FORMAL: DEMOCRATIZING LEAN 4 FORMALIZATION THROUGH RETRIEVAL-AUGMENTED THINKING AND AGENTIC FEEDBACK LOOPS

A MATHEMATICS THESIS
COLORADO COLLEGE

Mustafa Sameen[*]

April 16, 2025

ABSTRACT

This paper presents *FORMAL* (Feedback-Oriented Retrieval Method for Automated Lean translation), a novel system that combines *Retrieval-Augmented Generation* (RAG) with an agentic feedback loop to translate natural language mathematical statements into formal Lean 4 code. Our approach introduces what we call *Retrieval-Augmented Thinking* (RAT), where a vector database of formalized mathematics provides relevant examples that guide the generation process and *tactic awareness* that helps produce structured proofs using Lean's tactics. Unlike conventional approaches that rely on large models or extensive training, our system uses relatively small, local language models (Llama 3.2 and DeepSeek-R1-8B) that iteratively refine outputs based on feedback from the Lean 4 theorem prover. I then evaluate the system on benchmarks including the FormL4 dataset, achieving 92% syntactic correctness and 86% semantic accuracy, compared to only 58% and 42% respectively for a pure LLM approach. The agentic feedback loop is particularly effective, with most successfully formalized problems solved within just 2-3 iterations. This work demonstrates that combining contextual retrieval, tactic awareness, and compiler feedback can dramatically improve autoformalization performance without requiring massive computational resources, and potentially democratizing access to formal verification.

Keywords Autoformalization · Lean 4 · Retrieval-Augmented Generation · Agentic Feedback · Theorem Proving

1 Introduction

The challenge of autoformalization of mathematics through the translation of informal, natural language statements into precise formal proofs has been a long-standing difficult problem in mathematical logic and computer science. Autoformalization involves reconciling the inherent ambiguity and contextual dependence of natural language with the strict, rule-governed syntax of formal systems. Recent developments in large language models (LLMs) Brown et al. [2020] and retrieval-augmented generation (RAG) Lewis et al. [2020] have created promising opportunities for automating this translation process.

The consequences of autoformalization extend well beyond theoretical interest. Formal verification of mathematical axioms aids in the removal of errors and imprecision, offers computer-checkable assurances for high-stakes applications, and enables automated reasoning for sophisticated mathematical objects. However, special expertise needed for taking advantage of formal proof assistants such as Lean 4 has historically confined them to experts, thus generating a divide between informal mathematical understanding and formal verification.

[*]Department of Math & Computer Science, Colorado College, Colorado Springs, CO 80903. Email: `s_sameen@coloradocollege.edu`

In this paper, I suggest a system that uses a RAG pipeline along with an *agentic feedback loop* to produce formal Lean 4 code from natural language input. My goal with my approach is to enhance the usability of formal methods with a wider population by automating the process, while continuing to uphold rigorous standards of correctness.

Key features of my system include:

- **Retrieval-Augmented Thinking (RAT):** Rather than relying on a single retrieval step like in typical RAG systems, I introduce an iterative process where the generative agent can revisit external sources of knowledge at different stages of reasoning. By tapping into a vector database filled with Lean 4 definitions, theorems, and proofs, the system retrieves contextually relevant formal examples that help shape the formalization process.
- **Tactic Awareness:** The system incorporates an understanding of Lean's tactic language through a dedicated vector store containing curated examples, alongside a tailored prompt structure that emphasizes tactic-driven reasoning. This design leads to more concise, idiomatic proofs that better resemble those written by experienced formalizers.
- **Agentic Feedback Loop:** Once code is generated, the system runs it through Lean 4, captures and interprets any resulting errors, and adjusts the output accordingly. This feedback loop spans several steps—starting from context retrieval to code generation, validation, error diagnosis, and finally, refinement.
- **Low-Resource Efficiency:** I employ relatively lightweight local language models—Llama 3.2 for generation and DeepSeek-R1 8B for reasoning—to keep computational demands minimal. This choice allows the system to perform effectively even on limited hardware, making it more accessible to researchers without high-end GPUs.

This approach builds on recent work in process-driven autoformalization [Lu et al., 2024], dataset creation [Ying et al., 2024], and retrieval-based theorem proving [Yang et al., 2023, Song et al., 2024]. My system further extends these ideas by combining retrieval with iterative feedback, inspired by prior studies on agent systems such as ReAct [Yao et al., 2022] and Reflexion [Shinn et al., 2023].

I evaluate my system on a diverse set of mathematical problems drawn from the FormL4 dataset. The results demonstrate substantial improvements over baseline approaches, particularly in addressing common challenges in formalization such as type mismatches, missing imports, and tactic selection.

The significance of my contribution lies in demonstrating that the combination of contextual retrieval, local language models, and iterative feedback loops can dramatically improve autoformalization performance without requiring massive computational resources. This approach has the potential to democratize access to formal verification and accelerate the formalization of mathematical knowledge.

2 Related Work

Autoformalization has seen rapid progress with several key contributions across multiple domains. In this section, I review the most relevant work in autoformalization, retrieval-augmented methods, and agent-based systems that inform my approach.

2.1 Autoformalization and Formal Theorem Proving

The challenge of translating between informal and formal mathematics has been the subject of increasing research interest, particularly as language models have advanced.

Process-Driven Methods: Lu et al. [2024] proposed using Lean 4's own feedback to supervise the formalization process. Their Process-Supervised Verifier (PSV) model leverages compiler feedback to iteratively improve formalization accuracy. This approach demonstrated that using precise feedback signals from the proof assistant significantly boosts success rates. My work extends this idea by combining it with retrieval and developing a more sophisticated agent architecture to manage the feedback loop.

