LeanReasoner: Boosting Complex Logical Reasoning with Lean

Dongwei Jiang♡ Marcio Fonseca♢ Shay B. Cohen♢
♡ Johns Hopkins University ♢ University of Edinburgh
djiang21@jhu.edu m.fonseca@ed.ac.uk scohen@inf.ed.ac.uk

Abstract

Large language models (LLMs) often struggle with complex logical reasoning due to logical inconsistencies and the inherent difficulty of such reasoning. We use Lean, a theorem proving framework, to address these challenges. By formalizing logical reasoning problems into theorems within Lean, we can solve them by proving or disproving the corresponding theorems. This method reduces the risk of logical inconsistencies with the help of Lean’s symbolic solver. It also enhances our ability to treat complex reasoning tasks by using Lean’s extensive library of theorem proofs. Our method achieves state-of-the-art performance on the FOLIO dataset and achieves performance near this level on ProofWriter. Notably, these results were accomplished by fine-tuning on fewer than 100 in-domain samples for each dataset.1

1 Introduction

Logical reasoning, a bedrock of intelligence and a core capability of humans, has been a challenging issue for machine learning systems for a long time. LLMs, despite their impressive abilities to understand and generate natural language, often fall short when dealing with complex logical reasoning tasks. They frequently suffer from logical inconsistencies, where the model hallucinates and makes statements not grounded in premises, leading to spurious results (Saparov and He, 2023; Dasgupta et al., 2022).

Recent advances in AI have adopted a structured approach to tackle these reasoning problems by splitting them into symbolic formalization and problem-solving (He-Yueya et al., 2023; Pan et al., 2023; Ye et al., 2023). Specifically, the formalization step is often handled by a large language model, while problem-solving is handled by an off-the-shelf symbolic solver. In this approach, symbolic solvers essentially act as a rigorous check-point, ensuring that the model outputs align with logical rules, thereby mitigating the issue of logic inconsistency. In these approaches, solvers may range from being completely deterministic, like SymPy (He-Yueya et al., 2023), or relying on a combination of heuristics and basic machine learning techniques, as is the case with Pyke (Pan et al., 2023) and Z3 (Ye et al., 2023; de Moura and Bjørner, 2008). While this approach successfully addresses hallucinations, it still struggles with more complex problems.

As a powerful theorem prover and a versatile programming language, Lean (de Moura et al., 2015) presents a compelling solution to connect symbolic solvers with linguistic resources. Much like symbolic solvers, Lean has a strict checking system that ensures each reasoning step is certified. What distinguishes it, however, is its additional functionality as a programming language developed specifically for theorem proving. Every day, a substantial amount of code is written in Lean, capturing reasoning “nuggets” with step-by-step rationals that are useful for training LLMs. A few recent studies have already tapped into Lean for mathematical theorem-proving tasks (Polu et al., 2023; Han et al., 2022a; Lample et al., 2022), showing its potential in tackling difficult reasoning challenges.

In this paper, we propose LeanReasoner, a Lean-based framework to tackle logical reasoning problems. We use LLMs to formalize natural language context into Lean and fine-tune a custom model on these problems using a modest amount of data annotated ourselves. As we use LLMs to dynamically generate solutions within the Lean environment, our approach stands in stark contrast to the static, pre-defined solution-finding methods of LogicLM (Pan et al., 2023), which only rely on traditional techniques like forward and backward chaining, and SATLM (Ye et al., 2023), which operates within the Z3 environment using a suite of prede-

1Our code and data is available at https://github.com/Some-random/theorem-proving-reasoning.
terminated algorithms and heuristics. The adaptive nature of LLMs as a solution-finding tool allows our system to evolve continuously, harnessing a vast array of reasoning data and information.

Our contributions in this paper are three-fold.

• To our knowledge, this is the first attempt to use Lean, traditionally associated with mathematical theorem proving, for natural language logical reasoning. This effort highlights a possible intersection between mathematical theorem proving and logical reasoning.
• Our research revealed that incorporating pre-training data from mathematical theorem proofs enhances the development of a more effective solver for logical reasoning compared to previous techniques. Additionally, this approach enabled us to achieve SOTA results on FOLIO.
• We make available the training data accumulated in this research, comprising 100 Leanformalized logic reasoning problems from ProofWriter, along with 27 analogous formalizations from FOLIO. The corresponding proofs in Lean are also included.

2 Problem Definition and Notation

The task we aim to solve is logical reasoning, taking the form of multi-choice questions given a natural language context. The answer to the question can be logically deduced based on the context. The framework we use for solving the problem is Lean.² Lean is an open-source theorem-proving programming language with vibrant community support. Its current base includes over 100,000 theorems and 1,000,000 lines of code.³ We use Lean as a generic theorem prover, outside of mathematics.

The task and our solution to it, consist of the following components:

• **Context**, which is composed of natural language utterances, composing a set of rules and facts. For example: *Hudson is a cat, all cats are animals, and cats often meow.*

• **Question**, which denotes the posed question. For example, *Does Hudson often meow?*

• **Options** is a set of available answers (discrete categories) from which an answer can be chosen. For example, *True, False or Unknown.*

• **Formalized context** refers to the representation of context in Lean. For example, the formalized context for our example would be: *axiom A1 is_cat Hudson, axiom A2 ∀x, is_cat x → is_animal x and axiom A3 ∀x, is_cat x → often_meow x.*

• **Formalized question**: Given that Lean operates as a theorem prover, questions are transformed into dual theorems: one asserting the positive stance and the other negating it. For the given example, the formalized questions would be: *Theorem hudson_often_meows: often_meow Hudson and Theorem not_hudson_often_meows: ¬ often_meow Hudson.*

• **Goal**: In the context of proving theorems with Lean, a ”goal” is a logical statement that needs to be proven true, given a set of axioms and rules. When we set out to answer a question using the Lean prover, this question becomes our root goal. At that point, we can apply various instructions in Lean to simplify or break down this primary goal and generate intermediate goals.

