang, Beichen Zhang, Runji Lin, Keming Lu, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. Processbench: Identifying process errors in mathematical reasoning. volume abs/2412.06559, 2024. doi: 10.48550/ARXIV.2412.06559. URL https://doi.org/10.48550/arXiv.2412.06559.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=uccHPGDlao.

A Background: Formal Theorem Proving in Lean 4

Lean is a proof assistant based on dependent type theory. Theorem proving in Lean produces machine-verifiable proofs where every step is justified by the underlying type system. A theorem statement declares a mathematical claim with precise type annotations:

```
theorem add_comm (n m : Nat) : n + m = m + n := by
  sorry
```

This declares that natural number addition is commutative. The type signature `(n m : Nat)` introduces two natural numbers, and `n + m = m + n` specifies the proposition to prove. The keyword `by` initiates tactic mode, and `sorry` is a placeholder for an incomplete proof.

Proofs in Lean can be written in two modes: **term mode**, where proofs are constructed directly using lambda calculus expressions, and **tactic mode**, where proofs are built interactively using tactics such as `rw` (rewrite), `simp` (simplification), and `exact` (term provision) to transform proof states until no goals remain.

Lean's type checker verifies that proof terms match the types specified by theorem statements. If verification succeeds, the theorem is formally proved. Otherwise, Lean reports type mismatches, missing hypotheses, or tactic failures.

```
theorem add_comm (n m : Nat) : n + m = m + n := by
  rw [Nat.add_comm]
```

This applies `Nat.add_comm` from Lean's library to complete the proof. While we focus on Lean 4, our methodology generalizes to other proof assistants like Coq (Bertot & Castéran, 2004) and Isabelle (Nipkow et al., 2002) that share similar tactic-based proof construction.

B Background: Reward Model Formulations

Given a prompt x and two candidate responses y_1 and y_2, reward models determine which response is preferred.

LLM-as-Judge directly prompts a language model to compare two responses and output a preference judgment $I \in \{1, 2\}$.

Generative Reward Models (GenRMs) extend LLM-as-Judge by generating natural language reasoning before the final judgment, providing interpretable explanations for preference decisions.

C Error Injection Examples

Below we provide concrete examples for each error injection strategy.

S1: Minimal Single-Point Variations.

```
-- Correct proof
theorem ex2 (h1 : a < b) (h2 : b < c) : a < c :=
  lt_trans h1 h2

-- Incorrect: swapped hypotheses
theorem ex2 (h1 : a < b) (h2 : b < c) : a < c :=
  lt_trans h2 h1  -- ERROR: type mismatch
```

S2: Natural Language Justification.

```
-- Incorrect proof with misleading comments
theorem ex4 (n : Nat) : n * 0 = 0 := by
  -- First rewrite using commutativity
  rw [Nat.mul_comm]
  -- Now n * 0 becomes 0 * n, apply zero_mul
  rw [Nat.mul_zero]  -- ERROR: should be zero_mul
```

S3: Python Code Injection.

```
-- Incorrect: Python instead of Lean proof
theorem ex5 : 2 + 2 = 4 := by
  # Python verification
  assert 2 + 2 == 4
  print("Verified!")
```

S4: Forced LLM Mistakes.

```
-- Correct proof
theorem ex1 (n : Nat) : n + 0 = n := by
  rw [Nat.add_zero]

-- Incorrect: wrong lemma (zero_add instead of add_zero)
theorem ex1 (n : Nat) : n + 0 = n := by
  rw [Nat.zero_add]  -- ERROR: zero_add proves 0 + n = n
```

S5: Verbose Incorrect Proofs.

```
-- Correct proof
theorem ex6 (n : Nat) : n + 0 = n := by
  rw [Nat.add_zero]

-- Incorrect: verbose proof with subtle flaw
theorem ex6 (n : Nat) : n + 0 = n := by
  have h1 : n + 0 = 0 + n := by rw [Nat.add_comm]
  have h2 : 0 + n = n := by rw [Nat.zero_add]
  linarith  -- ERROR: linarith cannot close this goal; exact h2 needed
```

D Prompt Templates

Below we provide the prompts used for each error injection strategy. All prompts are used with Claude Opus 4.5. The theorem statement is inserted at the {} placeholder. For strategies requiring a correct proof as input (S1, S2), the proof is inserted at the second placeholder.

