Boosting Large Language Model Reasoning with Theorem Proving

Anonymous ACL submission

Abstract

Large Language Models (LLMs) frequently face challenges with complex reasoning tasks. A recent structured AI methodology addresses this by distinctly dividing tasks into symbolic formalization, managed by LLMs, and problem-solving, conducted by symbolic solvers. While solvers like SymPy and Pyke prevent hallucinations, they often struggle with advanced reasoning tasks. This study addresses their limitations by leveraging the extensive reasoning data in Lean, a programming language for theorem proving. Training a custom model using Lean's rich theorem proving data greatly enhances our model's reasoning capacity, allowing it to outperform traditional solvers. We achieve state-of-the-art result on FOLIO, a logical reasoning dataset, indicating the potential of our method for natural language reasoning.[1]

1 Introduction

Reasoning, a bedrock of intelligence and a core capability of humans, has long been a challenging issue for machine learning systems, even for the latest, powerful large language models (LLMs). LLMs, despite their impressive abilities to understand and generate natural language, often fall short when dealing with complex reasoning tasks. They frequently suffer from "hallucinations", wherein the model makes statements or predictions not grounded in its inputs, leading to spurious results (Saparov and He, 2023; Dasgupta et al., 2022).

Recent advances in AI have adopted a structured approach to tackling reasoning problems by splitting them into symbolic formalization and problem-solving phases (He-Yueya et al., 2023; Pan et al., 2023; Ye et al., 2023). The formalization step is often handled by a large language model (LLM), while problem-solving is tackled by an out-of-the-box solver. In this approach, symbolic reasoning essentially acts as a rigorous checkpoint, ensuring that the model outputs align with logical and factual standards, thereby mitigating the issue of hallucination. Here, solvers may range from being completely deterministic, like SymPy (He-Yueya et al., 2023), or rely 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. The limitation mainly lies in the capabilities of the solvers themselves; they lack the ability to extract and use the vast wealth of reasoning data and information available in large language resources as LLMs do. This absence of information integration leaves them underpowered when dealing with intricate reasoning tasks.

Serving as a powerful theorem prover and a versatile programming language, Lean (de Moura et al., 2015) presents a compelling solution to connect symbolic reasoning with extensive linguistic resources. Much like symbolic solvers, Lean has a strict check system, ensuring each reasoning step is certified. 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 automatic 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 develop a Lean-based framework to tackle natural language reasoning with datasets such as ProofWriter (Tafjord et al., 2021) and FOLIO (Han et al., 2022b). We use LLMs to formalize these datasets into Lean's formalized language, and fine-tune a custom model on these problems using a modest amount of data we collected ourselves. Our contributions in this paper are twofold.

- We show that incorporating theorem proving data

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

in training a custom model achieves competitive performance with substantially less training data. This strategy outperforms conventional out-of-the-box solvers, especially when tackling more complex problems. The model also obtained state-of-the-art results on FOLIO.

- We make available the training data gathered in this study, which includes 100 fully verified formalization of natural language reasoning problems from ProofWriter to Lean, as well as 27 similar translations from FOLIO. Additionally, we are releasing the corresponding theorem proofs for these problems.

2 Problem Definition and Notation

The underlying task we aim to solve is providing an answer to a natural query, where background natural language context is given, such that it would be possible to logically deduce the answer to the query based on the context. This task, referred to as *natural language reasoning*, along with our solution to it, consists 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 an available set of answers (discrete categories) from which an answer can be chosen. For example, *True*, *False* or *Unknown*.
- **Formalized context** is the formalization of the context in the underlying logical language, in our case, 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 Lean theorem proving context, 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 (or its formalized representation) becomes our root goal. As we apply various **Tactics** to simplify or break down this primary goal, we generate intermediate goals. These intermediate goals can be thought of as subproblems or sub-questions derived from the primary question. The proof process in Lean is essentially a journey from the root goal through a series of intermediate goals until we reach a point where all goals have been resolved based on our axioms and rules.

  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*. As we apply **Tactics**, 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 instructions in the Lean theorem prover language 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.