For instance, using our earlier examples, if the root goal is proving *Theorem hudson_often_meows: often_meow Hudson*, an intermediate goal might be proving that *Hudson is a cat*. We aim to resolve each intermediate goal using our provided context, gradually working our way towards proving the root goal. Once all intermediate goals are resolved, we have effectively proven our root goal, and the proof search concludes successfully.

• **Tactics** are the instructions in the Lean theorem proving language that are used to manipulate goals to obtain a proof for a given goal. For example, *apply A3 Hudson* is a tactic that uses modus ponens on the **Goal** *often_meow Hudson* and transforms it to a new **Goal** *is_cat Hudson*.

A diagram of these components and the relations between them is depicted in Figure 1. This procedure is framed within the language of the Lean theorem prover as a goal-satisfying process.

3 LeanReasoner

Our framework, LeanReasoner, is composed of four main components: a **formalizer**, a **tactic generator**, a **proof search** mechanism, and a **result interpreter**. The formalizer converts the context and question to formalized context and formalized
question. The tactic generator then generates tactics based on premises extracted from the formalized context. The proof search mechanism oversees tactic execution and goal expansion. The result interpreter analyses the output of the proof search and identifies the correct answer in the options. In this section, we detail each of those components.

### 3.1 Formalizer

As formalizers, we used OpenAI models text-davinci-003 (GPT-3) and GPT-4 (OpenAI, 2023). For text-davinci-003, we followed the same prompting approach as Logic-LM (Pan et al., 2023) to separate task specifications and problems, thereby enabling the model to continue with the task of formalization through next-token-prediction. For GPT-4, we used similar prompts but included the task specification in the system prompt.

There is no automatic way to assert all the entities, relationships, and constraints of the context have been captured by the formalized result. However, the syntax of the formalized result can be checked by Lean. Because correct syntax is a prerequisite for downstream theorem proving, if an error is encountered during compilation, we provide the error message generated by Lean along with the faulty formalization and ask the formalizer to regenerate the result. We further manually inspect the formalizer in §5. We note that we take a strict approach, and if the formalizer fails more than once, then the problem is counted as not being correctly solved.

### 3.2 Tactic Generator

The model we used for tactic generation is ReProver (Yang et al., 2023). This model contains two parts: a retriever that employs retrieval mechanisms to explicitly select premises when provided with the current goal, and a generator that generates tactics using the goal and the retrieved premises.

The division of the problem-solving task into premise selection and tactic generation simplifies the process and facilitates easier troubleshooting. It isolates the source of potential issues, be it in the premise selection or the tactic generation, thus reducing the complexity of the problem. Also, this division of responsibilities eases the burden on the tactic generator. Choosing the right premise with numerous distractions is challenging, especially in logical reasoning problems when several options might seem promising for the current step but will not ultimately lead to the desired goal.

The premise retrieval component of our process draws from the Dense Passage Retriever (DPR) (Karpukhin et al., 2020). Provided with a goal $g$ as the query and a set of candidate premises $P$, it generates a ranked list of $m$ premises from $P$. In DPR, both $g$ and $P$ are treated as raw texts that are embedded in a vector space. We then retrieve the top $m$ premises that maximize the cosine similarity between the goal and the premise. For tactic generation, we use a standard sequence-to-sequence model. The goal and the premises are concatenated together as a string to generate new tactics.

As a baseline, we also prompt GPT-4 to generate proofs. For cases when the chosen theorem to prove aligns with the answer (say the chosen theorem is the positive stance of the question and the answer is $Y\hat{E}S$), we present GPT-4 with the correct proof as part of the prompt. Conversely, if the answer does not align with the chosen theorem or the answer is $N\hat{O}$ $G\hat{a}l$ $o$s.
UNKNOW, the formalized theorem is unprovable. In those cases, we still encourage the model to engage in step-by-step reasoning, even though it will eventually hit a roadblock. An example of the prompt to GPT-4 can be found in Appendix A.1.

3.3 Proof Search

The proof search module controls the overall search process that selects tactics and maintains states during proof construction. Essentially, the goal of the search method is to build a proof tree that incrementally evolves the goal through tactic invocations. This approach was first introduced in GPT-F (Polu and Sutskever, 2020). LeanDoJo (Yang et al., 2023), a recently released framework that enables interaction with Lean programmatically, subsequently provided an implementation of this method, which we use for our study.

As a reference, the middle part of Figure 1 provides a practical illustration of this process. Starting from the root goal, for each given proof goal, we explore 64 possible tactics. All goals are maintained in a priority queue and expanded based on cumulative log probabilities of the goal. The cumulative log probability is defined as the summation of the log probabilities of the tactics that brought us to the goal from the root. This implies that we tend to expand those goals where our generative model has the highest global confidence.