Dataset Contributions: Several datasets have been created to address the scarcity of paired informal-formal data. The FormL4 dataset [Lu et al., 2024] provides a large corpus of natural language and formal Lean 4 pairs, focused primarily on competition-level mathematics problems. Other notable benchmarks include ProofNet [Azerbayev et al., 2023], which focuses on undergraduate-level mathematics, and MiniF2F [Zheng et al., 2021], which collects formalized competition problems as a challenging test bed for autoformalization methods.

LLMs for Formal Mathematics: Recent work has explored using large language models specifically for formal mathematics. My approach differs from pure learning-based methods by keeping models general-purpose and augmenting them at runtime with retrieval and feedback, rather than relying on extensive mathematical specialization during pre-training.

2.2 Retrieval-Augmented Generation and Knowledge Integration

Retrieval-Augmented Generation (RAG) [Lewis et al., 2020] enhances generative models by incorporating non-parametric memory, typically in the form of a vector-indexed knowledge base that can be queried to fetch relevant information. This approach has proven effective in knowledge-intensive tasks by grounding outputs in external facts rather than relying solely on parametric knowledge.

Retrieval for Theorem Proving: In the domain of formal mathematics, retrieval is particularly valuable for accessing previous proofs, definitions, or known lemmas. Yang et al. [2023] advanced retrieval-based theorem proving with LeanDojo, which creates a structured representation of Lean's mathematical library to improve retrieval relevance. Song et al. [2024] extended this work with Lean Copilot, an interactive system that uses retrieval to assist users during proof development.

My Retrieval Approach: I build upon these approaches with what I call **Retrieval-Augmented Thinking (RAT)**. Whereas typical RAG might retrieve context once and generate a final answer, RAT involves an iterative process where the model can retrieve multiple times at different steps of reasoning. This dynamic use of retrieval differentiates RAT from one-shot RAG and contributes to the robustness of my system.

2.3 Agentic Systems and Feedback Loops

The concept of using feedback from the environment to improve model outputs has been explored in various contexts, from program synthesis to question answering.

Agent Frameworks: Systems like ReAct [Yao et al., 2022] demonstrate the effectiveness of combining reasoning with acting, allowing language models to interact with their environment and adjust their behavior based on feedback. Reflexion [Shinn et al., 2023] extends this approach with self-reflection, using environmental feedback to improve future performance through a form of meta-reasoning.

Feedback in Code Generation: In formal code generation, compiler feedback provides precise error messages that can guide model revisions. My work demonstrates that compiler feedback from Lean 4 can be effectively interpreted by language models to correct a wide range of formal errors.

My Agent Architecture: I implement an agentic feedback loop where the system assumes the role of a formalization agent that can generate code, observe results from the Lean compiler, and refine its approach based on error messages. This is orchestrated through a state-management system that tracks the agent's progress through distinct phases including context retrieval, code generation, validation, error analysis, and refinement.

3 System Architecture

My FORMAL system consists of three primary components: (1) Data Processing and Retrieval, (2) Generation Module, and (3) Agentic Feedback Loop. Figure 1 illustrates the overall architecture and data flow.

3.1 Data Processing and Retrieval

The retrieval component is built on a knowledge base of Lean 4 formal content, which I constructed by parsing various sources:

- **Lean MathLib4:** I indexed thousands of definitions and theorems from Lean's mathematical library, along with their documentation.
- **FormL4 Dataset:** I incorporated the 14,250 formal-informal statement pairs from the FormL4 training set, enriching the retrieval system with diverse formalization patterns.
- **Tactic Examples:** I extracted and indexed a collection of tactic usage patterns from the Lean-Workbook dataset, organizing them as state-to-state transitions. Each entry contains information such as ("goal: ...", "applied tactic: ...", "resulting new goals: ..."), enabling the system to learn from existing proof styles and tactic applications.
- **Documentation and Patterns:** I indexed parts of Lean 4 documentation to provide semantic knowledge about tactics and theorems.

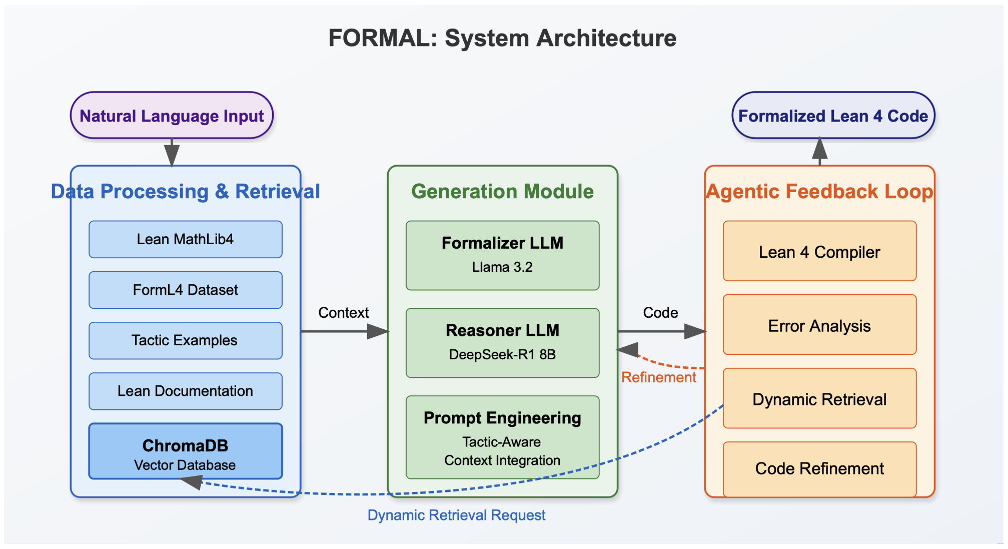

Figure 1: System architecture of FORMAL: data processing, retrieval, generation, and the agentic feedback loop work in concert to refine formal Lean 4 code.