The environment we use for theorem proving is Lean.[2] Lean is an open source theorem proving programming language, originally developed for mathematical theorem proving, with a 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.

3 Methodology

Our methodology is composed of four main components: a *formalizer*, a *tactic generator*, a *proof search* mechanism, and a *result interpreter*. The formalizer converts context and question to formalized context and formalized question. The tactic

[2] https://leanprover.github.io/.
[3] https://en.wikipedia.org/wiki/Lean_(proof_assistant).

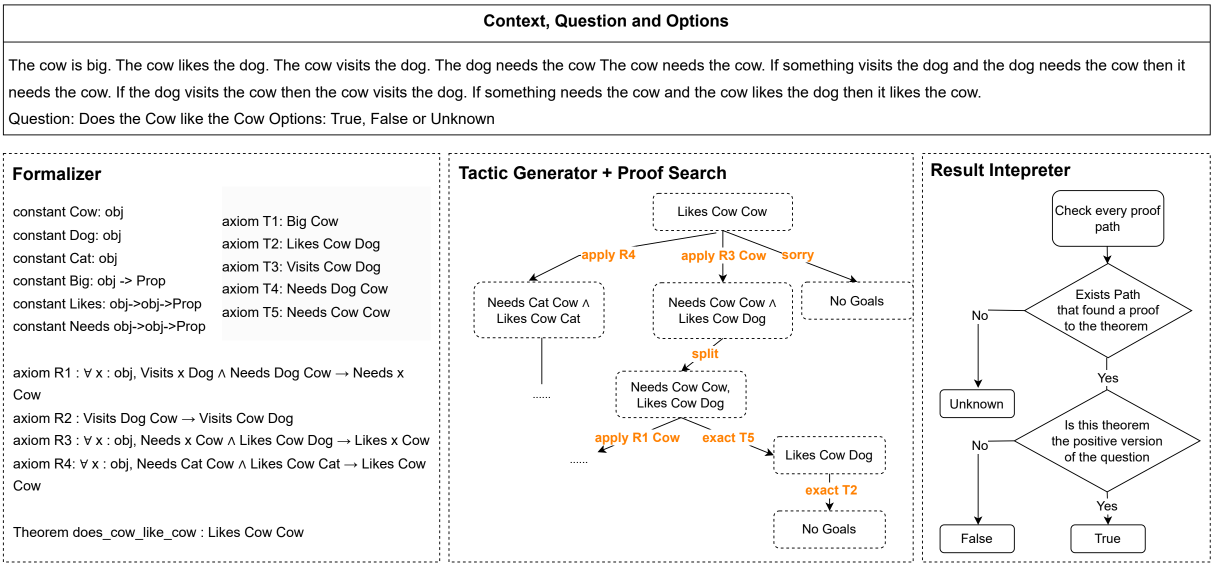

Figure 1: An overview of our approach: The natural language context is first processed by the "formalizer". It then advances to the proof search stage, where all the orange tactics are generated by the "tactic generator". Finally, the outcome is interpreted by the "result interpreter".

generator then generates tactics based on premises extracted from the formalized context. The proof search mechanism oversees tactic execution and goal expansion. Lastly, the result interpreter analyses the output of the proof search and identifies the correct answer in options. In this section, we provide detailed explanations of each component.

3.1 Formalizer

In this process of formalization, we used the 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 the task specification in the system prompt.

There is no definitive way to assert that a formalized result is correct since there is no deterministic Automated Theorem Prover (ATP) that can confirm the accuracy of formalization. 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 reformulate 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 example 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 employs retrieval mechanisms to explicitly select premises. When provided with the current state of proof, this generator retrieves a selected set of potentially useful premises from formalized context and generates tactic using both the goal and the retrieved premises.

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 state and the premise.

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. This division of duties also lightens the load for the tactic generator by allowing it to concentrate solely on its specific role, rather than grappling with the entirety of the problem. An added advantage of this approach is that it makes the system's reasoning steps more transparent and understandable.