To enhance search efficiency and circumvent potential loops, we have incorporated a mechanism that stops the expansion of a node $N$ if we have already explored another node $M$ with a state sequence that prefixes $N$. Essentially, if the current goal being explored contains all the elements of a previously explored goal, then it shouldn’t be further expanded. This is based on the observation that if we have already assessed the potential paths and outcomes for a specific goal, then exploring a more generalized version of the same goal is redundant. Such a mechanism avoids unnecessary repetitions, which streamlines the search process and improves the overall efficiency. Moreover, we define a valid proof as one that is devoid of “cheating” tactics (such as sorry) that tell Lean to assume that the current goal is completed, even though it has not been proven. This means that every path containing “cheating” tactics is disregarded.

Errors in the search process typically manifest in two ways: a timeout or an exhaustion of nodes to search. We have allocated a three-minute window for each search, which is usually sufficient. We provided more analysis of the errors made by tactic generator in the experiment section.

3.4 Result Interpreter

If the correct answer is Unknown, we only regard the result as correct if neither True nor False can be proven. All datasets investigated in this study only contain questions with only one correct answer. Consequently, if the proof system verifies more than one option, the response is immediately marked as incorrect.

4 Experimental Setup

We now describe our experimental setup: the datasets we used for evaluation and model training and the details of model training.

4.1 Evaluation Data

In our evaluation, we use two common logical reasoning datasets as testbeds:

**ProofWriter:** This deductive logical reasoning dataset presents problems in an intuitive language form. We incorporated the Open-World Assumption (OWA) subset as per (Pan et al., 2023), where each instance is characterized by a (problem, goal) pairing. The label for each pair contains True, False, or UNKNOW. It encompasses five segments based on the required reasoning depth. Our focus is the depth-5 subset, which is the most challenging one. To get a fair comparison against LogicLM, we used the same 600 sample tests, ensuring an even label distribution.

**FOLIO:** Unlike ProofWriter, FOLIO is constructed using first-order logic. This increases the complexity of the proving part. The dataset also presents problems in a more natural wording, with relationships that are considerably more complex. Such a combination of advanced logic and rich linguistic structure makes the formalization task in FOLIO substantially tougher than in ProofWriter. For our analysis, we turned to the entire FOLIO test set, encompassing 204 examples.

4.2 Training Data for Domain Adaptation

Regarding the data for model training, we collected 100 theorem proofs for ProofWriter and 27 theorem proofs for FOLIO, where each problem’s proof was either manually annotated or collected from successful proofs generated by GPT-4. The data collection took about eight days.
During data annotation, we adopted two divergent approaches for constructing proofs. One approach emulated a straightforward strategy, encompassing a detailed procedure with all of the intermediate steps and lemmas, similar to how we humans might derive proof when given theorem-proving tasks. Conversely, the second approach resembles the proof formats found in mathlib. We generate more succinct proofs of the same problem by reducing the number of intermediate lemmas and combining multiple tactics into a single compound tactic. The objective of having two annotations for the same problem was to examine the influence of annotation style on downstream logical reasoning. In the following experiments, we use Intuitive to refer to the first annotation style and Concise to denote the second annotation style. An illustrative example is available in Appendix C.

It is important to mention that despite the limited data collected, the reasoning patterns for logical reasoning likely mirror those found in mathematical reasoning, which were potentially learned during pretraining. The main purpose of this data collection is domain adaptation to transfer from math to natural language logical reasoning.

4.3 Model Training

We used the same model structure for pretraining as in the ReProver paper, namely, Google’s Byte-T5 (Xue et al., 2022). We also experimented with the pretrained ReProver from LeanDoJo (Yang et al., 2023), which was pretrained on mathlib. The fine-tuning of our collected data took about six hours on one A100 40G. The hyperparameters are the same as in the original LeanDoJo paper.

5 Results

We present our experimental results, including an examination of prompting-based baseline, experimental results for LeanReasoner, and a comparison between our work and other baselines.

5.1 Prompting-Based Baselines

Since there is no automated method to verify the accuracy of formalization, we conducted manual examinations of the formalized results to determine whether errors occur during formalization or proof generation stages. In this examination, only formalizations that correctly capture every fact, axiom, and rule are counted as accurate. We manually examined 100 questions from ProofWriter’s validation set and 40 questions from FOLIO’s training set. The findings have been summarized in Table 1.

Comparison of formalization accuracy. The formalization accuracy of ProofWriter is much higher than FOLIO. This can be attributed to its simpler language structure. In the case of FOLIO, although using LLM for formalization helped in filtering out unnecessary details from the natural language context, there still exists some common error patterns. We have illustrated typical GPT-4 error patterns in Appendix B using a composite sample derived from various error instances. Interestingly, Lean’s formalization accuracy is on par with both Prolog and FOL in Logic-LM. This consistency underscores Lean’s versatility, allowing it to uniformly represent different problem types within a single framework.

Adding textual comments increases formalization accuracy. We observed improved results when formalized code was paired with descriptive textual comments (example in Appendix A.1) sourced from the context. This approach further splits the formalization task into two subtasks: 1) linking textual context with formalized code and 2) generating formalized code based on the prior textual context. These textual cues acted as a bridge between raw text and formalized code, enhancing the performance of formalization.

GPT-3’s performance on formalization is worse than GPT-4. The distinction in performance between GPT-3 and GPT-4 is evident. While the formalization for simpler problems is the same, GPT-3 struggles with intricate logic and complex problems. As such, we opted not to use GPT-3 in further tests. Additionally, we experimented with the CodeLLAMA (Baptiste Rozière and et.al, 2023) model family for similar tasks, but found that their accuracy in formalization was significantly lower than that of GPT-3, achieving less than 30% on ProofWriter.

