AAAI Press Anonymous Submission Instructions for Authors Using LaTeX

Anonymous submission

Abstract

Large Language Models (LLMs) have shown capabilities in various natural language processing tasks, yet they often struggle with logical reasoning, particularly when dealing with complex natural language statements. To address this challenge, approaches that combine LLMs with symbolic reasoners have been proposed, where the LLM translates the natural language statements into symbolic representations, which are then verified by an external symbolic solver. However, ensuring the syntactic correctness of the symbolic representation remains challenging. In this paper, we introduce GCLLM (Grammar-Constrained Logic Language Model), a method to enhance both the semantic and syntactic accuracy of symbolic representations generated by LLMs. Our approach leverages dynamically selected, semantically similar few-shot examples in the prompt and constrains the LLM output to adhere to a specific grammar. We also implement a self-verification process that utilizes open-source LLMs to correct errors in the generated symbolic representations. Empirical results show that GCLLM improves the performance of similar frameworks by 2-4% in terms of logical reasoning accuracy. More significantly, our method enhances the syntactic accuracy of the generated formulas by up to 21%. These findings demonstrate that that improving the syntactic accuracy of the symbolic representation can help achieve higher semantic accuracy, and the improvement in syntactic accuracy ensures more consistent and dependable interactions with external symbolic solvers.

Introduction

In recent years, Large Language Models (LLMs) [5, 3, 2, 23, 1, 25] have shown increasing capabilities for logical reasoning, especially when guided with prompting techniques such as few-shot examples [19] and chain-of-thought (CoT) prompting [28].

However, the probabilistic nature of LLMs prevents them from reasoning in a true step-by-step and logical fashion, as their responses are simply generated by predicting the most likely token to follow the input. This makes them unreliable when it comes to logical reasoning tasks [30, 7].

To tackle this challenge, an increasingly popular approach is to treat LLMs as logical *parsers* rather than *reasoners*, using them to convert natural language problems into formal representations, to be then solver by an external symbolic solver.

Nevertheless, a persisting challenge is the fact that LLMs cannot guarantee syntactic accuracy [Sec.] of generated formal representations, especially when it comes to task-specific syntax that the model has not seen extensively during training. Even then, there is no guarantee that the output will strictly respect the syntax rules, which can lead to syntactically inaccurate symbolic representations.

To address these challenges, we introduce GCLLM (Grammar-Constrained Logic Language Model), a novel approach that enhances both the semantic and syntactic accuracy of symbolic representations generated by LLMs. Our method leverages dynamically selected, semantically similar few-shot examples [Sec.] in the prompt and constrains the LLM output to adhere to a specific grammar [Sec. . We also implement a self-verification process that utilizes open-source LLMs to correct errors in the generated symbolic representations.

Our empirical results [Sec.] demonstrate that GCLLM improves the performance of similar frameworks by 2-4% in terms of logical reasoning accuracy. Moreover, our method enhances the syntactic accuracy of the generated formulas by up to 21%.

The remainder of this paper is organized as follows: Section provides an overview of related work and background information. We then formally define the problem statement and research questions in Section . Following that, we detail our methodology, including the architecture of GCLLM and its components in Section . We describe our experimental setup and evaluation metrics in Sections and respectively, and present and analyze our results in Section . Finally, we discuss the implications of our findings and potential future directions in Section .

Preliminaries and Related Work

Semantics in ML

In the field of Machine Learning (ML), semantics are used for understanding and representing the meaning of data, particularly in natural language processing tasks. One of the key ways to capture semantic information is through the use of embeddings, which are dense vector representations of words, sentences, or even entire documents.

Mikolov et al. [17] show that words with similar meanings tend to have similar vector representations in the embedding space. For instance, in a well-trained embedding space, the vectors for ”king” and ”queen” would be closer to each other than to the vector for ”apple,” reflecting their semantic relationships.

In-Context Learning

In-context learning refers to the ability of a language model to perform tasks by conditioning on a few examples provided in the input context. This technique leverages the model’s capacity to infer patterns and solutions from the examples without additional fine-tuning. In the context of logical reasoning, in-context learning involves presenting the model with a few examples of natural language problems and their corresponding logical representations.

The properties described in Sec. [], allow us to find in-context examples that are similar to the input problem, which can then be used to guide the model’s generation process, using metrics such as cosine similarity.

Lewis et al. [15] proposed Retrieval-Augmented Generation (RAG), where a retriever extracts relevant passages from a corpus of documents, and passes them to the LLM to guide the generation. RAG demonstrates how retrieved examples can be seamlessly integrated into the generation process, potentially enhancing the accuracy of logical formula generation.

Wang et al. [26] proposed LLM-R, a framework that improves in-context learning by training dense retrievers to find high-quality examples using LLM feedback and knowledge distillation [12]. Experiments across different tasks showed significant performance gains, especially in classification tasks. Moreover, Tang et al. [22] have shown that LLMs can be sensitive to the semantics of the input for symbolic reasoning tasks.

Logical Reasoning with LLMS

Large Language Models (LLMs) have demonstrated capabilities in logical reasoning tasks, sparking research interest in this area. Wei et al. [28] introduced the Chain-of-Thought (CoT) prompting technique, which encourages LLMs to follow the provided examples, to break down complex reasoning tasks into intermediate steps. This approach has shown to be effective at enhancing LLMs’ performance on logical reasoning problems.

Building on CoT, Kojima et al. [14] proposed Zero-shot CoT, demonstrating that simply prompting LLMs to ”think step by step”, without providing any examples, can elicit reasoning capabilities without task-specific examples. This finding highlights the potential of LLMs to perform logical reasoning tasks with minimal guidance.

However, the probabilistic nature of LLMs can lead to inconsistencies in logical reasoning. To address this, Creswell et al. [4] introduced Faithful Reasoning, a framework that combines LLMs with classical automated reasoning tools to ensure logical consistency.

Recent research has focused on combining LLMs with formal logical reasoning systems to leverage the strengths of both approaches. Pan et al. [18] introduced Logic-LM, a framework that integrates LLMs with symbolic solvers to enhance reasoning capabilities. Logic-LM demonstrates improved performance on complex reasoning tasks by leveraging the LLM’s natural language understanding to convert reasoning problems into formal representation, that can be solved by formal solvers.