As a baseline, we also prompt GPT-4 to generate proofs. When the answer aligns with the chosen theorem (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 option, signifying that the formalized theorem is unprovable, we still encourage the model to engage in step-by-step reasoning, even though it will eventually hit a roadblock. In instances where the answer is UNKNOWN, implying that neither option can be proven, we provide step-by-step reasoning prompts for each option, acknowledging that the process will not result in a definitive answer. An example of the prompt to GPT-4 can be found in Appendix A.1.

3.3 Proof Search

Different from the tactic generator module that generates tactics, the proof search module controls the overall search process that selects tactics and maintains states during proof construction. LeanDoJo (Yang et al., 2023), a recently released framework that enables interaction with Lean programmatically, provides the mechanism to check the validity of tactics and execute tactics.

The search method involves building a proof tree, which incrementally evolves the goal through tactic invocations. This approach was first introduced in (Polu and Sutskever, 2020). LeanDoJo (Yang et al., 2023) subsequently provided an implementation of this method, which we utilized for our study. As a reference, the middle part of Figure 1 provides a practical illustration of this process. For each given proof goal, we explore 64 possible tactics, commencing from the root goal. 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. The resulting tendency is towards breadth-first exploration, as goals at greater depths have more parent tactics and hence a typically higher cumulative log probability. During the search process, there are no restrictions on the length of the priority queue.

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 a current goal or state being explored is a superset (or contains all the elements) of a previously explored goal, the current goal is not 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" keywords (such as "sorry") that tell Lean to assume that the current goal is completed, even though it hasn't been proven, meaning that every path containing "cheating" keywords 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 tactic generator in the experiment section.

3.4 Interpreting Results

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 have questions with only one correct option among the choices. 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, our model training and our baselines.

4.1 Datasets

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

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: 0,

≤ 1, ≤ 2, ≤ 3, and ≤ 5 hops. 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, increasing the complexity of the proving part. Beyond just the logic, the formalization for FOLIO is also challenging. The dataset presents problems in a more natural and intricate 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 Model Training

Regarding the data for model training, we collected 100 theorem proofs for ProofWriter, where each problem's proof was either manually annotated or collected from successful proofs generated by GPT-4. A similar approach was employed with FOLIO, albeit with 27 theorems. The data collection took about two days.

The pre-training model structure we adopted was the same as used 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 on our collected data took about six hours on one A100 40G.

4.3 Baselines

For all of our experiments, we tested reasoning ability against textual input to GPT-4. When benchmarking against GPT-4 for all datasets, we strived to leverage prompts from previous work to the greatest extent possible. Our principal focus was GPT-4's chain-of-thought (CoT) output.

For our own formalization, we use three examples as prompts for both ProofWriter and FOLIO. Because FOLIO uses the same context for different questions, we use a multi-question-style prompt for FOLIO where each prompt contains multiple questions, an example can be found in Appendix A.2.

For ProofWriter and FOLIO, we also compared our results against Logic-LM (Pan et al., 2023). Given that Z3 can also be used within Lean for problem-solving, we also employed Z3 on our formalized context using lean-smt package (Mohamed et al., 2022), which servers as a comparison against SATLM (Ye et al., 2023). In addition, our findings were compared with other benchmark on these two datasets.

5 Results

We describe the results of our experiments: an analysis of the formalization module, a description of how to improve the tactic generator module and a comparison of our work against the baselines.

5.1 Analysis of Formalization

To discern whether errors arise during the formalization or proving stages, and to pinpoint the exact mistakes in the formalization process, 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. These findings can be viewed in Table 1. Only those formalizations that correctly captured every fact, axiom, and rule were counted as accurate. The striking accuracy on ProofWriter can be attributed to its simpler language structure compared to FOLIO. In the case of FOLIO, using a large language model for formalization helped in filtering out unnecessary details from the natural language context, making it easier to understand the essence of the problem and do reasoning. We have illustrated typical GPT-4 formalization mistakes in B, using a composite sample derived from various error instances. Interestingly, Lean's formalization accuracy aligns closely with both Prolog and FOL in Logic-LM. This consistency underscores Lean's versatility, allowing it to uniformly represent both problem types.

We observed improved results when formalized code was paired with descriptive textual comments sourced from the context. This approach split the formalization task into two: 1) linking textual input with formalized code and 2) generating formalized code based on the prior textual comment. These textual cues acted as a bridge between raw text and formalized code, aiding the underlying computation processes.

