Generative Reward Models for Formal Theorem Proving: Methods, Benchmarks, and Tree Search Integration

by

Zeynel Abidin Uluşan

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

December 2025

Generative Reward Models for Formal Theorem Proving: Methods, Benchmarks, and Tree Search Integration

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Zeynel Abidin Uluşan

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Asst. Prof. Gözde G. Sahin (Advisor)

_____

Prof. Alper Erdogan

_____

Prof. Erkut Erdem

Date: _____

To my wife...

ABSTRACT

Generative Reward Models for Formal Theorem Proving: Methods, Benchmarks, and Tree Search Integration
Zeynel Abidin Uluşan
Master of Science in Computer Science and Engineering
December 2025

Automated theorem proving aims to generate machine-verifiable proofs with minimal human intervention, but effective proof search remains challenging. Tree search algorithms require guidance to navigate exponentially large proof spaces. However, existing approaches suffer from fundamental limitations. Log-probability heuristics conflate generation likelihood with step quality, binary feedback from proof assistants provides no notion of progress, and discriminative reward models compress complex reasoning into opaque scalar values. In addition, implementation fragmentation limits reproducibility, and the absence of evaluation frameworks blocks systematic development of reward models for formal mathematics.

This thesis makes three contributions addressing these challenges. First, we develop TreeThink, a modular Python library providing unified implementations of best-first search, beam search, and Monte Carlo tree search for formal theorem proving. The architecture cleanly separates search strategy, environment interaction, and model inference, enabling reproducible experimentation and fair comparison across algorithms. All components support asynchronous execution and batched inference for computational efficiency.

Second, we introduce a framework for generative reward models that produce natural language critiques evaluating proof steps. Unlike discriminative models that output scalar scores without explanation, our approach generates structured textual feedback describing correctness, progress toward goals, and strategic value, with embedded numerical scores for integration into tree search. A key design element is conditioning reward generation on environmental feedback from proof assistant execution, grounding evaluations in concrete compiler responses rather than predictions alone. We study both zero-shot deployment using pretrained models and reinforcement learning on proof trajectories to align critique scores with proof outcomes.

Third, we introduce FormalRewardBench, the first benchmark for evaluating reward models in formal theorem proving. The benchmark consists of preference pairs where correct Lean 4 proofs are paired with incorrect variants generated through five error injection strategies targeting realistic failure modes, including forced mistakes, minimal variations, complex incorrect proofs, natural language justification, and Python code injection. Quality control ensures errors are semantic rather than trivial, and evaluation protocols support both pointwise and pairwise reward models with position bias mitigation.

Together, these contributions advance neural theorem proving by providing modular infrastructure for systematic experimentation, richer guidance signals through interpretable natural language feedback grounded in execution, and evaluation frameworks that enable principled development of reward models for formal mathematics. Our work addresses key limitations in automated theorem proving and establishes foundations for more capable and interpretable proof search systems.

ÖZETÇE

Formel Teorem İspatlamada Üretken Ödül Modelleri: Yöntemler, Kıyaslamalar ve Ağaç Arama Entegrasyonu
Zeynel Abidin Uluşan
Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans
Aralık 2025

Otomatik teorem ispatı, insan müdahalesini en aza indirerek makine tarafından doğrulanabilir matematiksel kanıtlar üretmeyi amaçlar; ancak etkili ispat araması hâlâ önemli bir zorluktur. Ağaç arama algoritmaları, üstel olarak büyüyen ispat uzaylarında hangi durumların araştırılmaya değer olduğunu belirlemek için yönlendirmeye ihtiyaç duyar. Mevcut yaklaşımlar temel sınırlamalara sahiptir: log-olasılık temelli sezgiler üretim olasılığını adım kalitesiyle karıştırır, ispat yardımcılarından gelen ikili geri bildirim ilerleme hakkında bilgi vermez ve ayrık ödül modelleri karmaşık muhakemeyi açıklamasız tek bir sayıya indirger. Ayrıca, altyapı parçalanmışlığı tekrarlanabilirliği zorlaştırmakta ve değerlendirme çerçevelerinin yokluğu, biçimsel matematikte ödül modellerinin sistematik gelişimini engellemektedir.

Bu tez, söz konusu sorunları ele alan üç temel katkı sunmaktadır. İlk olarak, biçimsel teorem ispatı için en-iyi-önce arama, ışın arama ve Monte Carlo ağaç aramasını birleştiren modüler bir Python kütüphanesi olan TreeThink geliştirilmiştir. Mimari, arama stratejisini, ortam etkileşimini ve model çıkarımını açık biçimde ayırarak yeniden üretilebilir deneyler ve adil algoritma karşılaştırmaları yapılmasına olanak tanır. Tüm bileşenler, hesaplama verimliliği için asenkron yürütme ve toplu çıkarımı destekler.

İkinci olarak, ispat adımlarını değerlendiren doğal dil açıklamaları üreten üretici ödül modelleri için bir çerçeve sunulmuştur. Sayısal değerler üreten ayrık modellerin aksine, bu yaklaşım doğruluk, hedefe ilerleme ve stratejik değer hakkında yapılandırılmış metinsel eleştiriler üretir ve ağaç arama algoritmalarında kullanılmak üzere sayısal skorlar içerir. Kritik tasarım unsuru, değerlendirmeyi yalnızca tahminlere değil, ispat yardımcısının yürütme çıktılarından gelen çevresel geri bildirime koşullamaktır. Bu çerçeve, hem önceden eğitilmiş modellerle sıfır-atış (zero-shot) kullanım hem de ispat yörüngeleri üzerinde pekiştirmeli öğrenme ile eğitimi kap-

samaktadır.

Üçüncü olarak, biçimsel teorem ispatında ödül modellerini değerlendirmek için geliştirilen ilk kıyaslama seti olan FormalRewardBench tanıtılmaktadır. Bu kıyaslama, doğru Lean 4 ispatlarını; zorlanmış hatalar, minimal değişiklikler, karmaşık ancak hatalı ispatlar, doğal dil gerekçelendirme ve Python kodu enjeksiyonu gibi gerçekçi hata türlerini hedefleyen beş farklı stratejiyle üretilmiş yanlış varyantlarla eşleştiren tercih çiftlerinden oluşur. Kalite kontrol süreçleri, hataların yüzeysel değil anlamsal olmasını garanti eder ve değerlendirme protokolleri, konum yanlılığını azaltan noktasal ve ikili karşılaştırmaları destekler.

Bu katkılar bir arada ele alındığında, otomatik teorem ispatında sistematik deneyler için birleşik bir altyapı, yürütme geri bildirimiyle temellendirilmiş yorumlanabilir yönlendirme sinyalleri ve biçimsel matematikte ödül modellerinin geliştirilmesini mümkün kılan değerlendirme çerçeveleri sunulmaktadır. Bu çalışma, otomatik teorem ispatındaki temel boşlukları ele almakta ve daha yetenekli ve yorumlanabilir ispat arama sistemleri için sağlam bir temel oluşturmaktadır.

ACKNOWLEDGMENTS

I thank my advisor Asst. Prof. Gözde Gül Şahin for her guidance and support. Her insights shaped how I approach research, not just this thesis. I also thank Asst. Prof. Barış Akgün for his support and mentorship during my independent study on LLMs and tree search.

I want to thank Burak Sina Akbudak and Salih Can Erer, who worked with me as research interns at GGLab. Working with them made this research better.

To my lovely wife Yağmur Seda Uluşan: thank you for believing in me, keeping me motivated when things got hard, making my life easier in countless ways, and helping me with the figures. This work wouldn't exist without you.

I also thank my fellow GGLab members—Abed, Ali, Berfin, Ege, Hulki, and Tamta—for making the lab feel like a community. The discussions and shared struggles made the hard parts easier.

I thank Codeway Studios for providing an incredible work environment and introducing me to amazing people who inspired me throughout this journey. I especially thank Atakan, Batuhan, Onur, Burakcan, Mert and Yunus—but not limited to them.

To my friends Alper Buğra Aydın and Furkan Gülbey Yıldırım: thank you for your friendship and mental support during the thesis writing process.

I'm especially grateful to my mother, and to my parents and siblings, for always being there for me and believing in me.

This research used computational resources from Koç University's KUACC and Valar clusters, and TÜBİTAK's MareNostrum 5 supercomputer. Without these systems, this work wouldn't have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

LLM	Large Language Model
GRM	Generative Reward Model
MCTS	Monte Carlo Tree Search
BFS	Breadth-First Search
UCT	Upper Confidence Bounds applied to Trees
RL	Reinforcement Learning
vLLM	High-throughput inference engine for large language models
REPL	Read–Eval–Print Loop
PPO	Proximal Policy Optimization
GRPO	Group Relative Policy Optimization
RM	Reward Model
PRM	Process Reward Model
ORM	Outcome Reward Model
KL	Kullback–Leibler Divergence
CoT	Chain-of-Thought
SFT	Supervised Fine-Tuning
ITP	Interactive Theorem Proving
ATP	Automated Theorem Proving
GPU	Graphics Processing Unit
API	Application Programming Interface

Chapter 1

INTRODUCTION

1.1 Motivation

Mathematics demands absolute precision. A single error—a misapplied theorem, a confused variable, an overlooked edge case—can invalidate an entire proof. Traditional mathematical practice relies on peer review, but even published proofs sometimes contain subtle errors that go undetected for years [Gauthier and Brown, 2024].

Formal mathematics addresses this through mechanized verification. Systems like Lean [de Moura and Ullrich, 2021], Coq [The Coq Development Team, 2024], and Isabelle [Nipkow et al., 2002] check every proof step mechanically, eliminating ambiguity and ensuring correctness. This verification extends beyond pure mathematics to verified compilers, cryptographic implementations, and control systems for critical infrastructure [Huang et al., 2024]. However, writing formal proofs requires expertise in both mathematics and proof assistant syntax, often taking months or years of expert effort. This limits formal verification's reach and prevents wider adoption.

Automated theorem proving aims to generate formal proofs with minimal human intervention. Recent advances in large language models have enabled systems that generate plausible proof tactics and recognize common patterns [Polu and Sutskever, 2020, Yang et al., 2023, Xin et al., 2024]. However, the space of possible proof paths grows exponentially, and only a tiny fraction lead to success. Tree search provides a framework for systematically exploring this space by maintaining multiple candidate states and expanding promising ones, but its effectiveness depends critically on guidance—how the algorithm decides which proof states are worth exploring.

Existing guidance signals for proof search are sparse, conflated, or opaque, mo-

tivating richer reward modeling approaches. Additionally, most systems implement search from scratch, preventing code reuse and fair comparison. Finally, no benchmarks exist for evaluating outcome-level reward models in formal theorem proving, blocking progress on preference-based learning methods.

1.2 Research Questions

This thesis addresses three interconnected research questions:

RQ1: Can unified infrastructure enable more systematic research on tree search for theorem proving? Implementation fragmentation and tight coupling between components limit reproducibility and experimentation. We investigate whether modular design with transparent handling of computational efficiency can facilitate algorithm development and fair comparison. (Addressed in Chapter 3)

RQ2: Can generative reward models provide richer guidance than existing approaches? Current signals are sparse, conflated, or opaque. We explore whether natural language critiques that leverage execution feedback and mathematical reasoning can offer more effective and interpretable guidance for proof search. (Addressed in Chapter 4)

RQ3: Can systematic benchmarks advance reward modeling for formal mathematics? Without evaluation frameworks, we cannot measure understanding of proof quality or validate preference-based learning methods. We investigate whether controlled error injection can create challenging yet fair benchmarks for reward model development. (Addressed in Chapter 5)

1.3 Contributions

This thesis makes three main contributions addressing these research questions.

1.3.1 TreeThink: Unified Infrastructure for Tree Search

We present TreeThink, a modular library that provides common implementations of major tree search algorithms for formal theorem proving. The design separates

search strategy, environment interaction, and model inference through clean interfaces. This separation aims to enable researchers to experiment with different algorithms and reward models without reimplementing infrastructure, while transparent handling of computational operations addresses efficiency concerns.

1.3.2 Generative Reward Models for Dense Guidance

We introduce a framework for process-level reward models that generate natural language critiques rather than scalar values. These critiques explain proof step quality by conditioning on concrete execution feedback from the proof assistant. This approach aims to leverage the mathematical reasoning capabilities that language models develop during pretraining, while providing interpretable feedback that may facilitate debugging and systematic improvement. We explore both zero-shot deployment and training strategies to align critique quality with proof outcomes.

1.3.3 FormalRewardBench: Evaluation Framework for Reward Models

We develop the first benchmark for evaluating outcome-level reward models in formal theorem proving. The benchmark uses preference pairs where correct proofs are contrasted with incorrect variants generated through systematic error injection strategies. These strategies target realistic failure modes that language models encounter during proof generation. The framework supports multiple evaluation protocols to accommodate different reward model architectures and includes mechanisms to control for common biases.

1.4 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 provides background on formal theorem proving, reward modeling, and tree search. It surveys related work and identifies research gaps motivating our contributions.

Chapter 3 presents TreeThink's architecture, algorithm implementations, and integration mechanisms.

Chapter 4 develops the generative reward model framework, including design principles, training strategies, and experimental evaluation.

Chapter 5 introduces FormalRewardBench, describing design methodology, error injection strategies, and evaluation protocols with experimental findings.

Chapter 6 summarizes contributions, discusses limitations, and outlines future directions.

1.5 Summary

Automated theorem proving holds promise for making formal mathematics more accessible, yet current systems face challenges in search efficiency, guidance quality, and systematic evaluation. This thesis addresses these challenges through unified infrastructure for experimentation, interpretable guidance signals through natural language critiques, and evaluation frameworks for reward model development. These contributions aim to advance neural theorem proving by improving both research methodology and the quality of signals that direct proof search.

Chapter 2

BACKGROUND AND RELATED WORK

This chapter provides background on key concepts and surveys related work. We begin with foundational material on formal theorem proving, reward modeling.. We then review prior work in neural theorem proving, reward models for search guidance, and evaluation frameworks. Finally, we identify research gaps that motivate our contributions.

2.1 Background

2.1.1 Interactive Theorem Proving with Lean 4

Lean is a proof assistant based on dependent type theory [de Moura and Ullrich, 2021]. Every proof step must be verified by the system's kernel, which checks that terms have correct types according to the logical foundations. This mechanical verification eliminates ambiguity and ensures absolute correctness.

Proof Assistants

Proof assistants provide formal languages for writing mathematical proofs. Systems like Lean [de Moura and Ullrich, 2021], Isabelle [Nipkow et al., 2002], Coq [The Coq Development Team, 2024], and Agda [Norell, 2008] differ in their logical foundations and design philosophies, but all share the core principle of mechanized verification.

In Lean 4, theorem statements are expressed with precise type annotations. For example:

Listing 2.1: Commutativity of addition

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

This declares that addition of natural numbers is commutative. The type signature $(n\, m : \mathbb{N})$ introduces two natural number variables, and $n+m = m+n$ specifies the proposition to prove. The `by` keyword initiates tactic mode, where proofs are constructed interactively. The `sorry` placeholder indicates an incomplete proof.

Tactic-Based Proving

Lean supports two proof modes. Term mode constructs proofs directly using lambda calculus expressions. Tactic mode builds proofs interactively by applying tactics that transform proof states. Common tactics include `rw` (rewrite), `simp` (simplification), `apply` (lemma application), and `exact` (direct proof term).

Each tactic application transforms the current proof state—the goals remaining to be proven and the hypotheses available. When no goals remain, the proof is complete. If a tactic cannot apply or produces a type error, Lean reports failure with an error message.

Type Checking and Verification

Lean's type checker validates proofs through the Curry-Howard correspondence, which establishes a connection between propositions and types, and between proofs and programs [Sørensen and Urzyczyn, 2006]. A proof of proposition P is a term of type P. The type checker verifies that proof terms have the types claimed by theorem statements.

This verification is rigorous. Unlike informal mathematical proofs where steps might be sketched or "obvious" details omitted, formal proofs must justify every inference. This rigor eliminates errors but requires significant effort to formalize mathematics.

2.1.2 Reward Modeling and Reinforcement Learning

Reinforcement Learning from Human Feedback

RLHF has become central to aligning language models with human preferences [Ouyang et al., 2022, Bai et al., 2022]. The standard pipeline has three stages: supervised fine-

tuning on demonstration data, training a reward model on preference comparisons, and optimizing the policy with reinforcement learning algorithms like PPO [Schulman et al., 2017].

The reward model learns to predict which responses humans prefer. Given two responses to the same prompt, the model outputs scores indicating relative quality. These scores guide policy optimization through RL.

Bradley-Terry Model

The Bradley-Terry model [BRADLEY and TERRY, 1952] provides the foundation for preference-based reward modeling. It assumes each response has an underlying quality score, and the probability of preferring response y_1 over y_2 depends on the difference in their scores:

$$P(y_1 \succ y_2 \mid x) = \frac{\exp(r(x, y_1))}{\exp(r(x, y_1)) + \exp(r(x, y_2))} \tag{2.1}$$

where $r(x, y)$ is the reward function. This model converts pairwise preferences into a global ranking. Training maximizes the log-likelihood of observed preferences:

$$\mathcal{L}(\theta) = \mathbb{E}_{(x, y_{\text{chosen}}, y_{\text{rejected}})} \left[\log \sigma(r_\theta(x, y_{\text{chosen}}) - r_\theta(x, y_{\text{rejected}}))\right] \tag{2.2}$$

where σ is the sigmoid function.

Outcome vs Process Supervision

Reward mechanisms for reasoning tasks fall into two categories [Uesato et al., 2022, Lightman et al., 2024]. Outcome-based rewards focus only on final results—whether the solution is correct. Process-based rewards provide feedback at intermediate steps, evaluating each reasoning step independently.

2.2 Related Work

2.2.1 Formal Theorem Proving Systems

Formal theorem proving aims to generate machine-verifiable proofs for mathematical theorems. In interactive theorem proving, users write proofs in formal languages provided by proof assistants. The system mechanically verifies each step according to rigorous type-checking rules.

Recent advances in large language models have positioned neural systems as capable tools for automating formal proof construction [Polu and Sutskever, 2020, Yang et al., 2023, First et al., 2023]. Language models can learn to generate tactics, suggest lemmas, and propose proof strategies.

Benchmarks

Several benchmarks evaluate neural theorem provers:

MiniF2F [Zheng et al., 2021] contains 488 olympiad-level problems formalized across multiple proof assistants. Problems cover algebra, number theory, combinatorics, and geometry. The benchmark splits into validation and test sets of 244 problems each. Current systems achieve over 65% pass@1 on the test set [Wu et al., 2025, Li et al., 2025].

ProofNet [Azerbayev et al., 2023] provides 371 undergraduate mathematics problems in Lean, covering real analysis, linear algebra, and abstract algebra. These problems typically require longer proofs than miniF2F.

PutnamBench [Tsoukalas et al., 2024] contains 1099 problems from the Putnam competition. These are significantly harder, with current approaches solving only a small fraction.

These benchmarks use pass@k metrics—the percentage of problems where the system finds at least one correct proof in k attempts.

2.2.2 Neural Theorem Proving with Tree Search

Best-First Search Approaches

GPT-f [Polu and Sutskever, 2020] pioneered neural theorem proving with best-first tree search. The system trains transformer models on proof states and tactics, then uses log-probability-based heuristics to guide search.

