LeanReasoner: Boosting Complex Logical Reasoning with Lean

Anonymous ACL submission

Abstract

Large language models (LLMs) frequently face challenges with complex logical reasoning tasks. We address this issue with the help of Lean, a theorem proving framework. First, we formalize logical reasoning problems as theorems within Lean and then proceed to either prove or disprove them. This methodology serves dual purposes: it eliminates the possibility of logical inconsistencies typical in LLM outputs and effectively manages complex logical reasoning tasks. Central to our approach are the numerous theorem proofs written in Lean, which encapsulate human logical reasoning. Training a model with this data enhances its capability to address logical reasoning problems. Our method demonstrates state-of-the-art performance on 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 long been a challenging issue for machine learning systems, even for recent large language models (LLMs). 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, wherein the model hallucinates and makes statements or predictions 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). 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 checkpoint, ensuring that the model outputs align with logical rules, thereby mitigating the issue of logic inconsistency. Here, 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.

Serving 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 check system, ensuring each reasoning step is certified. What distinguishes it, however, is its functionality also 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 utilize 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

[1] Our code and data will be released upon publication.

chaining, and SATLM (Ye et al., 2023), which operates within the Z3 environment utilizing a suite of predetermined 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 formalizations of logic reasoning problems from ProofWriter to Lean, along with 27 analogous formalizations from FOLIO. The corresponding proofs in Lean are also included.

2 Problem Definition and Notation

The underlying task we aim to solve is logical reasoning, which takes the form of multi-choice questions given 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 1,000,0000 lines of code.[3] Lean can also be used as a generic theorem prover, not necessarily in the area of mathematics. This is the way we use it for our case.

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

- **Context**, which represents 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* $\forall x,$ *is_cat* $x \rightarrow$ *is_animal* x and *axiom A3* $\forall x,$ *is_cat* $x \rightarrow$ *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:* $\neg$ *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 addressed, 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 gen-*

[2] https://leanprover.github.io/.

[3] https://en.wikipedia.org/wiki/Lean_(proof_assistant).

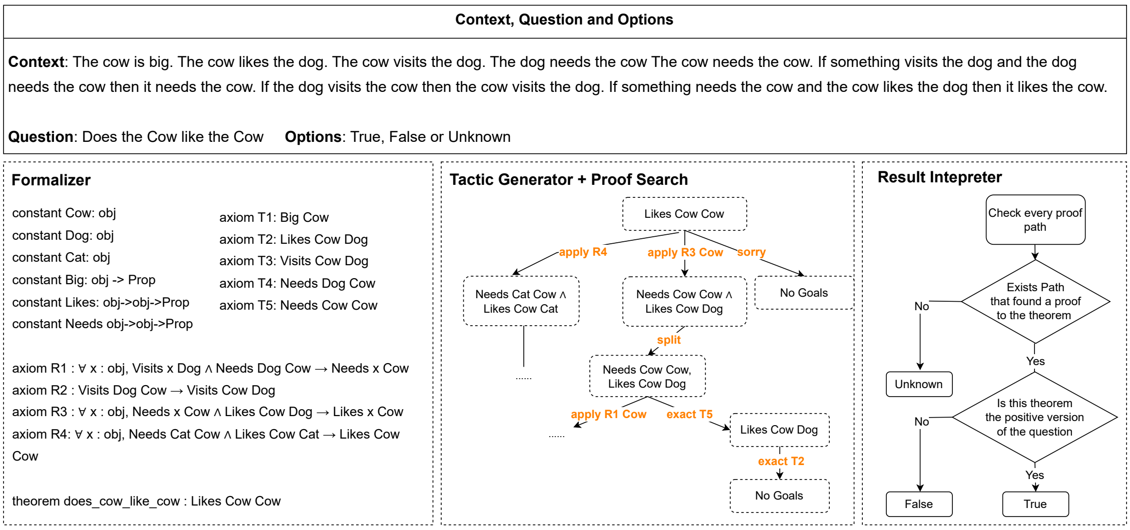

Figure 1: An overview of our approach: The natural language context is first processed by the "formalizer". It then advances to the proof search stage, where all the orange tactics are generated by the "tactic generator". Finally, the outcome is interpreted by the "result interpreter". All items inside stadiums are goals.

erator, a *proof search* mechanism, and a *result interpreter*. The formalizer converts 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 of each 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 the task specification 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 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, as 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 conduct manual inspections of 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 is challenging amidst numerous distractions, especially in logical reasoning problems when several options might seem promising for the current step but won't 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 YES), 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 UNKNOWN, 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, which 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 utilize 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 are expanded based on cumulative log probabilities of the goal, 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, thereby streamlining the search process and improving 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 hasn't 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 the 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 the tactic generator in the experiment section.