It is important to highlight that the compilation errors in the formalized Lean code were straightforward to correct. When issues arise during the Lean building process, we present the error message and the original formalized Lean code to LLM for re-formalization. If the subsequent attempt is unsuccessful, we simply categorize it as incorrect.

Model	ProofWriter			FOLIO		
	Formalize	**Prove**	**Total**	**Formalize**	**Prove**	**Total**
GPT-4 Base	94%	15%	80%	60%	10%	35%
GPT-4 Base Comments	99%	-	80%	75%	-	35%
GPT-4 Base Separate	-	5%	75%	-	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 Total accuracies for ProofWriter and FOLIO using the OpenAI language model API. 'Base Comments' provide annotations before each line of formalized code. In 'Base Separate', formalization and proof are segmented into two distinct prompts, reducing the workload on the LLM. For Logic-LM, proof accuracy is determined from correctly formalized problems, while total accuracy is calculated on all problems. For simplicity, we did not use the self-refinement technique when evaluating Logic-LM

The distinction in performance between GPT-3 and GPT-4 is evident. While the formalization for simple concepts is the same, GPT-3 struggles with intricate logic, highlighting its limitations. As such, we opted not to use GPT-3 in further tests.

The proof accuracy section of the table is determined by whether the generated proof can compile successfully in Lean. If the formalization of question to theorem is correct and the proof can be compiled without any error or warning, then we can be confident that the proof is valid. However, the accuracy of generated proof is very low. This could be due to overloading large language model with tasks, making it difficult to complete both on a single prompt. We attempted to separate formalization and proof, but the outcome remained disappointing, indicating GPT-3 and GPT-4's inability to perform proving tasks. Interestingly, the proof accuracy of Logic-LM wasn't as high as we expected. Upon replicating their code, we found the 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 inaccuracies in most of GPT-4's proofs, it achieved a high accuracy rate for final choices on ProofWriter (as shown in Total column). We believe this may be due to GPT-4's training exposure to it, potentially leading to a degree of memorization.

5.2 Enhanced Proving

In this section, we focus on training custom ReProver models to do tactic generation using our annotated training data. To isolate the impact of the tactic generator, we used all the accurate formalizations from the previous subsection. This gave us 99 test examples for ProofWriter and 14 for FOLIO. Furthermore, we annotated an additional 100 fully correct samples from the ProofWriter training set and 27 from the FOLIO training set. 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 by the ratio of ground truth premises intersecting with the top predicted premises to the total number of correct premises, represented as:

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

It is clear that relying on ReProver trained solely with math data yielded suboptimal results. This can be attributed to the limited set of tactics available for both ProofWriter and FOLIO. While these datasets have a confined tactic range, the model frequently makes mistakes by attempting to use other, unrelated tactics. The ReProver fine-tuning outperformed T5 fine-tuning in terms of overall results. Furthermore, the accuracy for FOLIO were noticeably poorer than those for 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**.

We proceeded to evaluate the overall proof results. Consistently, the ReProver fine-tuning model trained on math theorem proving data outperformed other approaches for both ProofWriter and FOLIO datasets. This advantage can be attributed to the limited data available for fine-tuning our tactic generator, thus highlighting the benefits of our approach. 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 Re-

Model	ProofWriter			FOLIO		
	Premise Selection		Proof	Premise Selection		Proof
	Rec@1	Rec@4	Acc	Rec@1	Rec@4	Acc
GPT-4 baseline	N/A		15%	N/A		10%
ReProver baseline	56.2%	81.3%	0%	23.5%	38.2%	0%
T5 fine-tuned	62.5%	100%	99%	54.8%	95.2%	71.4%
ReProver fine-tuned	75%	100%	99%	71.4%	96.8%	85.7%

Table 2: Recall@k for premise selection and overall proof accuracy across various tactic generator, encompassing the entire process from premise selection to tactic generation and proof search. We did not compute the Premise Selection accuracy for the GPT-4 baseline because prompting GPT-4 to select premises using Lean goals is challenging and is a primary concern in this context