The proof accuracy of prompting-based baseline is very low. The proof accuracy section of the table is determined by whether the generated proof can be validated successfully in Lean. If the formalization of the question as a theorem is correct and the proof can be validated without any error or warning, then we can treat the proof as
valid. However, the accuracy of rendered proofs is very low. The issue could stem from assigning too many tasks to the large language model, making it challenging to address both within a single prompt. Despite our efforts to separate formalization and proof, the results were still disappointing, which highlights GPT-3 and GPT-4’s struggle with generating correct Lean proof. Interestingly, the proof accuracy of Logic-LM wasn’t as high as we expected. Upon replicating their code, we found their chosen solver Pyke to be suboptimal, struggling to identify an answer when multiple search paths are available and some could result in loops.

The answer accuracy of prompting-based baseline is surprisingly high. Despite the low accuracy in most of GPT-4’s proofs, it achieved high accuracy for final choices on ProofWriter (as shown in column Answer). We believe this may be due to GPT-4’s training exposure to the dataset, potentially leading to a degree of memorization.

### 5.2 LeanReasoner

In this section, we focus on training our own models to do tactic generation using our annotated training data. To isolate the impact of erroneous formalization, we only used the accurate formalizations from the previous subsection for testing. This gave us 99 test examples for ProofWriter and 28 for FOLIO. All findings are detailed in Table 2.

**Fine-tuning on annotated data increases premise selection accuracy.** We first compare the results on premise selection using the metrics recall@1 and recall@4. The recall@k metric is defined as follows:

$$\text{recall@k} = \frac{|\text{GT}_\text{Prem} \cap \text{Pred}_\text{Prem}[0:k]|}{|\text{GT}_\text{Prem}|},$$

where GT_Prem means ground truth premises and Pred_Prem means top predicted premises. The suboptimal results of LeanReasoner pretrained solely with math data may be attributed to the domain mismatch between mathematical theorem proving and logical reasoning. The model frequently makes mistakes by attempting to use other, unrelated tactics that are useful in mathematical theorem proving (like `ring`, `linarith`) but not applicable in logical reasoning. Furthermore, the accuracy for FOLIO was noticeably poorer than that of ProofWriter. This disparity is likely due to FOLIO’s intricate logic and its need for a broader array of first-order logic tactics such as `cases`, `have`, and `contradiction`. In contrast, ProofWriter primarily employs tactics like `apply`, `exact`, and `split`.

**Pretraining on theorem proving data increases overall accuracy.** Regarding the overall proof results, LeanReasoner pretrained on math theorem proving data consistently outperformed other approaches for both ProofWriter and FOLIO datasets. This success indicates that our model effectively uses the logical “nuggets” found in mathematical theorem proofs. While the premise selector benefits from distinct cues and a limited range of choices, the realm of tactic generation is much broader. This vastness of options renders the ReProver baseline’s proof accuracy nearly negligible. But other than that, there is a strong correlation between premise selection accuracy and overall proof accuracy. While the benefits of a pretrained LeanReasoner may not be as noticeable for simpler datasets like ProofWriter, its value becomes evident for more complex datasets, such as FOLIO.

**Concise annotation gives better result on premise selection.** Fine-tuning with different an-
notations has a slight effect on premise selection and tactic generation in this small test set. When fine-tuned with Concise annotations, LeanReasoner would also try to generate concise proofs, which usually use compound tactics that offer more information for premise selection. However, the final proof accuracy has not changed on this small test set. Figure 2 displays an example of proofs for the same question, produced by the three primary methods we compared. In the absence of pretraining, the model struggles to identify an appropriate approach for solving the problem. It merely attempts to apply the next applicable theorem, lacking a clear objective. While Intuitive data offers numerous lemmas that assist in the thought process during proof-writing, these excessive lemmas do not aid LLMs in generating tactics effectively.

5.3 Other Baselines

Having demonstrated that pretraining on theorem-proving data yields superior performance, we proceed to benchmark our results against established baselines for both ProofWriter and FOLIO. The evaluation uses the same set of 600 problems from LogicLM and the entire FOLIO test set.

Our approach yields near-perfect accuracy on ProofWriter with significantly less data. As illustrated in Table 3, our approach yields near-perfect accuracy on the ProofWriter dataset. While other methods except Logic-LM and GPT-4 CoT use the entire training set of ProofWriter, our approach relies on just 100 examples, underscoring the efficiency of our method. Fine-tuning on Concise annotation does not bring any advantage to the final performance on this dataset.

<table>
<thead>
<tr>
<th>Method</th>
<th>Pretrained on Math Data</th>
<th>Fine-tuned on our Annotation</th>
<th>ProofWriter</th>
<th>FOLIO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Premise Selection</td>
<td>Proof Accuracy</td>
</tr>
<tr>
<td>GPT-4</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>15%</td>
</tr>
<tr>
<td>LeanReasoner</td>
<td>Yes</td>
<td>No</td>
<td>56.2%</td>
<td>100%</td>
</tr>
<tr>
<td>LeanReasoner</td>
<td>No</td>
<td>Intuitive</td>
<td>62.5%</td>
<td>100%</td>
</tr>
<tr>
<td>LeanReasoner</td>
<td>Yes</td>
<td>Intuitive</td>
<td>75%</td>
<td>100%</td>
</tr>
<tr>
<td>LeanReasoner</td>
<td>Yes</td>
<td>Concise</td>
<td>75%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 2: Comparative Analysis of Recall@k in premise selection and overall proof accuracy for 99 ProofWriter test samples and 28 FOLIO test samples. Note the proof accuracy here is different from Table 1 because it is directly linked to the final accuracy. The effects of pretraining and fine-tuning on LeanReasoner are evaluated using theorem-proving data and both Intuitive and Concise annotation sets, respectively. Premise Selection accuracy was not calculated for the GPT-4 baseline due to the complexities in prompting GPT-4 with Lean goals.