3.4 Result Intepreter

For options that include *Unknown*, we only regard the result as correct if no other options can be proven. All datasets investigated in this study only contain questions with only one correct option. 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, and labels can be categorized as TRUE, FALSE, or UNKNOWN. 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 Logic-LM, we used the same 600 sample tests, ensuring an even label distribution.

FOLIO: Unlike ProofWriter, FOLIO is constructed using intricate first-order logic. This increases the complexity of the proving part. The dataset presents problems in a more natural wording, with relationships that are considerably more complex. Such a combination of advanced logic and rich linguistic structure renders 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.

In annotating the data, we adopted two divergent approaches for constructing proofs. One approach emulated a straightforward strategy, encompassing a detailed procedure with numerous intermediate steps and lemmas, similar to how we might derive a proof when faced with 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 pre-trained ReProver from LeanDoJo (Yang et al., 2023), which was pre-trained on Mathlib 3. 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. We will release our code to facilitate reproducibility.

5 Results

We present our experimental results, including an examination of the formalization module, insights into enhancing the tactic generator module, and a comparison of our work with other baselines.

5.1 Analysis of Prompting Results

Since there is no automated method to confirm the accuracy of formalization, we conduct manual examinations of the formalized results to determine whether errors occur during the formalization or proof generation stages. Only formalizations that correctly captured every fact, axiom, and rule were counted as accurate. We prompted the LLM to formalize a selection of 100 questions from ProofWriter's validation set and 40 questions from FOLIO's training set and manually examined them. The findings have been summarized in Table 1.

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 a large language model 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.

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.

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 section of the table is determined by whether the generated proof can be validated successfully in Lean. If the formalization of the question to theorem is correct and the

Model	ProofWriter			FOLIO		
	Formalize	**Prove**	**Answer**	**Formalize**	**Prove**	**Answer**
GPT-4 Base	94%	15%	80%	60%	10%	35%
GPT-4 Base Comments	99%	15%	80%	75%	15%	35%
GPT-4 Base Separate	95%	5%	75%	60%	10%	40%
GPT-3 Base Comments	77%	12%	63%	45%	10%	35%
Logic-LM	98%	75.5%	74%	65%	69.2%	55%

Table 1: Formalization, Proof, and Answer choice accuracy of 100 ProofWriter samples and 40 FOLIO samples via OpenAI Language Model API, with manual annotation. 'GPT-4 Base' serves as our baseline, where few-shot examples include both formalization and proof generation in a single prompt. In 'GPT-4 Base Comments', we augment these examples with line-by-line comments in Lean code. For 'GPT-4 Base Separate', we divide the task into two parts, using separate prompts for formalization and proof generation. For simplicity, we did not use the self-refinement technique when evaluating Logic-LM.

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.

Despite the low accuracy in most of GPT-4's proofs, it achieved a 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 Enhanced Proving

In this section, we focus on training our own models to do tactic generation using our annotated training data. To isolate the impact of the tactic generator, 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.

We first compare the results on premise selection using the metrics recall@1 and recall@4. The recall@k metric is defined as follows:

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

where GT_Prem means ground truth premises and Pred_Prem means top predicted premises. It is not surprising that LeanReasoner pretrained solely with math data yielded suboptimal results. This can 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**.

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 utilizes the logical elements 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.

Fine-tuning with different annotations 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

Method	Pretrained on Math Data	Fine-tuned on the Annotation	ProofWriter			FOLIO		
			Premise Selection		Proof	Premise Selection		Proof
			Rec@1	Rec@4	Acc	Rec@1	Rec@4	Acc
GPT-4	N/A	N/A	N/A		15%	N/A		10%
LeanReasoner	Yes	No	56.2%	81.3%	0%	23.5%	38.2%	0%
LeanReasoner	No	Intuitive	62.5%	100%	99%	54.8%	95.2%	71.4%
LeanReasoner	Yes	Intuitive	75%	100%	99%	71.4%	96.8%	85.7%
LeanReasoner	Yes	Concise	75%	100%	99%	83.8%	97.4%	85.7%

Table 2: Comparative Analysis of Recall@k in premise selection and overall proof accuracy for 99 ProofWriter test samples and 28 FOLIO test samples. This test set comprises formally verified and manually inspected results. 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.