To address the instances where the formulas generated by the LLM are not syntactically valid, Pan et al. introduce a self-refinement loop, where the LLM is asked to analyze and fix the formulas if the symbolic solver throws an error. While this approach leads to a performance increase, there is no guarantee that the fixed formulas will be syntactically valid after refinement.

Feng et al. [6] proposed LoGiPT, a model aiming to combine the flexibility of LLMs with the precision of logical reasoning, by exposing the solving process of symbolic solvers, and using them to fine-tune an LLM to follow the same problem-solving process.

This approach sacrifices the interpretability and rigor of using an external symbolic solver for the convenience of an end-to-end framework. Moreover, the fact that fine-tuning is required makes LoGiPT less viable for data-scarce domains.

Wang et al. [27] proposed ChatLogic, a framework that enhances LLMs’ logical reasoning capabilities by combining them with a pyDatalog reasoning engine. In line with our work, their framework includes semantic and syntax correction modules to improve generation. Experiments on datasets like PARARULE-Plus showed that ChatLogic-augmented LLMs outperformed baselines in complex reasoning tasks.

In their work, Wang et al. approach syntax correction by prompting the LLM with the error info, to *guide* the correction process. However, they observe that ”syntax corrections are unreliable”. In our work, we instead *enforce* our syntax using grammar-constrained decoding [Sec.], to increase the reliability of syntax corrections.

Self Verification

Self-verification involves using the language model to verify and correct its own outputs. After generating an initial solution, the model is prompted with its own output, and asked to identify and fix any errors. This iterative process helps in refining the model’s outputs and ensuring higher accuracy and reliability.

For instance, consider a simple arithmetic task where a model is asked to solve $12 * 3 + 4$. If the model initially outputs 39, the self-verification process would involve asking the model to check its work. The model might then reason: ”Let’s verify this step-by-step. First, $12 * 3 = 36$. Then, $36 + 4 = 40$. The original answer was therefore incorrect.”

Weng et al. [29] demonstrated the effectiveness of self-verification for improving the reasoning capabilities of large language models. They propose a two-step process of forward reasoning followed by backward verification. In the forward reasoning step, the model generates candidate answers using CoT prompting. In the backward verification step, the model verifies these candidate answers by predicting masked conditions from the original problem.

This allows the model to generate interpretable verification scores for ranking the candidate answers, leading to im-

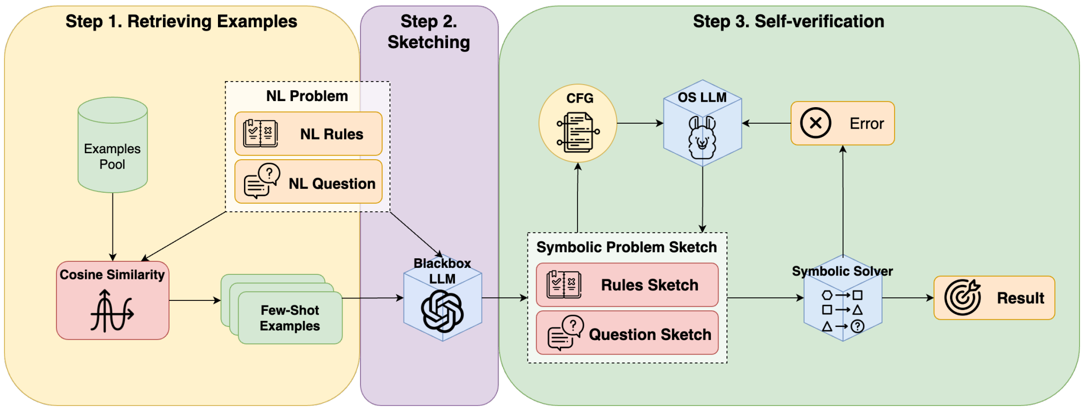

Figure 1: Full pipeline

Listing 1: An example of a CFG grammar

```
S ::= NP VP
NP ::= Det N
VP ::= V NP
Det ::= "the" | "a"
N ::= "cat" | "dog"
V ::= "chases" | "sees"
```

proved performance on various reasoning tasks without requiring additional training or fine-tuning.

Grammar-Constrained Decoding

Grammar-constrained decoding involves guiding the output of a language model using predefined grammatical rules. It is especially useful when the language model has not been extensively trained on domain-specific syntax, and can help in contexts where specific syntax is crucial.

Figure 1 shows an example of a CFG that can be used to generate sentences such as ”the dog chases the cat” or ”a cat sees the dog”, where $V = \{S, NP, VP\}$, $E = \{Det, N, V\}$ and $S = \{S\}$

figure

During the decoding process, the language model's output is restricted to sequences that can be derived from this grammar. The generation follows the production rules, building a parse tree from the root symbol. At each step, the model's vocabulary is filtered to include only grammatically valid tokens, and probabilities are redistributed among these options. This process continues, expanding non-terminals and backtracking when necessary, until a complete, syntactically correct sequence is generated. This ensures that the generated text adheres to the required syntactic structure, reducing parsing errors when interacting with external symbolic reasoners.

Sun et al. [21] proposed GrammarCNN, a method that incorporates grammar knowledge into convolutional neural network architectures. Their approach showed improved performance in tasks requiring structured output, such as code generation and mathematical expression parsing.

Looking at Language Models, Keskar et al. [13] introduced CTRL, a conditional language model trained to generate text with specific attributes, which can be selected by prepending the prompt with a **control code**. For example, one can prepend a prompt with ”Reviews Rating: 5.0” to have the model generate a positive review related to the rest of the prompt. However, this approach is limited by the need to train the model on the selected control codes, which is sometimes not feasible for domain-specific or data-scarce fields.

In order to overcome these limitations, Lu et al. [9] proposed a novel approach to constrain the outputs of language models without requiring access to their internal logits. They use a sketch-based method, where the output of a powerful black-box model is rewritten by a grammar-constrained LLM with logit access.

Our approach builds upon these foundations, adapting grammar-constrained decoding techniques to the specific challenges of generating logical formulas. We extend these methods by incorporating dynamic grammar updates based on the context of each reasoning problem, aiming to improve both the syntactic and semantic accuracy of the generated logical representations.

Problem Statement

Given a natural language reasoning problem x

$$x = (r^{NL}, q^{NL}) \tag{1}$$

where r^{NL} is a list of logical rules in natural language and q^{NL} is a natural language statement to be evaluated.