Method	Acc
Abs Biases (Gontier et al., 2022)	80.6%
MetaInduce (Yang et al., 2022)	98.6%
RECKONING (Chen et al., 2023b)	99.8%
GPT-4 CoT (Pan et al., 2023)	68.1%
Logic-LM (Pan et al., 2023)	79.3%
T5 fine-tuned	95.8%
ReProver fine-tuned	98.3%

Table 3: Accuracy with different methods on ProofWriter. Abs Biases stands for Abstraction Inductive Biases

Model	Acc
Codex (Han et al., 2022b)	56.0%
FOLNet (Chen, 2023)	70.6%
GPT-4 CoT (Pan et al., 2023)	70.6%
Logic-LM (Pan et al., 2023)	74.5%
Lean Z3 (SATLM)	77.5%
T5 fine-tuned	66.2%
ReProver fine-tuned	78.4%

Table 4: Accuracy comparisons across different methods for the FOLIO dataset. The Codex baseline employs an 8-shot prompt. The result from 'Lean Z3' is derived from lean-smt applied to formalized Lean Code

Prover 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 pre-trained ReProver baseline may not be as noticeable for simpler datasets like ProofWriter, its value becomes evident for more complex datasets, such as FOLIO.

5.3 Comparing Against Other Baselines

Having demonstrated that fine-tuning on pre-trained math theorem models 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 the ProofWriter paper, in addition to the entire FOLIO test set. Given the smaller test set used in the preceding section, it is of interest to also compare our approach with the model not pre-trained on theorem proving data on this larger set. Subsequently, we conduct an analysis of the errors made by the tactic generator in both the FOLIO and ProofWriter, exploring the reason our approach outperforms others.

As illustrated in Table 3, our approach yields results comparable to state-of-the-art methods for the ProofWriter dataset. While other methods except Logic-LM use the entire training set of ProofWriter, our approach relies on just 100 examples, underscoring the efficiency of our method.

Table 4 presents our performance on the FOLIO dataset. For a balanced comparison with SATLM, which utilizes 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 an assignment (a set of variable values) that makes the theorem true, which basically means the answer to the original question is True. "unsat" Stands for "unsatisfiable". When Z3 returns "unsat", it means that there is no possible assignment that can make the formula true. In other words, 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. It's worth noting that there can be instances where a problem is inaccurately formalized because the formalization accuracy on FOLIO is lower than on ProofWriter. If the answer

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

to the problem being formalized is unknown, this can inadvertently skew the model's performance, making it seem better than it truly is becasue our model can't prove neither the positive stance nor the negative stance of the problem. Nevertheless, to the best of our knowledge, our approach sets a new benchmark on the FOLIO dataset.

There are two types of error that occur during our proving process: timeout errors and running out of goals errors. The former arises when the time set for tactic generation and proof search is exhausted, while the latter occurs when generated tactics have errors, either due to syntactic invalidity or inability to be executed given the current goal, making them unprocessable by LeanDoJo. The likelihood of each error type can be influenced by the chosen beam size during the proof search. Our current approach utilizes a beam size of 64, meaning we generate 64 tactics for every goal we come across. At present, 81.8% of the errors from the ReProver fine-tuned model and 85.5% from the T5 fine-tuned model stem from timeouts. While a thorough inspection of every out-of-nodes error hasn't been conducted, a significant portion seems to arise from incorrect formalization.

6 Related Work

Several past studies (Chen, 2023; Creswell and Shanahan, 2022; Chen et al., 2023b) used neuro-symbolic methods to augment neural networks with symbolic 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 an adaptable framework that melds 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 ready-made solvers.

A common method to boost the reasoning skills of Large Language Models (LLMs) is by training them on data that requires some form of inference. As noted by (Lewkowycz et al., 2022), LLMs that are trained on data filled with science and math data do better on tasks that require reasoning, especially when using CoT prompting. Other work (Fu and Khot, 2022; Fu et al., 2023) suggests that LLMs get their advanced reasoning capabilities from being trained on code. This work is a natural extension of this idea to theorem proving data.