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

Table 3: Proof accuracy across different methods for the ProofWriter dataset. 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.

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

Table 4: Proof accuracy across different methods for the FOLIO dataset. The 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 FOLIO.

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 Comparing Against 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.

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 doesn't bring any advantage to the final performance on this dataset.

Table 4 presents our performance on FOLIO. For a fair comparison with SATLM that uses the Z3 solver, we used the lean-smt tool [4] 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

[4]https://github.com/ufmg-smite/lean-smt

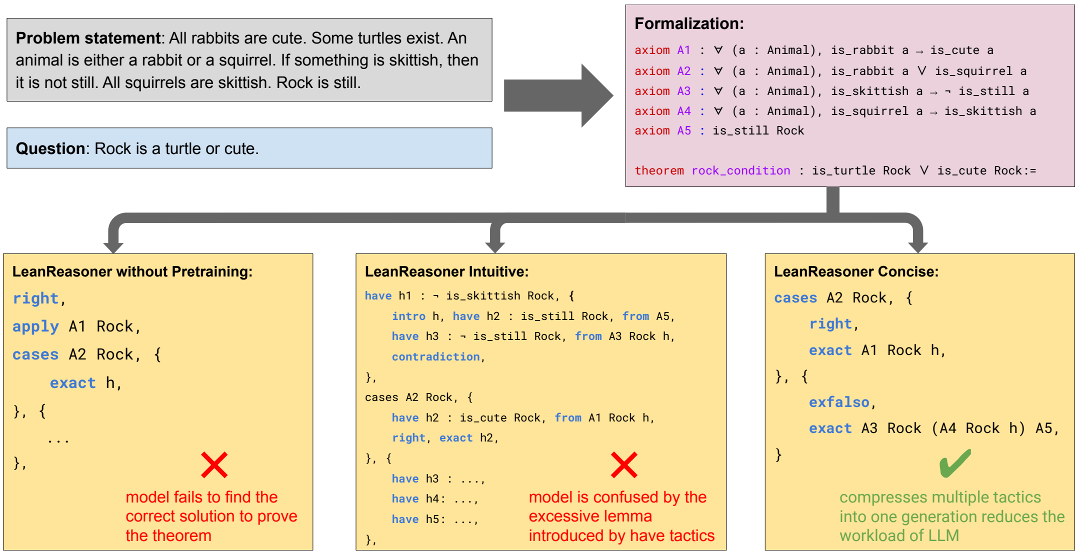

Figure 2: Sample proof generated by LeanReasoner without Pretraining, LeanReasoner finetuned on Intuitive data, and LeanReasoner finetuned on Concise data.

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 LeanReasoner, when fine-tuned on the **Intuitive** dataset, 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. In essence, 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.

It's 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

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 grapple with constraints 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) 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.

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 prompting. Other results by (Fu and Khot, 2022; Fu et al., 2023) suggest that powerful LLMs get 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. An extensive examination was conducted on errors from the formalization and proof generation stage. We also examined the performance enhancements from pretraining on theorem proving data and annotation styles. We offered a comprehensive comparison with other techniques that highlight our model's superior strengths. 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 represents a significant stride forward in the AI reasoning domain. Our method has not only shown a remarkable improvement in handling complex reasoning tasks but also offers a layer of mathematical rigor that bolsters the reliability of conclusions derived. However, as we elevate the reasoning prowess of LLMs, there's an amplified potential for embedded biases within the training data to manifest and magnify. Especially in reasoning scenarios, this can inadvertently lead to skewed logic or unintended favoritism in areas of utmost sensitivity 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.

Our integration of Lean provides LLMs with the unique advantage of elucidating detailed logical pathways, reinforcing the transparency of our reasoning process. Tracing reasoning step by step, not only bolsters confidence in the derived conclusions but also enhances the explainability of these systems. Such transparency is vital in fostering trust, especially when these models are used in pivotal decision-making processes. This robustness coupled with the broader societal implications of advanced reasoning in LLMs underscores the necessity for ongoing ethical reflection and proactive measures to address potential challenges.