We sample from two prompt variants for S1 strategy: *minimal difference pairs* and *subtle hypothesis confusion*.

S1: Minimal Single-Point Variations (Variant A)

System
You are a Lean 4 proof assistant that generates proofs with single-point variations.

Task
Given a correct Lean 4 proof, create a nearly identical version that differs by EXACTLY ONE small change that makes it incorrect.

Valid single-point changes

- Change ONE lemma name (e.g., add_comm → mul_comm)
- Change ONE variable name (e.g., n → m, h1 → h2)
- Change ONE tactic name (e.g., ring → omega, simp → rfl)
- Change ONE operator (e.g., + → *, <= → <)
- Remove/add ONE hypothesis in a single tactic call

Rules

- The modification must be MINIMAL (1–3 characters ideally)
- The incorrect proof must be syntactically valid
- The change should look plausible at first glance
- Keep all other parts of the proof EXACTLY the same
- The error should be logical, not a syntax error

User
Generate a minimally different INCORRECT version of this proof.

Input
Theorem: `{theorem}`
Proof: `{proof}`

Output
Return ONLY the modified proof with ONE subtle change.

S1: Minimal Single-Point Variations (Variant B)

System
You are a Lean 4 proof assistant that generates proofs with hypothesis misuse.

Task
Given a correct Lean 4 proof with multiple hypotheses, create a version that uses the WRONG hypothesis.

Strategies for hypothesis confusion

- Swap h1 and h2 where both have similar types
- Use ha instead of hb when both are inequalities
- Reference wrong assumption in calc chains
- Apply wrong induction hypothesis

- Confuse hypotheses with similar names

Rules

- Both hypotheses must exist in the context
- The types should be similar enough to be confusing
- Keep the rest of the proof structure identical
- The error should be in hypothesis SELECTION, not tactics

User
Generate a version that uses the WRONG hypothesis.

Input
Theorem: `{theorem}`
Proof: `{proof}`

Output
Return ONLY the modified proof with hypothesis confusion.

S2: Natural Language Justification

System
You are a technical assistant. Your task is to add clear, technical natural language comments to proofs, explaining the reasoning and steps. Do not focus on correctness, only on explaining the approach and logic.

User
Add natural language comments to explain the proof below. Write clear, technical comments that explain the reasoning. Only try to validate the proof with natural language explanations.

Input
THEOREM: `{theorem}`
CODE/PROOF: `{incorrect_proof}`

Output
Return the original proof with added natural language comments explaining the approach and reasoning.

S3: Python Code Injection

System
You are a helpful Python programming assistant. Your task is to write Python code that solves mathematical problems, given a Lean theorem as context. Output only the Python code, with no explanations or commentary.

User
Write Python code to solve the following theorem.

Input
Lean theorem (for context): `{theorem}`

Output
Output ONLY Python code.

S4: Forced LLM Mistakes

System
You are a Lean 4 proof assistant that makes common formal proof errors that language models typically encounter.

Task
Complete Lean 4 theorems with proofs that contain realistic formal proof errors:

1. Incorrect application of tactics (simp, rw, apply)
2. Wrong hypothesis selection or variable scoping
3. Mismatched term types or implicit argument errors
4. Incomplete case analysis or missing edge cases
5. Incorrect use of induction hypotheses
6. Wrong lemma instantiation or theorem application
7. Scope errors with bound variables
8. Misunderstanding proof state after tactic application

Rules

- Use valid Lean 4 syntax
- Make errors from automated proof generation
- Focus on formal reasoning mistakes, not basic math errors
- The proof should appear plausible at first glance

User
Complete the following Lean 4 theorem with an INCORRECT proof containing a typical proof generation error.