We aim to extract z, a symbolic representation of the reasoning problem x:

$$\hat{z} = \lambda(x) = (r^{SYM}, q^{SYM}) \tag{2}$$

where r^{SYM} is a list of symbolic rules, q^{SYM} is a symbolic statement to be evaluated, and λ is an LLM pipeline. Then we solve the reasoning problem by using a symbolic solver on the symbolic representations:

$$\hat{y} = \phi(\lambda(x)) \tag{3}$$

Where ϕ is a symbolic solver and z is the result (True, False or Uncertain), with the goal of maximizing the accuracy between the ground truth answers y and the predicted answers $\hat{y}$

Research questions

Our work focuses on the following research questions:

R1: Can we use **embedding distance** to select appropriate examples for a given reasoning problem, in order to improve the **semantic accuracy** of the logical representations?

R2: Can we combine the **self-verification** abilities of LLMs and **Grammar Constrained Decoding (GCD)** to improve the **syntactic accuracy** of the generated logical representations? How do they affect the semantic accuracy?

Contributions

The primary contributions of this work are as follows:

Automatic Example Selection: We propose a method to automatically select in-context examples using embedding distances.

Grammar-Constrained Self-verification: We implement self-verification process, with a dynamically updated grammar constraint mechanism, ensuring that the generated text adheres to the required syntactic structure. This significantly reduces parsing errors when interacting with external symbolic reasoners.

Datasets: We propose three datasets derived from the FOLIO [11] and LogicNLI [24] datasets:

- *FOLIO-Prover9* [Sec.]: A dataset that converts FOLIO's logical representations into Prover9-compatible syntax, enabling direct use with the Prover9 [16] symbolic solver.
- *FOLIO-Refinement* [App.]: A synthetic dataset created by introducing syntax errors into FOLIO's FOL formulas, designed to evaluate and improve self-verification techniques.
- *LogicNLI-Prover9* [App.]: A dataset that transforms LogicNLI's tree-like logical structures into First-Order Logic notation compatible with Prover9, expanding the range of problems that can be evaluated using our framework.

Methodology

To address the challenges of generating accurate and syntactically valid logical representations from natural language problems, we propose a three-step pipeline: Retrieving Relevant Examples Sketching and Self-verification.

In Step 1, we implement a dynamic example selection method using embedding similarity to provide more relevant context for each test sample. This aims to improve the semantic accuracy of generated logical formulas by tailoring the few-shot examples to the specific problem.

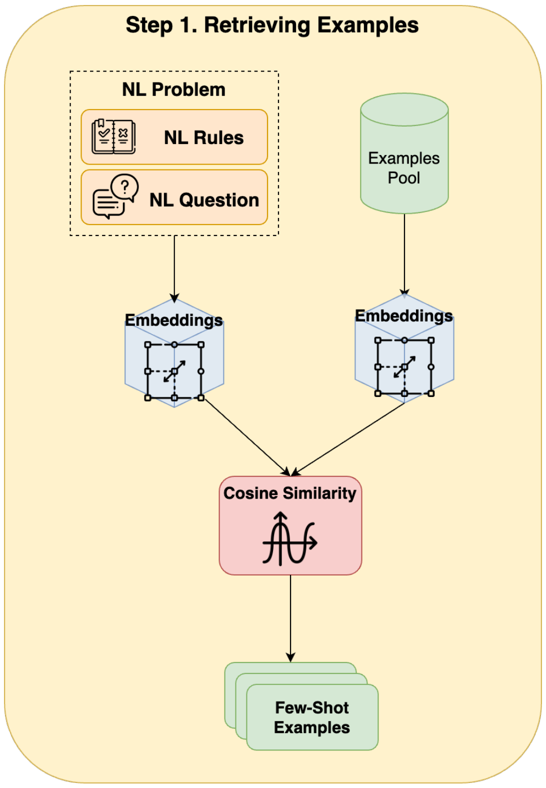

Figure 2: Pipeline for examples retrieval

In Step 2, we leverage a powerful black-box language model to generate an initial symbolic problem sketch from the natural language input. This serves as the foundation for our logical representations, capturing the core structure and relationships expressed in the problem.

Finally, in Step 3, we combine execution error correction, parsing error correction, and grammar-constrained generation. This step is designed to improve the syntactic validity [] of the generated logical formulas.

We illustrate our methodology in Figure 1. For details on the design of our prompts, see Sec. .

Step 1: Retrieving Relevant Examples

The dynamic example selection method aims to provide the language model with context that is more relevant to the specific problem at hand. By choosing examples that are semantically similar to the test sample, we hypothesize that the model will be better equipped to generate accurate logical representations.

For each test sample, we retrieve the most similar problems from our datasets' training sets by ranking them according to *cosine score* of their embeddings (Fig. 2)

Let $P^{train} = \{(x_i^{train}, z_i^{train}, y_i^{train})_{i=1}^{|P^{train}|}\}$ and $P^{test} = \{(x_j^{test}, z_j^{test}, y_j^{test})_{i=1}^{|P^{test}|}\}$ be the set of problems in the training split and in the test split, respectively.

For each NL problem $x_i^{train} \in P^{train}$ and $x_j^{test} \in P^{test}$, we use OpenAI's `text-embedding-3-large` model to compute the text embeddings and generate pairs $(x_i^{train}, e_i^{train})$ and (x_j^{test}, e_j^{test}). We chose this model due to its long context window of 8192 tokens, which allows us to embed entire problems without truncation, and its state-of-the-art performance in capturing semantic relationships [[mteb cite]].

Then, for each test pair (x_j^{test}, e_j^{test}), we compute the *cosine similarity score* with all training pairs $(x_i^{train}, e_i^{train})$:

$$\frac{e_j^{test} \cdot e_i^{train}}{||e_j^{test}|| \times ||e_i^{train}||} \tag{4}$$

For each test sample x_j^{test}, we select the top k most similar training examples for a problem, ranked by cosine similarity, to obtain a set of examples D_j for the test sample.

$$D_j = \{(x_i^{train}, z_i^{train}\}_{i=1}^{k} \tag{5}$$

Where z_i^{train} is the symbolic problem corresponding to the natural language problem x_i^{train}.