<table>
<thead>
<tr>
<th>Method</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full training set method</td>
<td></td>
</tr>
<tr>
<td>Abs Biases (Gontier et al., 2022)</td>
<td>80.6%</td>
</tr>
<tr>
<td>MetaInduce (Yang et al., 2022)</td>
<td>98.6%</td>
</tr>
<tr>
<td>RECKONING (Chen et al., 2023b)</td>
<td>99.8%</td>
</tr>
<tr>
<td>Zero-shot method</td>
<td></td>
</tr>
<tr>
<td>GPT-4 CoT (Pan et al., 2023)</td>
<td>68.1%</td>
</tr>
<tr>
<td>Logic-LM (Pan et al., 2023)</td>
<td>79.3%</td>
</tr>
<tr>
<td>Our method (finetuned on 100 samples)</td>
<td></td>
</tr>
<tr>
<td>LeanReasoner without Pretraining</td>
<td>95.8%</td>
</tr>
<tr>
<td>LeanReasoner fine-tuned on Intuitive</td>
<td>98.3%</td>
</tr>
<tr>
<td>LeanReasoner fine-tuned on Concise</td>
<td>98.3%</td>
</tr>
</tbody>
</table>

Table 3: The fine-tuned LeanReasoner has been pretrained on mathlib. Full training set method means the model has been trained on the full training set of ProofWriter. Fine-tuning on Concise achieves near-perfect accuracy with significantly less data.

<table>
<thead>
<tr>
<th>Method</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full training set method</td>
<td></td>
</tr>
<tr>
<td>Roberta (Han et al., 2022b)</td>
<td>62.1%</td>
</tr>
<tr>
<td>FOLNet (Chen, 2023)</td>
<td>70.6%</td>
</tr>
<tr>
<td>Zero-shot method</td>
<td></td>
</tr>
<tr>
<td>GPT-4 CoT (Pan et al., 2023)</td>
<td>70.6%</td>
</tr>
<tr>
<td>Logic-LM (Pan et al., 2023)</td>
<td>74.5%</td>
</tr>
<tr>
<td>Lean Z3 (SATLM)</td>
<td>77.5%</td>
</tr>
<tr>
<td>Our method (finetuned on 27 samples)</td>
<td></td>
</tr>
<tr>
<td>LeanReasoner without Pretraining</td>
<td>66.2%</td>
</tr>
<tr>
<td>LeanReasoner fine-tuned on Intuitive</td>
<td>78.4%</td>
</tr>
<tr>
<td>LeanReasoner fine-tuned on Concise</td>
<td>82.6%</td>
</tr>
</tbody>
</table>

Table 4: Result from “Lean Z3” is derived from lean-smt applied to formalized Lean Code. The fine-tuned LeanReasoner has been pretrained on mathlib. Full training set method means the model has been trained on the full training set of ProofWriter. Our approach achieves state-of-the-arts performance on FOLIO.
Our approach achieves state-of-the-arts performance on FOLIO. Table 4 presents our performance on FOLIO. For a fair comparison with SATLM that uses the Z3 solver, we used the lean-smt tool\(^5\) on our formalized Lean code. This tool produces outcomes in the form of “sat/unsat”. In Z3, “sat” stands for “satisfiable.” When Z3 returns “sat” as the result, it means that there exists a set of variable values that makes the theorem true. On the other hand, “unsat” stands for “unsatisfiable”. When Z3 returns “unsat”, it means that the formula is inherently contradictory and cannot be satisfied under any circumstance. We interpret these results similarly to “found a proof/didn’t find a proof” using our result interpreter. Due to the extensive length of proofs for FOLIO problems, we observed that when LeanReasoner is fine-tuned on the Intuitive dataset, it often allocates an excessive amount of time for exploration and occasionally enters loops. In contrast, generating shorter proofs tends to ease the discovery of the proof. While the tactics generated when fine-tuned on the Concise dataset are more challenging to produce, the bottleneck for LeanReasoner on FOLIO resides in the search process.

Challenges in Benchmarking. It is important to acknowledge that there can be scenarios where errors in problem formalization or proof generation may occur, yet the final answer is still deemed correct. A case in point is when the answer to a problem is Unknown, and errors arise in these stages. In such instances, the model would struggle to prove either the positive or negative theorem. However, with our result interpreter, these instances would still be classified as correct despite the underlying issues in problem handling.

6 Related Work

Combining LLM with symbolic solver. Several past studies (Chen, 2023; Creswell and Shanahan, 2022; Chen et al., 2023b) used symbolic solvers to augment neural networks with logical reasoning. Many of these approaches have limitations, like the necessity for custom or specialized module designs that lack broad applicability. Recent work (Pan et al., 2023; Ye et al., 2023; Poesia et al., 2023; Olausson et al., 2023) presents a more general framework that combines contemporary LLMs with symbolic logic, bypassing the need to train or craft intricate modules tailored for specific problems. While our research aligns with these, we do not exclusively rely on off-the-shelf solvers.