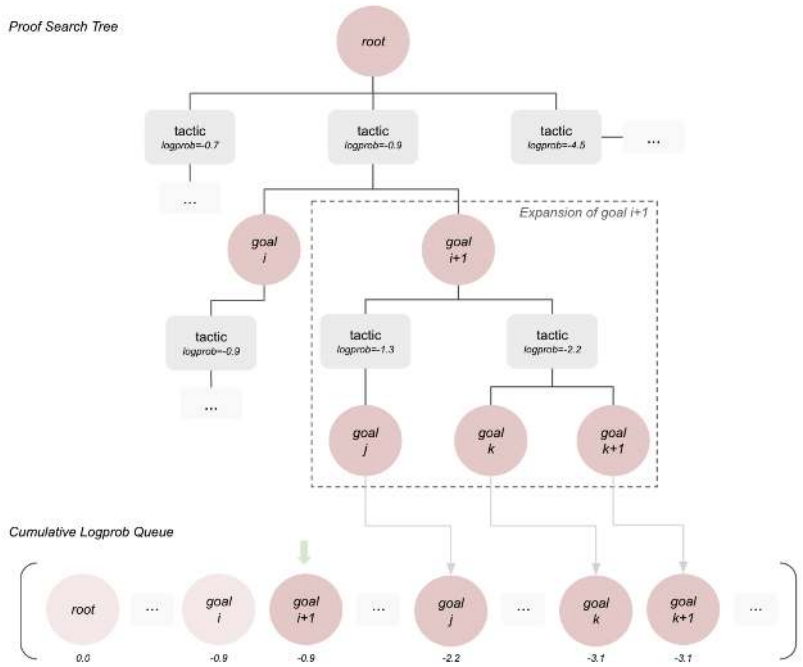

Figure 2.1: Proof search consists in maintaining a proof tree where multiple tactics are explored for each goal, starting from the root goal. Goals are expanded by cumulative (tactic) logprob priority. Figure is directly taken from [Polu and Sutskever, 2020]

BFS-Prover [Lample et al., 2022] addresses this through length normalization and strategic data filtering with DPO guided by compiler feedback. This refined best-first search matches more complex approaches while maintaining efficiency.

InternLM2.5-StepProver [Wu et al., 2025] demonstrates large-scale expert iteration on 80,000+ problems. They show that vanilla best-first search achieves average proof length 1.66, while critic-guided search reaches 4.44. They also identify a log-

linear relationship between compute and problems solved. The system achieves 65.9% on miniF2F-test.

HunyuanProver [Li et al., 2025] combines synthetic data generation with guided tree search. Built on Hunyuan 7B, it achieves 68.4% on miniF2F-test and proves 4 IMO statements.

Monte Carlo Tree Search

TacticZero [Wu et al., 2021] formulated theorem proving as a Markov Decision Process and learned proof search strategies with deep RL. The work showed learned MCTS policies can beat fixed heuristics.

Early MCTS methods allocated equal computational effort to all states. DT-Solver [Wang et al., 2023] introduced dynamic budget allocation based on state confidence, guided by proof-level value functions.

DeepSeek-Prover-V1.5 [Xin et al., 2024] advanced MCTS through RMaxTS, which handles sparse rewards by giving intrinsic curiosity-based rewards to under-explored states. This encourages diverse exploration.

2.2.3 Reward Models for Search Guidance

Discriminative Reward Models

InternLM2.5-StepProver trains a critic model using preference-based learning similar to RLHF [Wu et al., 2025]. They create path pairs (states closer to completion preferred) and sibling pairs (successful paths preferred over failed ones).

The STILL-1 framework [Jiang et al., 2024] systematically explores design choices: discriminative vs generative, outcome vs process supervision, ranking vs scoring. They combine reward scores with self-consistency from rollout samples.

Verifiable Rewards

Verifiable reward models use external systems for ground truth feedback [Snell et al., 2024]. In theorem proving, proof assistants provide perfect verification—any tactic

they accept is logically sound. This reliability prevents reward hacking and overoptimization issues that plague learned models.

2.2.4 Generative Reward Models

Generative reward models produce natural language explanations instead of scalars, offering improved interpretability and reasoning capabilities.

Foundational Work

LLM-as-Judge approach [Zheng et al., 2023] prompts strong models to evaluate responses without training. However, these exhibit systematic biases: position bias (favoring certain orderings), verbosity bias (preferring longer text), and self-enhancement bias (rating own outputs higher) [Panickssery et al., 2024, Ye et al., 2024].

In contrast, GenRM [Mahan et al., 2024] trains models to generate reasoning traces justifying preference judgments. This combines benefits of RLHF and RLAIF [Bai et al., 2022]. The key insight is that language models have reasoning capabilities underutilized when restricted to scalar outputs.

GenPRM [Zhao et al., 2025] extends to process rewards with explicit chain-of-thought reasoning before scoring each step. This enables test-time compute scaling—a 1.5B GenPRM can outperform GPT-4 on ProcessBench through longer generation.

Recent Advances

DeepSeek-GRM [Liu et al., 2025] demonstrates inference-time scaling. A 27B model with Self-Principled Critique Tuning and multi-sample voting matches 340B+ models through strategic compute allocation. Multiple critiques are generated and aggregated through ensemble methods.

Code Generation Context

ORPS [Yu et al., 2025] combines process and outcome supervision for code generation. Traditional process supervision needs expensive data and suffers misalignment. ORPS uses executable verification through tree search with runtime feedback and self-critique mechanisms.

This integration of execution results with learned evaluation is directly relevant to theorem proving, where proof assistants provide immediate deterministic verification.

Application to Formal Mathematics

Generative reward models remain underexplored in formal theorem proving, where evaluation differs fundamentally from informal reasoning. Formal proofs are governed by strict syntactic and type-theoretic constraints, such that small errors can invalidate entire proof steps. Beyond syntax, every step must be logically sound with respect to axioms and inference rules. Assessing proof quality therefore requires domain-specific mathematical understanding, rather than simple pattern matching.

2.2.5 Evaluation Frameworks

General Reward Model Benchmarks

RewardBench [Lambert et al., 2025] is the main benchmark for reward models, covering chat, safety, and reasoning tasks in natural language. It measures accuracy at selecting preferred responses from pairs. However, it doesn't evaluate formal mathematical reasoning.

Process Reward Benchmarks

ProcessBench [Zheng et al., 2025] and PRMBench [Song et al., 2025] evaluate process reward models on informal math problems. Models must identify correct and incorrect reasoning steps in natural language solutions.

Both operate in informal domains where flexibility in expression is acceptable and multiple solution paths are valid. This differs fundamentally from formal theorem

proving.

The Formal-Informal Gap

A proof that seems correct in natural language may have type errors, wrong lemma applications, or misused hypotheses that only formal verification catches. Conversely, a formally correct proof might seem unclear informally.

Example: Writing "$a^p \equiv a \pmod{p}$ by Fermat's Little Theorem" is acceptable informally. In Lean, you must import the theorem, instantiate types correctly, prove p is prime, and handle type casting between formulations.

2.3 Research Gaps and Motivation

Our analysis reveals three critical research gaps that motivate our contributions.

2.3.1 Gap 1: Lack of Unified Tree Search Infrastructure

Most prior work implements search algorithms from scratch for specific systems [Polu and Sutskever, 2020, Wu et al., 2025, Li et al., 2025], developing custom search infrastructure, environment interfaces, and evaluation protocols. This fragmentation makes comparison difficult—when different systems use different implementations, it's unclear whether performance differences stem from the algorithm, the policy model, the reward model, or implementation details. Researchers must reimplement search strategies from incomplete descriptions, and testing a new reward model across multiple algorithms requires implementing each separately.

Sequential execution of model calls severely underutilizes modern hardware. When each proof state generates tactics one at a time, GPUs sit mostly idle between inference requests [Kwon et al., 2023]. Search strategy, environment interaction, and model inference are typically tightly coupled—adding a new search algorithm requires modifying environment code, changing the proof assistant requires updating search implementations, and integrating different reward models requires significant refactoring.

We develop TreeThink to address these issues through unified implementations of best-first search, beam search, and MCTS with shared infrastructure, asynchronous execution and batched inference handled transparently, and modular design separating search strategy from environment interaction and model inference.

2.3.2 Gap 2: Limited Guidance Signals for Proof Search

Current guidance mechanisms face fundamental limitations. Proof assistants provide binary signals—a tactic either succeeds or fails. This sparsity makes it difficult to distinguish proofs that nearly succeeded from those that were fundamentally wrong, or to identify which specific steps caused failures. Simple heuristics like policy model probabilities conflate generation likelihood with step quality—high-probability tactics might represent common patterns that don't advance the proof, while lower-probability tactics might make genuine progress.

Discriminative reward models provide scalar values without explanation. When a model assigns low value to a proof state, the reasoning remains opaque—whether from type errors, strategic concerns, or other factors is unclear. This opacity makes debugging difficult and prevents systematic improvement. Moreover, discriminative models learn direct numerical prediction without the intermediate reasoning that natural language articulation provides, potentially missing opportunities to leverage the mathematical reasoning capabilities that language models develop during pretraining.

We develop generative reward models that produce natural language critiques explaining correctness, progress, and strategic value. By conditioning on environmental feedback from proof assistant execution and generating explicit reasoning, these models can leverage broader knowledge while providing interpretable assessments that facilitate debugging and improvement.

2.3.3 Gap 3: No Evaluation Framework for Outcome Reward Models

Existing theorem proving benchmarks measure success using pass@k metrics, but cannot evaluate understanding of proof quality. When a prover fails, distinguish-

ing near-successes from fundamental errors is impossible—both receive the same binary outcome. Multiple correct proofs may vary in elegance, length, or clarity, yet success-based metrics treat them identically. Modern reward modeling techniques like DPO [Rafailov et al., 2023] and RLHF [Ouyang et al., 2022] require preference data for training and evaluation, but no benchmarks provide such preferences for formal proofs.

Without systematic evaluation frameworks, collecting preference data remains ad-hoc and inconsistent across research efforts. Measuring whether reward models accurately capture proof quality becomes difficult without ground truth preferences. Recent advances like GenRM [Mahan et al., 2024], GenPRM [Zhao et al., 2025], and DeepSeek-GRM [Liu et al., 2025] show promise in informal domains but lack formal evaluation. This absence prevents systematic application of preference learning to theorem proving.

We develop FormalRewardBench to enable systematic evaluation. The benchmark provides preference pairs where correct proofs are paired with incorrect variants generated through five error injection strategies capturing realistic failure modes. Quality control ensures errors are semantic rather than trivial, and evaluation protocols support both pointwise and pairwise reward models with position bias mitigation.

2.4 Summary

This chapter provided background on formal theorem proving, reward modeling, and tree search algorithms, then surveyed related work in neural theorem proving systems, reward models for search guidance, and evaluation frameworks.

Three critical research gaps emerge:

1. **Lack of unified infrastructure.** Implementation fragmentation, inefficient execution, and tight coupling limit research progress in tree search for theorem proving.

2. **Limited guidance signals.** Sparse verification feedback, heuristics, and lim-

ited interpretability and reasoning constrain proof search effectiveness.

3. **No evaluation framework.** The absence of systematic benchmarks for reward models blocks progress on preference-based learning and modern reward modeling techniques.

Our three contributions address these gaps: TreeThink provides unified efficient infrastructure for tree search, process level generative reward models offer richer interpretable guidance, and FormalRewardBench enables systematic evaluation of outcome level reward models in formal mathematics.

The following chapters present each contribution in detail, demonstrating how they advance the state of neural theorem proving.

Chapter 3

TREETHINK: A MODULAR LIBRARY FOR TREE SEARCH IN THEOREM PROVING

3.1 Overview

Effective theorem proving requires sophisticated search strategies that can navigate the exponentially large space of possible proof steps. While language models have shown impressive capabilities in generating individual proof tactics [Polu and Sutskever, 2020, Yang et al., 2023], their integration into systematic search procedures remains challenging. Search algorithms must balance exploration and exploitation, manage computational resources efficiently, and interact with proof environments that provide formal verification.

We develop TreeThink, a modular Python library designed specifically for tree search in formal theorem proving. The library provides clean abstractions that separate search strategy from environment interaction and model inference, enabling researchers to easily experiment with different algorithms and configurations. TreeThink handles the complexities of asynchronous execution, batched inference, and proof environment management, allowing users to focus on search algorithm design and reward model development.

The key design principles of TreeThink are:

- **Modularity**: Search algorithms, reward models, and proof environments are separate components with well-defined interfaces.
- **Efficiency**: All algorithms support asynchronous execution and batched inference to maximize GPU utilization.
- **Extensibility**: New search strategies and reward models can be easily inte-

grated through base class inheritance.

- **Reproducibility**: Deterministic execution modes and comprehensive logging facilitate experimental reproducibility.

3.2 Architecture

3.2.1 Core Components

TreeThink's architecture consists of four primary components that work together to enable efficient proof search:

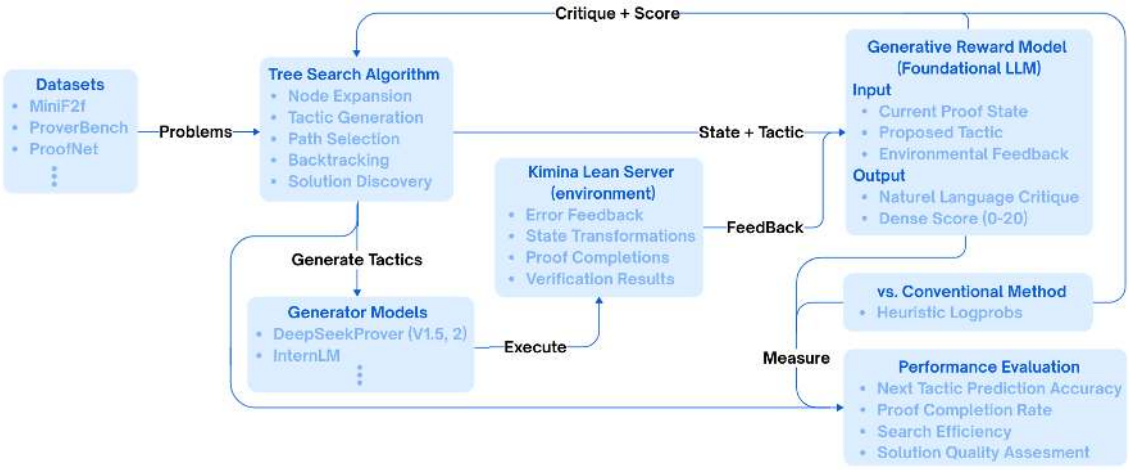

Figure 3.1: TreeThink diagram showing the integration of tree search algorithms, generator models, formal verification environment, and generative reward models

Search Algorithm Layer The search algorithm layer implements various tree search strategies through a common interface. Each algorithm inherits from a base `Method` class to define the core search loop and base `Node` class for node management operations. This abstraction allows different algorithms to share common infrastructure while implementing their specific selection and expansion strategies.

Environment Interface The environment interface manages interaction with the Lean 4 proof assistant through the Kimina server [Santos et al., 2025]. This component handles:

- Proof state management and persistence
- Tactic execution and verification
- Error handling and recovery

The environment interface abstracts away the details of proof assistant communication, presenting a simple API where users can execute tactics on proof states and receive structured responses indicating success, failure, or completion.

Language Model Integration Language model inference is handled through vLLM's AsyncLLMEngine [Kwon et al., 2023], which provides efficient batched inference with continuous batching and paged attention. TreeThink supports two types of language model calls:

- **Next Proof Step Generation**: Given a proof state, the policy model generates next proof step that might advance the proof.
- **State Evaluation**: The reward model evaluates proof steps, generating natural language critiques and value estimates.

Both operations are performed asynchronously with automatic batching across multiple search nodes, significantly improving throughput compared to sequential execution.

Reward Model Integration The reward model component converts natural language critiques into scalar rewards that guide search decisions. This involves:

- Prompt construction with proof context and trajectory
- Critique generation through language model inference
- Critique parsing and reward extraction

The modular design allows users to implement custom reward extraction functions while reusing the infrastructure for critique generation.

3.2.2 Node Representation

Search algorithms operate on a tree of proof states, where each node represents a particular configuration of the proof. TreeThink represents nodes with the following attributes shown in Table 3.1.

Table 3.1: Node structure in TreeThink's search tree

Field	**Description**
State	The current proof state, including goals and context
Parent	Reference to the parent node in the search tree
Action	The tactic that was applied to reach this state
Children	List of child nodes generated by expanding this state
Value	Estimated value from the reward model
Visits	Number of times this node has been visited (for MCTS)
Depth	Distance from the root node

This representation is efficient and supports all implemented search algorithms while maintaining enough information for detailed logging and analysis.

3.3 Search Algorithms

TreeThink implements three tree search algorithms that differ in their exploration strategies and computational requirements. All algorithms share common infrastructure for node expansion, environment interaction, and reward model evaluation, but employ different policies for selecting which nodes to expand.

3.3.1 Best-First Search

Best-First Search (BFS-Prover) maintains a priority queue of open proof states and always expands the most promising node according to a heuristic function. In TreeThink, the heuristic combines the policy model's prior probability with the reward model's value estimate:

$$h(s) = \alpha \cdot \log \pi(s) + (1 - \alpha) \cdot V(s) \tag{3.1}$$

where $\pi(s)$ is the policy model's probability for the action leading to state s, $V(s)$ is the reward model's value estimate, and α controls the relative weighting.

Algorithm 1 Best-First Search in TreeThink

Initialize frontier $\mathcal{F} \leftarrow \{s_0\}$ with root state
while $\mathcal{F} \neq \emptyset$ and budget not exhausted **do**
  $s^* \leftarrow \arg\max_{s \in \mathcal{F}} h(s)$
  $\mathcal{F} \leftarrow \mathcal{F} \setminus \{s^*\}$
  if s^* is proven **then**
    return proof from s^*
  end if
  $\mathcal{A} \leftarrow$ generate_tactics(s^*)
  for $a \in \mathcal{A}$ **do**
    $s' \leftarrow$ execute(s^*, a)
    if s' is valid **then**
      $V(s') \leftarrow$ evaluate(s')
      $\mathcal{F} \leftarrow \mathcal{F} \cup \{s'\}$
    end if
  end for
end while
return failure

Best-First Search prioritizes depth and focuses computational effort on the most promising branches of the search tree. This makes it particularly effective when the reward model provides reliable value estimates, as it can quickly pursue high-value paths to completion. However, it may suffer from overcommitment if the reward model is miscalibrated, potentially missing correct proofs on initially lower-ranked branches.

The TreeThink implementation of BFS includes several optimizations:

- **Lazy Evaluation**: Child nodes are only evaluated when they become competitive for selection, reducing wasted computation.

- **Duplicate Detection**: A hash table tracks visited states to avoid re-expanding equivalent proof configurations.

3.3.2 Beam Search

Beam Search expands the proof tree level-by-level, maintaining a fixed-size beam of the most promising states at each depth. This approach balances breadth and efficiency, exploring multiple proof paths in parallel while pruning unpromising branches early.

At each level d, TreeThink expands all states in the current beam $\mathcal{B}_d$ in parallel. For each state $s \in \mathcal{B}_d$, we:

1. Generate k candidate tactics using the policy model

2. Execute each tactic to create potential child states

3. Filter out invalid states (syntax errors, failed execution)

4. Evaluate valid children with the reward model

The next beam $\mathcal{B}_{d+1}$ consists of the top-k states across all expansions, ranked by reward model scores:

$$\mathcal{B}_{d+1} = \text{top-}k \left(\bigcup_{s \in \mathcal{B}_d} \text{children}(s) \right) \tag{3.2}$$

Beam Search offers several advantages for theorem proving:

- **Parallelism**: All states in a beam are expanded simultaneously, enabling efficient batched inference.

- **Diversity**: By maintaining multiple hypotheses at each level, beam search can recover from early mistakes.