References

Fabian Gloeckle Baptiste Rozière, Jonas Gehring and et.al. 2023. Code llama: Open foundation models for code. abs/2308.12950.

Jianshu Chen. 2023. Learning language representations with logical inductive bias. In *ICLR*. OpenReview.net.

Wenhu 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.

Zeming Chen, Gail Weiss, Eric Mitchell, Asli Celikyilmaz, and Antoine Bosselut. 2023b. RECKONING: reasoning through dynamic knowledge encoding. *CoRR*, abs/2305.06349.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems.

Antonia Creswell and Murray Shanahan. 2022. Faithful reasoning using large language models. abs/2208.14271.

Ishita Dasgupta, Andrew K. Lampinen, Stephanie C. Y. Chan, Antonia Creswell, Dharshan Kumaran, James L. McClelland, and Felix Hill. 2022. Language models show human-like content effects on reasoning.

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: an efficient SMT solver. In *TACAS*.

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The lean theorem prover (system description). In *CADE-2*.

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*.

Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. 2023. Complexity-based prompting for multi-step reasoning. In *ICLR*. OpenReview.net.

Nicolas Gontier, Siva Reddy, and Christopher Pal. 2022. Does entity abstraction help generative transformers reason? *Trans. Mach. Learn. Res.*, 2022.

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

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Luke Benson, Lucy Sun, Ekaterina Zubova, Yujie Qiao, Matthew Burtell, David Peng, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Shafiq R. Joty, Alexander R. Fabbri, Wojciech Kryscinski, Xi Victoria Lin, Caiming Xiong, and Dragomir Radev. 2022b. FOLIO: natural language reasoning with first-order logic.

Joy He-Yueya, Gabriel Poesia, Rose E. Wang, and Noah D. Goodman. 2023. Solving math word problems by combining language models with symbolic solvers.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring Mathematical Problem Solving With the MATH Dataset. In *NeurIPS*, Menlo Park, Calif. AAAI Press.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick S. H. Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *EMNLP*.

Philippe Laban, Wojciech Kryscinski, Divyansh Agarwal, Alexander R. Fabbri, Caiming Xiong, Shafiq Joty, and Chien-Sheng Wu. 2023. Llms as factual reasoners: Insights from existing benchmarks and beyond.

Guillaume Lample, Timothée Lacroix, Marie-Anne Lachaux, Aurélien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. 2022. Hypertree proof search for neural theorem proving. In *NeurIPS*.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay V. Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. Solving quantitative reasoning problems with language models. In *NeurIPS*.

OpenAI. 2023. GPT-4 technical report.

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

Gabriel Poesia, Kanishk Gandhi, Eric Zelikman, and Noah D. Goodman. 2023. Certified reasoning with language models.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. 2023. Formal mathematics statement curriculum learning. In *ICLR*.

Stanislas Polu and Ilya Sutskever. 2020. Generative language modeling for automated theorem proving. abs/2009.03393.

Abulhair Saparov and He He. 2023. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. In *ICLR*. OpenReview.net.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, and et al. 2022. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models.

Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. 2022. Byt5: Towards a token-free future with pre-trained byte-to-byte models. *Trans. Assoc. Comput. Linguistics*, 10.

Kaiyu Yang, Jia Deng, and Danqi Chen. 2022. Generating natural language proofs with verifier-guided search. In *EMNLP*.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. Leandojo: Theorem proving with retrieval-augmented language models.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2023. Satisfiability-aided language models using declarative prompting.

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.
```

```
Output:
universe u

constant obj : Type u

constant Cat : obj
constant Cow : obj
constant Rabbit : obj
constant Tiger : obj

constant Blue : obj → Prop
constant Nice : obj → Prop
constant Likes : obj → obj → Prop
constant Needs : obj → obj → Prop
constant Round : obj → Prop
constant Red : obj → Prop
constant Chases : obj → obj → Prop

axiom T1 : Blue Cat
axiom T2 : Nice Cat
axiom T3 : Likes Cat Cow
axiom T4 : Needs Cat Cow
axiom T5 : Needs Cat Rabbit
axiom T6 : ¬ Needs Cow Rabbit
axiom T7 : Needs Cow Tiger
axiom T8 : Round Rabbit
axiom T9 : Nice Tiger
axiom T10 : ¬ Likes Tiger Cow