Input
`{theorem}`

Output
Output only the completed proof, starting after `by`.

S5: Verbose Incorrect Proofs

System
You are a Lean 4 proof assistant. Your task is to produce a long, complex Lean 4 proof with many sophisticated steps and intermediate lemmas. Use advanced tactics, case splits, and multiple proof techniques.

User
Produce a LONG, COMPLEX Lean 4 proof with many sophisticated steps and intermediate lemmas.

Constraints

- Use many steps (lemmas, case splits, advanced tactics)
- Make the proof elaborate and thorough
- Include multiple proof techniques
- Output only Lean 4 code of the proof body after `by`

Input
Theorem: `{theorem}`

Output
Return only the proof body after `by`.

E Full Overall Results by Model Family

Table 4 presents overall performance grouped by model category, corresponding to the ranked view in Table 1.

Table 4: Overall performance on FormalRewardBench grouped by model category. Pointwise: independent scoring accuracy. Pairwise: accuracy with position consistency requirement.

Model	Pointwise	Pairwise
Frontier LLMs		
Claude Opus 4.5	**70.1**	**59.8**
Claude Sonnet 4.5	62.0	45.7
Gemini 2.5 Flash	50.9	25.2
GPT-5.2	48.9	39.7
GPT-4.1	44.0	32.1
GPT-4o	44.0	23.1
GPT-5.1	41.8	30.2
Claude Sonnet 4	49.8	32.4
Judge LLMs		
Con-J-Qwen2-7B	52.8	42.8
Selene-1-Llama-3.3-70B	46.8	44.4
CompassJudger-1-7B	43.6	30.8
LMUnit-Qwen2.5-72B	41.2	36.8
CompassJudger-1-14B	40.3	35.2
Skywork-Critic-Llama-3.1-70B	39.2	25.6
RISE-Judge-Qwen2.5-7B	18.9	32.0
Skywork-Critic-Llama-3.1-8B	0.0	22.0
General-Purpose LLMs		
Qwen2.5-72B-Instruct	39.8	27.6
DeepSeek-Coder-V2-Lite	38.3	28.0
Qwen2.5-Coder-32B-Instruct	36.4	37.6
Qwen2.5-Math-7B-Instruct	8.9	0.0
Theorem Proving Specialized		
Gödel-Prover-V2-32B	36.4	24.4
Gödel-Prover-SFT	36.4	0.8
DeepSeek-Prover-V2-7B	13.7	9.4
DeepSeek-Prover-V1.5-SFT	12.6	6.4
DeepSeek-Prover-V1.5-RL	11.7	9.2
Gödel-Prover-V2-8B	0.0	0.4

F Full Error Category Results

Table 5 presents pairwise accuracy by error strategy for all evaluated models, including those omitted from the main text for space.

Table 5: Full pairwise accuracy (%) by error strategy. S1: verbose incorrect, S2: Minimal Variations, S3: NL Justification, S4: Forced Mistakes, S5: Python Injection.

Model	S1	S2	S3	S4	S5
Frontier LLMs					
Claude Opus 4.5	**60**	**46**	**52**	**50**	72
Claude Sonnet 4.5	42	42	32	42	56
GPT-5.2	36	26	40	46	38
GPT-4.1	32	24	30	26	38
GPT-4o	24	18	18	18	30
Gemini 2.5 Flash	10	46	20	12	30
Claude Sonnet 4	32	28	30	40	32
Judge LLMs					
Selene-1-70B	20	44	44	18	96
LMUnit-72B	22	32	24	12	94
CompassJudger-14B	2	28	34	12	**100**
CompassJudger-7B	2	20	30	8	94
RISE-Judge-7B	0	16	40	8	96
Skywork-Critic-70B	10	22	10	6	80
Skywork-Critic-8B	6	24	18	8	54
General-Purpose LLMs					
Qwen2.5-Coder-32B	4	32	44	14	94
DS-Coder-V2-Lite	12	22	12	22	72
Specialized					
Gödel-V2-32B	22	8	28	18	46
DS-Prover-V2-7B	10	16	24	20	48
Prover-V1.5-RL	10	2	6	12	16
Prover-V1.5-SFT	4	4	0	4	20
Gödel-SFT	0	0	0	4	0
Gödel-V2-8B	0	0	0	0	2
Qwen2.5-Math-7B	0	0	0	0	0