Algorithm 2 Beam Search in TreeThink

Initialize beam $\mathcal{B}_0 \leftarrow \{s_0\}$ with root state
$d \leftarrow 0$
while $\mathcal{B}_d \neq \emptyset$ and $d < d_{\max}$ **do**
  $\mathcal{C} \leftarrow \emptyset$ // candidates for next beam
  for $s \in \mathcal{B}_d$ in parallel **do**
    if s is proven **then**
      return proof from s
    end if
    $\mathcal{A}_s \leftarrow$ generate_tactics(s)
    for $a \in \mathcal{A}_s$ **do**
      $s' \leftarrow$ execute(s, a)
      if s' is valid **then**
        $V(s') \leftarrow$ evaluate(s')
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{s'\}$
      end if
    end for
  end for
  $\mathcal{B}_{d+1} \leftarrow$ top-k states from $\mathcal{C}$ by V
  $d \leftarrow d + 1$
end while
return failure

- **Predictable Cost**: The computational budget is determined primarily by beam width and maximum depth, making resource planning straightforward.

However, beam search can struggle with proofs requiring varying numbers of steps along different branches, as it commits to a fixed depth schedule. TreeThink addresses this limitation by supporting variable beam widths and adaptive depth limits based on proof progress.

3.3.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) builds a search tree incrementally through repeated simulations, balancing exploration of uncertain regions with exploitation of promising paths. TreeThink implements an MCTS variant inspired by AlphaZero [Silver et al., 2017] but adapted for theorem proving with generative reward models.

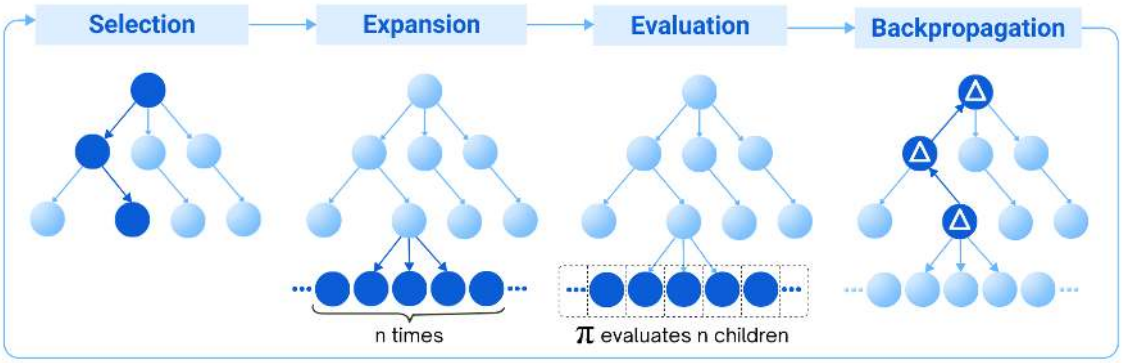

Figure 3.2: Steps of Monte Carlo Tree Search

Each MCTS iteration consists of four phases:

Selection Starting from the root, we traverse the tree by repeatedly selecting child nodes according to the UCT (Upper Confidence Bounds for Trees) formula:

$$\mathrm{UCT}(s,a) = Q(s,a) + c \cdot \sqrt{\frac{\ln N(s)}{N(s,a)}} \tag{3.3}$$

where:

- $Q(s, a)$ is the mean value estimate for taking action a from state s
- $N(s)$ is the visit count for state s
- $N(s, a)$ is the visit count for action a from state s
- c is an exploration constant that trades off exploration and exploitation

This selection continues until we reach a leaf node (a node that has not been fully expanded).

Expansion At the selected leaf node s, we generate k candidate tactics using the policy model. Unlike traditional MCTS which generates actions on-demand, we perform batch expansion: we generate tactics for multiple leaf nodes simultaneously and execute them in parallel through the proof environment.

Each candidate tactic a is executed to create a child state s'. Only valid states (those that do not result in execution errors) are added as children to the search tree.

Evaluation Our generative reward model evaluates each individual child state-tactic pair resulting from node expansion. For each child node, the model receives the current proof state, the proposed tactic, and environmental feedback, then produces a natural language critique and value estimate $V(s', a)$ that captures the promise of reaching state s' via action a.

Backpropagation The value estimate $V(s)$ is propagated back through the parent to the root, updating statistics for all ancestors:

$$N(s, a) \leftarrow N(s, a) + 1 \tag{3.4}$$

$$Q(s, a) \leftarrow \frac{N(s, a) \cdot Q(s, a) + V(s)}{N(s, a) + 1} \tag{3.5}$$

These updates incrementally refine our estimates of action values, with $Q(s, a)$ representing the average reward obtained from taking action a in state s across all simulations.

Algorithm 3 Monte Carlo Tree Search in TreeThink

Initialize root node s_0
for $i = 1$ to $n_{\text{simulations}}$ **do**
  $s \leftarrow s_0$
  path $\leftarrow [s_0]$
  while s is not a leaf node and not proven **do**
    $a^* \leftarrow \arg\max_a \text{UCT}(s, a)$
    $s \leftarrow$ child of s via action a^*
    path $\leftarrow$ path $+[s]$
  end while
  if s is proven **then**
    return proof from s
  end if
  $\mathcal{A} \leftarrow$ generate_tactics(s)
  for $a \in \mathcal{A}$ **do**
    $s' \leftarrow$ execute(s, a)
    if s' is valid **then**
      Add s' as child of s via action a
      $V(s', a) \leftarrow$ evaluate(s', a) {Evaluate each child}
    end if
  end for
  {Backpropagate from parent node}
  for $s_j \in$ path **do**
    Update $Q(s_j, a_j)$ and $N(s_j, a_j)$ using child values
  end for
end for
return failure

The main limitation of MCTS is computational cost: it requires many iterations to build accurate value estimates, and each iteration involves tree traversal overhead. TreeThink mitigates this through batched expansion and evaluation, processing multiple MCTS simulations in parallel.

3.4 Asynchronous Execution and Batched Inference

A critical design requirement for TreeThink is computational efficiency. Language model inference, particularly for generative reward models, is expensive—a single evaluation can take hundreds of milliseconds on modern GPUs. Sequential execution, where each operation waits for the previous to complete, would result in severe underutilization of computational resources.

TreeThink addresses this challenge through two complementary mechanisms:

3.4.1 Asynchronous Execution

All search algorithms in TreeThink use asynchronous execution where multiple operations proceed concurrently. This is implemented using Python's `asyncio` framework, which enables cooperative multitasking without the overhead of thread or process creation.

Consider beam search as an example. At each level, we need to:

1. Generate tactics for all states in the beam
2. Execute each tactic in the proof environment
3. Evaluate resulting states with the reward model

In synchronous execution, these operations would proceed sequentially: generate tactics for state 1, execute them, evaluate results, then move to state 2, and so on. With asynchronous execution, we can interleave these operations: while waiting for tactic generation for state 1, we can start executing previously generated tactics for other states, or begin reward model evaluation for completed executions.

This approach significantly reduces wall-clock time by overlapping I/O operations and computation. Network communication with the Lean server, disk I/O for model weights, and GPU computation all become opportunities for parallel progress.

3.4.2 Batched Inference

While asynchronous execution handles concurrency, batched inference addresses GPU efficiency. Modern language models achieve much higher throughput when processing multiple inputs simultaneously due to parallel matrix operations.

TreeThink uses vLLM's AsyncLLMEngine [Kwon et al., 2023], which implements continuous batching: as soon as multiple inference requests are available, they are automatically grouped and processed together on the GPU. This happens transparently—users write code that appears to make independent inference calls, but under the hood, vLLM batches them dynamically.

For example, during beam search, all states in the current beam require tactic generation. Rather than generating tactics for each state sequentially, TreeThink submits all generation requests concurrently. vLLM batches these requests and processes them in a single forward pass through the model, achieving much higher throughput than sequential generation.

Similarly, when multiple nodes need evaluation by the reward model, their critiques are generated in batch. This is particularly important for MCTS, where multiple simulations can be running in parallel—their evaluation requests are automatically batched together.

3.5 Integration with Lean 4

TreeThink integrates with the Lean 4 proof assistant [de Moura and Ullrich, 2021] through the Kimina server [Santos et al., 2025], which provides a JSON-RPC interface for programmatic interaction with Lean. This integration handles several important technical challenges:

3.5.1 Proof State Management

Lean proof states contain goals (propositions to be proven), hypotheses (available assumptions), and context (definitions and previous results). TreeThink maintains lightweight serializations of these states in tree nodes, allowing the search algorithm to track proof progress without keeping full Lean states in memory.

When a node needs to be expanded, TreeThink reconstructs the full Lean state by replaying the sequence of tactics from the root to that node. This approach trades some computation (replaying tactics) for memory efficiency and enables search trees with thousands of nodes.

3.5.2 Tactic Execution

Generated tactics are sent to the Kimina server for execution in the Lean environment. The server returns one of three outcomes:

- **Success**: The tactic was valid and produced one or more new goal states
- **Completion**: The tactic successfully completed the proof (no remaining goals)
- **Failure**: The tactic had a syntax error or could not be applied to the current state

TreeThink handles each outcome appropriately: successful tactics create child nodes in the search tree, completed proofs trigger proof extraction and search termination, and failures are logged but do not create children.

3.5.3 Error Handling and Recovery

Communication with external systems can fail in various ways: network timeouts, server crashes, or resource exhaustion. TreeThink implements robust error handling with automatic retry logic and graceful degradation. If a tactic execution fails due to a transient error, the system retries with exponential backoff. If the server becomes

unresponsive, ongoing search operations are suspended and can be resumed when the connection is restored.

This reliability is essential for long-running proof searches that may interact with Lean thousands of times over hours of computation.

3.6 Logging and Analysis

TreeThink includes comprehensive logging facilities that record all aspects of the search process:

- Node expansions with timestamps and resource usage
- Generated tactics and execution results
- Reward model evaluations and critiques
- Search tree structure and statistics

This detailed logging enables post-hoc analysis of search behavior, helping researchers understand why certain proofs succeeded or failed, identify patterns in reward model predictions, and debug search algorithm implementations.

The library also provides visualization tools for search trees, showing the evolution of value estimates, visit counts, and selection probabilities over time. These visualizations have proven valuable for understanding algorithm behavior and communicating results.

3.7 Summary

TreeThink provides a complete, efficient, and extensible framework for tree search in formal theorem proving. The library's key contributions are:

- Unified implementations of Best-First Search, Beam Search, and Monte Carlo Tree Search
- Asynchronous execution and batched inference for computational efficiency

- Clean abstractions that separate search strategy from environment and model integration
- Robust integration with Lean 4 through the Kimina server
- Comprehensive logging and visualization for analysis and debugging

By handling the engineering challenges of efficient proof search, TreeThink enables researchers to focus on developing better search strategies and reward models. The modular design ensures that insights from one component—such as improved reward model architectures—can immediately benefit all search algorithms without modification.

Chapter 4

GENERATIVE REWARD MODELS FOR GUIDING SEARCH IN FORMAL THEOREM PROVING

4.1 Introduction

Tree search algorithms require guidance to decide which proof states are worth exploring. In formal theorem proving, this guidance determines not only efficiency but feasibility, as the space of possible proof paths grows exponentially. Most existing systems rely on one of three signals: log-probability heuristics from the policy model [Polu and Sutskever, 2020, Xin et al., 2025], binary feedback from the proof assistant [Xin et al., 2024], or discriminative reward models that output scalar values. [Wu et al., 2025] Each of these approaches has fundamental limitations.

Log-probability heuristics treat the policy model's confidence as a proxy for step quality. However, likelihood reflects how typical a tactic is under the training distribution, not whether it meaningfully advances the proof. High-probability tactics often correspond to common but unproductive patterns, while low-probability tactics may represent creative steps that make genuine progress. This bias toward short, familiar sequences systematically penalizes longer or less conventional reasoning.

Binary feedback from proof assistants is reliable but extremely sparse. The system reports only whether a tactic succeeds or fails, without indicating partial progress or proximity to a valid proof. A proof attempt that fails due to a single incorrect lemma receives the same signal as one that is fundamentally unsound. This makes credit assignment difficult and provides little guidance for search.

Discriminative reward models aim to address sparsity by producing dense scalar signals. While effective in some settings, these models compress complex reasoning into a single number. When a proof state receives a low score, the cause is opaque: the model does not indicate whether the issue arises from a type error, an invalid

inference, or a poor strategic choice. This lack of interpretability hinders debugging and limits systematic improvement. Moreover, discriminative models rely heavily on large preference datasets or expensive expert iteration, learning value estimation directly from labeled outcomes without explicit reasoning.

We propose generative reward models (GRMs) that produce natural language critiques instead of scalar rewards alone. For each proof step, a GRM generates structured textual feedback that explains correctness, progress toward the goal, and strategic value. These critiques include an explicit numerical score usable by tree search algorithms for ranking and selection, while the accompanying text preserves information that scalar outputs discard.

Crucially, this approach leverages the foundational reasoning capabilities and mathematical priors acquired by large language models during pretraining. Rather than learning value estimation solely from preference data, GRMs can reason explicitly about proof structure, logical consistency, and mathematical strategy. This supports zero-shot evaluation of proof states in practice, depending on base model quality. In contrast to scalar reward models, which must infer value implicitly, generative models expose their reasoning process directly through language.

A key design choice is conditioning reward generation on environmental feedback from the proof assistant. When a tactic executes, the assistant returns the resulting proof state, execution status, and error messages. GRMs condition on this concrete feedback, grounding their evaluations in actual proof assistant feedback responses rather than the tactic text alone. As a result, a tactic that fails due to a type error receives qualitatively different feedback from one that succeeds but fails to simplify the goal, enabling finer-grained guidance.

We consider two deployment strategies. Zero-shot GRMs operate without additional training, generating critiques directly from pretrained language models given structured prompts and execution feedback. While immediately applicable, their performance depends on the base model's reasoning ability. To improve alignment with proof outcomes, we also train GRMs using online reinforcement learning. The training objective encourages critiques whose scores correlate with proof progress: steps closer to completion receive higher scores, while failed or regressive steps re-

ceive lower ones. We employ policy gradient methods, specifically Group Relative Policy Optimization (GRPO), to optimize the model.

This chapter presents the GRM framework in detail. Section 4.2 describes the model design, output structure, and conversion from critiques to rewards. Section 4.3 details the reinforcement learning procedure. Section 4.4 explains how GRMs integrate with tree search algorithms.

4.2 Design

4.2.1 Output Representation

Reward models can produce different types of output. Discriminative models output scalars that directly encode quality. Generative models output natural language that explains evaluation. We use a hybrid: structured text with an embedded numerical score.

This hybrid design offers the best of both worlds: the numerical score enables efficient integration with tree search algorithms that require comparable values for ranking and selection, while the natural language critique provides human-interpretable explanations for debugging, analysis, and model improvement. Moreover, the act of generating explanatory text before producing a score may improve the score's accuracy—similar to how chain-of-thought prompting enhances reasoning in language models. [Wei et al., 2022] The structured critique also enables richer feedback loops: researchers can identify systematic errors in the model's reasoning, and the textual explanations can potentially be used to fine-tune or guide future iterations of the reward model.

For a proof state s_t, proposed tactic a_t, environment response f_t, and trajectory history $\tau_{<t}$, the generative reward model produces:

$$C_t = \mathcal{R}_\theta(s_t, a_t, f_t \mid \tau_{<t}) \tag{4.1}$$

where C_t is a natural language assessment. The critique discusses whether the tactic is correct, whether it makes progress, and whether the strategy is sound. At

the end, it provides a score on a 0-20 scale.

Example critique:

> *"This tactic applies the commutativity lemma to simplify the left side of the equation. The execution succeeded and reduced the number of goals from 2 to 1. However, the resulting goal is not obviously simpler—we may need additional rewrites. Score: 14/20"*

This format preserves interpretability while remaining compatible with tree search algorithms that need numerical values. We extract the score through simple parsing and use it for node ranking.

4.2.2 Scoring Pattern

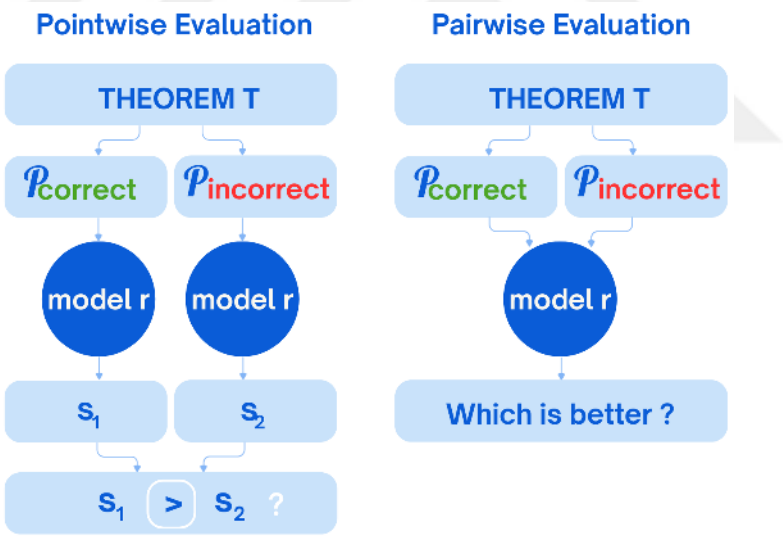

Figure 4.1: The difference between pointwise evaluation and pairwise evaluation

Reward models can evaluate proof steps independently (pointwise) or by comparing multiple candidates (pairwise). We use pointwise evaluation. Each proof step gets assessed on its own merits without reference to alternatives.

This decision simplifies integration with search algorithms. Best-first search and MCTS need value estimates for individual states, not pairwise comparisons. Gen-

erating multiple critiques for comparison would multiply inference cost. Pointwise evaluation provides the necessary signals efficiently.

While TreeThink includes a tournament-style node evaluator implementation—which generates multiple candidate expansions and selects the best through pairwise comparisons—we reserve the full integration of pairwise evaluation insights for future work. The tournament evaluator demonstrates how comparative judgments can refine node selection, but its sequential nature creates a wall-clock time bottleneck: pairwise comparisons cannot be parallelized as effectively as pointwise evaluations, where all nodes can be scored simultaneously. The core search algorithms in this thesis therefore rely on the more efficient pointwise approach. Exploring how to effectively leverage pairwise signals within MCTS value backpropagation and best-first priority queuing—while maintaining computational efficiency—remains an open direction for improving proof search quality.

4.2.3 Critique Structure

The critique has three parts:

Correctness Assessment Did the tactic execute successfully? Are there type errors or missing hypotheses? The model conditions on the environment response f_t, which includes execution status and error messages. If Lean reports "unknown identifier", the critique should note this. If execution succeeds, the critique can say so explicitly.

Progress Evaluation Does the new proof state move toward completion? The model compares the previous state s_{t-1} with the current state s_t. Fewer goals suggests progress. Simpler goals might be easier to solve. Goals that match known lemmas are promising. The critique explains what changed and whether it's helpful.

Strategic Value Is this a sensible direction? Some tactics make progress but head down dead ends. Others don't simplify goals but set up future steps. The model uses mathematical knowledge to assess whether the approach is sound. For

example, "applying induction here seems premature—we should first simplify the expression."

Numerical Score After the textual analysis, the critique ends with a score from 0 to 20. Low scores (0-5) indicate serious errors or clearly wrong directions. Medium scores (6-14) suggest the step is valid but its value is unclear. High scores (15-20) indicate strong progress or particularly insightful moves.

4.2.4 Environmental Feedback Integration

The critical design element is using environmental feedback. When we evaluate a proof step, the tactic has already executed in Lean. We know whether it worked. We have the new proof state. We can see error messages if it failed.