-- If someone likes the cat and they chase the cat
    then they are blue
axiom R1 : ∀ x : obj, Likes x Cat ∧ Chases x Cat →
    Blue x
-- If someone likes the cow and they are red then
    the cow is round
axiom R2 : ∀ x : obj, Likes x Cow ∧ Red x → Round
    Cow
-- If someone needs the tiger and they need the cat
    then they do not chase the cow
axiom R3 : ∀ x : obj, Needs x Tiger ∧ Needs x Cat
    → ¬ Chases x Cow
-- If someone needs the cat and the cat is blue
    then the cat is red
axiom R4 : ∀ x : obj, Needs x Cat ∧ Blue Cat →
    Red Cat
-- If someone is round then they need the cat
axiom R5 : ∀ x : obj, Round x → Needs x Cat
-- If someone likes the cat and the cat needs the
    rabbit then the cat chases the tiger
axiom R6 : ∀ x : obj, Likes x Cat ∧ Needs Cat
    Rabbit → Chases Cat Tiger
-- If the rabbit needs the tiger then the rabbit is
    not round
axiom R7 : ∀ x : obj, Needs Rabbit Tiger → ¬
    Round Rabbit


theorem cow_chases_cow : Chases Cow Cow :=
begin
  have H1 : Needs Cow Tiger := T7,
  have H2 : Needs Cat Cow := T4,
  have H3 : ¬ Chases Cow Cow := R3 Cow (and.intro
    H1 H2),
  contradiction,
  sorry,
end

-- Looks like the theorem cannot be proven in Lean.
-- So let's try to prove the negation of the
    theorem, which is Cow does not chase Cow

theorem not_cow_chases_cow : ¬ Chases Cow Cow :=
begin
  apply R3 Cow,
  split,
  exact T7,
  apply R5,
  apply R2 Cat,
  split,
  exact T3,
  apply R4 Rabbit,
  split,
  apply R5,
  exact T8,
  exact T1,
end

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

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.

```
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: There are six types of wild
turkeys: Eastern wild turkey, Osceola wild
turkey, Gould's wild turkey, Merriam's wild
turkey, Rio Grande wild turkey, and Ocellated
wild turkey. Tom is not an Eastern wild turkey.
Tom is not an Osceola wild turkey. Tom is
also not a Gould's wild turkey, or a
Merriam's wild turkey, or a Rio Grande wild
turkey. Tom is a wild turkey.

Question 1: Based on the above information, is the
following statement true, false, or uncertain?
Tom is an Ocellated wild turkey.
Question 2: Based on the above information, is the
following statement true, false, or uncertain?
Tom is an Eastern wild turkey.
Question 3: Based on the above information, is the
following statement true, false, or uncertain?
Joey is a wild turkey.
```

```
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_wild_turkey : Turkey → Prop
constant is_eastern_wild_turkey : Turkey → Prop
constant is_osceola_wild_turkey : Turkey → Prop
constant is_goulds_wild_turkey : Turkey → Prop
constant is_merriams_wild_turkey : Turkey → Prop
constant is_rio_grande_wild_turkey : Turkey → Prop
constant is_ocellated_wild_turkey : Turkey → Prop

-- 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 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
-- Tom is a wild turkey.
axiom A6 : is_wild_turkey Tom
-- There are six types of wild turkeys: Eastern
     wild turkey, Osceola wild turkey, 'Goulds wild
     turkey, 'Merriams wild turkey, Rio Grande wild
     turkey, and Ocellated wild turkey.
axiom turkey_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
-- Tom is an Ocellated wild turkey.
-- Let's first prove the positive case.
theorem tom_is_ocellated_wild_turkey :
     is_ocellated_wild_turkey Tom :=
begin
    cases turkey_type_exclusivity Tom A6, {
        have h1 : ¬ is_eastern_wild_turkey Tom,
     from A1,
        contradiction,
    }, {
        cases h, {
            have h1 : ¬ is_osceola_wild_turkey
     Tom, from A2,
            contradiction,
        }, {
            cases h, {
                have h1 : ¬ is_goulds_wild_turkey
     Tom, from A3,
                contradiction,
            }, {
                cases h, {
                    have h1 : ¬
     is_merriams_wild_turkey Tom, from A4,
                    contradiction,
                }, {
                    cases h, {
                        have h1 : ¬
     is_rio_grande_wild_turkey Tom, from A5,
                        contradiction,
                    }, {
                        assumption,
                    }
                }
            }
        }
    }
end

-- 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
-- Tom is an Eastern wild turkey.
-- Let's first prove the positive case.
theorem tom_is_eastern_wild_turkey :
     is_eastern_wild_turkey Tom :=
begin
    sorry, -- Placeholder for proof
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
-- Joey is a wild turkey.
-- Let's first prove the positive case.
theorem joey_is_wild_turkey : is_wild_turkey Joey :=
begin
    sorry, -- Placeholder for proof
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
```