Boosting the reasoning skill of LLM by training on reasoning data. A common way to boost the reasoning skills of LLMs is by training them on data that requires some form of reasoning. As noted by Lewkowycz et al. (2022), LLMs trained with science and math data do better on tasks that require reasoning, especially when using CoT.

---

\(^5\)https://github.com/ufmg-smite/lean-smt
prompting. Other results by Fu and Khot (2022) and Fu et al. (2023) suggest that powerful LLMs obtain advanced reasoning capabilities from being trained on code. This work is an extension of this idea to theorem proving.

7 Conclusion

We introduced LeanReasoner, a framework based on Lean that augments the logical reasoning abilities of LLMs. We follow an extensive examination of errors from the formalization and proof generation stages that are present in our framework. We also examine the performance enhancements from pretraining on theorem-proving data and annotation styles (concise v.s. intuitive). We offered a comprehensive comparison with other techniques that highlight the strengths of our model. Our results underscore the potential of integrating theorem-proving frameworks with LLMs in advancing logical reasoning.

Limitations

Despite our promising results, our method encounters limitations when dealing with problems that involve commonsense and factual reasoning. In these cases, it is challenging to retrieve all the necessary information and accurately represent it in Lean. Consider MMLU (Hendrycks et al., 2020) and SummEdits (Laban et al., 2023): MMLU requires the model to possess extensive world knowledge, while SummEdits involves determining consistency in summaries of different edits. In both instances, the ability to represent the complexity and nuance of real-world knowledge in Lean is severely limited.

Further complications arise when dealing with math word problems (Cobbe et al., 2021) and similar tasks (Hendrycks et al., 2021), where the goal is to derive a numeric solution rather than a proof. The theorem-proving approach, while effective for certifying the validity of logical reasoning, does not directly yield a numerical answer. Lastly, our method grapples with problems found in more complicated reasoning datasets like TheoremQA (Chen et al., 2023a). These problems require an advanced understanding of complex concepts and the ability to formalize these concepts into Lean. Our current framework struggles with this level of complexity, underscoring the need for more sophisticated formalization techniques and a deeper integration between language understanding and theorem proving.

Even in the context of symbolic problems, there are challenges. For instance, consider the LogicalDeduction task from the BigBench dataset (Srivastava et al., 2022). Although this problem appears straightforward, employing Lean to solve it is neither the most practical nor the most efficient approach. Lean, as a theorem prover, is excellent in abstract reasoning and proof construction, but when faced with tasks involving constraints and variable possibilities, it falls short. To solve the problems in LogicDeduction, using Lean would require us to formalize the concepts of ordering and relative positioning. Even after doing so, generating proof would necessitate significant labor and wouldn’t necessarily yield a readily interpretable answer. In contrast, a Constraint Satisfaction Problem (CSP) solver can effectively manage constraints and generate potential solutions efficiently.

Ethical Considerations

Incorporating Lean’s theorem-proving capabilities into LLMs offers a layer of mathematical rigor that improves the reliability of conclusions derived. However, LLMs are known to be susceptible to data biases, which may manifest in critical applications. This issue can inadvertently lead to skewed logic or unintended bias in sensitive domains such as medical diagnoses or legal interpretations. While our method’s foundation in Lean’s theorem proving data acts as a rigorous check, complete reliance on it is not foolproof. A proactive approach in reviewing both training data and model outcomes is essential to uphold unbiased reasoning.

Acknowledgements

We thank the reviewers and members of the Cohort for the valuable feedback and comments on the paper. We appreciate the provision of computing resources through the Baskerville cluster at the University of Birmingham.
References


Wenhui Chen, Ming Yin, Max Ku, Pan Lu, Yixin Wan, Xueguang Ma, Jianyu Xu, Xinyi Wang, and Tony Xia. 2023a. Theoremqa: A theorem-driven question answering dataset.


Hao Fu, Yao; Peng and Tushar Khot. 2022. How does gpt obtain its ability? tracing emergent abilities of language models to their sources. Yao Fu’s Notion.


Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. 2022a. Proof artifact co-training for theorem proving with language models. In ICLR.


OpenAI. 2023. GPT-4 technical report. CoRR.

Liangming Pan, Alon Albalak, Xinyi Wang, and et.al. 2023. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning.


A Prompts for Formalization

A.1 Prompts for ProofWriter

In subsection 5.1, we discussed various formalization approaches. In this section, we present the results using the GPT-4 Base Comments method on ProofWriter when the answer is False. As evident from the last line, the predicted outcome from GPT-4 can be derived easily.

System Message:
You are a logician with a background in mathematics that translates natural language reasoning text to Lean code so that these natural language reasoning problems can be solved. During the translation, please pay close attention to the predicates and entities. There is an additional requirement: I also want you to try to prove the theorem you translated to Lean. If you can prove the theorem, give me True at the end of the answer. If you can prove the negation of the theorem, write False at the end of the answer. If you can neither prove the original theorem nor the negation of the theorem, please give me Unknown at the end of the answer.