G Pointwise Accuracy by Error Strategy

Table 6 reports pointwise accuracy broken down by error injection strategy, complementing the pairwise results in Table 2.

Table 6: Pointwise accuracy (%) by error strategy. S1: verbose incorrect, S2: Minimal Variations, S3: NL Justification, S4: Forced Mistakes, S5: Python Injection.

Model	**S1**	**S2**	**S3**	**S4**	**S5**
Frontier LLMs					
Claude Opus 4.5	60	52	68	70	78
Claude Sonnet 4.5	56	42	72	56	64
GPT-5.2	46	40	34	52	52
GPT-4.1	42	32	38	44	42
GPT-4o	36	40	32	54	44
GPT-5.1	38	20	22	46	32
Gemini 2.5 Flash	4	20	4	12	16
Judge LLMs					
Con-J-Qwen2-7B	62	54	20	32	94
Selene-1-70B	10	46	64	18	96
CompassJudger-7B	14	40	38	30	90
LMUnit-72B	14	32	44	26	90
CompassJudger-14B	12	42	44	16	86
Skywork-Critic-70B	16	52	34	12	68
RISE-Judge-7B	2	16	16	6	20
General-Purpose LLMs					
Qwen2.5-72B-Inst.	8	34	66	10	80
Qwen2.5-Coder-32B	2	34	50	12	84
DS-Coder-V2-Lite	4	50	40	22	48
Qwen2.5-Math-7B	0	0	0	2	0
Specialized					
DS-Prover-V2-7B	16	6	8	12	2
Prover-V1.5-SFT	16	4	8	10	6
Prover-V1.5-RL	16	12	4	6	2
Gödel-SFT	6	0	0	0	0

Several patterns complement the pairwise analysis. Claude Opus 4.5 achieves the highest pointwise accuracy on Forced Mistakes (70%) and NL Justification (68%), confirming its balanced capability across error types. Con-J-Qwen2-7B stands out among judge models with 62% on verbose incorrect and 94% on Python Answer, yet drops to 20% on NL Justification, suggesting vulnerability to misleading natural language explanations. Specialized provers show uniformly low scores across all categories, reinforcing that the generation–evaluation gap is not an artifact of position bias in pairwise evaluation.

H Benchmark Stability

We analyze how model accuracy changes as sample size increases to validate that 250 preference pairs provide reliable evaluation. Figure 3 shows accuracy trajectories for selected models as pairs are incrementally added. Model rankings stabilize around 250 pairs, indicating that our benchmark size is sufficient for reliable aggregate comparison while remaining cost-efficient.

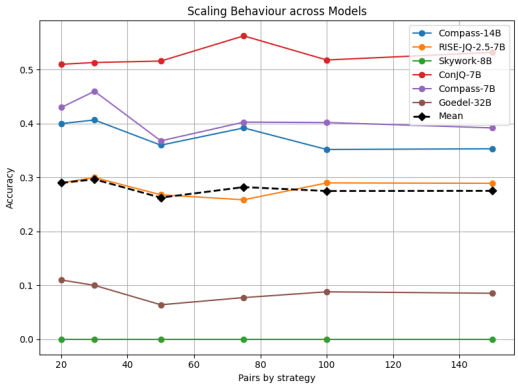

Figure 3: Model accuracy as a function of benchmark size. Accuracy stabilizes after 250 pairs.