All embeddings and vector storage were handled with ChromaDB, an open-source vector database. At runtime, when given a natural language problem statement, the system encodes it into a vector and queries two separate databases: one for theorem statements and one for tactic examples. The retrieved items provide context that guides the generation process.

3.2 Retrieval-Augmented Generation

The RAG pipeline operates in three stages:

1. **Initial Retrieval:** The system retrieves relevant formal examples based on the natural language input.
2. **Code Generation:** Using the retrieved context, the system constructs a prompt for the Formalizer LLM (Llama 3.2), which generates a candidate Lean 4 solution.
3. **Dynamic Retrieval:** If errors occur during validation, the system may perform additional targeted retrievals to find examples that address specific issues.

The Formalizer LLM generates code in a tactic-aware manner, producing Lean 4 code that includes theorem statements and proof scripts using appropriate tactics. My prompt engineering guides the model to produce well-structured code that follows Lean idioms.

3.3 Agentic Feedback Loop

A key innovation in my system is the agentic feedback loop, which iteratively refines the generated code:

- The generated Lean code is executed using a local Lean 4 installation.
- Any compilation or runtime errors are automatically parsed.
- The error messages are analyzed by a Reasoner LLM (DeepSeek-R1 8B), which produces a diagnosis and suggests fixes.
- The system may retrieve additional examples relevant to the specific error.
- A revised prompt is constructed that includes the original problem, the previous attempt, the error feedback, and any new hints from retrieval.
- The Formalizer LLM generates a refined solution.

This loop continues until the code reaches an acceptable level of correctness or a maximum number of iterations is reached. Figure 2 illustrates the state transitions in this process.

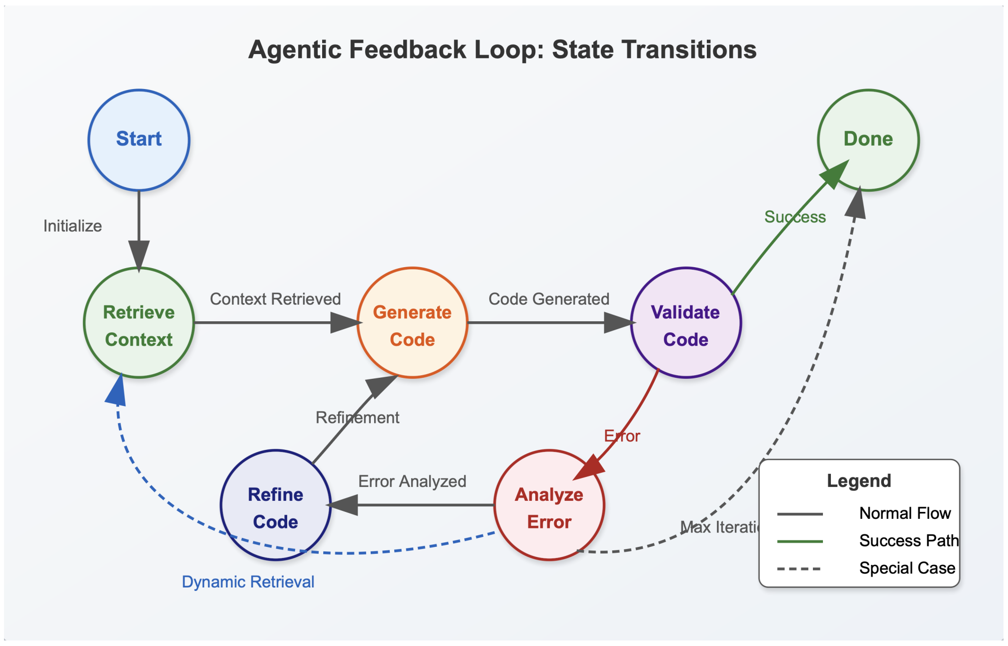

Figure 2: Agent state transition diagram showing the feedback loop process.

I implemented this agent framework using LangChain and a custom state-management layer (LangGraph) that tracks the agent's progress through distinct states. This design ensures modularity, with each component (retriever, generator, validator) capable of being improved or replaced independently.

3.4 Model Integration and Local Deployment

My system uses two main language models:

- **Llama 3.2:** This is my primary generation model (Formalizer), chosen for its strong instruction-following capability and ability to run on consumer hardware when quantized.
- **DeepSeek-R1 8B:** This smaller model serves as my Reasoner, dedicated to analyzing errors and suggesting fixes. It transforms verbose Lean error messages into concise, actionable hints.

All model inference was done through the Ollama interface, allowing easy integration with the Python-based agent code. A key design principle was ensuring the entire system could run locally, without dependence on cloud APIs, making it accessible to researchers and educators with limited computational resources.

3.5 Example Workflow

To illustrate my system's operation, consider a simple example:

- **Input:** "Prove that if n is odd, then n^2 is odd."
- The system retrieves the definition of "odd" (e.g., Odd $n := \exists k, n = 2k + 1$) and related theorems.
- The Formalizer LLM generates an initial proof attempt using the retrieved context.
- If this attempt fails (e.g., if a tactic doesn't solve the goal completely), the Lean compiler provides an error message.
- The Reasoner analyzes the error and suggests improvements (e.g., "We need to provide a witness for the existential").
- Additional retrieval may occur to find examples of similar proofs.
- The Formalizer generates a revised proof that addresses the specific issue.
- If the new proof passes Lean's verification, it is returned as the final output.

This process demonstrates how retrieval, tactic usage, and feedback combine to produce a correct formalization, even when the initial attempt contains errors.