Step 2: Sketching

In this step, we follow the approach of Geng et al. [9], using a black-box LLM M_{sk} to sketch the symbolic problem from the NL problem x_j^{test} (Fig. 3). This step is required because parsing Natural Language problems into symbolic representations is a complex task, and the small open-source models we use for self-verification are not powerful enough to complete it succesfully.

The model interprets a task description T_{sk}[Lst. 7], together with a set of examples D_j, to convert the NL statements into formulas:

$$\hat{y}_j^{test} = \underset{s \in S}{\operatorname{argmax}}\, M_{sk}(s|T_{sk}, D_j, x_j^{test}) \tag{6}$$

where S is the set of all possible string sequences.

Step 3: Self-verification

Our self-verification mechanism consists of two main components: *execution error* correction and *parsing error* correction. After sketching the symbolic problems $\hat{y}$, we attempt to solve them using the symbolic solver ϕ. If the solver fails, either due to a parsing error or execution error, we expose the error to an LLM, and exploit its self-verification abilities to correct the mistake [Fig. 4].

For execution errors, we observe that they are mainly related to mistakes with the *whole symbolic problem* as scope. For example, the sketch $\hat{y}$ might contain a predicate `Love(x)` in one formula and `Love(x,y)` in another, which is not allowed since the arity of a predicate cannot change. Therefore, we provide a black-box LLM with the full sketch $\hat{y}_j^{test}$ and the error message, prompting it to regenerate it completely.

For parsing errors, we observe that they are related to mistakes with *a single formula* as scope. For example, the sketch $\hat{y}$ might contain a formula `Love(x,y)` $\wedge \exists$`x`, which is not allowed since quantifier must always be at the beginning of a formula. In these cases, we use an open-source (OS) LLM to generate valid symbolic formulas constrained by a dynamically updated CFG.

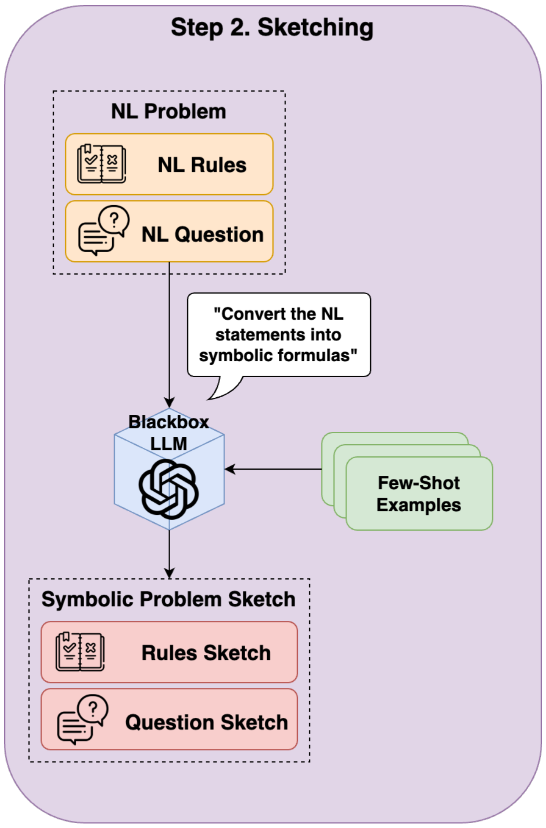

Figure 3: Pipeline for sketching symbolic problems

Execution error Given a valid sketch w_j, and a solving error g, we prompt a black-box LLM M_{sk} with a task T_{ex} [Lst. 11] and a set of error correction examples D_{ex} to correct the sketch:

$$\hat{y}_j^{n+1} = \underset{s \in S}{\operatorname{argmax}}\, M_{sk}(s|T_{ex}, D_{ex}, g, \hat{y}_j^{n}) \tag{7}$$

Parsing error Given an invalid symbolic formula in a sketch $\hat{f}_l \in \hat{y}_j$, $0 \leq l \leq |\hat{y}_j|$, and its corresponding natural language statement $s_l \in x_j$, we aim to generate a valid symbolic formula $\hat{h}_l$.

To make sure that the new symbolic formula is syntactically correct, we constrain the LLM's generation with a Context-Free-Grammar (CFG) $G = (\mathcal{V}, \mathcal{E}, \mathcal{R}, \mathcal{S})$, where $\mathcal{V}$ is the finite set of non-terminals; $\mathcal{E}$ is the finite set of terminals; $\mathcal{R}$ is the finite set of production rules and $\mathcal{S}$ is the starting symbol.

Since black-box models do not provide logit access, which is needed to implement the constrained generation, we experiment with various open-source LLMs.

Moreover, in order to make the CFG tailored to $\hat{y}_j$, we extract the predicates and use them to dynamically update the set of terminals $\mathcal{E}$.

To do so, we first extract the sketch's logic predicates $\hat{Q}$, and update the set of terminals with $\hat{Q} \subset \mathcal{E} \in G$.