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.
Turkey 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 Bob : 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":

  $\text{Goulds Tom} \lor \text{Eastern Tom} \lor \text{Osceola Tom}$

- The formalization of numbers is incorrect, it should be:

  $\text{constant weight} : \text{Turkey} \rightarrow \mathbb{N}$

  $\text{axiom A3} : \text{weight Bob} = 86$

  $\text{axiom A4} : \text{weight Tom} = 73$

- The formalization of logic is incorrect, it should be:

  $\neg\ \text{bite\_causes\_death}\ \ t\ \land\ \text{bite\_causes\_itching}\ \ t) \lor$ $(\text{bite\_causes\_death} \land \neg\ \text{bite\_causes\_itching}\ t$

- There is an incorrect division of concepts that would make the proving impossible, the correct version should be:

  $\forall\ (t : \text{Turkey}), \text{Eastern}\ t \rightarrow \text{bite\_causes\_death}\ t$

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:
```
constant Thing : Type

constant is_eel : Thing → Prop
constant is_fish : Thing → Prop
constant is_plant : Thing → Prop
constant is_animal : Thing → Prop
constant is_paper : Thing → Prop
constant breathes : Thing → Prop

constant sea_eel : Thing

-- All eels are fish.
axiom A1 : ∀ (t : Thing), is_eel t → is_fish t
-- No fish are plants.
axiom A2 : ∀ (t : Thing), is_fish t → ¬ is_plant t
-- A thing is either a plant or animal.
axiom A3 : ∀ (t : Thing), is_plant t ∨ is_animal t
-- Nothing that breathes is paper.
axiom A4 : ∀ (t : Thing), breathes t → ¬ is_paper
    t
-- All animals breathe.
axiom A5 : ∀ (t : Thing), is_animal t → breathes t
-- If a sea eel is either an eel or a plant, then a
     sea eel is an eel or an animal.
axiom A6 : (is_eel sea_eel ∨ is_plant sea_eel) →
     (is_eel sea_eel ∨ is_animal sea_eel)
```

Intuitive Proof:
```
theorem not_sea_eel_is_paper : ¬ is_paper sea_eel
     :=
begin
    cases A3 sea_eel, {
        have h1 : ¬ is_fish sea_eel, {
            intro h,
            have temp := A2 sea_eel h,
            contradiction,
        },
        have h2 : ¬ is_eel sea_eel, {
            intro h,
            have temp := A1 sea_eel h,
            contradiction,
        },
        have h3 : is_eel sea_eel ∨ is_plant
     sea_eel, {
            right,
            assumption,
        },
        have h4 : is_eel sea_eel ∨ is_animal
     sea_eel := A6 h3,
        cases h4, {
            contradiction,
        }, {
            have h5 : breathes sea_eel := A5
     sea_eel h4,
            have h6 : ¬ is_paper sea_eel := A4
     sea_eel h5,
            contradiction,
        }
    }, {
        have h1 : breathes sea_eel := A5 sea_eel h,
        have h2 : ¬ is_paper sea_eel := A4 sea_eel
     h1,
        contradiction,
    }
end
```

Concise Proof:
```
theorem not_sea_eel_is_paper : ¬ is_paper sea_eel
     :=
begin
    cases A3 sea_eel, {
        cases A6 (or.inr h), {
            have h1 := A2 sea_eel (A1 sea_eel h_1),
            contradiction,
        }, {
            exact A4 sea_eel (A5 sea_eel h_1),
        }
    }, {
        exact A4 sea_eel (A5 sea_eel h),
    }
end
```