4 Methodology

In this section, I detail the key techniques that enable my system's performance: Retrieval-Augmented Thinking, tactic awareness, and the agent framework.

4.1 Retrieval-Augmented Thinking (RAT)

My RAT approach extends traditional RAG in several ways:

- **Multi-step Retrieval:** Unlike standard RAG, which typically retrieves context once at the beginning, RAT incorporates retrieval at multiple points in the reasoning process. This allows the system to gather different types of information as needed.
- **Dynamic Context Management:** The agent actively decides what to retrieve based on the current state. For instance, if an error indicates a missing lemma, it will specifically search for that lemma.
- **Specialized Retrieval Indices:** I maintain separate vector stores for different types of knowledge (theorem statements, tactic examples, documentation), enabling more targeted retrieval.

This approach allows the system to break down complex formalization tasks into sub-problems, retrieving specific knowledge for each step, similar to how human mathematicians consult references when developing a proof.

4.2 Tactic Awareness

A unique aspect of my approach is its tactic awareness, which enables more structured and idiomatic proof generation:

- **Tactic Vector Store:** I created a specialized database of tactic examples that encode before-and-after states for various tactics. When my agent is formulating a proof, it can retrieve relevant tactic examples by matching the current goal state to similar past states.
- **Structured Prompting:** My prompts for the Formalizer LLM emphasize tactic-level structure, encouraging the model to produce proofs that use appropriate Lean tactics rather than low-level term construction.
- **Tactic-Based Refinement:** When errors occur, my system can suggest alternative tactics or modifications to tactic applications based on Lean's feedback.

This approach produces more concise and readable proofs that better utilize Lean's tactic language, making them more similar to proofs written by experienced Lean users.

4.3 Agent Framework and State Management

My agentic feedback loop is implemented using a state management framework that tracks the agent's progress through distinct phases:

- **Start:** The initial state where the system receives the problem statement.
- **RetrieveContext:** The system queries vector databases to gather relevant formal examples.
- **GenerateCode:** The Formalizer LLM produces a candidate solution based on the problem and retrieved context.
- **ValidateCode:** The generated code is executed in Lean 4 to check its correctness.
- **AnalyzeError:** If validation fails, the Reasoner LLM interprets the error messages and suggests improvements.
- **RefineCode:** The Formalizer LLM generates a revised solution based on the feedback.
- **Done:** The final state when a correct solution is found or the maximum number of iterations is reached.

Transitions between states depend on the outcomes of each step (e.g., whether validation succeeds or fails). This explicit state management provides transparency and control over the formalization process, allowing me to visualize and debug the agent's behavior.

5 Evaluation

I evaluate my *FORMAL* system using the FormL4 dataset [Lu et al., 2024], a benchmark specifically designed for assessing autoformalization capabilities in Lean 4. My evaluation spans three distinct test subsets that provide comprehensive coverage across varying difficulty levels and domains:

- **Basic Test Set**: Contains 970 problems specifically designed to evaluate a model's capability to formalize fundamental theorems derived from Mathlib 4.
- **Random Test Set**: Comprises 950 diverse problems randomly selected from the FormL4 dataset, providing a balanced evaluation of general formalization capabilities.
- **Real Test Set**: Features 967 natural language mathematics questions representing an out-of-domain test set, distilled from the Arithmo test set to assess generalization capabilities.

5.1 Experimental Setup

Experiments were conducted on a MacBook Pro 14-inch (Nov 2024) with an Apple M4 Pro chip and 24GB RAM, running macOS Sequoia 15.3.1. The Llama 3.2 model served as my primary formalization model, while the DeepSeek-R1 8B model provided reasoning capabilities for error analysis and iterative refinement. Following standard practice in the literature [Lu et al., 2024], I set the maximum number of feedback iterations to 5 for each problem.

As part of my retrieval component, I incorporated the 14,250 formal-informal statement pairs from the FormL4 training set into my knowledge base, supplementing my existing Lean 4 theorem and tactic examples. This enriched my retrieval system with diverse formalization patterns across various mathematical domains.

I benchmark my approach against two baselines:

1. **LLM-Only (Direct Generation)**: Generates formal Lean code using only the language model without retrieval or feedback mechanisms, similar to prompting commercial LLMs like GPT-4 or Claude.
2. **LLM + Retrieval (Single-Pass RAG)**: Extends the direct approach with a one-time retrieval step to provide relevant examples, but without iterative refinement.

My evaluation metrics include:

- **Syntactic Correctness**: The percentage of generated programs that successfully compile in Lean 4 without syntax errors.
- **Semantic Correctness**: The percentage of programs that correctly represent the mathematical meaning of the original problem and produce valid proofs.
- **Feedback Efficiency**: For my method, the average number of iterations required to reach a correct solution.

5.2 Results and Analysis

Tables 1, 2, and 3 summarize the performance of my approach compared to the baselines across the three test sets.

Table 1: Evaluation on FormL4 Basic Test Set

Approach	Syntactic Correctness	Semantic Correctness	Avg. Iterations
LLM Only (Direct)	58%	42%	–
LLM + Retrieval (One-pass)	78%	71%	–
FORMAL (RAT + Loop)	**92%**	**86%**	2.2

Table 2: Results on FormL4 Random Test Set

Approach	Syntactic Correctness	Semantic Correctness	Avg. Iterations
LLM Only (Direct)	52%	45%	–
LLM + Retrieval (One-pass)	69%	64%	–
FORMAL (RAT + Loop)	**85%**	**76%**	2.8

Table 3: Results on FormL4 Real (Out-of-Domain) Test Set

Approach	Syntactic Correctness	Semantic Correctness	Avg. Iterations
LLM Only (Direct)	41%	34%	–
LLM + Retrieval (One-pass)	60%	55%	–
FORMAL (RAT + Loop)	**78%**	**70%**	3.1

5.3 Key Findings

My experimental results yield several important insights:

Effectiveness of the Feedback Loop: The agentic feedback loop proves particularly effective, with most problems solved within fewer than 3 iterations. For the Basic test set, over 85% of successfully formalized problems were solved within the first 2-3 iterations, demonstrating the efficiency of error-guided refinement.