Input:
Textual context: The cat is blue. The cat is nice. The cat likes the cow. The cat needs the cow. The cat needs the rabbit. The cow does not need the rabbit. The cow needs the tiger. The rabbit is round. The tiger is nice. The tiger does not like the cow. If someone likes the cat and they chase the cat then they are blue. If someone likes the cow and they are red then the cow is round. If someone needs the tiger and they need the cat then they do not chase the cow. If someone needs the cat and the cat is blue then the cat is red. If someone is round then they need the cat. If someone likes the cat and the cat needs the rabbit then the cat chases the tiger. If the rabbit needs the tiger then the rabbit is not round.

Question: Based on the above information, is the following statement true, false, or unknown? The cow chases the cow.
A.2 Prompts Used for FOLIO

For FOLIO, the prompts differ slightly from those used for ProofWriter. Since FOLIO's textual context remains consistent across multiple questions, we concatenate questions sharing the same context and prompt the large language model in a single instance. An illustrative example is provided below.

---

The negation of the theorem can be proven in Lean. So
---
The answer is False

The theorem cannot be proven in Lean. So let's try to prove the negation of the theorem instead.

The answer is True
Output:
-- Introducing basic types for entities in our
universe.
constant Turkey : Type
constant Tom : Turkey
constant Joey : Turkey
-- Defining properties for the turkeys in our
universe.
constant is_eastern_turkey : Turkey → Prop
constant is_osceola_turkey : Turkey → Prop
constant is_merriams_turkey : Turkey → Prop
constant is_rio_grande_turkey : Turkey → Prop
constant is_goulds_turkey : Turkey → Prop
constant is_ocellated_turkey : Turkey → Prop
-- Context from the information
constant is_ocellated_wild_turkey : Turkey
constant is_rio_grande_wild_turkey : Turkey
constant is_merriams_wild_turkey : Turkey
constant is_goulds_wild_turkey : Turkey
constant is_osceola_wild_turkey : Turkey
constant is_eastern_wild_turkey : Turkey
constant is_wild_turkey : Turkey
-- Formalizing the Questions
axiom turkey_type_exclusivity : ∀ (t : Turkey), bites t ∨ stings t
axiom A6 : bites Bob
axiom A7 : ∀ (t : Turkey),
eastern_bite_causes_death t
13
begin
-- Let's try the negative case.
theorem not_tom_is_ocellated_wild_turkey : ¬
is_ocellated_wild_turkey Tom :=
begin
exact A2,
end
-- The answer is Unknown
-- Question 2
begin
end
-- Let's try the negative case.
theorem not_tom_is_eastern_wild_turkey : ¬
is_eastern_wild_turkey Tom :=
begin
exact A1,
end
-- The answer is False
-- Question 3
begin
end
-- Let's try the negative case.
theorem not_joey_is_wild_turkey : ¬ is_wild_turkey Joey :=
begin
sorry, -- Placeholder for proof
end
-- The answer is Unknown
end
begin
theorem tom_is_eastern_wild_turkey :
begin
end
-- Let's first prove the positive case.
-- Tom is an Eastern wild turkey.
-- Question 2
begin
end
-- The answer is Unknown
begin
end
-- Let's first prove the positive case.
-- Tom is a wild turkey.
begin
end
-- Context from the information
-- Tom is not an Eastern wild turkey.
axiom A1 : ¬ is_eastern_wild_turkey Tom
-- Tom is not an Osceola wild turkey.
axiom A2 : ¬ is_osceola_wild_turkey Tom
-- Tom is also not a Gould's wild turkey.
axiom A3 : ¬ is_goulds_wild_turkey Tom
-- Tom is also not a Merriam's wild turkey.
axiom A4 : ¬ is_merriams_wild_turkey Tom
-- Tom is not a Rio Grande wild turkey.
axiom A5 : ¬ is_rio_grande_wild_turkey Tom
-- There are six types of wild turkeys: Eastern
wild turkey, Osceola wild turkey, 'Goulds wild
turkey, 'Merriam's wild turkey, Rio Grande wild
turkey, and Ocellated wild turkey.
axiom type_exclusivity : ∀ (t : Turkey),
is_wild_turkey t → is_eastern_wild_turkey t ∨ is_osceola_wild_turkey t ∨
is_goulds_wild_turkey t ∨ is_merriams_wild_turkey t ∨
is_rio_grande_wild_turkey t ∨ is_ocellated_wild_turkey t
-- Formalizing the Questions
-- Question 1
begin
end
-- Tom is an Ocellated wild turkey.
theorem tom_is_ocellated_wild_turkey :
begin
end
-- Let's try the negative case.
theorem not_tom_is_eastern_wild_turkey :
begin
end
-- The answer is Unknown
begin
end
-- Question 2
begin
end
-- Tom is a wild turkey. Bob is an Eastern turkey.
begin
end
-- The answer is False
begin
end
-- Question 3: If Bob bites you, you will die.
Theorem 3: If Bob bites you, you will die.

B Example Errors During Formalization
This is an artificial example highlighting typical
errors encountered in the formalization stage.

Input:
There are three types of wild turkeys: Eastern
turkey, Osceola turkey, Gould's turkey. Tom is
neither an Eastern turkey nor an Osceola turkey. Tom
is a wild turkey. Bob is a Eastern turkey.
Bob weights 86 pounds and Tom weights 73 pounds.
Turkeys either bites or stings. Bob bites. Eastern
turkey’s bite would cause death.

Question 1: Tom is a Gould’s turkey.
Question 2: Bob is heavier than Tom.
Question 3: If Bob bites you, you will die.