By conditioning on execution results, we ground evaluation in reality. The model doesn't guess whether a tactic will work—it knows. Critiques can reference specific error messages, discuss how the proof state changed, and reason about what worked or failed concretely.

Consider a tactic that rewrites using lemma X. Without environmental feedback, the model might say "this should simplify the goal" based on typical behavior. With environmental feedback showing "lemma X not found in scope", the critique can identify the specific error and suggest importing the necessary module.

4.2.5 From Critique to Reward

Tree search algorithms need scalar rewards for ranking. We extract these from critiques through parsing. The model generates text ending with "Score: N/20" where N is an integer. Simple pattern matching extracts this value.

We normalize scores to [0, 1] by dividing by 20. This gives the reward $r_t = s(C_t)/20$ where s extracts the numerical score. The normalization makes rewards compatible with standard RL algorithms and tree search implementations.

If parsing fails (malformed output, no score, invalid number), we assign a default low reward. During training, these failures become rare as the model learns the required format.

4.2.6 Prompt Design

See Appendix B for the full prompt templates used for zero-shot and trained GRMs.

4.3 Training Procedure

Zero-shot GRMs work immediately but their quality depends entirely on the base model. Training improves performance by teaching the model to recognize good and bad proof steps through reinforcement learning.

4.3.1 Training Objective

We want critiques whose scores align with actual proof quality. Steps that lead to successful proofs should score higher than steps that lead to failures. Steps close to completion should score higher than early exploratory steps.

The challenge is defining ground truth. We don't have human annotations of step quality. What we have is proof outcomes—whether search eventually succeeded or failed—and step positions in trajectories. We derive training labels from these.

4.3.2 Ground Truth Labels

For each step in a successful proof trajectory, we assign a label based on its position. If a proof has n steps and this is step i, the label is:

$$y_t = \frac{i}{n} \tag{4.2}$$

This captures the intuition that later steps are closer to completion and thus more valuable. The first step in a 10-step proof gets label 0.1, the last step gets 1.0. Steps in failed trajectories get label 0.

This labeling scheme is intentionally coarse. We assign zero labels to all steps in failed trajectories, even if some steps appear locally correct. The goal is not to estimate intrinsic step correctness, but to align reward signals with eventual proof success. A step that is locally reasonable but consistently leads to dead ends should

not be reinforced. While this may under-reward partial progress in failed proofs, it provides a stable and outcome-aligned signal without requiring human annotations.

4.3.3 Reward Function

The training reward encourages the model to assign higher scores to steps with higher labels. For a critique C_t with extracted score $s(C_t) \in [0, 20]$ and ground truth label $y_t \in [0, 1]$, we define:

$$r(C_t, y_t) = \begin{cases} \frac{s(C_t)}{20}(1 + \alpha y_t) & \text{if } y_t > 0 \text{ (successful path)} \\ \frac{20 - s(C_t)}{20} & \text{if } y_t = 0 \text{ (failed step)} \end{cases} \tag{4.3}$$

For successful steps, reward increases with both the predicted score and the ground truth label. The hyperparameter α controls how much we weight proximity to completion. Higher α emphasizes getting near-solution steps right.

For failed steps, reward increases when the model assigns low scores. A failed step that gets score 2/20 produces high reward because the model correctly identified it as bad. A failed step that gets score 18/20 produces low reward because the model was overconfident.

This reward formulation reflects two design principles. First, for successful trajectories, the model should assign higher scores to steps that are both predicted as high quality and closer to proof completion. The proximity term encourages discrimination between early exploratory steps and decisive steps near the end of a proof. Second, for failed trajectories, the model should be penalized for overconfidence. Assigning high scores to failed steps produces low reward, while correctly assigning low scores yields high reward. This asymmetry explicitly discourages false positives, which are particularly harmful in search guidance.

We use a simple linear formulation to keep optimization stable and interpretable. More complex shaping functions are possible, but we found this structure sufficient to produce consistent improvements in practice.

4.3.4 Policy Gradient Optimization

We optimize using GRPO (Group Relative Policy Optimization) [Shao et al., 2024], a policy gradient method that normalizes rewards within mini-batches. This reduces variance and improves training stability.

Standard policy gradients compute:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=1}^{|\tau|} \nabla_\theta \log p_\theta(C_t \mid \text{context}_t)\, r(C_t, y_t) \right] \tag{4.4}$$

where π_θ is the current policy (the GRM) and context_t includes the proof state, tactic, environment feedback, and history.

GRPO adds batch normalization. For a mini-batch $\mathcal{B}$ of trajectories, we compute:

$$\tilde{r}_t = \frac{r(C_t, y_t) - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}} + \epsilon} \tag{4.5}$$

where $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ are the mean and standard deviation of rewards in the batch. The gradient becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{(\tau, \{y_t\}) \sim \mathcal{B}} \left[\sum_{t=1}^{|\tau|} \nabla_\theta \log p_\theta(C_t \mid \text{context}_t)\, \tilde{r}_t \right] \tag{4.6}$$

Normalization provides three benefits. First, relative comparison within batches reduces the impact of reward scale variations across different problems. Second, it prevents gradient explosion from outlier rewards—one extremely high or low reward doesn't dominate the batch. Third, group-based statistics provide more stable learning signals than global baselines that require estimating value functions.

We choose GRPO over standard PPO-style objectives because reward normalization across groups of proof steps naturally fits the highly heterogeneous reward distributions encountered in theorem proving, where different problems and proof lengths produce vastly different raw reward scales. [Liu et al., 2025]

More information on policy gradient optimization can be found in Appendix A

4.3.5 Training Procedure

Training happens in iterations. At each iteration:

1. **Collect Trajectories:** Run proof search on a batch of problems using the current policy model and current GRM. This produces trajectories—sequences of proof states, tactics, and outcomes.

2. **Generate Critiques:** For each step in each trajectory, generate a critique using the current GRM. This includes the environmental feedback from when we originally executed the step during search.

3. **Compute Labels:** Assign ground truth labels based on trajectory outcomes and step positions.

4. **Compute Rewards:** Calculate training rewards for each critique-label pair.

5. **Update Parameters:** Apply policy gradient update using GRPO normalization.

The on-policy nature ensures the GRM learns to evaluate proof attempts from the current policy's distribution. Since we train the GRM while keeping the policy fixed, the tactic distribution remains stable, allowing the GRM to specialize on the specific types of proof states and tactics this policy generates. This alignment is crucial: the GRM must accurately score the actual proof paths the policy explores during search, not hypothetical paths from a different distribution. We leave the co-evolution scenario—where the policy and GRM are trained iteratively in concert, with the policy adapting to improved value estimates and the GRM continuously adjusting to the shifting distribution of proof attempts—as future work. Such iterative refinement could potentially lead to stronger proof search capabilities but introduces additional training complexity and computational overhead.

4.3.6 Two-Stage Training

We use a two-stage procedure inspired by recent work on generative reward models. The first stage teaches format, the second stage optimizes reward alignment.

Stage 1: Rejective Fine-Tuning (Cold Start) The model must learn to generate critiques in the correct structured format with appropriate scoring patterns. We collect a dataset of proof steps—some successful, some failed—and generate training examples where the model should produce well-formatted critiques.

This stage uses supervised learning. We don't worry about reward optimization yet. The goal is getting the model to consistently output critiques with textual analysis followed by "Score: N/20". We use rejection sampling: generate multiple candidates, keep only those with valid format, train on these.

After cold start, the model reliably produces structured critiques and shows basic discrimination between correct and incorrect step and this stage significantly reduces training instability in the subsequent reinforcement learning phase.

Stage 2: GRPO Training With format learned, we transition to online RL. Now we collect trajectories from actual proof search, generate critiques, compute rewards based on alignment with outcomes, and update using policy gradients.

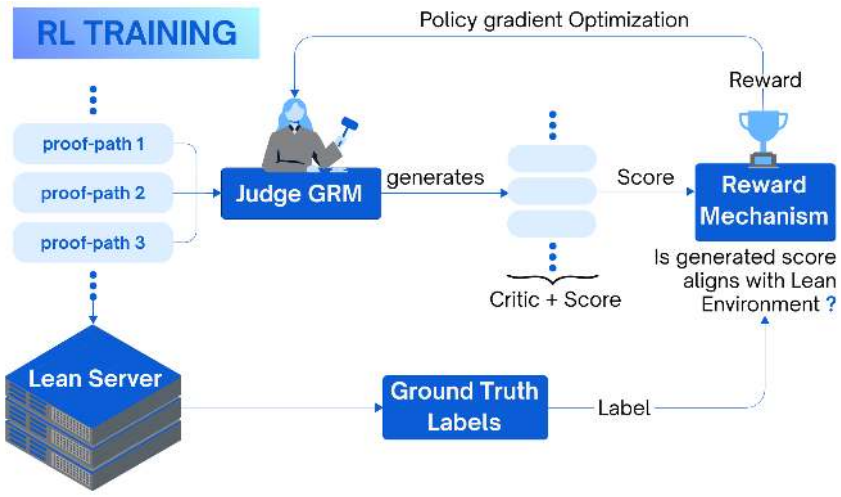

Figure 4.2: Reinforcement Learning Training Loop

This stage teaches the model to distinguish high-quality from low-quality proof steps based on whether they lead to success. The model learns that steps near completion in successful proofs should score high, early steps in failed proofs should score low, and so on.

Hyperparameters and training details are provided in Appendix C.

4.3.7 Model Initialization

We initialize the GRM from the same base model as the policy. This ensures consistency in mathematical understanding and reduces distribution mismatch. If the policy is DeepSeek-Prover-V2, the GRM starts from the same checkpoint.

Shared initialization means the GRM already understands Lean syntax, common proof patterns, and mathematical reasoning before training. We only need to teach it to evaluate rather than generate, and to align its evaluations with proof outcomes.

4.4 Integration with Tree Search

Tree search algorithms use GRM outputs to guide exploration. The integration process converts natural language critiques into the numerical values these algorithms need.

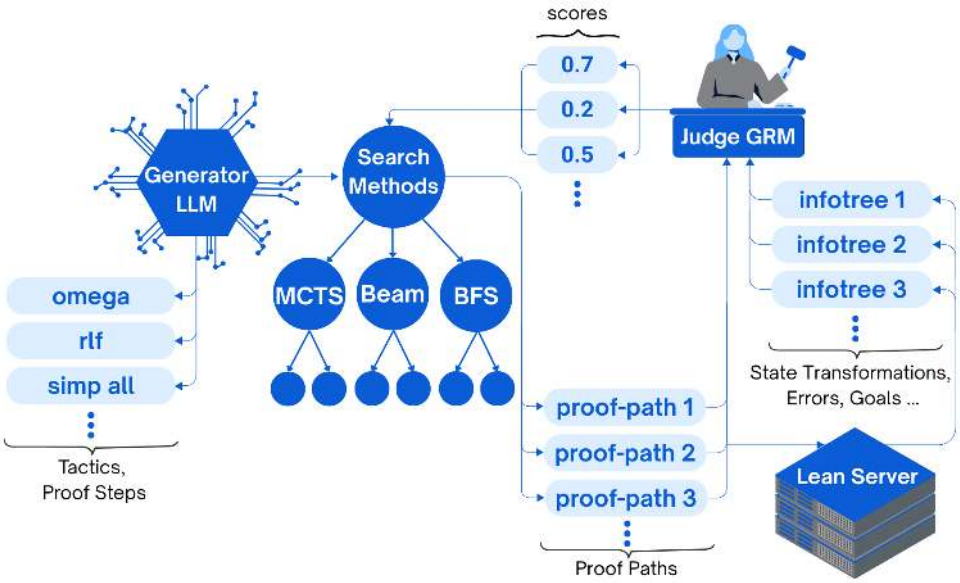

Figure 4.3: Overview of GRM Guided Search Pipeline

4.4.1 Value Estimation

All three search algorithms in TreeThink—best-first search, beam search, and MCTS—need value estimates for proof states. The GRM provides these through its numerical scores.

When a node is expanded, we generate candidate tactics using the policy model, execute them in the proof environment, and evaluate each resulting state with the GRM. For state s, tactic a, and environment response f, we get critique $C = \mathcal{R}_\theta(s, a, f \mid \tau)$. We extract score $s(C)$ and normalize to reward $r = s(C)/20$.

This reward guides search decisions. Best-first search uses it to rank the frontier. Beam search uses it to select the top-k states at each level. MCTS uses it to update Q-values during backpropagation.

4.4.2 Combining with Policy Priors

Some search algorithms combine reward model scores with policy model probabilities. For best-first search, we define a heuristic:

$$h(s) = \alpha \cdot \log p_\pi(a \mid s) + (1 - \alpha) \cdot r_{\text{GRM}}(s) \tag{4.7}$$

where $p_\pi(a \mid s)$ is the policy model's probability for the tactic that led to state s, and $r_{\text{GRM}}(s)$ is the normalized GRM score. The parameter $\alpha \in [0, 1]$ controls weighting.

This combination balances two signals. Policy priors capture which tactics the model thinks are likely based on training data. GRM scores capture which states are actually promising based on environmental feedback and strategic analysis. This heuristic is used for best-first search. Beam search relies solely on GRM scores for ranking, while MCTS integrates GRM outputs directly into value backpropagation. Consequently, the mixing coefficient α is set to zero for these methods.

4.4.3 Handling Inference Cost

Generative reward models are expensive. Generating a 100-token critique takes much longer than computing a scalar from a discriminative model. This cost affects search efficiency.

TreeThink addresses this through batched inference. Rather than generating critiques sequentially, we collect multiple evaluation requests and process them together. vLLM's AsyncLLMEngine batches requests dynamically, achieving much higher throughput.

We also cache critiques. If we evaluate the same proof state multiple times during search (which happens in MCTS), we reuse the previously generated critique rather than generating a new one. This cache hit rate can be substantial, especially in MCTS where states are visited repeatedly.

We do not claim that GRM-guided search is computationally cheaper than discriminative alternatives; rather, batching and caching make it practical at the scales considered in this work.

4.4.4 Error Handling

GRM generation can fail in various ways. The model might produce malformed output without a parseable score. It might generate text that doesn't address the proof step. It might hang or exceed length limits.

We handle these robustly with retry logic and fallback mechanisms. If parsing fails, we retry up to a fixed number of attempts before assigning a neutral score and logging the failure. If generation times out or repeatedly fails, we use a threshold-based fallback: if the policy model's probability exceeds a predetermined threshold, we use it as the reward; otherwise, we assign a neutral score. These failures become rare after training, but handling them gracefully prevents search crashes.

4.4.5 Experimental Setup

Dataset and Metrics

MiniF2F contains 488 olympiad-level problems formalized in Lean 4, split into validation (244 problems) and test (244 problems) sets. Problems cover algebra, number theory, combinatorics, and geometry. We report pass rate—the percentage of problems where search finds a verified correct proof.

Computational Budget

We control computational budget using three-component notation: passes $\times$ children $\times$ expansions. The passes parameter specifies independent search attempts per problem with different random seeds. The children parameter controls how many candidate tactics are generated from each expanded proof state. The expansions parameter limits the maximum number of nodes expanded in a single search pass.

For example, budget $8\times4\times2048$ means 8 independent search passes, 4 candidate tactics per expansion, and up to 2048 total expansions per pass. This notation makes resource allocation explicit and comparable across experiments.

This notation explicitly separates sources of computational cost, enabling controlled and transparent comparison across methods.

Models

We use DeepSeek-Prover-V2-7B as the policy model across all experiments. This isolates the effect of reward model choice—differences in results come from the reward model, not the policy. For comparison, we also test other specialized models (Goedel-LM) and general-purpose models (Qwen 2.5, Llama 3.1) as reward models where relevant.

4.4.6 Research Question 1: Zero-Shot Effectiveness

Question

Does a generative reward model that receives environmental feedback and generates natural language critiques improve tree search performance compared to conventional log-probability heuristics, without any fine-tuning?

Experimental Design

We compare three configurations using best-first tree search at budget 4×4×512:

1. **Logprobs baseline**: Use policy model's log-probabilities as node scores, following the GPT-f approach

2. **Zero-shot GRM without environmental feedback**: Generate critiques based only on proof context and proposed tactic, without seeing execution results

3. **Zero-shot GRM with environmental feedback**: Generate critiques after tactic execution, conditioning on proof state changes, execution status, and error messages

All GRM configurations use DeepSeek-Prover-V2-7B. Prompts are identical except for the environmental feedback component. We evaluate on miniF2F-test.

Results

Table 4.1 shows the results. The logprobs baseline achieves 53.7% on miniF2F-test. Zero-shot GRM without environmental feedback performs significantly worse at 47.1%—actually below the baseline. This suggests that language model reasoning alone, without execution grounding, cannot reliably evaluate proof steps.

Table 4.1: Impact of environmental feedback on zero-shot GRM performance. Policy and reward model: DeepSeek-Prover-V2-7B. Search: Best-first. Budget: 4×4×512.

Configuration	miniF2F-test (%)
Logprobs baseline	53.7
Zero-shot GRM (no env feedback)	47.1
Zero-shot GRM (with env feedback)	61.4

When environmental feedback is added, performance jumps to 61.4%—a 14.3 point improvement over the no-feedback version and 7.7 points over logprobs. This demonstrates that grounding GRM evaluations in concrete compiler responses is essential. The model sees what actually happened when the tactic ran: whether it succeeded, what error occurred if it failed, how the proof state changed.

The improvement over logprobs is substantial. Traditional heuristics fuse generation likelihood with step quality, while GRMs with environmental feedback can separately assess whether tactics advance the proof based on execution results.

Analysis

Why does environmental feedback matter so much? Consider evaluating the tactic `apply lemma_X`. Without execution feedback, the model must guess whether `lemma_X` is in scope, whether its types match the current goal, and whether applying it will help. These predictions are unreliable.

With execution feedback, the model knows the outcome. If Lean reports "unknown identifier lemma_X", the critique can identify this specific error. If execution

succeeds but produces a more complex goal, the critique can note that the tactic was valid but counterproductive. This grounding in reality improves evaluation accuracy.

The worse-than-baseline performance without environmental feedback suggests that ungrounded language model reasoning can be misleading. The model generates plausible-sounding evaluations that don't reflect actual proof quality, leading search astray.

This setting is intentionally included as a stress test, highlighting that language model reasoning alone is insufficient without execution grounding.

4.4.7 Research Question 2: Specialized vs General Models

Question

Do theorem proving-specialized language models perform better as zero-shot generative reward models than general-purpose models with strong mathematical reasoning capabilities?

Experimental Design

We compare four models as zero-shot GRMs, all with environmental feedback:

Specialized models:

- DeepSeek-Prover-V2-7B: Trained on formal theorem proving data
- Gödel-LM Prover V2-8B: Another theorem proving specialist

General-purpose models:

- Qwen 2.5-7B Instruct: Strong general instruction-following model
- Llama 3.1-8B Instruct: Another capable general-purpose model

All use DeepSeek-Prover-V2-7B as policy. All use best-first search. Budget is $8\times8\times512$ for most models (Qwen uses $4\times8\times512$ due to inference cost). This isolates reward model quality—the policy and search algorithm are identical, only the GRM changes.

Results

Table 4.2 shows performance across models. DeepSeek-Prover-V2 achieves 66.4%, substantially outperforming general-purpose models. Qwen 2.5 reaches 53.2% and Llama 3.1 reaches 50.0%.

Table 4.2: Zero-shot GRM performance by model type. All use environmental feedback, DeepSeek-Prover-V2-7B as policy, and best-first search.