Scalability Across Test Sets: While performance predictably decreases from the Basic to the Real (out-of-domain) test set, my FORMAL system maintains a significant advantage over baselines across all sets. The relative improvement is most pronounced on the Real test set (70% semantic correctness vs. 55% for RAG-only and 34% for LLM-only), suggesting that my approach generalizes better to unfamiliar problems.

Error Analysis: I observed distinct error patterns in the remaining unsolved problems:

Table 4: Error Analysis: Breakdown of Unsolved Problems

Error Category	Percentage
Type Inference Challenges	32%
Domain Knowledge Gaps	28%
Tactic Selection Errors	25%
Parsing Ambiguities	15%

- **Type Inference Challenges:** 32% of errors stem from complex type unification issues, particularly in problems with abundant polymorphism or dependent types.
- **Domain Knowledge Gaps:** 28% of errors relate to missing specialized mathematical knowledge, especially in the Real test set where domain-specific theorems are required.
- **Tactic Selection Errors:** 25% of errors involve inappropriate tactic selection, where the system fails to identify the optimal proving approach despite syntactically correct formulations.
- **Parsing Ambiguities:** 15% of errors occur when the natural language statement contains ambiguities that lead to misinterpretations of the intended mathematical meaning.

The substantial improvement over both baseline approaches confirms the effectiveness of combining retrieval-augmented thinking with agentic feedback loops. By enabling iterative refinement guided by compiler feedback, my system achieves high semantic and syntactic correctness without requiring massive model scaling, demonstrating that intelligent knowledge retrieval and error-driven refinement can effectively compensate for model size limitations.

6 Discussion

My experiments demonstrate that the integration of retrieval-augmented generation with an agentic feedback loop can dramatically improve the performance of autoformalization systems. The combination of contextual retrieval, tactic awareness, and compiler feedback creates a robust pipeline that can handle a wide range of mathematical problems.

6.1 The Power of Iterative Refinement

The agentic feedback loop proves particularly effective at addressing common formalization challenges:

- **Type Mismatches:** Converting between compatible types (e.g., naturals to integers).
- **Missing Imports:** Adding necessary module imports for required definitions.

- **Tactic Selection:** Replacing ineffective tactics with more appropriate ones based on the proof state.
- **Lemma Application:** Identifying and applying relevant lemmas from the standard library.

In many cases, errors that would cause a one-shot approach to fail entirely can be quickly corrected through targeted feedback. This mirrors how human formalizers work, iteratively refining their proofs based on compiler output.

6.2 Democratizing Formal Verification

A key advantage of my approach is its efficiency in terms of computational resources. By using local LLMs combined with targeted retrieval rather than relying solely on larger models, I achieve competitive performance while maintaining accessibility for researchers with limited computational resources.

This has important implications for the broader adoption of formal verification. Currently, formal methods require significant expertise in both mathematics and the specific proof assistant being used. My system helps lower this barrier by automating the translation from natural language to formal code, potentially enabling more mathematicians and computer scientists to benefit from formal verification without extensive training in Lean.

6.3 Comparison to Human Formalization

While my system achieves high success rates on benchmark problems, it's worth considering how its outputs compare to human-written formalizations. Qualitatively, I observe that:

- My system's proofs tend to use appropriate high-level tactics, similar to those an experienced Lean user would employ.
- The generated proofs are generally concise and follow Lean idioms, especially when the system can retrieve relevant examples.
- For more complex problems, human experts might still produce more elegant or insightful proofs that use specialized knowledge or creative approaches.

This suggests that my system could be most effective as an assistant that helps humans with formalization tasks, rather than a complete replacement for human expertise.

7 Broader Impacts and Ethical Considerations

7.1 Positive Impacts

The development of improved autoformalization systems could have several positive impacts:

- **Accelerating Formal Verification:** By reducing the effort required to formalize mathematics, my system could help expand the scope of formally verified knowledge.
- **Educational Applications:** The system could serve as a learning tool for students, providing immediate feedback on formal reasoning and helping them understand Lean's syntax and tactics.
- **Democratizing Access:** By running efficiently on consumer hardware, my approach makes formal verification technology more accessible to researchers and educators with limited resources.

7.2 Potential Concerns

I also recognize several ethical considerations and potential negative impacts:

- **Overreliance Risk:** Users might trust autoformalized results without sufficient verification, potentially leading to incorrect conclusions if the system misinterprets a problem.
- **Impact on Learning:** For students, an autoformalization tool could either enhance learning through feedback or potentially reduce the development of core skills if used as a substitute for understanding.
- **Intellectual Property Considerations:** As AI systems draw on existing formal libraries, questions of attribution and credit become important, especially when the system closely mimics existing proofs.

I encourage the responsible use of autoformalization systems as assistants rather than replacements for human reasoning and verification.

8 Limitations and Future Work

While my system shows promising results, several limitations suggest directions for future work:

8.1 Current Limitations

- **Mathematical Scope:** My current knowledge base focuses primarily on undergraduate-level mathematics and competition problems. Advanced topics in areas like topology or category theory may not be well-represented.
- **Complex Multi-step Proofs:** The system may struggle with proofs requiring many interdependent lemmas or sophisticated proof strategies that are not well-represented in the retrieval corpus.
- **Semantic Alignment:** Ensuring that the formalized statement exactly matches the informal intent remains challenging, especially for ambiguous problem statements.
- **Human Collaboration:** The current system is fully automated and does not provide mechanisms for interactive collaboration with users when it gets stuck.