Output:
constant Turkey : Type
constant Eastern Osceola Goulds : Turkey → Prop
constant weight_86 weight_73 : Turkey → Prop
constant bites stings : Turkey → Prop
constant eastern_bite_causes_death : Turkey → Prop
constant Tom : Turkey
axiom A1 : ¬ Eastern Tom ∧ ¬ Osceola Tom
axiom A2 : Eastern Bob
axiom A3 : weight_86 Bob
axiom A4 : weight_73 Tom
axiom A5 : ∀ (t : Turkey), bites t ∨ stings t
axiom A6 : bites Bob
axiom A7 : ∀ (t : Turkey),
eastern_bite_causes_death t
There are a few errors in the above code, including:

- There is a missing axiom that corresponds to “There are three types of wild turkeys”:
  Goulds Tom ∨ Eastern Tom ∨ Osceola Tom

- The formalization of numbers is incorrect, it should be:
  \[ \text{constant weight : Turkey} \to \mathbb{N} \]
  \[ \text{axiom A3 : weight Bob = 86} \]
  \[ \text{axiom A4 : weight Tom = 73} \]

- The formalization of logic is incorrect, it should be:
  \[ \neg \text{bite\_causes\_death}\ t \land \neg \text{bite\_causes\_itching}\ t \lor \neg \neg \text{bite\_causes\_death}\ t \]

- There is an incorrect division of concepts that would make the proving impossible, the correct version should be:
  \[ \forall (t : \text{Turkey}), \text{Eastern}\ t \land \text{bite\_causes\_death}\ t \]

C Example Proof Annotation with Different Annotation Styles

Here we’re showing two example proofs created on the same problem with ‘Intuitive’ annotation style and ‘Concise’ annotation style.

Input:
"Textual Context": All eels are fish. No fish are plants. A thing is either a plant or animal. Nothing that breathes is paper. All animals breathe. If a sea eel is either an eel or a plant, then a sea eel is an eel or an animal.

"Question": Based on the above information, is the following statement true, false, or uncertain? Sea eel is a paper.

Formalized Context:
\[
\begin{align*}
\text{constant Thing : Type} \\
\text{constant is\_eel : Thing} \to \text{Prop} \\
\text{constant is\_fish : Thing} \to \text{Prop} \\
\text{constant is\_plant : Thing} \to \text{Prop} \\
\text{constant is\_animal : Thing} \to \text{Prop} \\
\text{constant breathes : Thing} \to \text{Prop} \\
\text{constant sea\_eel : Thing} \\
\end{align*}
\]

\[ \text{-- All eels are fish.} \]
\[ \text{axiom A1 : \forall (t : \text{Thing}), is\_eel}\ t \to \text{is\_fish}\ t \]

\[ \text{-- No fish are plants.} \]
\[ \text{axiom A2 : \forall (t : \text{Thing}), is\_fish}\ t \to \neg \text{is\_plant}\ t \]

\[ \text{-- A thing is either a plant or animal.} \]
\[ \text{axiom A3 : \forall (t : \text{Thing}), is\_plant}\ t \lor \text{is\_animal}\ t \]

\[ \text{-- Nothing that breathes is paper.} \]
\[ \text{axiom A4 : \forall (t : \text{Thing}), breathes}\ t \to \neg \text{is\_paper}\ t \]

\[ \text{-- All animals breathe.} \]

Intuitive Proof:
\[
\text{theorem not\_sea\_eel\_is\_paper : \neg \text{is\_paper}\ \text{sea\_eel}} := \begin{align*}
\text{cases A3 sea\_eel,} \\
\text{cases A6 (or.inr h),} \\
\text{have h1 : \neg \text{is\_fish}\ \text{sea\_eel},} \\
\text{intro h,} \\
\text{have temp := A2 sea\_eel h,} \\
\text{contradiction,} \\
\text{have h2 : \neg \text{is\_eel}\ \text{sea\_eel},} \\
\text{intro h,} \\
\text{have temp := A1 sea\_eel h,} \\
\text{contradiction,} \\
\text{have h3 : \neg \text{is\_eel}\ \text{sea\_eel} \lor \text{is\_plant}\ \text{sea\_eel},} \\
\text{right,} \\
\text{assumption,} \\
\text{have h4 : \neg \text{is\_eel}\ \text{sea\_eel} \lor \text{is\_animal}\ \text{sea\_eel} := A6 h_3,} \\
\text{cases h4,} \\
\text{have h5 : breathes\ \text{sea\_eel} := A5 sea\_eel h_4,} \\
\text{have h6 : \neg \text{is\_paper}\ \text{sea\_eel} := A4 sea\_eel h_5,} \\
\text{contradiction,} \\
\text{have h1 : breathes\ \text{sea\_eel} := A5 sea\_eel h,} \\
\text{have h2 : \neg \text{is\_paper}\ \text{sea\_eel} := A4 sea\_eel h_1,} \\
\text{contradiction,} \\
\end{align*}
\]

Concise Proof:
\[
\text{theorem not\_sea\_eel\_is\_paper : \neg \text{is\_paper}\ \text{sea\_eel}} := \begin{align*}
\text{begin} \\
\text{cases A3 sea\_eel,} \\
\text{cases A6 (or.inr h),} \\
\text{have h1 := A2 sea\_eel (A1 sea\_eel h_1),} \\
\text{contradiction,} \\
\text{have h4 := A4 sea\_eel (A5 sea\_eel h_1),} \\
\text{exact A4 sea\_eel (A5 sea\_eel h_1),} \\
\end{align*}
\]