Model	Type	Budget	Pass (%)
DeepSeek-Prover-V2-7B	Specialized	8×8×512	66.4
Goedel-LM Prover V2-8B	Specialized	8×8×512	55.7
Qwen 2.5-7B Instruct	General	4×8×512	53.2
Llama 3.1-8B Instruct	General	8×8×512	50.0

The gap between DeepSeek-Prover-V2 and Qwen—13.2 points despite identical 7B parameter counts—demonstrates that domain-specific pretraining on formal mathematics directly improves proof step evaluation. Even without fine-tuning, specialized models better recognize valid Lean tactics, understand proof progress, and assess strategic value.

Interestingly, performance varies even among specialized models. DeepSeek-Prover-V2 (66.4%) outperforms Goedel-LM (55.7%) by 10.7 points. This suggests that training data quality and model architecture matter beyond just specialization.

One potential contributing factor is that the policy and reward models share the same base model in the DeepSeek setting, which may reduce distribution mismatch between generation and evaluation. In contrast, Gödel-LM is used solely as a reward model while the policy remains DeepSeek-Prover-V2. While this alignment effect may partially explain the performance gap, the magnitude of the difference indicates that model-specific pretraining and architectural choices also play a significant role. A more systematic study of policy–reward model pairing is left for future work.

Analysis

Why do specialized models excel? They've seen thousands of formal proofs during training. They understand Lean syntax, common proof patterns, and which tactics typically help in different situations. This knowledge transfers to evaluation even without explicit reward modeling training.

General-purpose models struggle despite strong performance on broad reasoning benchmarks. Qwen and Llama are capable instruction-followers with good mathematical reasoning in natural language. But formal theorem proving requires precise understanding of type systems, tactic semantics, and proof assistant behavior. These systems operate differently from natural language math.

The result has practical implications: when building systems with GRMs, use specialized models if available. The performance gain is substantial even in zero-shot settings. Fine-tuning general models might close the gap, but starting from a specialized base is more efficient.

4.4.8 Research Question 3: Impact of Training

Question

Can generative reward models trained with reinforcement learning on proof trajectories provide better guidance than zero-shot models?

Training Setup

We train a GRM using the two-stage procedure described in Section 4.3. Starting from DeepSeek-Prover-V2-7B, we first perform rejective fine-tuning to learn critique format, then apply GRPO-based online RL for reward alignment.

Training data comes from approximately 23,000 problems in the Lean-Workbook dataset. We generate 8 complete proof attempts per problem through autoregressive generation, validate against Lean, and construct training examples from intermediate proof steps. This yields 100,000 critique generation examples covering various proof stages, error types, and success patterns.

The cold start stage uses supervised learning on this data to ensure the model generates well-formatted critiques with appropriate scoring. The GRPO stage uses online RL with rule-based rewards that encourage distinguishing high-quality responses while maintaining format.

Results

Table 4.3 shows progression through training stages. Zero-shot DeepSeek-Prover-V2 achieves 66.4% at budget 8×8×512 with best-first search. After cold start rejective fine-tuning, performance improves modestly to 66.9%. This validates that the model learns the critique format and shows basic improvement.

Table 4.3: GRM performance through training stages. Policy: DeepSeek-Prover-V2-7B. Search: Best-first. Budget: 8×8×512.

Training Stage	miniF2F-test (%)
Zero-shot (no training)	66.4
After cold start (rejective FT)	66.9
After GRPO training	70.3

GRPO training brings performance to 70.3%—a 3.9 point total improvement over zero-shot baseline. This demonstrates that the two-stage procedure successfully teaches the model to recognize productive proof steps. The rejective fine-tuning provides stable initialization, and GRPO optimization aligns scores with proof outcomes.

Scaling to Higher Budgets

The trained GRM's effectiveness scales across computational budgets and search algorithms. Table 4.4 shows results at different resource levels.

At budget 8×4×2048, trained GRM with best-first search reaches 70.8%—a 9.7 point improvement over the logprobs baseline (61.1%). With MCTS, performance

Table 4.4: Trained GRM performance across budgets and algorithms.

Configuration	Budget	miniF2F-test (%)
Logprobs + BFTS	8×4×2048	61.1
Trained GRM + BFTS	8×4×2048	70.8
Trained GRM + MCTS	8×4×2048	71.6
Trained GRM + BFTS	128×8×2048	73.7
Trained GRM + MCTS	128×8×2048	76.9

further improves to 71.6%. This shows the trained GRM provides effective guidance for multiple search algorithms.

At higher compute (128×8×2048), the gains persist. Best-first search with trained GRM achieves 73.7%. MCTS reaches 76.9%, establishing new state-of-the-art on miniF2F-test and exceeding prior best results like BFS-Prover (70.83%) and HunyuanProver (68.4%).

Analysis

The training procedure successfully improves GRM quality. The scaling behavior is encouraging. As we increase computational budget, the trained GRM continues to provide value. This suggests the learned evaluations are robust—they help across different numbers of search passes, expansion depths, and algorithms.

The state-of-the-art results at 128×8×2048 budget demonstrate that GRMs can guide competitive theorem proving systems. The 76.9% performance matches or exceeds specialized systems with comparable model sizes and computational resources. While higher budgets naturally improve performance, the consistent gap between GRM-guided search and log-probability baselines across budgets indicates that gains are not solely due to increased compute.

Why does training help? The RL objective teaches the model to distinguish steps that lead to success from those that lead to failure. Steps near proof completion should score higher than early exploratory steps. Failed tactics should score lower

than successful ones. The model learns these patterns from thousands of proof trajectories, developing intuition about what makes progress.

The cold start stage is important—it ensures stable training by providing strong initialization. Without format learning first, GRPO might produce malformed outputs that can't be parsed into rewards. The two-stage approach addresses this.

4.4.9 Comparison with State-of-the-Art

Table 4.5 compares our trained GRM approach with recent search-based theorem proving systems on miniF2F-test.

Table 4.5: Comparison with state-of-the-art systems on miniF2F-test.

Method	Size	Budget	Pass (%)
Prior Work			
Hypertree Proof Search	600M	64×5k	41.0
InternLM2.5-StepProver+BFS+CG	7B	256×32×600	65.9
HunyuanProver v16+BFS+DC	7B	600×8×400	68.4
BFS-Prover	7B	2048×2×600	70.8
Our Work			
Logprobs + BFTS	7B	8×4×2048	61.1
Trained GRM + BFTS	7B	8×4×2048	70.8
Trained GRM + MCTS	7B	8×4×2048	71.6
Trained GRM + BFTS	7B	128×8×2048	73.7
Trained GRM + MCTS	7B	128×8×2048	**76.9**

At comparable budgets and model sizes, trained GRM matches or exceeds prior results. The 70.8% with BFTS at 8×4×2048 matches BFS-Prover's 70.8% at a different budget configuration. With MCTS, we reach 71.6%. At higher compute, 76.9% exceeds all listed prior work.

These results demonstrate that GRMs provide competitive guidance for theorem proving. The natural language critiques with environmental feedback grounding

enable effective proof search.

These findings validate the GRM framework and demonstrate that generative reward models can provide strong guidance for automated theorem proving systems.

4.4.10 Summary of Experimental Findings

Three key findings emerge from our experiments:

1. **Environmental feedback is essential.** Zero-shot GRMs without execution grounding perform worse than simple log-probability heuristics. Adding environmental feedback yields a 14.3 point improvement, demonstrating that grounding in compiler responses is critical for accurate evaluation.

2. **Specialized models excel as GRMs.** Domain-specific pretraining on formal mathematics provides substantial advantages. DeepSeek-Prover-V2 outperforms Qwen 2.5 by 13.2 points despite identical parameter counts, showing that theorem proving knowledge directly transfers to proof evaluation.

3. **Training improves performance.** GRPO-based fine-tuning on proof trajectories increases effectiveness by 3.9 points at moderate budgets, with gains persisting at higher compute. Trained GRMs achieve state-of-the-art results, reaching 76.9% on miniF2F-test with MCTS at 128×8×2048 budget.

We focus only on miniF2F because it provides a standardized and widely adopted benchmark for formal theorem proving. Extending evaluation to larger benchmarks such as ProofNet or PutnamBench is left for future work due to computational constraints.

4.5 Summary

We presented generative reward models for formal theorem proving. The key ideas are:

- Structured natural language critiques with embedded numerical scores

- Conditioning on environmental feedback from proof assistant execution
- Online RL training using GRPO on proof trajectories with outcome-based labels
- Integration with tree search through value estimation and batched inference

The GRM framework provides richer guidance than existing approaches. Zero-shot GRMs work immediately with base models. RL-trained GRMs learn to align scores with proof success through policy gradient optimization. Integration with TreeThink enables systematic evaluation across search algorithms.

Chapter 5

FORMALREWARDBENCH: A BENCHMARK FOR REWARD MODELS

5.1 Introduction

Reward models guide reinforcement learning in many domains. For language model alignment, reward models learn human preferences from comparison data. [Ouyang et al., 2022] For code generation, they distinguish correct from incorrect solutions. [Huang et al., 2025]For theorem proving, they should evaluate proof quality.

But no benchmark exists for reward models in formal theorem proving. RewardBench [Lambert et al., 2025] evaluates reward models in natural language domains. ProcessBench [Zheng et al., 2025] evaluates process reward models for informal math. Neither addresses formal proofs. This gap blocks progress on preference-based learning for automated theorem proving.

The challenge is fundamental. Natural language math allows flexibility—multiple valid approaches, subjective judgments of clarity, approximations that are "close enough". Formal proofs require absolute correctness. Every step must be syntactically valid, type-correct, and logically sound. A proof that seems reasonable in natural language might contain type errors or lemma misapplications that only formal verification catches.

Current theorem proving benchmarks measure success: did the prover find a correct proof? They use pass@k metrics—the percentage of problems solved in k attempts. While useful for measuring overall capability, these metrics can't evaluate proof understanding. When a prover fails, we don't know if the attempt was close to working or fundamentally wrong. When it succeeds, we can't assess which tactics were insightful versus merely correct.

This limitation matters for modern reward modeling approaches. Methods like

RLHF, DPO, and generative reward models need to compare proof attempts and provide nuanced feedback. A proof failing from one wrong lemma is preferable to one with fundamental logical errors, but pass@k treats them identically. Training reward models requires preference data that existing benchmarks can't provide.

We introduce FormalRewardBench, the first benchmark for evaluating outcome reward models in formal theorem proving. The core idea is controlled error injection: start with verified correct proofs, systematically generate incorrect variants, create preference pairs, evaluate whether models correctly prefer the correct version.

Our error injection strategies target realistic failure modes. Language models generating formal proofs make specific types of mistakes: wrong hypotheses, incorrect lemma applications, type mismatches, plausible but unsound reasoning. We design error buckets that capture these patterns, creating challenging but fair evaluation.

The benchmark supports both pointwise and pairwise evaluation. Pointwise evaluation gives independent scores to each proof and checks whether correct proofs score higher. Pairwise evaluation shows both proofs simultaneously and checks whether models directly prefer the correct one. Both modes are useful for different reward model architectures.

This chapter describes FormalRewardBench in detail.

5.2 Benchmark Design

5.2.1 Design Principles

Three principles guide our design:

Controlled Difficulty Errors should be challenging but fair. We don't want trivial syntax errors that any model would catch. We also don't want subtle logical flaws that require deep mathematical insight. The sweet spot is errors that plausible models might miss but capable models should detect.

We achieve this through systematic error injection. Start with correct proofs, apply specific transformations, verify the results are actually wrong but not obvi-

ously so. This control lets us adjust difficulty and ensures coverage of different error types.

Realistic Failure Modes Errors should reflect how language models actually fail at theorem proving. We analyzed common mistakes from deployed systems: applying lemmas with wrong hypotheses, confusion between similar variables, type mismatches from integer/real mixing, plausible but incorrect logical steps.

Each error bucket targets a specific failure mode. This makes the benchmark practically relevant—good performance suggests models will handle real errors from actual proof search.

Verifiable Ground Truth Every proof must be verifiably correct or incorrect. We run all generated proofs through Lean's type checker. Correct proofs must pass completely. Incorrect proofs must fail for semantic reasons, not trivial syntax errors.

This verification is critical. Without it, we couldn't be certain our "incorrect" proofs are actually wrong, or that errors are meaningful rather than superficial. Lean's type checker provides ground truth we can trust.

5.2.2 Benchmark Structure

FormalRewardBench consists of preference pairs:

$$\mathcal{D} = \{(T^{(i)}, P^{(i)}_{\text{correct}}, P^{(i)}_{\text{incorrect}})\}_{i=1}^{N} \tag{5.1}$$

where $T^{(i)}$ is a theorem statement, $P^{(i)}_{\text{correct}}$ is a verified correct proof, and $P^{(i)}_{\text{incorrect}}$ is a verified incorrect proof.

The task is preference prediction: given T, P_{correct}, and $P_{\text{incorrect}}$, identify which proof is correct. We evaluate whether models can consistently prefer the correct variant.

5.2.3 Error Injection Strategies

We design five error injection strategies (buckets), each targeting different dimensions of proof correctness.

Bucket 1: Forced Mistakes Generate proofs with common errors that language models make. These include applying tactics that don't apply in the current context, using wrong hypotheses when multiple are available, instantiating lemmas with incorrect terms, or making invalid logical steps that seem plausible.

The constraint is plausibility. Errors shouldn't be obvious—they should look like mistakes a model trying to prove the theorem might make. A human reading the proof should find it superficially reasonable until checking against Lean.

Example: In a proof about integer inequalities, using hypothesis $h_1 : a < b$ when the correct one is $h_2 : a \leq b$. Both are available, both are about the same variables, but only one fits the current goal.

Bucket 2: Minimal Single-Point Variations Make the smallest possible change that breaks correctness. This tests fine-grained discrimination. The correct and incorrect proofs differ by literally one token—a variable name, a lemma name, a type annotation.

Valid variations include:

- Swapping similar variables (n vs m, x vs y)
- Using wrong hypothesis among similar ones (h_1 vs h_2)
- Type mismatches ($\mathbb{N}$ vs $\mathbb{Z}$, $\mathbb{R}$ vs $\mathbb{Q}$)
- Wrong lemma from same family (commutativity vs associativity)

These changes are visually difficult to spot but semantically critical. Models must carefully track variable scopes, type constraints, and available hypotheses.

We also create pairs where both proofs are incorrect but one is "less wrong"—a minimal error versus a fundamental flaw. This tests whether models can distinguish error severity.

Bucket 3: Complex Incorrect Proofs Generate long proofs with many steps and intermediate lemmas but flawed reasoning. The prompt requests sophisticated structure, multiple proof techniques, detailed argumentation. The result looks impressive—many lines, advanced tactics, intermediate results—but contains logical errors in the chain.

This tests whether models are deceived by superficial complexity. A short correct proof should be preferred over a long incorrect proof, even if the long version seems more thorough. Models should not assume that length is an indicator of correctness.

Example: A 20-line proof that applies induction, uses several lemmas, and has detailed case analysis, but makes a subtle error in one case that invalidates the entire argument.

Bucket 4: Natural Language Justification Take an incorrect proof and add natural language comments explaining the reasoning as if it were correct. The proof is still wrong, but now it has plausible justifications for each step.

This is a particularly challenging case. Models must identify formal errors even when incorrect proofs come with convincing explanations. The formal content is incorrect, but the informal comments are persuasive. Models that focus too much on natural language reasoning will be mistaken.

We also create pairs where a preferred incorrect proof has helpful comments while a dispreferred incorrect proof lacks them. This tests whether models can distinguish "well-explained but wrong" from "poorly-explained and wrong" when both fail formally.

Listing 5.1: Example of misleading comments in Lean code: The first comment claims to apply commutativity but uses `wrong_lemma` instead. The second comment instructs to use `h_2`, but `h_1` is the correct hypothesis.

```
-- First, apply commutativity to swap terms
rw [wrong_lemma]
-- Now we can simplify using the hypothesis
exact h_2
```

Bucket 5: Python Code Injection Replace or supplement Lean tactics with Python code. The Python might be completely correct—it actually solves the math problem computationally. But the task requires a formal Lean proof, so this is fundamentally wrong despite being mathematically valid.

This tests task adherence. Models trained extensively on code might prefer Python syntax even when it's inappropriate. Correct code in the wrong language doesn't represenet a valid formal proof.

Listing 5.2: Example of informal verification instead of formal proof: Instead of proving $n + m = m + n$ formally in a theorem prover, the code only provides a runtime test function that checks specific instances.

```
def test_commutativity(n, m):
    return n + m == m + n
```

The Python is right, but it's not a formal proof in Lean's logic.

5.2.4 Quality Control

Every generated incorrect proof undergoes validation:

1. **Syntactic Validity:** Must parse in Lean 4 without syntax errors
2. **Semantic Incorrectness:** Must fail Lean's type checker
3. **Non-Trivial Error:** Must fail for semantic reasons (type errors, wrong tactics, invalid logic), not parsing issues
4. **Plausibility Check:** Must not be obviously wrong to human inspection

We manually inspect samples from each bucket to verify they meet quality standards. Proofs that fail validation are discarded and regenerated.

5.3 Construction Pipeline

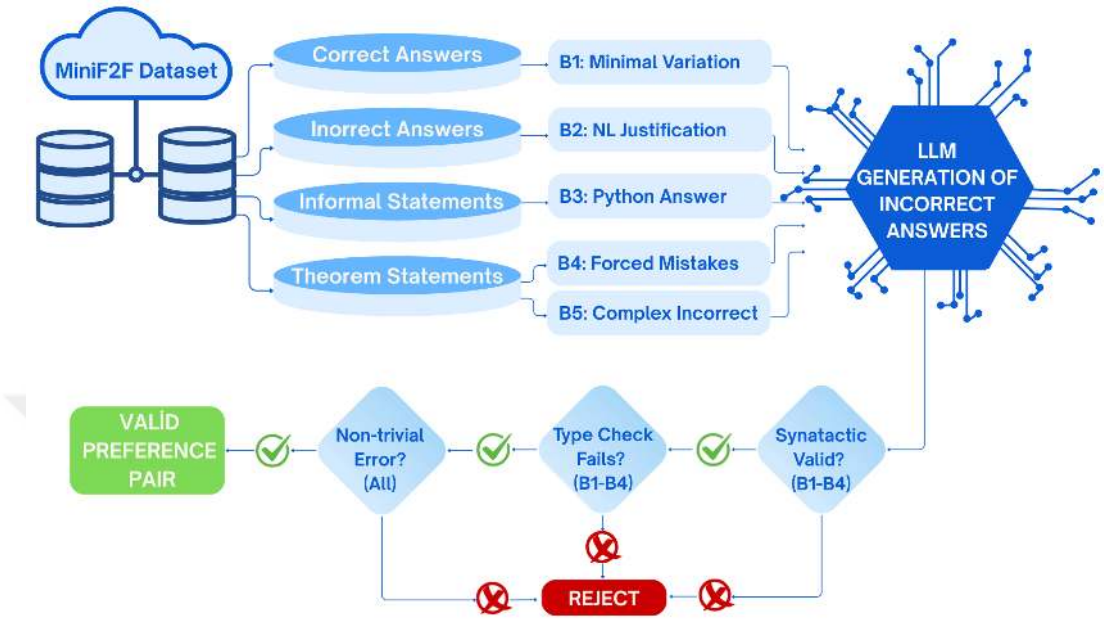

Figure 5.1: FormalRewardBench construction pipeline.

5.3.1 Source Data

We build on miniF2F, a curated set of 488 olympiad-level problems formalized in Lean 4. These problems cover diverse mathematical domains: algebra, number theory, combinatorics, geometry. Each comes with a verified correct proof.