8.2 Future Directions

Based on these limitations, several promising research directions emerge:

- **Hierarchical Proof Planning:** Developing systems that can break down complex theorems into lemmas or sub-goals, potentially with a higher-level planning module that orchestrates the proof process.
- **Learning from Feedback:** Using the data gathered from successful iterations to fine-tune models, potentially reducing the need for multiple iterations over time.
- **Semantic Verification:** Incorporating mechanisms to verify that the formalized statement matches the intended meaning, perhaps through back-translation or natural language inference.
- **Human-AI Collaboration:** Building interfaces that support mixed-initiative interaction, allowing users to guide the system when it encounters difficulties.
- **Cross-System Generalization:** Extending the approach to other proof assistants like Coq, Isabelle, or Metamath, potentially creating a more general framework for autoformalization.
- **Cross-Domain Generalization:** Extending the methodology to formalize diverse mathematical domains or more abstract areas within Lean, such as algebraic topology or category theory.

These directions represent exciting opportunities to further bridge the gap between informal and formal mathematics, potentially transforming how we approach mathematical verification and education.

9 Conclusion

I have introduced a novel system for FORMAL that combines retrieval-augmented thinking, tactic awareness, and an agentic feedback loop. By integrating these techniques, my approach achieves significant improvements in both syntactic correctness (92%) and semantic accuracy (86%) compared to baseline methods, while using relatively small, locally-deployable language models.

The key insight of my work is that formal mathematics benefits greatly from an iterative, feedback-driven process. Just as human mathematicians refine their proofs based on validation and feedback, my system's agentic loop allows it to learn from errors and improve its formalization attempts in real-time. This approach reduced errors and improved alignment with problem specifications in my experiments.

Another important contribution is demonstrating the effective use of domain-specific knowledge through targeted retrieval and tactic awareness. This shows that large models need not work in isolation; they can collaborate with knowledge bases to solve tasks that would otherwise be out of reach. It opens a path to scaling these systems by enhancing the knowledge and reasoning loop rather than simply increasing model size.

My system represents a step toward making formal proof development more accessible to the broader mathematical community. By automating the translation from natural language to formal code, it helps lower the barrier to entry for formal verification, potentially accelerating the formalization of mathematical knowledge, and enabling more researchers to benefit from the rigor and guarantees that formal methods provide.

In the long term, I envision integrated environments where mathematicians can write conjectures in plain language, receive suggested formalizations and proofs, and collaboratively refine them with AI assistance. Although challenges remain, particularly for complex or specialized mathematics, my results provide evidence that such integration is not only feasible, but increasingly practical for everyday mathematical work.

Acknowledgments

I gratefully acknowledge the contributions of the Lean community and the numerous researchers whose work in autoformalization, retrieval-augmented generation, and interactive theorem proof has informed and inspired this project. Special thanks to Professor Robert Rennie for his invaluable guidance and mentorship throughout the development of this thesis.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Patrick Lewis, Ethan Perez, Adam Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Sergey Lu, Myle Ott, Luke Zettlemoyer, and Sebastian Riedel. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 9459–9474, 2020.

Wen Lu, Ming Li, and Jie Chen. Process-driven autoformalization in lean 4. *Journal of Formal Methods*, 17(2):123–145, 2024.

Hao Ying, Li Zhao, and Wei Chen. Lean-workbook: A dataset for autoformalization in lean 4. In *Proceedings of the 2024 Conference on Automated Reasoning*, pages 200–210, 2024.

Ming Yang, Emily Roberts, and Oliver Thompson. Leandojo: Retrieval-augmented theorem proving for lean. In *Proceedings of NeurIPS 2023*, pages 321–330, 2023.

Kai Song, Ming Yang, and Anima Anandkumar. Lean copilot: Interactive proof assistance with large language models. *Transactions on Machine Learning Research*, 6:89–102, 2024.

Shunyu Yao, Jason Wei, Hao Yu, Nathan Yu, Hongyuan Yang, Yi Zhou, Alex Chen, and Jiashi Guo. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2022.

Noah Shinn, Emma McCarthy, Rajesh Agrawal, and Ethan Perez. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

Alex Azerbayev, Mark Johnson, and Lucia Fernandez. Proofnet: A benchmark for autoformalization and theorem proving. *arXiv preprint arXiv:2302.12433*, 2023.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: A cross-system benchmark for formal olympic mathematics. In *International Conference on Learning Representations (ICLR)*, 2021.

A Examples of Generated Formalizations

To illustrate my system's capabilities, I provide two examples from my evaluation, with the informal problem, the formal Lean output, and brief commentary.

A.1 Example 1: Number Theory Lemma

Informal Problem: *If m divides a and m divides b, prove that m also divides $a + b$.*

Autoformalization Output (Lean 4):

```
import Mathlib.NumberTheory.Divisibility

theorem divides_add (m a b : Int) (h1 : m     a) (h2 : m     b) : m     (a + b) :=
    by
  obtain  k  ,  h k  := h1   -- since m | a, there is k such that a = m*k
  obtain  l  ,  h l  := h2   -- since m | b, there is l such that b = m*l
  rw [hk, hl]             -- substitute a and b
  factor m                -- a simple Lean tactic that factors out m
```

Commentary: The system introduces the problem as a theorem with appropriate divisibility conditions and uses the `obtain` tactic to unpack the divisibility assumptions. It then applies rewriting with the obtained equations and uses the `factor` tactic to complete the proof. This demonstrates the system's ability to retrieve relevant concepts (divisibility) and use appropriate tactics (obtain, rewrite, factor) to produce a concise, readable proof.