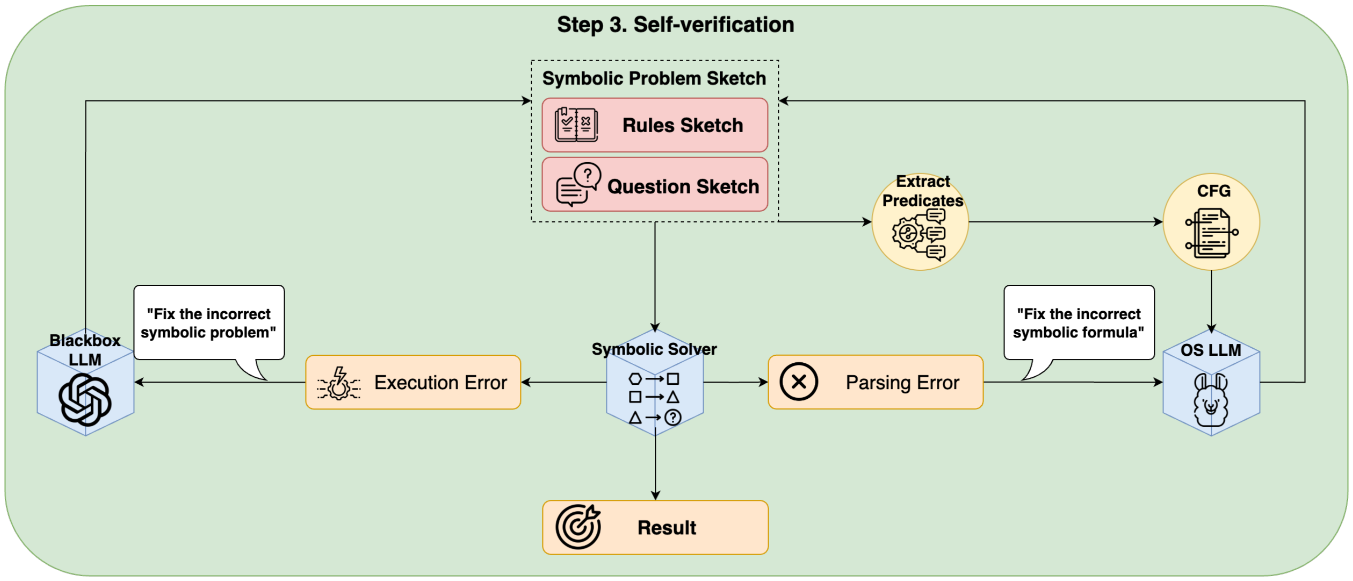

Figure 4: Pipeline for self-verification

Then, we initialize an open-source LLM with logit access M_{logit}, constrained by the grammar G, to which we provide a set of examples $D_{par} = \{(s_i^{train}, f_i^{train}, h_i^{train})\}_{i=1}^{k}$ and a task description T_{par} [Lst. 9].

Finally, we generate the valid formula, constraining M_{logit}'s generation as described in [10].

$$\hat{h}_l = \underset{s \in S^*}{\operatorname{argmax}}\, M_{logit}(s|T_{par}, D_{par}, s_l, \hat{f}_l) \quad (8)$$

Where S^* is the set of sequences that can be generated following grammar G.

Experiments

In this section, we illustrate the experiments we conducted to evaluate our framework. To account for the stochastic nature of LLM outputs, we run multiple trials for each configuration, reporting mean scores and standard deviations.

Baselines

End-to-end reasoning with LLM We leverage LLMs to perform direct reasoning on the NL problems without any intermediate logical representation. The process is as follows:

1. Input: We provide the LLM with the NL rules and question from the dataset, as well as some examples.
2. Prompting: We instruct the LLM to analyze the given information and determine whether the statement in question is True, False, or Uncertain based solely on the provided rules.
3. Output: The LLM generates a direct answer (True, False, or Uncertain) along with a brief explanation of its reasoning.

Logic-LM We implement the work of Pan et al. [18], Logic-LM, which does not make use of either Example Retrieval or GCD. We compare the performance of their framework to ours in order to assess the impact of these techniques.

Retrieving Relevant Examples

To investigate the impact of example selection on the performance of our pipeline, we implement and compare two approaches for providing few-shot examples to the language model:

Static Examples In this approach, we select a fixed set of examples from the training set that are used for all test samples. These examples are chosen to cover a range of logical structures and problem types representative of the dataset.

Dynamic Examples This method involves selecting examples tailored to each specific test sample. We implement this approach as described in .

To quantify the impact of dynamic example selection, we compare the semantic accuracy [Sec.] of the generated logical representations for both static and dynamic approaches. This comparison allows us to assess whether tailoring examples to each test sample leads to improved performance in logical reasoning tasks.

Symbolic Solver

For our symbolic solver, we chose Prover9 [16], a widely accepted automated theorem prover for first-order logic (FOL). Following the implementation approach of Pan et al. in Logic-LM [18], we integrated Prover9 into our pipeline through Python's NLTK library, to evaluate both the syntactic validity and the outcome of the generated formulas.

Grammar-Constrained Self-verification

To evaluate our grammar-constrained self-verification approach, we implement it using various open-source LLMs with logit access, constrained by our dynamically updated Context-Free Grammar (CFG).

We use the process described in equation 8, where the LLM's generation is constrained by the grammar G, which is dynamically updated based on the predicates extracted from the initial sketch.

Fine-tuned Self-verification

We also evaluate the impact of fine-tuning on self-verification performance. To do so, we propose **FOLIO-Refinement**, a synthetic dataset created by introducing syntax error into the FOL formulas of the FOLIO dataset.

After fine-tuning the OS LLMs, we integrate them into our self-verification pipeline, replacing their non-fine-tuned counterparts. We then compared the performance of the fine-tuned models against both the non-fine-tuned versions and the unconstrained self-verification approach.

Self-verification with Backup

Finally, we introduce an additional strategy to handle samples that remain unexecutable after the self-verification process. Instead of assigning these samples the control label as described in Sec. , we adopt an approach similar to LogicLM [18]. For these cases, we directly employ an LLM to generate the answer to the logical problem.

To implement this strategy we promp the LLM with the original natural language problem and question, following the approach described in Sec. . We then use the LLM's output as the final answer for samples that could not be processed through our main pipeline.

Our aim is to provide a response for all input samples, potentially improving the overall performance of our system. This approach also enables us to compare the effectiveness of our logical reasoning pipeline against direct LLM inference on challenging cases.

We measure the performance of the experiments in Sections - in terms of syntactic validity [Sec.] and semantic accuracy [Sec.] of the generated logical representations.

Evaluation

Control Label

To account for problems that our pipeline cannot run, we propose an additional label (**control label**) other than the ones described in , called "N/A". This label is assigned to programs that, after the self-verification rounds have finished, are still not runnable by the symbolic solver.

Since we chose to evaluate our results using the F1-score, as described in , and since no sample in our datasets is labelled as "N/A", the problems that are assigned it can only ever be **False Positives**.

Therefore, all problems that cannot be ultimately represented as a syntactically correct logical representation, will lead to a degradation of **Precision**, and consequently of **F1-score** as a whole.

This choice allows us to fairly evaluate our pipeline even when it's unable to produce an output for a given test problem.

Metrics

Semantic Accuracy We measure the semantic accuracy of the generated programs by running them through the symbolic solver, and comparing the output with the ground truth.

