LLMs Can Learn Theorem Libraries Through Dialogue to Become Effective Autoformalizers

Ziyu Zhou
Czech Technical University
zhouziyu@cvut.cz

Ziwei Li
Czech Technical University
liziwei@fel.cvut.cz

Abstract

In this work, we leverage large language models to summarize theorem library files into auxiliary prompts for the autoformalization task. Under a pass@5 budget, our approach achieves 100% syntactic accuracy on the MiniF2F-test. The main advantage of this simple pipeline is that it enables low-cost and timely adaptation to the latest theorem libraries, a capability that is difficult to achieve with traditional fine-tuning-based methods.

1 Motivation and Background

Automatically translating natural language mathematical propositions into formal mathematical languages is known as autoformalization. Formal mathematical languages can be used to verify the correctness of mathematical proofs and serve as training data for reasoning in large models, and are therefore considered to be of great significance [14]. However, the manual formalization of mathematics by human scientists is extremely costly.

The current paradigm mainly focuses on fine-tuning large language models (LLMs) on datasets that align natural language with formal language [13][16][12][15]. A key difficulty with this approach is that formalization tasks heavily rely on rapidly updated theorem libraries that are not forward-compatible, while existing datasets often become outdated relative to the latest syntax updates. A frequently auxiliary approach is to augment generative models with retrieval from the theorem library[6]. However, existing works (e.g., [6] and [1]) employ retrievers (such as the BGE series models [3]) that are trained on natural language, without fine-tuning for autoformalization tasks, and they still struggle to keep up with theorem library version updates.

It has been shown that large language models demonstrate robust aptitude for executing rule-based tasks [8]. Inspired by this, we aim to investigate whether dialogue-based large language models can autonomously learn a formal language's theorem library, summarize the underlying concepts and syntactic rules, and then use those rules to carry out autoformalization. This approach has the advantage of being able to track the latest theorem libraries and even private projects in real time, providing strong generality. Another motivation of this work is to build a widely usable autoformalizer at the lowest possible cost. Mathematicians often lack the computational resources required to train and deploy models. Thus, our goal is to build a solution entirely based on public APIs, allowing users to reproduce and extend the tool at low cost to meet personalized needs. At the same time, this also demonstrates that LLMs possess sufficient capabilitiesgiven appropriate prompts, they can solve complex tasks.

This paper reports an ongoing project. Our upcoming research plans can be found in the 4 section.

2 Methodology

2.1 Pipeline

This project is implemented almost entirely through prompt engineering. The chosen formal language is Lean 4 [7], which relies on the Mathlib4 theorem library. First, a series of Mathlib4 library files are provided

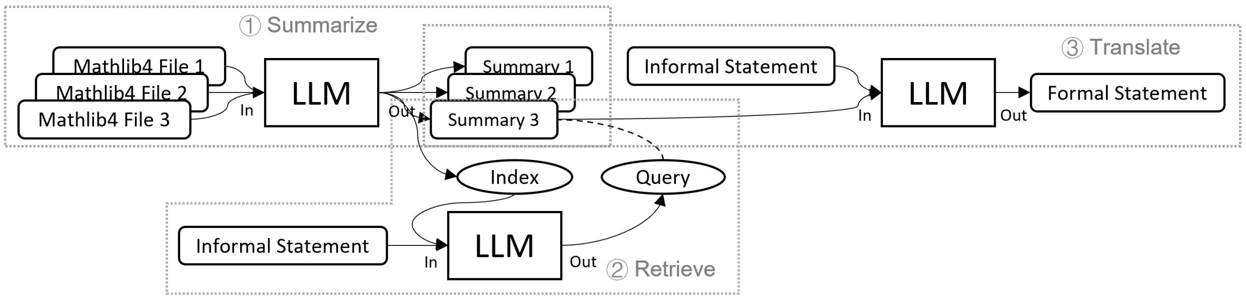

Figure 1: Autoformalizer pipeline. Step 1: Let the LLM summarize key definitions and syntactic rules from Mathlib4, and maintain an index for retrieval. Step 2: Given a natural language statement, the LLM locates the corresponding keywords in the index file and matches the summary files. Step 3: Feed both the natural language statement and the summary files into the LLM to obtain the translation.

to a base model. The model is prompted using a template to summarize the mathematical concepts and the syntactic rules declared in these files. In practice, theorems in the mathematical library are not directly used in autoformalization tasks, they can instead serve as examples of how definitions are used. Therefore, when submitting files to the model, most of the theorems can be removed.