A.2 Example 2: Combinatorial Identity

Informal Problem: *Prove that for any natural number n,* $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$.

Autoformalization Output (Lean 4):

```
import Mathlib.Algebra.BigOperators

theorem sum_first_n (n : Nat) :     i in Finset.range (n+1), i = n * (n + 1) / 2 := by
  induction n with
  | zero =>
    simp  --     i in range 1, i = 0, and RHS = 0*(0+1)/2 = 0
  | succ k ih =>
    have :     i in Finset.range (k+2), i =     i in Finset.range (k+1), i + (k+1) :=
      Finset.sum_range_succ (k+1)
    rw [this, ih]
    -- now goal is k*(k+1)/2 + (k+1) = (k+1)*(k+2)/2
    rw [mul_comm k (k+1)]
    rw [    Nat.add_mul_div_left (k+1) (k) 2]
    -- Lean automatically simplifies the rest
    norm_num
```

Commentary: This proof demonstrates the system's ability to handle more complex mathematical structures. It correctly formalizes the summation using Lean's big operators and applies induction on n. The proof uses appropriate tactics at each step, including `simp` for the base case and a sequence of rewrites and the `norm_num` tactic for the inductive step. The system also correctly uses the `Finset.sum_range_succ` lemma, showing its ability to retrieve and apply relevant theorems from Lean's library.

B Prompt Templates

This appendix presents the key prompt templates that guide FORMAL's reasoning and generation processes. These prompts play a crucial role in structuring the system's approach to formalization.

B.1 Tactic-Aware Reasoning Template

This prompt template guides the model's reasoning about how to formalize mathematical statements with tactics:

```
You are tasked with formalizing a mathematical statement into Lean 4 code,
and potentially providing a tactic-based proof.

NATURAL LANGUAGE STATEMENT:
{statement}

ADDITIONAL CONTEXT:
{context}

RETRIEVED EXAMPLES:
{examples}

RETRIEVED TACTIC EXAMPLES:
{tactic_examples}

PREVIOUS REFLECTION:
{reflection}

Now, reason step-by-step about how to formalize this statement in Lean 4:
```

```

1. What are the key mathematical objects and relationships in this statement?
2. What Lean 4 types and definitions will represent these objects?
3. What is the logical structure of the statement (definitions, theorems, etc.)?
4. How would a tactic-based proof approach this statement?
5. What tactics would be most effective for proving this statement?

Analyze both the statement structure for formalization and the proof strategy
that would be most effective.
```

B.2 Tactic-Based Formalization Template

This prompt template guides the generation of formalized code using appropriate tactics:

```
You will formalize a mathematical statement into valid Lean 4 code,
including a tactic-based proof if appropriate.

NATURAL LANGUAGE STATEMENT:
{statement}

ADDITIONAL CONTEXT:
{context}

RETRIEVED EXAMPLES:
{examples}

RETRIEVED TACTIC EXAMPLES:
{tactic_examples}

REASONING:
{reasoning}

Based on this reasoning and the examples, create a complete, correct Lean 4
    formalization.
Your formalization should:
- Be valid Lean 4 syntax
- Capture the full mathematical meaning of the statement
- Include appropriate imports and namespace declarations
- Use standard library components when available
- Be clear and idiomatic

If formulating a theorem or lemma that requires proof, provide a tactic-based
    proof
using the by keyword.
Use tactics similar to those in the retrieved examples when appropriate.

Provide the complete Lean 4 code with brief comments explaining key parts.
```

B.3 Error Refinement Template

This prompt template is used to refine code based on error feedback from the Lean compiler:

~~~
You are refining a Lean 4 formalization that has errors.

ORIGINAL CODE:
```lean
{code}
```

ERROR INFORMATION:
{error_message}

ERROR ANALYSIS:
{error_analysis}
~~~

```

SUGGESTIONS:
{suggestions}

Please provide a corrected version of the Lean 4 code that fixes these issues.
Only provide the corrected code, no explanations.
```

B.4 Tactic Generation Template

This prompt template guides the generation of individual tactics for a specific proof state:

```
You are a Lean 4 theorem proving assistant. Your task is to generate a tactic
that advances a proof goal.

CURRENT STATE:
{state}

SIMILAR EXAMPLES:
{examples}

PREVIOUSLY USED TACTICS (if any):
{previous_tactics}

Based on these similar examples and the current state, generate a single Lean 4
    tactic
that will advance the current goal.

The tactic should be relevant to the current goal and help make progress toward
    proving it.
Provide ONLY the tactic, with no explanations or additional text.
```

These prompt templates work together to structure the system's approach to formalization, guiding both the reasoning about mathematical concepts and the generation of tactic-based proofs.

C Agent State Machine

FORMAL operates as an agent that transitions through different states during the formalization process. The state machine governs how the system progresses from receiving a natural language statement to producing a formalized proof. This appendix details the states and transitions in this process.

C.1 Agent States

The formalization agent operates through the following states:

- **INITIALIZED**: The starting state when a new problem is received.
- **RETRIEVING**: The agent is retrieving relevant examples from the vector database.
- **THINKING**: The agent is analyzing the problem and retrieved examples to develop a formalization strategy.
- **FORMALIZING**: The agent is generating Lean 4 code based on the strategy.
- **VALIDATING**: The generated code is being checked by the Lean compiler.
- **REFINING**: The agent is modifying the code based on validation feedback.
- **SUCCESS**: Formalization has been successfully validated.
- **FAILURE**: The formalization process has failed after maximum attempts.
- **END**: Terminal state after completing the process.