MiniF2F provides ideal source material. The problems are challenging but not impossibly hard. The proofs are clean and well-structured.

We use both original miniF2F proofs and proofs generated by strong theorem proving models. This increases diversity in proof style and length. Some proofs are short and direct, others are longer with multiple steps. This variety ensures the benchmark covers different complexity levels.

5.3.2 Generation Process

For each source theorem with correct proof P_{correct}:

1. **Select Bucket:** Randomly choose one of five error injection strategies

2. **Generate Candidates:** Prompt language model with bucket-specific instructions

3. **Validate Candidates:** Run each through Lean, keep only those that fail semantically

4. **Quality Filter:** Remove trivially wrong or obviously broken variants

5. **Sample Final:** Randomly select one validated candidate as $P_{\text{incorrect}}$

We generate multiple candidates per theorem and bucket. This gives us options and lets us select the best example—not too easy, not impossibly subtle.

5.3.3 Prompt Engineering

Each bucket uses a carefully designed prompt. The prompt includes:

- The correct proof as reference
- Bucket-specific instructions (e.g., "make a minimal single-token change")
- Constraints on error type
- Examples of acceptable and unacceptable errors

Prompt details can be found in Appendix D

5.3.4 Bucket Distribution

We aim for balanced coverage across buckets. Each should contribute roughly equally to the final benchmark. This ensures models can't exploit distributional biases—they must handle all error types.

In practice, some buckets are harder to generate than others. Bucket 2 (minimal variations) requires many candidates to find truly minimal changes that break correctness. Bucket 5 (Python injection) is easier because any Python code fails Lean verification.

We generate more candidates for difficult buckets and sample to achieve desired distribution.

5.4 Evaluation Protocols

5.4.1 Pointwise Evaluation

In pointwise evaluation, the reward model scores each proof independently. Given theorem T, correct proof P_{correct}, and incorrect proof $P_{\text{incorrect}}$, the model produces:

$$s_{\text{correct}} = r(T, P_{\text{correct}}) \tag{5.2}$$

$$s_{\text{incorrect}} = r(T, P_{\text{incorrect}}) \tag{5.3}$$

where r is the reward function. A sample is classified as correct if:

$$\mathbb{1}[s_{\text{correct}} > s_{\text{incorrect}}] = 1 \tag{5.4}$$

This protocol works for discriminative reward models that output scalars and generative reward models that produce scores. Both can assign independent values to proofs.

5.4.2 Pairwise Evaluation

In pairwise evaluation, the model directly compares both proofs. Given T, P_{correct}, and $P_{\text{incorrect}}$ simultaneously, the model outputs a judgment:

$$\hat{I} = \text{judge}(T, P_{\text{correct}}, P_{\text{incorrect}}) \tag{5.5}$$

where $\hat{I} \in \{\text{correct}, \text{incorrect}\}$ indicates preference. A sample is correct if:

$$\mathbb{1}[\hat{I} = \text{correct}] = 1 \tag{5.6}$$

This protocol suits LLM-as-Judge approaches that naturally handle pairwise comparison. The model sees both proofs and generates a preference judgment.

5.4.3 Position Bias Mitigation

Language models exhibit position bias—they favor responses in certain positions regardless of quality. To control for this, we require consistency across orderings.

For each preference pair, we evaluate twice: once with proofs in original order $(P_{\text{correct}}, P_{\text{incorrect}})$, once with them swapped $(P_{\text{incorrect}}, P_{\text{correct}})$. A sample is correct only if both evaluations give the right answer:

$$\text{Correct}_{\text{consistent}} = \text{Correct}_{\text{original}} \wedge \text{Correct}_{\text{swapped}} \tag{5.7}$$

This is a strict criterion. Models must prefer the correct proof regardless of position. Inconsistent judgments suggest the model relies on position rather than content.

5.4.4 Metrics

Accuracy The primary metric is accuracy—the percentage of examples where the model correctly prefers the correct proof:

$$\text{Accuracy} = \frac{1}{N} \sum_{i=1}^{N} \text{Correct}^{(i)} \tag{5.8}$$

For pairwise evaluation with position bias mitigation, we report accuracy with consistency requirement.

Per-Bucket Accuracy We also report accuracy broken down by error injection bucket. This reveals which error types are easy or hard for different models. A model might excel at detecting Python injection (Bucket 5) but fail on minimal variations (Bucket 2).

We verify benchmark validity through several checks:

Automatic Verification Every proof runs through Lean's type checker. Correct proofs must pass fully. Incorrect proofs must fail. This automated verification ensures ground truth accuracy.

Manual Inspection We manually review samples from each bucket. Correct proofs should be genuinely correct—not relying on sorry or axioms. Incorrect proofs should have meaningful errors—not trivial syntax mistakes.

5.5 Comparison with Existing Benchmarks

5.5.1 RewardBench

RewardBench evaluates reward models in natural language. It covers chat, safety, and reasoning tasks. The reasoning portion includes informal math problems. RewardBench is valuable for general reward modeling but doesn't address formal mathematics. The verification and precision requirements differ fundamentally.

5.5.2 ProcessBench and PRMBench

ProcessBench and PRMBench evaluate process reward models on informal math. Models must identify correct and incorrect reasoning steps in natural language solutions.

These benchmarks are closer to our work—they evaluate mathematical reasoning at step level. But they operate in informal space where ambiguity is acceptable and multiple solution styles are valid.

FormalRewardBench requires absolute correctness. A proof that is "mostly right" or "on the right track" is still wrong if Lean rejects it. This precision distinguishes formal evaluation.

5.5.3 Theorem Proving Benchmarks

MiniF2F, ProofNet, and PutnamBench measure theorem proving success: did the prover find a valid proof? They use pass@k metrics.

FormalRewardBench serves a different purpose. We assume proofs have been generated and ask whether models can evaluate them. This complements success metrics—you might solve problems without understanding why proofs work, or understand proof quality without being able to generate proofs yourself.

Our benchmark enables training outcome reward models for evaluating proof understanding beyond binary verification, and applying modern preference learning/ reinforcement learning techniques to formal mathematics.

5.6 Experimental Evaluation

We evaluate multiple classes of models on FormalRewardBench to understand current capabilities and limitations in formal proof evaluation. Experiments examine overall performance, compare specialized versus general models, and analyze difficulty across error types.

5.6.1 Experimental Setup

Models Evaluated

We evaluate three categories of models:

Judge LLMs Models specifically designed or fine-tuned for preference evaluation and judgment tasks. These are trained on diverse preference datasets and optimized for comparing responses across domains. We test:

- ZiyiYe Con-J-Qwen2-7B: Preference-trained judge model
- AtlaAI Selene-1-Llama-3.3-70B: Large-scale judge model
- CompassJudger-1-7B and 14B: Judge models at different scales
- ContextualAI LMUnit-qwen2.5-72b: Large judge model
- Skywork-Critic-Llama-3.1-8B and 70B: Critic models at different scales
- RISE-Judge-Qwen2.5-7B: Recent judge model

General-Purpose LLMs Strong instruction-following models not specifically trained for judgment. These represent capable general assistants with strong code understanding:

- Qwen 2.5-72B-Instruct: Large general-purpose model
- Qwen 2.5-Coder-32B-Instruct: Code-specialized variant
- DeepSeek-Coder-V2-Lite-Instruct: Code-focused model

Theorem Proving Specialized Models Models specifically trained for formal theorem proving. These have been trained on formal proof generation datasets and optimized for Lean 4:

- DeepSeek-Prover-V2-7B: Latest specialized prover
- DeepSeek-Prover-V1.5-SFT and RL: Earlier versions with different training

Evaluation Metrics

We report accuracy—the percentage of preference pairs where the model correctly identifies the correct proof as superior. For pairwise evaluation, we also report consistency-based accuracy requiring correct predictions across both proof orderings.

5.6.2 Overall Performance

Table 5.1 presents overall performance across all evaluated models. Results reveal striking patterns in model capabilities.

Key Observations

Even the best model—Con-J-Qwen2-7B at 52.4% pointwise—only marginally exceeds random guessing. This demonstrates that formal proof evaluation might be extremely challenging.

Table 5.1: Overall performance on FormalRewardBench. Pointwise scores show accuracy when independently scoring proofs; pairwise scores show accuracy with position consistency requirement.

Model	**Pointwise (%)**	**Pairwise (%)**
Judge LLMs		
ZiyiYe Con-J-Qwen2-7B	52.4	-
AtlaAI Selene-1-Llama-3.3-70B	46.8	44.4
CompassJudger-1-7B-Instruct	42.4	30.8
ContextualAI LMUnit-qwen2.5-72b	41.2	36.8
CompassJudger-1-14B-Instruct	40.0	35.2
Skywork-Critic-Llama-3.1-70B	36.4	25.6
RISE-Judge-Qwen2.5-7B	17.2	32.0
Skywork-Critic-Llama-3.1-8B	0.0	22.0
General-Purpose LLMs		
Qwen 2.5-72B-Instruct	39.6	-
Qwen 2.5-Coder-32B-Instruct	36.4	37.6
DeepSeek-Coder-V2-Lite-Instruct	32.8	28.0
Theorem Proving Specialized		
DeepSeek-Prover-V2-7B	12.8	23.6
DeepSeek-Prover-V1.5-SFT	12.0	6.4
DeepSeek-Prover-V1.5-RL	11.2	9.2

Many models perform substantially below 50%, indicating they consistently prefer incorrect proofs. This is worse than random—the models have learned systematic biases that hurt performance.

Judge LLMs achieve much higher accuracy than theorem proving specialized models. The best judge model (Con-J-Qwen2-7B at 52.4%) outperforms the best specialized model (DeepSeek-Prover-V2-7B at 12.8%) by 39.6 points—a factor of 4x difference.

This finding is counterintuitive. One might expect that models trained specifically on formal proof generation would better understand proof correctness. However, the opposite holds—general preference learning capabilities transfer more effectively to formal proof evaluation than domain-specific proof generation training.

At first glance, this result may appear to conflict with Chapter 4. However, the settings differ in important ways. Chapter 4 studies *process-level* reward modeling, where models evaluate individual proof steps with access to execution feedback, and theorem-proving specialists perform best. In contrast, FormalRewardBench evaluates *outcome-level* preferences over complete proofs, often without step-by-step traces, and includes judge models explicitly trained for comparative evaluation. These judge models bring strong general evaluation priors that transfer better to global correctness discrimination. The contrast highlights how supervision level, evaluation granularity, and model role (generator vs. judge) critically shape reward model performance.

The gap between pointwise and pairwise scores reveals position bias effects. For example, Selene-1-Llama-3.3-70B achieves 46.8% pointwise but 44.4% pairwise. RISE-Judge drops from 17.2% pointwise to 32.0% pairwise—actually improving with the consistency requirement, suggesting the model has inverted position bias preferring the second position.

The consistency requirement is strict—models must prefer the correct proof regardless of ordering. This substantially reduces accuracy for most models, showing that many judgments depend on position rather than content quality.

Analysis

Why do most models perform so poorly? Several factors contribute:

Many errors in FormalRewardBench are subtle—wrong variables, type mismatches, incorrect lemma applications. Detecting these requires carefully tracking scopes, types, and available hypotheses. Surface-level pattern matching is insufficient.

Our error injection strategies create proofs that look plausible. They have correct syntax, reasonable structure, and sometimes even correct reasoning for most steps. Only careful verification reveals the errors. Models must go beyond "does this look right" to "is this actually correct."

5.6.3 Specialized vs Judge Models

To understand the surprising performance gap, we examine differences between model categories in detail.

Aggregated Comparison

Table 5.2 shows aggregated results by category. Judge LLMs average 34.6% pointwise accuracy compared to only 6.5% for specialized models—a 28.1 point gap.

Table 5.2: Comparison between theorem proving specialized models and judge LLMs on FormalRewardBench.

Model Category	**Pointwise (%)**	**Pairwise (%)**
Judge LLMs (Average)	34.6	32.4
General-Purpose (Average)	27.6	21.9
Specialized (Average)	6.5	6.7
Best Judge	52.4	44.4
Best General	39.6	37.6
Best Specialized	12.8	23.6

Even the best specialized model (DeepSeek-Prover-V2 at 12.8%) performs dra-

matically worse than the best judge model (Con-J-Qwen2 at 52.4%)—a 39.6 point gap. General-purpose models fall in between at 27.6% average, suggesting that broad capabilities help even without judge-specific training.

Analysis: Why Do Judge Models Excel?

Several factors explain the performance gap:

Judge models are trained on diverse preference data across many domains—chat, code, math, reasoning. This training develops general capabilities for comparing responses and identifying quality differences. These skills transfer to formal proof evaluation even though formal proofs weren't in the training data.

Specialized models are optimized for proof generation—producing tactics that advance proofs. This is a different skill than evaluating whether given proofs are correct. Generation training focuses on positive examples and common patterns. Evaluation requires recognizing errors, including uncommon mistakes.

Judge models see many types of errors during training—wrong answers, logical flaws, incorrect reasoning, formatting issues. This exposure helps them recognize problems even in unfamiliar domains. Specialized models primarily train on correct proofs, limiting error recognition.

Judge models develop meta-reasoning about response quality—what makes one response better than another, how to articulate differences, what aspects matter for evaluation. These meta-skills apply across domains including formal mathematics.

5.6.4 Performance by Error Type

Different error injection strategies pose varying difficulty levels. Table 5.3 breaks down pairwise accuracy across the five buckets.

Bucket Difficulty Analysis

Bucket 5: Python Answer (Easiest) Python injection errors achieve 56.1% average accuracy—the only bucket above random baseline. Several models achieve

Table 5.3: Pairwise accuracy breakdown by error injection strategy. B1: Complex Lean Wrong, B2: Forced Mistake, B3: Minimal Difference, B4: NL Justification, B5: Python Answer.

Model	**B1**	**B2**	**B3**	**B4**	**B5**
Judge LLMs					
Selene-1-Llama-3.3-70B	20.0	18.0	44.0	44.0	96.0
LMUnit-qwen2.5-72b	22.0	12.0	32.0	24.0	94.0
CompassJudger-1-14B	2.0	12.0	28.0	34.0	100.0
CompassJudger-1-7B	2.0	8.0	20.0	30.0	94.0
RISE-Judge-Qwen2.5-7B	0.0	8.0	16.0	40.0	96.0
Skywork-Critic-70B	10.0	6.0	22.0	10.0	80.0
Skywork-Critic-8B	6.0	8.0	24.0	18.0	54.0
General-Purpose					
Qwen2.5-Coder-32B	4.0	14.0	32.0	44.0	94.0
DeepSeek-Coder-V2-Lite	12.0	22.0	22.0	12.0	72.0
Specialized					
DeepSeek-Prover-V2-7B	10.0	20.0	16.0	24.0	48.0
DeepSeek-Prover-V1.5-RL	10.0	12.0	2.0	6.0	16.0
DeepSeek-Prover-V1.5-SFT	4.0	4.0	4.0	0.0	20.0
Average Accuracy	8.4	11.8	18.1	19.8	56.1

near-perfect performance: CompassJudger-1-14B at 100%, Selene-1 at 96%, RISE-Judge at 96%.

This bucket is easier because the error is categorical—Python code instead of Lean tactics. Models trained on code recognize this mismatch. The task violation is obvious even without understanding the mathematical content. Models can apply the simple rule "Python is not a valid Lean proof."

However, even on this easiest bucket, specialized models struggle. DeepSeek-Prover-V2 achieves only 48.0%, and earlier versions perform much worse. This suggests specialized models have difficulty with task adherence when faced with code they might otherwise consider correct.

Bucket 1: Complex Wrong (Hardest) Complex incorrect proofs achieve only 8.4% average accuracy—dramatically below random. Most models score near 0%, showing they consistently prefer the wrong proof.

Long sophisticated-looking proofs deceive models. Multiple steps, intermediate lemmas, and detailed argumentation signal "this is a serious proof attempt." Models mistake complexity for correctness. The actual logical flaw hidden in the chain is difficult to detect.

This finding reveals a critical weakness: models are susceptible to surface-level sophistication. They need better reasoning about logical soundness rather than proof appearance.

Bucket 2: Forced Mistakes (Very Hard) Forced mistakes achieve 11.8% average accuracy. These errors reflect common language model mistakes—wrong hypotheses, incorrect lemma applications. Models struggle to detect errors they themselves might make.

This suggests limited self-awareness. Models that generate these errors during proof search cannot reliably recognize them during evaluation. Improving performance here requires either better understanding of formal proof correctness or meta-reasoning about common failure modes.

Bucket 3: Minimal Differences (Hard) Single-token variations achieve 18.1% average accuracy. Detecting these requires careful attention to variable scopes, type constraints, and hypothesis availability.

The best models show some capability—Selene-1 at 44.0%, Qwen2.5-Coder at 32.0%—but most perform poorly. Specialized models particularly struggle, with DeepSeek-Prover-V2 at only 16.0%. This suggests that fine-grained tracking of formal details is difficult even for domain-specific models.

Bucket 4: NL Justification (Hard) Natural language justification errors achieve 19.8% average accuracy. Some models are misled by plausible explanations—RISE-Judge at 40.0% performs relatively well, but others like Skywork-Critic-70B drop to 10.0%.

This tests whether models rely on formal content versus informal explanations. Good performance requires ignoring convincing text and checking formal correctness independently. The mixed results suggest many models are swayed by plausible justifications even when the formal proof is wrong.

Analysis by Model Category

Judge models excel at Python detection (B5) with most above 90%, show moderate performance on minimal differences (B3) and NL justification (B4) with 20-40% for the best models, and struggle with complex wrong (B1) and forced mistakes (B2) typically below 20%.

This pattern suggests judge models have task adherence capabilities (detecting Python) and some fine-grained discrimination ability (minimal differences), but struggle with sophisticated deception (complex proofs) and domain-specific errors (forced mistakes).

Specialized models perform poorly on all buckets, even relatively easy ones. DeepSeek-Prover-V2 achieves 48.0% on Python (B5)—barely above random despite this being the easiest bucket. On all other buckets, performance is 10-24%, and earlier versions perform even worse.

This uniform failure suggests specialized models lack the evaluation capabilities

needed for reward modeling. Their training on proof generation doesn't transfer to proof evaluation, even on error types they should recognize.

General-purpose models show mixed results. Qwen2.5-Coder performs well on Python (94%) and decently on minimal differences (32%) and NL justification (44%), but poorly on complex wrong (4%) and forced mistakes (14%). DeepSeek-Coder shows different strengths—better on forced mistakes (22%) but worse on NL justification (12%).

This inconsistency suggests general models have some relevant capabilities but lack systematic proof evaluation skills.

5.6.5 Commercial Model Performance

We also evaluate Claude Sonnet 4 on FormalRewardBench through the API. Results provide a point of comparison with frontier commercial models.

Claude Sonnet 4 achieves 32.4% pairwise accuracy overall—above specialized models but below the best judge models. Performance across buckets shows: Complex Wrong (32.0%), Forced Mistakes (40.0%), Minimal Differences (28.0%), NL Justification (30.0%), Python Answer (32.0%).

Interestingly, Claude shows relatively balanced performance across buckets, unlike other models that excel at Python detection while failing elsewhere. The 40% on forced mistakes is notably strong compared to most models' performance on this difficult bucket.

However, even frontier models struggle with formal proof evaluation. Claude's 32.4% overall is barely above random on many buckets. This reinforces that formal proof understanding remains challenging even for the most capable language models.

5.6.6 Error Analysis

We manually examine cases where models fail to understand patterns in mistakes.

Common Failure Modes

Preferring Longer Proofs Many models show bias toward longer proofs regardless of correctness. In Bucket 1 (complex wrong), models consistently prefer the long incorrect proof over the short correct one. They seem to equate effort with quality.