For each label ("True", "False" or "Uncertain"), we define True Positives, False Positives and False Negatives as follows:

- **True Positive (TP)**: an instance that is correctly predicted as its actual label. (e.g. is predicted as "True" and is actually "True")
- **False Positive (FP)**: An instance that is incorrectly predicted as a particular label when it actually belongs to a different label. (e.g., predicted as "True" when it is actually "False" or "Uncertain")
- **False Negative (FN)**: An instance that actually belongs to a particular label but is incorrectly predicted as a different label. (e.g., actually "True" but predicted as "False" or "Uncertain")

Note that, when a prediction is a FP for one class, it is also a FN for another. For example, if a statement is actually "False" but the model predicts it as "True", this is an FP for the "True" label and simultaneously a False Negative (FN) for the "False" label.

We calculate both multi-label and single-label metrics as follows:

1. **Single-label:** For each label, we measure
   - **Precision**, calculated as $\frac{TP}{TP+FP}$
   - **Recall**, calculated as $\frac{TP}{TP+FN}$
   - **F1-score**, calculated as $2 * \frac{precision*recall}{precision+recall}$
2. **Multi-label:** For multi-label metrics, we use weighted-averaging, by obtaining the metrics for each label, and then calculating the average weighted by each label's frequency:

$$F1_{\text{weighted}} = \frac{\sum_{i=1}^{n} w_i \cdot F1_i}{\sum_{i=1}^{n} w_i} \tag{9}$$

   where $F1_i$ is the F1-score for label i, w_i is the support for label i, and n is the number of labels.

We choose the F1-score as our primary metric for evaluating semantic accuracy due to its balanced nature in capturing both precision and recall. This choice is particularly relevant for our multi-class classification task, where we need to account for True, False, and Uncertain labels.

This approach allows us to fairly evaluate our pipeline even when it's unable to produce an output for a given test problem, as discussed in Section []. By using the weighted F1-score, we ensure that the control label, which has zero support in the ground truth, appropriately penalizes the overall score when the pipeline fails to produce a valid output.

		FOLIO	LogicNLI
Sketcher	**Example selection method**	**F1-Score**	**F1-Score**
GPT-3.5-Turbo	Static Examples	$50.5^{\pm 0.5}\%$	$23.17^{\pm 0.0}\%$
	Dynamic Examples	$47.5^{\pm 1.2}\%$	$20.97^{\pm 0.1}\%$
GPT-4o	Static Examples	$66.12^{\pm 1.3}\%$	$44.58^{\pm 0.0}\%$
	Dynamic Examples	$\mathbf{67.45^{\pm 0.9}}\%$	$\mathbf{46.09^{\pm 0.2}}\%$

Table 1: Pipeline performance with static and dynamic example retrieval.

Syntactic Accuracy We measure the syntactic accuracy of the generated programs by running them through the symbolic solver, and observing the percentage of generated programs that can be ran by the solver without incurring in an error.

Datasets

FOLIO The FOLIO dataset [11]contains natural language problems that need to be converted into logical form for reasoning. It is made of a train split, containing 1000 samples and a test split, containing 203 samples. The label balancing is shown in table 2. We chose it as it is one of the most widely accepted dataset to assess Natural Language Reasoning with First-Order-Logic

```
"Natural Language Rules":[
  "All squares have four sides.",
  "All four-sided things are shapes."]

"Logical Rules": [
  "∀x (Square(x) → FourSides(x))",
  "∀x (FourSides(x) → IsShape(x))"]

"Natural Language Question":
  "All squares are shapes."

"Logical Question":
  "∀x (Square(x) → IsShape(x))"

"Answer": "True"
```

Figure 5: A sample of the FOLIO dataset

Each entry [Fig. 5] in the dataset contains:

- Natural Language Rules (r^{NL}): A list of premises described in natural language. Each rule describes a specific condition or fact.
- Natural Language Question (q^{NL}): A statement that needs to be evaluated based on the given rules.
- Logical Rules (r^{LOG}): The logical form of the premises.
- Logical Question (q^{LOG}): The logical form of the statement to be evaluated.
- Answer (y): The label of the logical problem.

During our experiments, we found that some of the logical representations in the FOLIO dataset were not directly compatible with the symbolic solver we used, Prover9 [16]. This incompatibility was due to differences in the syntactical requirements of Prover9 compared to the logical forms provided in the dataset.

Label	**FOLIO**	**LogicNLI**
True	34%	33%
False	29%	33%
Uncertain	37%	33%

Table 2: Label balance of FOLIO and LogicNLI

To address this issue, we propose **FOLIO-Prover9**, a new dataset created by converting the incompatible logical representations into ones that Prover9 could process, using the syntax described in . We ensured that these converted representations were logically equivalent to the original ones, thereby maintaining the integrity of the symbolic problems.

To do so, we extracted all incompatible formulas from the *dev* split, manually converted them, and finally ran them through Prover9, ensuring that the output was the same as the sample's ground truth.

Moreover, we found that the samples in the *train* split did not contain any Logical Question q^{LOG}. Therefore, we manually annotate **75** randomly selected samples with the corresponding Logical Questions, to be used in the example retrieval [Sec.].

LogicNLI The LogicNLI dataset [24] evaluates the first-order logic (FOL) reasoning capabilities of language models. It contains a train split with 16,000 instances and a test split with 2,000 instances. Each entry [1] includes:

- Natural Language Rules and Facts(r^{NL}): A list of rules and facts described in natural language. Each rule describes a specific condition or relations between facts.
- Natural Language Questions ($\{q_i^{NL},\ \ 0 < i < n\}$): A list of statements that need to be proven based on the given rules.
- Answers ($\{z_i,\ \ 0 < i < n\}$)

In order to use this dataset with Prover9 [16], we need to convert its logical representations r^{NL} into a format that the solver can use. To do so, we propose **LogicNLI-Prover9** [App.], a new dataset created by automatically converting the formulas of LogicNLI using the syntax described in .

Results and discussion

Our experiments yielded several insights into the performance of GCLLM. Overall, we observed improvements in both syntactic validity and semantic accuracy of generated logical formulas compared to baseline methods.