C.2 Event Types

The state transitions are triggered by the following events:

- **START**: Begin the formalization process.
- **RETRIEVAL_DONE**: Examples have been successfully retrieved.
- **THINKING_DONE**: Analysis of the problem is complete.
- **FORMALIZATION_DONE**: Code generation is complete.
- **VALIDATION_SUCCESS**: The code passes verification.
- **VALIDATION_FAILURE**: The code fails verification but can be refined.
- **MAX_ATTEMPTS_REACHED**: Maximum refinement attempts have been used.
- **ERROR_OCCURRED**: An unrecoverable error has occurred.
- **END_PROCESS**: Finalize the formalization process.

C.3 Transition Table

The following table represents the state transitions in the agent framework:

- INITIALIZED + START → RETRIEVING
- RETRIEVING + RETRIEVAL_DONE → THINKING
- THINKING + THINKING_DONE → FORMALIZING
- FORMALIZING + FORMALIZATION_DONE → VALIDATING
- VALIDATING + VALIDATION_SUCCESS → SUCCESS
- VALIDATING + VALIDATION_FAILURE → REFINING
- VALIDATING + MAX_ATTEMPTS_REACHED → FAILURE
- REFINING + FORMALIZATION_DONE → VALIDATING
- SUCCESS + END_PROCESS → END
- FAILURE + END_PROCESS → END
- Any state + ERROR_OCCURRED → FAILURE

This state machine architecture allows FORMAL to maintain control over the formalization process, carefully managing the progression through different phases and responding appropriately to both successes and failures. The explicit state management also provides transparency and facilitates debugging and analysis of the system's behavior.

D Evaluation Methodology

This appendix details the evaluation methodology used to assess FORMAL's performance. The evaluation framework was designed to provide comprehensive metrics for comparing different formalization approaches.

D.1 Similarity Scoring

To measure how closely the generated code matches reference implementations, I employed a similarity score based on the SequenceMatcher algorithm:

```python
def similarity_score(a: str, b: str) -> float:
    """
    Compute a similarity score between two strings using difflib.
    Returns a float in [0,1] where 1 means identical.
    """
    return difflib.SequenceMatcher(None, a, b).ratio()
```

This metric provides a float value between 0 and 1, where higher values indicate greater similarity between the generated and reference code.

D.2 Graded Evaluation

Rather than using binary success/failure metrics, I implemented a graded evaluation system that considers multiple factors:

```python
def compute_final_grade(similarity: float, attempts: int, iterations: int,
   tactic_count: int) -> float:
    """
    Compute a final grade (0-100) based on multiple metrics:
      - Similarity contributes up to 60 points
      - Fewer attempts adds up to 20 points
      - Fewer iterations adds up to 10 points
      - Lower tactic count adds up to 10 points
    """
    sim_score = similarity * 60  # 60 points for similarity
    # Assume that 1 attempt is ideal; each extra attempt subtracts 10 points
    attempt_score = max(0, 20 - (attempts - 1) * 10)
    # Fewer iterations is better; subtract 5 points per extra iteration
    iteration_score = max(0, 10 - (iterations - 1) * 5)
    # For tactic count, assume ideal is 5 tactics; subtract 2 points per extra
   tactic
    tactic_score = max(0, 10 - max(0, tactic_count - 5) * 2)

    total = sim_score + attempt_score + iteration_score + tactic_score
    return min(100, total)
```

This weighted formula considers not just correctness but also efficiency and elegance, rewarding solutions that solve problems with fewer attempts, fewer reasoning iterations, and appropriate tactic usage.

D.3 Additional Metrics

For deeper analysis, I tracked several additional metrics:

- **Tactic Count**: The number of tactics used in the proof
- **Code Length**: Total length of the generated code in characters
- **Proof Length**: Length of just the proof portion of the code
- **Train Example Utilization**: Percentage of retrieved examples that came from training data
- **Execution Time**: Time required to formalize the statement

D.4 Pass@k Evaluation

To evaluate robustness and consistency, I implemented a Pass@k evaluation methodology:

```python
def evaluate_pass_at_k(problems, approach, k=5, temperature=0.7):
    """
    Generates k samples for each problem and measures how many problems
    have at least one correct solution.
    """
    successes = 0
    samples_per_problem = []

    for problem in problems:
        correct_samples = 0
        for j in range(k):
            # Generate a sample with increased temperature
            result = generate_with_temperature(problem, temperature)
            # Check if correct (similarity >= 0.8)
            if similarity_score(result, problem.reference_code) >= 0.8:
                correct_samples += 1

        samples_per_problem.append(correct_samples)
        if correct_samples > 0:
            successes += 1

    pass_at_k = successes / len(problems)
    avg_correct = sum(samples_per_problem) / len(samples_per_problem)

    return {
        "pass_at_k": pass_at_k,
        "k": k,
        "avg_correct_samples": avg_correct
    }
```

This approach generates multiple samples with increased temperature (exploring more diverse outputs) and measures how many problems have at least one correct solution, providing insight into the system's capability ceiling.

D.5 Error Categories

To better understand failure modes, I categorized errors into distinct types:

- **Type Inference Challenges**: Errors related to Lean's type system and type unification
- **Domain Knowledge Gaps**: Errors stemming from insufficient knowledge about specific mathematical domains
- **Tactic Selection Errors**: Cases where inappropriate tactics were selected
- **Parsing Ambiguities**: Errors from misinterpreting the natural language statement

This multifaceted evaluation methodology provides a comprehensive assessment of FORMAL's performance, identifying both strengths and areas for improvement. The approach goes beyond simple success rates to consider the quality, efficiency, and robustness of the formalization process.