Along with generating summary files, we also maintain an index that records which domains the model has learned and maps each concept to its corresponding summary file. After completing the Mathlib4 learning process, for any given formalization task, we first submit the library index to the model and ask which domains from the index are involved in the current natural language statement. The model returns a query result, which corresponds to several summary files. We then present the relevant summaries to the model and it uses the knowledge in these summaries to formalize the input natural language.

2.2 Evaluation Metrics

Mathematically equivalent propositions can have different forms, while small differences in phrasing can lead to completely different meanings. We use Bidirectional Extended Definitional Equivalence (BEq) [6] and LLM evaluation to assess correctness. The LLM evaluation process is to first informalize the formal proposition into natural language, and then determine the equivalence of the two propositions in natural language.

The idea behind BEq is that if the prediction and the ground truth can trivially entail each other, then they are semantically equivalent. Trivially here means using only a limited set of simple strategies, such as renaming variables and unfolding definitions. Since formal language proof assistants often lack strong automation, we use LLM-based tools to perform this mutual entailment. Specifically, we use DeepSeek-Prover-V2-671B [10] as the proving model and Lean 4's official REPL as the interface for model-to-Lean interaction.

Our evaluation pipeline is as follows: for all outputs that conform to Lean 4 syntax, we first attempt to prove them using the `exact?` heuristic. If that fails, we fall back to using the proving model. If this also fails, the output will be handed over to LLM for verification. Any output that passes any step in this pipeline is deemed correct.

It's important to note that this process is still imperfect. The large models BEq relies on may fail to discover proofs, and LLMs are also prone to false positives [6]. In addition, some work [18] is attempting to train an evaluator. Evaluating the correctness of autoformalization still remains one of the main challenges in this field.

3 Experiments

We use the MiniF2F-test dataset [17] as our benchmark. The base model is DeepSeek-R1-0528 [4]. Direct interaction with this model serves as our baseline. We also compare our method with Kimina-Autoformalizer-7B [12], which is currently the best-performing model trained on several aligned datasets between aligned natural language and Lean 4. We tested our translator on Lean version v4.23.0-rc2, which is the latest release as of the time of writing. At the same time, we evaluated the baseline model Kimina-Autoformalizer on v4.9.0, because the baseline model was trained on data from this Lean version, and appendix C shows that its performance has significantly declined on newer versions. Table 1 reports syntactic accuracy and semantic equivalence accuracy for one-shot and five-shot sampling. Syntactic accuracy shows how many model predictions that pass Lean 4 syntax check, and semantic equivalence accuracy shows how many model predictions that are semantic equivalent to the ground truth label. We observe that the performance of the base model after learning the theorem library is even higher than the state-of-the-art fine-tuned model. In particular, the search results can indicate the correct usage of some mathematical concepts, thereby significantly improving syntactic accuracy. We list some case analysis results in appendix B.

Table 1: Results on MiniF2F-test. Our improved approach (DeepSeek-R1-0528+Retrieval) leads to a substantial improvement in accuracy, even surpassing that of fine-tuned models (Kimina-Autoformalizer).