			FOLIO			
Sketcher	**Strategy**	**Refiner**	**F1**	**F1 + Backup**	**Executable samples (%)**	**F1**
GPT-3.5-Turbo	Chain-of-Thought	N/A	**0.5189**	N/A	N/A	**0.3577**
	LogicLM	GPT-3.5-Turbo	0.4871	**0.6245**	62.74	[HTML]FE0000 0
	GCLLM	Llama 2 7b	0.4953	0.6151	72.57	0.2133
		Llama 2 7b	0.5040	0.5754	**83.57**	0.2137
		Llama 2 7b	0.4962	0.6194	71.42	0.2186
		Llama 2 7b	0.4961	0.5773	83.25	0.2161
GPT-4o	Chain-of-Thought	N/A	**0.7160**	N/A	N/A	**0.7263**
	LogicLM	GPT-4o	[HTML]FE0000 0.0558	0.7138	[HTML]FE0000 3.43	[HTML]FE0000
	GCLLM	Llama 2 7b	0.6726	0.7339	87.84	0.4825
		Llama 2 7b	0.6772	0.7251	91.95	0.4700
		Llama 2 7b	0.6745	**0.7377**	87.84	0.4621
		Llama 2 7b	0.6898	0.7329	**93.92**	0.4717

Table 3: F1 Score of the generated logic programs after 3 rounds of self-verification.
: Grammar-Constrained Model
: Fine-tuned Model

The dynamic example selection approach [Tab. 1] showed mixed results, with slight improvements for more advanced models like GPT-4o, but a decrease in performance for GPT-3.5-Turbo. This suggests that the effectiveness of semantic similarity-based example selection may depend on the underlying capabilities of the language model.

Self-verification with grammar-constrained LLMs [Tab. 3] led to improvements in the **syntactic accuracy** [Sec.] of generated programs of up to 21%, albeit with a slight trade-off in **semantic accuracy** [Sec.] compared to self-verification with black-box LLMs. The trade off appears to be bigger when inferring on the LogicNLI [Sec.] dataset, and we suspect that this is due to its logical problems having more rules compared to FOLIO.

Conversely, fine-tuning the models seems to to lead to slight improvements 1% in the semantic accuracy on the FOLIO dataset, at the cost of the syntactic accuracy. It is interesting to notice how combining the two techniques seems to achieve the best balance of semantic and syntactic accuracy, pointing to the fact that the two techniques complement each other nicely in this setting.

Finally, we observe that the LogicLM framework is sometimes almost completely unable to generate symbolic problems that are compatible with Prover9 (Table 3, results in red). Upon further inspection, this is caused by the fact that the Sketchers (GPT-3.5-Turbo and GPT-4o) do not follow the instructions in the prompt, making the LogicLM framework unable to parse the responses and properly submitting them to the symbolic solver [Lst. 13].

Dymanic Examples

Our results in table 1 show that selecting the few-shot examples dynamically, according to embedding similarity with the test sample, improves the **semantic accuracy** of the generated symbolic problems by $1-2\%$ when using GPT-4o, and degrades it by 3% when using GPT-3.5-Turbo. This seems to suggest that when it comes to generating logical formulas from Natural Language statements, these models are fairly agnostic to the semantic similarity [Sec.] of the few-shot-examples with the test problem.

Self verification

The results of our self-verification experiments are presented in Table 3. Overall, we observed improvements in both syntactic validity and semantic accuracy of generated logical formulas across different configurations. The table shows the performance of various sketcher-refiner combinations, including grammar-constrained and fine-tuned models, as well as the impact of implementing a backup strategy for unexecutable samples.

Grammar-constrained self-verification The impact of Grammar-Constrained self-verification can be seen by comparing the rows with and without the symbol. For example, with GPT-4o as the sketcher and Llama 2 7b as the refiner, implementing grammar constraints increased the F1 score from 0.6726 to 0.6772, as well as improving the percentage of executable samples from 87.84% to 91.95%. This demonstrates that our grammar-constrained approach effectively enhances the syntactic validity of generated formulas, albeit with a slight trade-off in **semantic accuracy** [Sec.] in some cases.

Fine-tuned self-verification The results of fine-tuning on our FOLIO-Refinement dataset are indicated by the symbol. For instance, with GPT-4o as the sketcher and Llama 2 7b as the refiner, fine-tuning slightly improved the F1 score from 0.6726 to 0.6745. However, when combined with grammar constraints (), we see a more substantial improvement, with the F1 score increasing to 0.6898 and executable samples reaching 93.92%. These results suggest that while fine-tuning alone may have limited impact, it can complement Grammar-Constrained techniques effectively.

Self-verification with backup The ”F1 + Backup” column reveals an interesting insight about the effectiveness of different approaches to logical reasoning. While GCLLM with self-verification and backup shows improvements over the standard LogicLM approach, it's notable that Chain-of-Thought (CoT) reasoning with black-box LLMs still achieves the best overall performance. For GPT-3.5-Turbo, the CoT approach yields an F1-score of 0.5189, which outperforms all GCLLM configurations. Similarly, for GPT-4o, the CoT method achieves an F1-score of 0.7160, surpassing the best GCLLM configuration. These results suggest that when it comes to solving complex logical problems, the direct reasoning capabilities of advanced language models, guided by CoT prompting, remain highly effective.

Conclusion and Future Work

This paper introduced GCLLM, a framework enhancing both semantic and syntactic accuracy of symbolic representations generated by LLMs for logical reasoning tasks. By leveraging dynamically selected few-shot examples and grammar-constrained decoding, our approach improved logical reasoning accuracy by 2-4% and **syntactic accuracy** [Sec.] by up to 21% compared to similar frameworks.

We successfully addressed one of the key limitations in LogicLM [18]: ensuring syntactic correctness of LLM-generated symbolic representations. Our grammar-constrained decoding and self-verification process improved the reliability of generated formulas and their compatibility with the external symbolic solver.

For future work, it would be interesting to explore how these results translate to tasks beyond logical reasoning. The principles of grammar-constrained decoding could potentially apply to a wide range of structured output tasks, such as mathematical formula derivation, theorem proving and constraint satisfaction.

Another research direction could be to generate a synthetic dataset of symbolic formulas with an LLM, and then using that dataset to fine-tune the open-source refiner. This would provide insights on whether fine-tuning the refiner on large amounts of synthetic data can lead to a significant improvement in semantic/syntactic accuracy, potentially compensating the current trade off of GCD.

Finally, it would be insightful investigating the performance of grammar-constrained massive LLMs, like the GPT [2], Gemini [23] or some large variant of Llama [25]. While we focused on smaller, open-source models due to practical constraints, applying these techniques to state-of-the-art massive language models could potentially reduce the trade off between syntactic and semantic accuracy. This could encourage the development of new methods for constraining the output of black-box models or finding ways to efficiently implement grammar constraints in the decoding process of larger models.