Following Natural Language In Bucket 4, models often prefer incorrect proofs with convincing explanations over correct proofs without commentary. The natural language text overrides formal verification signals.

Accepting Familiar Patterns Models accept tactics that look common even when they don't apply. In Bucket 2, wrong lemma applications are overlooked when the lemma name is familiar and the context seems appropriate.

Missing Type Errors Bucket 3 minimal differences often involve type mismatches—using Natural where Integer is required. Models frequently miss these, suggesting weak understanding of type systems.

Task Confusion In Bucket 5, some specialized models prefer Python solutions, possibly because their training on code generation makes Python look "correct" even when it's the wrong language for the task.

5.6.7 Summary of Experimental Findings

FormalRewardBench evaluation reveals several key findings:

1. **Formal proof evaluation is extremely challenging.** Most models perform at or below the 50% random baseline, with even the best model (52.4%) only marginally exceeding random performance.

2. **Judge models dramatically outperform specialists.** Judge LLMs achieve 34.6% average accuracy compared to 6.5% for specialized theorem provers—a 4x performance gap. The best judge model (52.4%) outperforms the best specialized model (12.8%) by 39.6 points.

3. **Error type dramatically affects difficulty.** Python injection errors are relatively easy (56.1% average), while complex wrong proofs (8.4%) and forced mistakes (11.8%) are extremely hard. This reveals that categorical errors are more detectable than subtle logical flaws.

4. **Position bias is substantial.** The gap between pointwise and pairwise scores shows many judgments depend on position rather than content. Consistency requirements substantially reduce accuracy for most models.

5. **Sophistication deceives models.** Long complex-looking proofs with logical flaws consistently fool models. Surface-level sophistication is mistaken for correctness.

These findings establish FormalRewardBench as a challenging testbed for reward model development and reveal fundamental limitations in current approaches to formal proof evaluation. The benchmark enables systematic research on improving reward models for automated theorem proving.

5.7 Summary

We presented FormalRewardBench, the first benchmark for evaluating reward models in formal theorem proving. Key contributions:

- Five error injection strategies capturing realistic failure modes
- Systematic construction pipeline with quality control
- Evaluation protocols for both pointwise and pairwise models
- Position bias mitigation for reliable pairwise evaluation
- Analysis framework examining model performance by error type

The benchmark provides a testbed for developing reward models in formal mathematics. By creating challenging but fair preference pairs, we enable systematic

evaluation of proof understanding. This addresses a critical gap in preference-based learning for automated theorem proving.

Chapter 6

CONCLUSION

Automated theorem proving represents a fundamental challenge in artificial intelligence: constructing logically sound arguments in formally verifiable systems. The challenge is not merely generating plausible steps, but systematically exploring exponentially large proof spaces while maintaining correctness at every stage. This thesis investigated three interconnected aspects of this challenge—how to build infrastructure that enables systematic experimentation, how to provide guidance signals that are both effective and interpretable, and how to evaluate whether systems truly understand proof quality.

6.1 Problem Reframing

The central difficulty in automated theorem proving is not the generation of individual proof tactics—language models have demonstrated competence at producing plausible steps. Rather, the challenge lies in guidance: deciding which of exponentially many possible proof paths to explore. Existing approaches to guidance face a fundamental tension. Simple heuristics are fast but unreliable, conflating generation likelihood with step quality. Binary feedback from proof assistants is perfectly accurate but provides no notion of progress or strategic direction. Discriminative reward models offer dense signals but compress rich proof context into opaque scalar values.

This guidance problem is compounded by the absence of systematic ways to measure progress. Without evaluation frameworks, we cannot determine whether reward models understand proof correctness or merely recognize superficial patterns. Without shared infrastructure, comparing approaches becomes difficult and advances in one component rarely transfer to others. These structural challenges

constrain progress as much as algorithmic limitations.

6.2 Lessons Learned

Our work yields several insights about guidance, evaluation, and infrastructure for automated theorem proving.

6.2.1 Guidance as Epistemic Tool

Reward models serve not only as scoring mechanisms, but also as epistemic tools that influence how search algorithms interpret proof spaces. When guidance signals are sparse or unclear, search algorithms cannot distinguish between promising directions and dead ends, or between proofs that almost worked and those that were fundamentally flawed. More informative signals — those that explain correctness, assess progress and evaluate strategy — enable more informed exploration.

The effectiveness of natural language critiques suggests that the reasoning capabilities that language models develop during pretraining can be used for evaluation purposes. Instead of learning to map proof states directly to numbers, models that explicitly articulate reasoning may access broader knowledge and produce more reliable assessments. This finding challenges the assumption that evaluation requires maximal compression of information. In fact, making reasoning explicit can sometimes improve performance rather than hinder it.

6.2.2 Grounding in Execution Feedback

The critical importance of conditioning on execution feedback highlights a deeper principle: prediction is more challenging than assessment. When reward models attempt to predict the success of tactics, they must reason about proof assistant semantics, type systems and scope rules. However, when they assess what actually happened after execution, they work with concrete facts, such as whether tactics succeeded, what errors occurred and how proof states changed.

This grounding in concrete outcomes rather than predictions substantially improves the accuracy of the evaluation. This suggests that integrating symbolic veri-

fication with learned models — letting proof assistants provide ground truth about correctness while models provide strategic assessment — may be more effective than attempting to learn verification itself.

6.2.3 Evaluation Shapes Research Direction

The difficulty of formal proof evaluation on systematic benchmarks reveals gaps in current understanding. Categorical errors—violations of task requirements—are relatively easy to detect. Subtle logical errors—tactics that execute successfully but lead to dead ends, proofs with sophisticated structure but flawed reasoning—remain extremely challenging. This difficulty is not merely a measurement problem but indicates fundamental limitations in how models understand formal correctness.

The counterintuitive finding that general preference learning capabilities transfer better to formal proof evaluation than domain-specific proof generation training challenges assumptions about specialization. It suggests that evaluation and generation may be distinct skills requiring different types of knowledge. Systems optimized for producing proofs are not automatically good at assessing them. This separation has implications for how we train and deploy models in formal domains.

6.3 Limitations and Future Directions

Several open questions suggest directions for continued research.

6.3.1 Scale and Generalization

Our experiments operate at specific scales—models up to tens of billions of parameters, search budgets of thousands of expansions, benchmarks of hundreds of problems. Whether findings hold at dramatically different scales remains unclear. Similarly, our evaluation focuses primarily on competition mathematics formalized in one proof assistant. Understanding the boundaries of generalization would clarify which insights are fundamental versus which are artifacts of particular domains or systems.

6.3.2 From Automation to Assistance

Rather than attempting to automate proving entirely, guidance systems could support interactive formalization. As mathematicians write proofs, systems could provide real-time feedback, suggest alternatives, and explain why certain approaches might succeed or fail. This paradigm shift from automation to assistance might make formal mathematics more accessible while leveraging human mathematical insight.

6.3.3 Cross-System and Theoretical Understanding

Important open questions remain about generalization across proof assistants and theoretical foundations of guidance mechanisms. Do principles of formal proof evaluation transfer across different systems? What properties must guidance signals have to enable efficient search? How does proof space structure affect optimal exploration strategies? Understanding these questions would move the field beyond empirical trial toward principled design.

6.4 Broader Implications

Advances in automated theorem proving extend beyond mathematics. Formal verification increasingly underpins critical systems—compilers, operating systems, cryptographic implementations, hardware designs. Making verification more accessible could improve reliability across industries and help students develop mathematical reasoning through interaction with systems that explain their decisions.

However, the limitations revealed by this work emphasize continued need for human oversight. As proving systems improve, maintaining appropriate skepticism about their outputs remains essential. The accessibility question also matters—open infrastructure and public benchmarks help ensure that advances benefit broad participation rather than only well-resourced institutions.

6.5 Concluding Thoughts

This thesis demonstrates that guidance can be richer than scalars, that grounding in execution improves evaluation, and that systematic benchmarks reveal fundamental limitations in current approaches. These insights shift how we should think about the problem. Effective automated theorem proving requires not just more powerful models but infrastructure that enables systematic experimentation, guidance mechanisms that balance effectiveness with interpretability, and evaluation frameworks that distinguish genuine understanding from superficial pattern recognition.

The gap between current capabilities and human-level formal mathematics remains substantial, but the path forward is clearer. Progress requires sustained advances across multiple fronts—better search algorithms, richer guidance signals, improved training methods, deeper theoretical understanding. The foundations established here aim to facilitate this progress by providing infrastructure for experimentation, frameworks for evaluation, and insights about what works and why.

The vision is systems that make formal mathematics more accessible, verification more routine, and mathematical reasoning more transparent. Realizing this vision requires both technical innovation and careful attention to how automated systems can effectively augment human capabilities. This work takes steps toward reshaping how we approach automated theorem proving—from viewing it as a generation problem to recognizing it as a problem of guidance, evaluation, and systematic understanding.

BIBLIOGRAPHY

[Azerbayev et al., 2023] Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. (2023). Proofnet: Autoformalizing and formally proving undergraduate-level mathematics.

[Bai et al., 2022] Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C., Chen, C., Olsson, C., Olah, C., Hernandez, D., Drain, D., Ganguli, D., Li, D., Tran-Johnson, E., and ..., E. P. (2022). Constitutional ai: Harmlessness from ai feedback. *ArXiv*, abs/2212.08073.

[BRADLEY and TERRY, 1952] BRADLEY, R. A. and TERRY, M. E. (1952). Rank analysis of incomplete block designs: The method of paired comparisons. *Biometrika*, 39(3-4):324–345.

[de Moura and Ullrich, 2021] de Moura, L. and Ullrich, S. (2021). The lean 4 theorem prover and programming language. In Platzer, A. and Sutcliffe, G., editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer.

[First et al., 2023] First, E., Rabe, M. N., Ringer, T., and Brun, Y. (2023). Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1229–1241, New York, NY, USA. Association for Computing Machinery.

[Gauthier and Brown, 2024] Gauthier, T. and Brown, C. E. (2024). A Formal Proof of R(4,5)=25. In Bertot, Y., Kutsia, T., and Norrish, M., editors, *15th International Conference on Interactive Theorem Proving (ITP 2024)*, volume 309

of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:18, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Huang et al., 2024] Huang, L., Ebersold, S., Kogtenkov, A., Meyer, B., and Liu, Y. (2024). Lessons from formally verified deployed software systems (extended version).

[Huang et al., 2025] Huang, S., Feng, S., He, X., Yan, X., Zhang, B., and BAI, L. (2025). Rewardcode: Training generalist code reward model via pairwise reinforcement learning.

[Jiang et al., 2024] Jiang, J., Chen, Z., Min, Y., Chen, J., Cheng, X., Wang, J., Tang, Y., Sun, H., Deng, J., Zhao, W. X., Liu, Z., Yan, D., Xie, J., Wang, Z., and Wen, J.-R. (2024). Enhancing llm reasoning with reward-guided tree search.

[Kwon et al., 2023] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. (2023). Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[Lambert et al., 2025] Lambert, N., Pyatkin, V., Morrison, J., Miranda, L., Lin, B. Y., Chandu, K., Dziri, N., Kumar, S., Zick, T., Choi, Y., Smith, N. A., and Hajishirzi, H. (2025). RewardBench: Evaluating reward models for language modeling. In Chiruzzo, L., Ritter, A., and Wang, L., editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1755–1797, Albuquerque, New Mexico. Association for Computational Linguistics.

[Lample et al., 2022] Lample, G., Lacroix, T., Lachaux, M., Rodriguez, A., Hayat, A., Lavril, T., Ebner, G., and Martinet, X. (2022). Hypertree proof search for neural theorem proving. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A., editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

[Li et al., 2025] Li, Y., Du, D., Song, L., Li, C., Wang, W., Yang, T., and Mi, H. (2025). Hunyuanprover: A scalable data synthesis framework and guided tree search for automated theorem proving.

[Lightman et al., 2024] Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., and Cobbe, K. (2024). Let's verify step by step. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

[Liu et al., 2025] Liu, Z., Wang, P., Xu, R., Ma, S., Ruan, C., Li, P., Liu, Y., and Wu, Y. (2025). Inference-time scaling for generalist reward modeling.

[Mahan et al., 2024] Mahan, D., Phung, D. V., Rafailov, R., Blagden, C., Lile, N., Castricato, L., Fränken, J.-P., Finn, C., and Albalak, A. (2024). Generative reward models.

[Nipkow et al., 2002] Nipkow, T., Wenzel, M., and Paulson, L. C. (2002). *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg.

[Norell, 2008] Norell, U. (2008). Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, page 230–266, Berlin, Heidelberg. Springer-Verlag.

[Ouyang et al., 2022] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., and Lowe, R. (2022). Training language models to follow instructions with human feedback. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A., editors, *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc.

[Panickssery et al., 2024] Panickssery, A., Bowman, S. R., and Feng, S. (2024). Llm evaluators recognize and favor their own generations. In *Proceedings of the 38th*

International Conference on Neural Information Processing Systems, NIPS '24, Red Hook, NY, USA. Curran Associates Inc.

[Polu and Sutskever, 2020] Polu, S. and Sutskever, I. (2020). Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393.

[Rafailov et al., 2023] Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., and Finn, C. (2023). Direct preference optimization: Your language model is secretly a reward model. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S., editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

[Santos et al., 2025] Santos, M. D., Wang, H., de Saxcé, H., Wang, R., Baksys, M., Unsal, M., Liu, J., Liu, Z., and Li, J. (2025). Kimina lean server: Technical report.

[Schulman et al., 2016] Schulman, J., Moritz, P., Levine, S., Jordan, M. I., and Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. In Bengio, Y. and LeCun, Y., editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.

[Shao et al., 2024] Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. (2024). Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300.

[Silver et al., 2017] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv*, abs/1712.01815.

[Snell et al., 2024] Snell, C., Lee, J., Xu, K., and Kumar, A. (2024). Scaling llm test-time compute optimally can be more effective than scaling model parameters.

[Song et al., 2025] Song, M., Su, Z., Qu, X., Zhou, J., and Cheng, Y. (2025). Prm-bench: A fine-grained and challenging benchmark for process-level reward models. In Che, W., Nabende, J., Shutova, E., and Pilehvar, M. T., editors, *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*, pages 25299–25346. Association for Computational Linguistics.

[Sørensen and Urzyczyn, 2006] Sørensen, M. H. and Urzyczyn, P. (2006). *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., USA.

[Sutton et al., 1999] Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. In Solla, S., Leen, T., and Müller, K., editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press.

[The Coq Development Team, 2024] The Coq Development Team (2024). The Coq reference manual – release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman.

[Tsoukalas et al., 2024] Tsoukalas, G., Lee, J., Jennings, J., Xin, J., Ding, M., Jennings, M., Thakur, A., and Chaudhuri, S. (2024). Putnambench: evaluating neural theorem-provers on the putnam mathematical competition. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA. Curran Associates Inc.

[Uesato et al., 2022] Uesato, J., Kushman, N., Kumar, R., Song, H. F., Siegel, N. Y., Wang, L., Creswell, A., Irving, G., and Higgins, I. (2022). Solving math word problems with process- and outcome-based feedback. *CoRR*, abs/2211.14275.

[Wang et al., 2023] Wang, H., Yuan, Y., Liu, Z., Shen, J., Yin, Y., Xiong, J., Xie, E., Shi, H., Li, Y., Li, L., Yin, J., Li, Z., and Liang, X. (2023). DT-solver: Auto-

mated theorem proving with dynamic-tree sampling guided by proof-level value function. In Rogers, A., Boyd-Graber, J., and Okazaki, N., editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12632–12646, Toronto, Canada. Association for Computational Linguistics.

[Wei et al., 2022] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A., editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

[Wu et al., 2021] Wu, M., Norrish, M., Walder, C., and Dezfouli, A. (2021). Tacticzero: learning to prove theorems from scratch with deep reinforcement learning. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA. Curran Associates Inc.

[Wu et al., 2025] Wu, Z., Huang, S., Zhou, Z., Ying, H., Yuan, Z., Zhang, W., Lin, D., and Chen, K. (2025). Internlm2.5-stepprover: Advancing automated theorem proving via critic-guided search.

[Xin et al., 2024] Xin, H., Ren, Z. Z., Song, J., Shao, Z., Zhao, W., Wang, H., Liu, B., Zhang, L., Lu, X., Du, Q., Gao, W., Zhu, Q., Yang, D., Gou, Z., Wu, Z. F., Luo, F., and Ruan, C. (2024). Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *CoRR*, abs/2408.08152.

[Xin et al., 2025] Xin, R., Xi, C., Yang, J., Chen, F., Wu, H., Xiao, X., Sun, Y., Zheng, S., and Ding, M. (2025). BFS-prover: Scalable best-first tree search for LLM-based automatic theorem proving. In Che, W., Nabende, J., Shutova, E.,

and Pilehvar, M. T., editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32588–32599, Vienna, Austria. Association for Computational Linguistics.

[Yang et al., 2023] Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. (2023). Leandojo: theorem proving with retrieval-augmented language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

[Ye et al., 2024] Ye, F., Yang, M., Pang, J., Wang, L., Wong, D. F., Yilmaz, E., Shi, S., and Tu, Z. (2024). Benchmarking llms via uncertainty quantification. In Globerson, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J., and Zhang, C., editors, *Advances in Neural Information Processing Systems*, volume 37, pages 15356–15385. Curran Associates, Inc.

[Yu et al., 2025] Yu, Z., Gu, W., Wang, Y., Jiang, X., Zeng, Z., Wang, J., Ye, W., and Zhang, S. (2025). Reasoning through execution: Unifying process and outcome rewards for code generation.

[Zhao et al., 2025] Zhao, J., Liu, R., Zhang, K., Zhou, Z., Gao, J., Li, D., Lyu, J., Qian, Z., Qi, B., Li, X., and Zhou, B. (2025). Genprm: Scaling test-time compute of process reward models via generative reasoning.

[Zheng et al., 2025] Zheng, C., Zhang, Z., Zhang, B., Lin, R., Lu, K., Yu, B., Liu, D., Zhou, J., and Lin, J. (2025). ProcessBench: Identifying process errors in mathematical reasoning. In Che, W., Nabende, J., Shutova, E., and Pilehvar, M. T., editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1009–1024, Vienna, Austria. Association for Computational Linguistics.

[Zheng et al., 2021] Zheng, K., Han, J. M., and Polu, S. (2021). Minif2f:

a cross-system benchmark for formal olympiad-level mathematics. *CoRR*, abs/2109.00110.

[Zheng et al., 2023] Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., and Stoica, I. (2023). Judging llm-as-a-judge with mt-bench and chatbot arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Appendix A

REINFORCEMENT LEARNING METHODS: FROM BASICS TO GRPO

This appendix provides background on reinforcement learning methods used for training generative reward models. We start with fundamental concepts, then explain policy gradient algorithms, and finally detail why we chose GRPO for our training procedure.

A.1 Reinforcement Learning Basics

A.1.1 Markov Decision Processes

Reinforcement learning formalizes sequential decision-making through Markov Decision Processes (MDPs). An MDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ where:

- $\mathcal{S}$ is the state space
- $\mathcal{A}$ is the action space
- $P(s' \mid s, a)$ is the transition probability
- $R(s, a)$ is the reward function
- $\gamma \in [0, 1)$ is the discount factor

In our setting, states correspond to partial proof trajectories, actions correspond to generated critiques, and rewards measure alignment with proof outcomes.

A.1.2 Policies and Value Functions

A policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ maps states to probability distributions over actions. In language model fine-tuning, the policy is parameterized by neural network weights θ, written π_θ.