Method	Syn. Pass@1	Sem. Pass@1	Syn. Pass@5	Sem. Pass@5
DeepSeek-R1-0528	75.4%	67.2%	90.6%	87.3%
DeepSeek-R1-0528+Retrieval	93.0%	79.5%	100.0%	96.3%
Kimina-Autoformalizer	88.1%	64.8%	99.6%	90.6%

4 Limitation and Future Work

This is an ongoing project. Some simple techniques can significantly improve performance, such as feeding back error messages from the Lean compiler to the LLM, although this has not been employed in the present work. In the future, we plan to test more base models and larger-scale benchmarks such as ProofNet [2] and Putnambench [11], and we aim to develop a more powerful generation pipeline, including but not limited to:

- Using LeanExplore [1] , LeanSearch [5] or other retrieval methods to extract the most relevant theorem library files;
- Enforcing the autoformalizer's output to follow Lean 4's syntactic norms, such as by filling in concrete syntax trees or simulating the behavior of the Lean Parser, as similar methods described in [9];
- Enhancing BEq metrics to provide more precise error localization, not just binary feedback;
- Exploring formalization methods for mathematical proofs, not just propositions.

References

[1] Justin Asher. Leanexplore: A search engine for lean 4 declarations, 2025.

[2] Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics, 2023.

[3] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation, 2023.

[4] et.al. DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[5] Guoxiong Gao, Haocheng Ju, Jiedong Jiang, Zihan Qin, and Bin Dong. A semantic search engine for mathlib4. *arXiv preprint arXiv:2403.13310*, 2024.

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

[7] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pages 625–635. Springer, 2021.

[8] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[9] Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, et al. A survey on text-to-sql parsing: Concepts, methods, and future directions. *arXiv preprint arXiv:2208.13629*, 2022.

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

[11] George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition, 2024.

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

[13] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368, 2022.

[14] Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075*, 2024.

[15] et.al. Yutong Wu. Stepfun-formalizer: Unlocking the autoformalization potential of llms through knowledge-reasoning fusion, 2025.

[16] Lan Zhang, Marco Valentino, and Andre Freitas. Formalizing complex mathematical statements with llms: A study on mathematical definitions. *arXiv preprint arXiv:2502.12065*, 2025.

[17] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

[18] et.al. Zhongyuan Peng. Criticlean: Critic-guided reinforcement learning for mathematical formalization, 2025.

A Prompts Used in Experiments

Summarizer

```
You are an expert on mathematics and Lean 4 formal language. Your current task is to translate the
     given natural language math problem into Lean 4 language. For example, you probably encounter
     theorem translation tasks like this:
```

```
INPUT: Show that log_{5^2} 5^4 = 2

OUTPUT:
‘‘‘lean
import MiniF2F

open BigOperators Real Nat Topology

theorem log_base_5pow2_5pow4 : Real.logb ((5 : ℝ) ^ 2) ((5 : ℝ) ^ 4) = 2 := by
  sorry
‘‘‘

However, since you are not familiar with the mathematical concepts and corresponding syntax rules
    in the theorem library Mathlib that the latest version of Lean 4 language relies on (for
    example, you may not know logarithms are represented in mathlib as Real.logb), I will provide
    you with a part of the Mathlib theorem library files. You need to summarize what information
    in them you must follow in your future translation work. You can organize these rule sets in a
    text file that you think is appropriate and package them into a file for future reference.

You should follow the template below for summarizing the Mathlib theorem library files:

‘‘‘
# Translation Rules for <math concept>
## Core Definitions
<Definitions>
## Syntax Rules
<Rules>
## Translation Examples
<Examples>
‘‘‘

Here is an example summary for Mathlib.Analysis.SpecialFunctions.Log.Base:

# Translation Rules for Logarithmic Expressions

## Core Definitions
1. Real logarithm base ‘b‘ is represented as ‘Real.logb b x‘
2. Natural logarithm is ‘Real.log‘ (base e)
3. The division ‘log x / log b‘ is equivalent to ‘logb b x‘

## Syntax Rules
1. All real numbers must be explicitly typed as ‘ℝ‘ using ‘(x : ℝ)‘
2. Exponentiation uses ‘^‘ operator (not ‘**‘)
3. For bases and arguments that are numerals or variables:
   - Wrap numerals in parentheses and type them: ‘(5 : ℝ)‘
   - Variables should be declared with type ‘ℝ‘

## Translation Examples
1. Basic logarithmic equality:
   Natural: log 25 = 2
   Lean: ‘Real.logb (5 : ℝ) (25 : ℝ) = 2‘

2. Logarithm with exponentiation:
   Natural: log(5^4) = 4
   Lean: ‘Real.logb (5 : ℝ) (5 : ℝ) ^ 4 = 4‘

3. Logarithmic operations:
   Natural: log(xy) = logx + logy
   Lean: ‘Real.logb (5 : ℝ) (x * y) = Real.logb (5 : ℝ) x + Real.logb (5 : ℝ) y‘
   (with hypotheses ‘hx : x  0‘, ‘hy : y  0‘)

-----------

The demonstration is over, now you have to complete this task. The following code is from
```