LLMs' intersection with theorem proving has recently become an important topic in NLP. Although some studies delve into various theorem provers (Polu and Sutskever, 2020; Jiang et al., 2023), a consistent focus has been observed around Lean. A distinct advantage of Lean is its array of open-source tools (Yang et al., 2023) which simplify data collection and enable easy interaction with external tools. Predominant research on theorem proving with Lean encompasses strategies such as harnessing intricate proving artifact as seen in (Han et al., 2022a), resorting to curriculum learning (Polu et al., 2023) which capitalizes on theorem provers' ability to verify proofs to generate more training data, and high-level planning reminiscent of the tactics used by AlphaGo as detailed by Lample et al. (2022). For future work, we posit that these methodologies could potentially be repurposed for natural language reasoning.

7 Conclusion

We augmented LLMs with reasoning capabilities by integrating into them Lean, a theorem proving programming language, originally developed for mathematical theorem proving. We examined the source of errors from the formalization of natural language and from proving based on such formalization. We also examined the performance enhancements from pretraining on theorem proving data, and offered a comprehensive comparison with other techniques that highlights our model's superior strengths. Our results underscore the potential of integrating theorem proving frameworks with LLMs in advancing natural language reasoning.

Looking ahead, we aim to improve our method's ability to capture complex real-world situations, especially those filled with commonsense that's hard to express as symbols. One way to attack this problem might be to separate general knowledge representation from logical reasoning. Furthermore, in future work we would like to devise better ways to exploit the reasoning abilities inherent in theorem proving data. This will allow us to solve reasoning tasks more effectively, given that this is a unique resource that involves step-by-step logic and reasoning with a well-defined method of verifying the correctness of an answer.

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's formal language. Consider the MMLU (Hendrycks et al., 2020) and SummEdits (Laban et al., 2023) datasets as examples. MMLU requires the model to possess extensive world knowledge and problem-solving abilities, 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, limiting its utility in these scenarios. Lastly, our method grapples with problems found in more complicated theorem proving datasets like TheoremQA (Chen et al., 2023a). These problems require advanced understanding of natural language, alongside the ability to formalize complex theorems 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 a problem from the LogicalDeduction task of the BigBench dataset (Srivastava et al., 2022), involving the arrangement of three books on a shelf: a black book, an orange book, and a blue book. The problem states that the blue book is to the right of the orange book, and the orange book is to the right of the black book. The question is to confirm whether the black book is the leftmost. 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. In this particular problem, using Lean would require us to formalize the concepts of ordering and relative positioning. Even after doing so, generating a proof would necessitate significant manual labor and wouldn't necessarily yield a readily interpretable answer. In contrast, a Constraint Satisfaction Problem (CSP) solver, which is specifically designed to handle such problems, can effectively manage constraints like the relative order of books and generate potential solutions efficiently.

Ethical Considerations

Incorporating Lean's theorem proving capabilities into Large Language Models (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. By tracing reasoning step by step, it 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

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 co-training 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.

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. 2023. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *ICLR*. OpenReview.net.

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*.

Abdalrhman Mohamed, Wojciech Nawrocki, Tomaz Gomes Mascarenhas, and Haniel Barbosa. 2022. lean-smt. `https://github.com/ufmg-smite/lean-smt`.

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.

Oyvind Tafjord, Bhavana Dalvi, and Peter Clark. 2021. Proofwriter: Generating implications, proofs, and abductive statements over natural language. In *ACL*.

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 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 keep 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 whether the original theorem or 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 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 keep 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 whether the original theorem or 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, 'Goulds wild turkey, 'Merriams 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, 'Goulds 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": *Goulds Tom* ∨ *Eastern Tom* ∨ *Osceola Tom*
- The formalization of numbers is incorrect, it should be: constant weight : Turkey → , axiom A3 : weight Bob = 86, axiom A4 : weight Tom = 73
- The formalization of logic is incorrect, it should be: *(¬ bite_causes_death t ∧ bite_causes_itching t)* ∨ *(bite_causes_death ∧ ¬ bite_causes_itching t)*
- There is an incorrect division of concepts that would make the proving impossible, the correct version should be: ∀ *(t : Turkey), Eastern t* → *bite_causes_death t*