The goal is to find a policy that maximizes expected cumulative reward. We define:

Return The discounted sum of rewards along a trajectory:

$$R(\tau) = \sum_{t=0}^{T-1} \gamma^t r_t \tag{A.1}$$

where $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$ is a trajectory.

State Value Function The expected return from state s:

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau) \mid s_0 = s] \tag{A.2}$$

Action Value Function The expected return from taking action a in state s:

$$Q^\pi(s, a) = \mathbb{E}_{\tau\sim\pi}[R(\tau) \mid s_0 = s, a_0 = a] \tag{A.3}$$

Advantage Function The relative value of taking action a versus the average action:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s) \tag{A.4}$$

The advantage function is central to variance reduction in policy gradient methods. It tells us how much better (or worse) an action is compared to what we would expect on average.

A.1.3 The Optimization Problem

The objective is to find parameters θ that maximize expected return:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{A.5}$$

Two main approaches exist: value-based methods (Q-learning, DQN) that learn value functions and derive policies from them, and policy-based methods that directly optimize the policy. We focus on policy gradient methods because they work naturally with continuous action spaces and stochastic policies, both important for language model fine-tuning.

A.2 Policy Gradient Theorem

A.2.1 Core Result

The policy gradient theorem [Sutton et al., 1999] provides the foundation for policy optimization. It states that the gradient of the expected return with respect to policy parameters is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \, R(\tau) \right] \tag{A.6}$$

This remarkable result says we can estimate the gradient by sampling trajectories and weighting the log-probability gradients by the trajectory return. Intuitively, we push up the probability of actions in high-reward trajectories and push down the probability of actions in low-reward trajectories.

A.2.2 Why This Works

The key insight is the log-derivative trick:

$$\nabla_\theta \pi_\theta(a \mid s) = \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s) \tag{A.7}$$

This allows us to convert expectations over policy probabilities into expectations

we can sample:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{A.8}$$

$$= \nabla_\theta \int P(\tau \mid \theta) R(\tau)\, d\tau \tag{A.9}$$

$$= \int \nabla_\theta P(\tau \mid \theta) R(\tau)\, d\tau \tag{A.10}$$

$$= \int P(\tau \mid \theta) \nabla_\theta \log P(\tau \mid \theta) R(\tau)\, d\tau \tag{A.11}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau \mid \theta) R(\tau)] \tag{A.12}$$

Since trajectory probability factors as $P(\tau \mid \theta) = \prod_{t=0}^{T-1} \pi_\theta(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t)$ and transition probabilities don't depend on θ, we get the policy gradient formula above.

A.3 Variance Reduction Techniques

A.3.1 Baseline Subtraction

The first improvement is subtracting a baseline $b(s_t)$ from returns:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t)(R(\tau) - b(s_t))\right] \tag{A.13}$$

This is unbiased because $\mathbb{E}[b(s_t)] = b(s_t)$ is independent of the action. The baseline reduces variance without introducing bias. A good baseline is the value function $V^\pi(s_t)$—this measures the expected return from state s_t, so subtracting it gives a centered measure of how much better the actual trajectory was.

A.3.2 Advantage Functions

Instead of using full trajectory returns, we can use the advantage function. This leads to the gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) A^\pi(s_t, a_t)\right] \tag{A.14}$$

The advantage $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$ has much lower variance than returns because it measures the relative quality of the action, not the absolute

return. An action can be good (positive advantage) even if the overall return is low, and vice versa.

A.3.3 Generalized Advantage Estimation

Computing exact advantages requires knowing Q^π and V^π. In practice, we estimate advantages from sampled returns. Generalized Advantage Estimation (GAE) [Schulman et al., 2016] provides a family of estimators that trade off bias and variance:

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{A.15}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference error. The parameter $\lambda \in [0, 1]$ controls the trade-off. When $\lambda = 0$, we get the one-step TD estimate (low variance, high bias). When $\lambda = 1$, we get the Monte Carlo estimate (high variance, low bias).

A.4 Actor-Critic Methods

Actor-critic methods maintain two networks:

- **Actor** π_θ: The policy network that selects actions
- **Critic** V_ϕ: The value network that estimates state values

The critic enables advantage estimation without full trajectory rollouts. At each step, we compute:

$$\hat{A}_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{A.16}$$

The actor is updated using policy gradients with these advantage estimates. The critic is updated to minimize the value prediction error:

$$\mathcal{L}_{\text{critic}} = \mathbb{E}[(V_\phi(s_t) - R_t)^2] \tag{A.17}$$

where R_t is the actual observed return from state s_t.

Memory Considerations For large language models, maintaining two networks doubles memory requirements. Each network copy requires storing all parameters and optimizer states. For a 7B parameter model with AdamW, this means roughly 28GB per network (4 bytes per parameter, plus 8 bytes per parameter for optimizer states). The full actor-critic setup thus requires at least 56GB, often exceeding single GPU capacity.

A.5 Proximal Policy Optimization

A.5.1 Motivation

Policy gradient methods are sensitive to step size. Too small and learning is slow. Too large and the policy changes drastically, potentially collapsing to poor behavior that's hard to recover from.

PPO [Schulman et al., 2017] addresses this through a clipped objective that constrains policy updates. The key insight is to limit how much the new policy π_θ can deviate from the old policy $\pi_{\theta_{\text{old}}}$.

A.5.2 The PPO Objective

PPO defines a probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{A.18}$$

The clipped objective is:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right] \tag{A.19}$$

where ϵ is typically 0.2. The clipping operation ensures:

- If $A_t > 0$ (good action), we increase $\pi_\theta(a_t \mid s_t)$ but not beyond ratio $1+\epsilon$
- If $A_t < 0$ (bad action), we decrease $\pi_\theta(a_t \mid s_t)$ but not below ratio $1-\epsilon$

This prevents destructively large policy updates while still allowing meaningful improvement.

A.5.3 Full PPO Algorithm

The complete PPO algorithm for language model fine-tuning:

Algorithm 4 PPO for LLM Fine-tuning

Initialize policy π_θ, value network V_ϕ, reference policy π_{ref}
for iteration $= 1, 2, \ldots$ **do**
  Collect rollouts $\mathcal{D} = \{(x_i, y_i, r_i)\}$ using π_θ
  Compute advantages $\hat{A}_i$ using V_ϕ and rewards r_i
  for epoch $= 1, \ldots, K$ **do**
    for minibatch $\mathcal{B} \subset \mathcal{D}$ **do**
      Compute policy loss: L^{CLIP} with advantages $\hat{A}$
      Compute value loss: $L^{\text{value}} = \frac{1}{2}\sum(V_\phi(s) - R)^2$
      Compute KL penalty: $L^{\text{KL}} = \beta\mathbb{E}[D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})]$
      Total loss: $L = L^{\text{CLIP}} + c_1 L^{\text{value}} - c_2 L^{\text{KL}}$
      Update θ and ϕ using gradient descent
    end for
  end for
  Update reference policy: $\pi_{\text{ref}} \leftarrow \pi_\theta$
end for

A.5.4 Memory Requirements

PPO for LLMs requires four model copies in memory:

1. **Policy** π_θ: The model being trained
2. **Reference** π_{ref}: Fixed copy for computing probability ratios
3. **Value** V_ϕ: The critic network
4. **Reward** R: Often a separate reward model (not always needed if using rule-based rewards)

For a 7B parameter model, this totals over 100GB of GPU memory with optimizer states. This requires multiple high-memory GPUs and complex distributed training setups.

A.6 Group Relative Policy Optimization

A.6.1 Core Idea

GRPO [Shao et al., 2024] further simplifies the training pipeline while maintaining competitive performance. The key insight is to normalize rewards relative to the current batch rather than maintaining baselines across iterations.

For a batch $\mathcal{B}$ of (x, y, r) tuples, GRPO computes:

$$\tilde{r}_i = \frac{r_i - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}} + \epsilon} \tag{A.20}$$

where $\mu_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_i r_i$ and $\sigma_{\mathcal{B}} = \sqrt{\frac{1}{|\mathcal{B}|} \sum_i (r_i - \mu_{\mathcal{B}})^2}$ are the batch mean and standard deviation.

A.6.2 The GRPO Objective

The full GRPO objective combines normalized rewards with KL regularization:

$$\mathcal{L}_{\text{GRPO}}(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{B}} \left[\tilde{r}(x, y) \log \pi_\theta(y \mid x) - \beta D_{\text{KL}}(\pi_\theta(y \mid x) \| \pi_{\text{ref}}(y \mid x)) \right] \tag{A.21}$$

The KL term prevents the policy from diverging too far from the reference policy, similar to PPO's trust region. The coefficient β controls the strength of regularization.

A.6.3 Why Batch Normalization Works

Group-relative normalization provides several benefits:

Different prompts may naturally produce different reward scales. A math problem might yield rewards in [0, 100] while a coding problem yields rewards in [0, 1]. Batch normalization makes the training invariant to these scales—what matters is relative quality within each batch.

The batch mean $\mu_{\mathcal{B}}$ acts as a baseline. Completions with above-average rewards get positive normalized scores, those with below-average rewards get negative scores. This baseline is always well-calibrated because it's computed from the current policy's samples.

Normalization by standard deviation $\sigma_{\mathcal{B}}$ prevents gradient explosion from outlier rewards. Without normalization, a single very high or very low reward could dominate the batch gradient. Standardization ensures all examples contribute meaningfully.

A.6.4 Implementation Details

The GRPO update loop is straightforward:

Algorithm 5 GRPO Training Loop

Initialize policy π_θ, reference policy $\pi_{\text{ref}} \leftarrow \pi_\theta$
for iteration $= 1, 2, \ldots$ **do**
  Sample batch of prompts $\{x_1, \ldots, x_B\}$
  Generate completions $\{y_1, \ldots, y_B\}$ using π_θ
  Compute rewards $\{r_1, \ldots, r_B\}$ using reward function
  Compute batch statistics: $\mu_{\mathcal{B}} = \text{mean}(\{r_i\})$, $\sigma_{\mathcal{B}} = \text{std}(\{r_i\})$
  Normalize rewards: $\tilde{r}_i = (r_i - \mu_{\mathcal{B}})/(\sigma_{\mathcal{B}} + \epsilon)$
  for epoch $= 1, \ldots, K$ **do**
    for minibatch $\mathcal{M} \subset \mathcal{B}$ **do**
      Compute policy loss: $L^{\text{policy}} = -\frac{1}{|\mathcal{M}|} \sum_{i \in \mathcal{M}} \tilde{r}_i \log \pi_\theta(y_i \mid x_i)$
      Compute KL penalty: $L^{\text{KL}} = \beta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})$
      Total loss: $L = L^{\text{policy}} + L^{\text{KL}}$
      Update θ using gradient descent on L
    end for
  end for
  if iteration mod $N_{\text{ref}} == 0$ **then**
    Update reference: $\pi_{\text{ref}} \leftarrow \pi_\theta$
  end if
end for

The reference policy is updated periodically (every N_{ref} iterations) rather than every iteration, which reduces computational overhead.

A.6.5 Why We Choose GRPO

For training generative reward models, GRPO offers the best trade-off between simplicity, memory efficiency, and performance:

1. Requires only two 7B models (50GB) instead of four (100GB), enabling training on fewer GPUs

2. Eliminates the need to train a separate critic, simplifying the training pipeline

3. Batch normalization provides reliable gradient estimates without manual hyperparameter tuning for baseline learning rates

4. The batch mean serves as a well-calibrated baseline that adapts automatically as the policy improves

Appendix B

PROMPT TEMPLATES

This appendix provides the complete prompt templates used for generative reward models.

B.1 Zero-Shot Evaluation Prompt (Without Environmental Feedback)

```
You are evaluating Lean 4 proof steps (tactics, hypotheses,
   let bindings, structural elements).

Rate 0-20 based on:
- Progress: Did we get closer to the goal?
- Simplification: Did complexity decrease?
- Utility: Will this step be useful later?
- Appropriateness: Is this the right approach?

Consider the step type:
- Tactics: Focus on goal transformation
- Hypotheses (have/suffices): Focus on usefulness
- Let bindings: Focus on abstraction value
- Structural: Focus on necessity

Output format:
Analysis: [1 sentence on impact]
Score: \boxed{NUMBER}

Always give the rating in \\boxed{RATING} format. Your task
   is only to rate the proof steps.
```

```
Here you can see the proof so far, the applied tactic, the
   open goals. Depends on these information, you should judge
    the quality of the proof step application.


\# Proof So Far:
\{proof\_so\_far\}

\# Open Goals:
\{current\_goals\}

\# Applied Proof Step:
\{applied\_proofstep\}

If goals are not provided, you should judge the quality of
   the proof step application based on the proof so far. Put
   your score in $\boxed{\{\}}$.
```

B.2 Zero-Shot Evaluation Prompt (With Environmental Feedback)

```
You are evaluating Lean 4 proof steps (tactics, hypotheses,
   let bindings, structural elements).

Rate 0-200 based on:
- Progress: Did we get closer to the goal?
- Simplification: Did complexity decrease?
- Utility: Will this step be useful later?
- Appropriateness: Is this the right approach?

Consider the step type:
- Tactics: Focus on goal transformation
- Hypotheses (have/suffices): Focus on usefulness
```

```
- Let bindings: Focus on abstraction value
- Structural: Focus on necessity

Output format:
Analysis: [1 sentence on impact]
Score: \boxed{NUMBER}

Always give the rating in \\boxed{RATING} format. Your task
   is only to rate the proof steps.

Here you can see the proof so far, the applied tactic, the
   open goals, errors if exist and solved goals. Depends on
   these information, you should judge the quality of the
   proof step application.


\# Proof So Far:
\{proof\_so\_far\}

\# Open Goals:
\{current\_goals\}

\# Applied Proof Step:
\{applied\_proofstep\}

(if error\_message\_from\_lean:)
\# Error Message from Lean:
\{error\_message\_from\_lean\}

\# Solved Goals:
\{solved\_goals\}
```

```
If goals are not provided, you should judge the quality of
   the proof step application based on the proof so far. Put
   your score in $\boxed{\{\}}$.
```

Appendix C

TRAINING HYPERPARAMETERS

This appendix details all hyperparameters used for training generative reward models.

C.1 Model Configuration

Parameter	Value
Base Model	DeepSeek-Prover-V2-7B
Vocabulary Size	100,000
Context Length	4096 tokens
Precision	bfloat16

Table C.1: Base model configuration

C.2 Cold Start (Rejective Fine-Tuning)

Training Data We use 100,000 critique examples constructed from proof trajectories. Examples include both successful and failed steps with manually verified labels.

Rejection Sampling During cold start, we generate multiple critique candidates per example and keep only those with valid format (ending with "Score: X/20" where X is 0-20).

Parameter	Value
Learning Rate	5e-6
Batch Size	32
Gradient Accumulation Steps	4
Effective Batch Size	128
Training Steps	5,000
Warmup Steps	500
Weight Decay	0.01
Max Gradient Norm	1.0
Optimizer	AdamW
β_1	0.9
β_2	0.95
ϵ	1e-8

Table C.2: Cold start hyperparameters

C.3 GRPO Training

Data Collection At each iteration, we run proof search on 50 randomly sampled problems from Lean-Workbook, generating 8 proof attempts per problem. This yields approximately 400 trajectories per iteration.

Trajectory Filtering We filter trajectories to balance successful and failed attempts—50

C.4 Learning Rate Scheduling

We use cosine annealing with warmup for both training stages:

$$\text{lr}(t) = \begin{cases} \text{lr}_{\max} \cdot \frac{t}{t_{\text{warmup}}} & t < t_{\text{warmup}} \\ \text{lr}_{\min} + \frac{1}{2}(\text{lr}_{\max} - \text{lr}_{\min})(1 + \cos(\frac{t - t_{\text{warmup}}}{t_{\max} - t_{\text{warmup}}}\pi)) & t \geq t_{\text{warmup}} \end{cases} \tag{C.1}$$

Parameter	Value
Learning Rate	1e-6
Batch Size	16 trajectories
Steps per Trajectory	8-15 (varies)
Effective Samples per Batch	128-240
Training Iterations	10,000
Warmup Iterations	1,000
α (proximity weight)	0.2
ϵ (normalization)	1e-5
Max Gradient Norm	1.0
Optimizer	AdamW
β_1	0.9
β_2	0.99

Table C.3: GRPO training hyperparameters

where $\text{lr}_{\text{min}} = 0.1 \cdot \text{lr}_{\text{max}}$ for cold start and $\text{lr}_{\text{min}} = 0$ for GRPO.

Appendix D

ERROR INJECTION PROMPTS

This appendix provides the prompts used to generate incorrect proofs for FormalRewardBench.

D.1 Bucket 1: Forced Mistakes

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

Your task is to complete Lean 4 theorems with proofs that
   contain realistic formal proof errors. Focus on mistakes
   that occur during formal proof generation:

Common LLM 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
9) Incorrect handling of definitional equality
10) Wrong unification or pattern matching

Rules:
- Use valid Lean 4 syntax
```

```
- Make errors that would occur during automated proof
   generation
- Focus on formal reasoning mistakes, not basic mathematical
   errors
- At first glance, the proof should appear plausible and
   aesthetically pleasing,
- Do not add any commentary or hint about what you are doing
   or what you changed. (Like the reason of incorrectness etc
   )
- Do not mention about the task you are doing.
Complete the following Lean 4 theorem with a INCORRECT proof
   that contains a typical formal proof generation error. Do
   not include any commentary.

```lean4
{}
```

Output only the completed proof (Lean 4), starting after `by
   `.
```

D.2 Bucket 2: Minimal Variations

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

Your 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,
   add_assoc     mul_assoc)
```

```
- 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 change
   ideally)
- The incorrect proof must be syntactically valid (parseable)
- The change should be subtle enough that it looks plausible
   at first glance
- Keep all other parts of the proof EXACTLY the same
- The error should be a logical/mathematical mistake, not a
   syntax error
- Do not add any commentary or hint about what you are doing
   or what you changed. (Like the reason of difference etc)
- Do not mention about the task you are doing.

Generate a minimally different INCORRECT version of this
   proof.

Original CORRECT proof:
```lean4
Theorem
{}
Proof:
{}
```

Return ONLY the modified proof code with ONE subtle change
   that makes it incorrect. No explanations, no markdown.
```

D.3 Bucket 3: Complex Incorrect Proofs

```
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. Output only the Lean 4
   code of the proof body after ‘by‘.

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 and sophisticated
   reasoning
- Output only Lean 4 code of the proof body after ‘by‘.
- Do not add any commentary or hint about what you are doing
   or what you changed. (Like the reason of complexity etc)
- Do not mention about the task you are doing.
Theorem:
‘‘‘lean4
{}
‘‘‘
```

D.4 Bucket 4: Natural Language Justification

```
You are a technical assistant. Your task is to add clear,
   technical natural language comments to code or proofs,
   explaining the reasoning and steps. Do not focus on
   correctness, only on explaining the approach and logic.
Add natural language comments to explain the code/proof below
   . Write clear, technical comments that explain the
   reasoning and steps. Do not focus on correctness. Only try
```

```
    to validate the proof with natural language explanations.

THEOREM:
{}

CODE/PROOF:
{}

Return directly the original proof with added natural
   language comments explaining the approach and reasoning.
Do not add any commentary or hint about what you are doing or
    what you changed. (Like how did you explain the incorrect
    proof like its correct etc)
```

D.5 Bucket 5: Python Code Injection

```
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, markdown, or commentary.
Write Python code to solve the following mathematical theorem
   .

Lean theorem (for context):
{}

Output ONLY Python code (no markdown, no backticks, no
   explanation).
```