Retriever

```
You are a theorem library retriever in a translation system from natural language to Lean 4 code. I
    will provide you with a natural language mathematical proposition and an index that links
    mathematical concepts and the file names of Lean 4 theorem library summary files. Your task is
    to find all the mathematical concepts involved in the given natural language proposition from
    the index and then output the corresponding file names.

Here is a template for a task:

INDEX: <math concept> ||| <file_name>
INPUT: <natural language statement>
OUTPUT:
...Thinking...
<result>
<file_name1>
<file_name2>
</result>

Here is an example:

INDEX: Logarithmic Expressions ||| Mathlib_Analysis_SpecialFunctions_Log_Base
INPUT: Show that $\log_5^2 5^4=2\$.
OUTPUT:
In the given natural language statement, the mathematical concept involved is Logarithmic
    Expressions. According to the index, this concept corresponds to the file name
    Mathlib_Analysis_SpecialFunctions_Log_Base.
<result>
Mathlib_Analysis_SpecialFunctions_Log_Base
</result>

Here is your task:
INDEX:

{index}

INPUT:
{informal_stmt}

OUTPUT:
```

Translator

```
You are an expert on mathematics and Lean 4 formal language. Your current task is to translate the
    given natural language math problem into Lean 4 language. Follow the file header provided
    below, do not re-import libraries and re-open namespaces. You should begin your output with
    "theorem" and give the theorem an appropriate name. Do not give a proof, just fill it with ‘by
    sorry‘ after the theorem statement. Use lean code blocks in markdown format to mark the lean
    proposition. You may need to use the concepts and syntax specifications in the theorem
    library. I have prepared a cheat sheet for you:

{cheat_sheet}

HEADER:
‘‘‘lean
{header}
‘‘‘

TASK TEMPLATE:

Math Proposition:
<natural language math problem>

Lean Proposition:
‘‘‘lean
<Lean 4 code>
‘‘‘

YOUR TASK:

Math Proposition:
{informal_stmt}
```

An Example Summary File: `Mathlib.Data.Int.GCD`

```
# Translation Rules for Integer GCD and Bézout’s Lemma

## Core Definitions
1. Greatest common divisor:
   - ‘Nat.gcd‘ for natural numbers
   - ‘Int.gcd‘ for integers
2. Extended GCD coefficients:
   - ‘Nat.gcdA x y‘ returns coefficient ‘a‘ for ‘x‘
   - ‘Nat.gcdB x y‘ returns coefficient ‘b‘ for ‘y‘
   - ‘Int.gcdA‘ and ‘Int.gcdB‘ handle integer inputs
3. Least common multiple:
   - ‘Nat.lcm‘ for natural numbers
   - ‘Int.lcm‘ for integers

## Syntax Rules
1. Natural number literals:
   - Use ‘(x : ℕ)‘ for natural numbers
   - Example: ‘(5 : ℕ)‘
2. Integer literals:
   - Use ‘(x : ℤ)‘ for integers
   - Negative: ‘(-3 : ℤ)‘
```

```
3. GCD expressions:
   - For naturals: ‘Nat.gcd x y‘
   - For integers: ‘Int.gcd x y‘
4. Bézout’s lemma statements:
   - Use ‘gcd_eq_gcd_ab‘ theorem
   - Format: ‘(gcd x y : ℤ) = x * gcdA x y + y * gcdB x y‘

## Translation Examples
1. Basic GCD property:
   Natural: gcd(6, 9) = 3
   Lean: ‘Nat.gcd (6 : ℕ) (9 : ℕ) = 3‘

2. Bézout’s identity:
   Natural: There exist a,b such that gcd(15,25) = 15a + 25b
   Lean: ‘(Nat.gcd (15 : ℕ) (25 : ℕ) : ℤ) = 15 * Nat.gcdA 15 25 + 25 * Nat.gcdB 15 25‘

3. Integer GCD:
   Natural: gcd(-4, 6) = 2
   Lean: ‘Int.gcd (-4 : ℤ) (6 : ℤ) = 2‘

4. LCM property:
   Natural: lcm(4, 6) = 12
   Lean: ‘Int.lcm (4 : ℤ) (6 : ℤ) = 12‘

5. Divisibility statement:
   Natural: If a divides b*c and gcd(a,c)=1, then a divides b
   Lean: ‘theorem dvd_of_dvd_mul_left_of_gcd_one {a b c : ℤ} (habc : a | b * c) (hab : gcd a c = 1)
     : a | b‘
```

B Case Study on Syntax Repair

Natural Language Problem:

> The greatest common divisor of positive integers m and n is 6. The least common multiple of m and n is 126. What is the least possible value of $m + n$? Show that it is 60.

An example of code that was incorrectly translated by the model:

```
theorem min_sum_of_nats_with_gcd_6_lcm_126 :
(∃ m n : ℕ, 0 < m ∧ 0 < n ∧ gcd m n = 6 ∧ lcm m n = 126 ∧ m + n = 60) ∧
(∀ m n : ℕ, 0 < m → 0 < n → gcd m n = 6 → lcm m n = 126 → 60 ≤ m + n) := by sorry
```

The summaries of `Mathlib.Data.Int.GCD` (presented in Appendix A), `Mathlib.Data.PNat.Prime`, `Mathlib.NumberTheory.Divisors` and `Mathlib.Data.Finset.Basic` are provided to the translator, giving the correct result:

```
theorem gcd_lcm_sum_min :
(∃ m n : ℕ, 0 < m ∧ 0 < n ∧ Nat.gcd m n = 6 ∧ Nat.lcm m n = 126 ∧ m + n = 60) ∧
(∀ m n : ℕ, 0 < m → 0 < n → Nat.gcd m n = 6 → Nat.lcm m n = 126 → 60 ≤ m + n) := by sorry
```

As can be seen, the model has learned the correct syntax of the greatest common divisor ‘`gcd`‘ and the least common multiple ‘`lcm`‘.

C Kinima-Autoformalizer syntax pass rate on different versions of Lean

We examined the output of the baseline model Kinima-Autoformalizer across three Lean versions: v4.9.0 is the dependency version of baseline and the Deepseek Prover series models [10], and it was released one year ago (July 1, 2024); v4.19.0 was released six months ago (March 1, 2025); and v4.23.0-rc2 is the latest version (August 15, 2025). From this, we can clearly observe that newer versions of Lean provide increasingly less support for older syntax. Especially, the most recent version of Lean has completely deprecated the syntax using '`in`' to specify the index range in summation and product operators. This change significantly affects the syntactic accuracy of models trained on the older version of the mathematics library.

Table 2: Kinima-Autoformalizer syntax pass rate on Lean v4.19.0 and v4.23.0-rc2

Lean version	Syn. Pass@1	Syn. Pass@5
v4.9.0	215/244	243/244
v4.19.0	213/244	242/244
v4.23.0-rc2	200/244	237/244