Acknowledgment

References

[1] ???? The Claude 3 Model Family: Opus, Sonnet, Haiku.

[2] Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

[3] Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901.

[4] Creswell, A.; Shanahan, M.; and Higgins, I. 2022. Selection-Inference: Exploiting Large Language Models for Interpretable Logical Reasoning. arXiv:2205.09712.

[5] Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805.

[6] Feng, J.; Xu, R.; Hao, J.; Sharma, H.; Shen, Y.; Zhao, D.; and Chen, W. 2023. Language Models can be Logical Solvers. arXiv:2311.06158.

[7] Friedman, R. 2023. Large Language Models and Logical Reasoning. *Encyclopedia*, 3(2): 687–697.

[8] gbnfdocumentation. ???? Ggerganov/llama.cpp, GBNF Guide.

[9] Geng, S.; Döner, B.; Wendler, C.; Josifoski, M.; and West, R. 2024. Sketch-Guided Constrained Decoding for Boosting Blackbox Large Language Models without Logit Access. arXiv:2401.09967.

[10] Geng, S.; Josifoski, M.; Peyrard, M.; and West, R. 2024. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. arXiv:2305.13971.

[11] Han, S.; Schoelkopf, H.; Zhao, Y.; Qi, Z.; Riddell, M.; Zhou, W.; Coady, J.; Peng, D.; Qiao, Y.; Benson, L.; Sun, L.; Wardle-Solano, A.; Szabo, H.; Zubova, E.; Burtell, M.; Fan, J.; Liu, Y.; Wong, B.; Sailor, M.; Ni, A.; Nan, L.; Kasai, J.; Yu, T.; Zhang, R.; Fabbri, A. R.; Kryscinski, W.; Yavuz, S.; Liu, Y.; Lin, X. V.; Joty, S.; Zhou, Y.; Xiong, C.; Ying, R.; Cohan, A.; and Radev, D. 2024. FOLIO: Natural Language Reasoning with First-Order Logic. .

[12] Hinton, G.; Vinyals, O.; and Dean, J. 2015. Distilling the Knowledge in a Neural Network. arXiv:1503.02531.

[13] Keskar, N. S.; McCann, B.; Varshney, L. R.; Xiong, C.; and Socher, R. 2019. CTRL: A Conditional Transformer Language Model for Controllable Generation. arXiv:1909.05858.

[14] Kojima, T.; Gu, S. S.; Reid, M.; Matsuo, Y.; and Iwasawa, Y. 2023. Large Language Models are Zero-Shot Reasoners. arXiv:2205.11916.

[15] Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; tau Yih, W.; Rocktäschel, T.; Riedel, S.; and Kiela, D. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401.

[16] McCune, W. 2005–2010. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`.

[17] Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.; and Dean, J. 2013. Distributed Representations of Words and Phrases and their Compositionality. arXiv:1310.4546.

[18] Pan, L.; Albalak, A.; Wang, X.; and Wang, W. Y. 2023. Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning. arXiv:2305.12295.

[19] Parnami, A.; and Lee, M. 2022. Learning from few examples: A summary of approaches to few-shot learning. *arXiv preprint arXiv:2203.04291.*

[20] recursivegrammarissue. ???? Ggerganov/llama.cpp, Issue #7572, "Bug: GBNF repetition rewrite results in unsupported left recursion".

[21] Sun, Z.; Zhu, Q.; Mou, L.; Xiong, Y.; Li, G.; and Zhang, L. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, 7055–7062.

[22] Tang, X.; Zheng, Z.; Li, J.; Meng, F.; Zhu, S.-C.; Liang, Y.; and Zhang, M. 2023. Large Language Models are In-Context Semantic Reasoners rather than Symbolic Reasoners. arXiv:2305.14825.

[23] Team, G.; Anil, R.; Borgeaud, S.; Wu, Y.; Alayrac, J.-B.; Yu, J.; Soricut, R.; Schalkwyk, J.; Dai, A. M.; Hauth, A.; et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805.*

[24] Tian, J.; Li, Y.; Chen, W.; Xiao, L.; He, H.; and Jin, Y. 2021. Diagnosing the First-Order Logical Reasoning Ability Through LogicNLI. In Moens, M.-F.; Huang, X.; Specia, L.; and Yih, S. W.-t., eds., *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 3738–3747. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics.

[25] Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.-A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971.*

[26] Wang, L.; Yang, N.; and Wei, F. 2024. Learning to Retrieve In-Context Examples for Large Language Models. arXiv:2307.07164.

[27] Wang, Z.; Liu, J.; Bao, Q.; Rong, H.; and Zhang, J. 2024. ChatLogic: Integrating Logic Programming with Large Language Models for Multi-Step Reasoning. arXiv:2407.10162.

[28] Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q. V.; Zhou, D.; et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35: 24824–24837.

[29] Weng, Y.; Zhu, M.; Xia, F.; Li, B.; He, S.; Liu, S.; Sun, B.; Liu, K.; and Zhao, J. 2023. Large Language Models are Better Reasoners with Self-Verification. arXiv:2212.09561.

[30] Ye, X.; and Durrett, G. 2022. The unreliability of explanations in few-shot prompting for textual reasoning. *Advances in neural information processing systems*, 35: 30378–30392.

LogicNLI-Prover9

The LogicNLI dataset does not annotate the samples using First-Order-Logic notation, but using a tree-like structure that contains the logical relations between predicates, variables and constants, as shown in Listing 1. Since this format is not compatible with the Prover9 solver, we curate a new dataset, **LogicNLI-Prover9**, by converting the logic trees into First-Order-Logic notation.

The conversion was accomplished by systematically translating each element of the tree-like structure into corresponding First-Order-Logic (FOL) expressions. For each logical fact, the subject and attribute were extracted and formatted into a standard FOL predicate notation, with polarity indicating the presence or negation of the attribute. Rules were more complex, requiring the handling of quantifiers (universal $\forall$ or existential $\exists$), and conjunctions or disjunctions within premises and conclusions. By iterating through the dataset, premises were identified and grouped, taking into account shared subjects to apply quantifiers appropriately. Each rule was then translated into an implication ($\rightarrow$) or bi-conditional ($\leftrightarrow$) statement in FOL, depending on its type. Finally, questions were similarly converted into FOL expressions, resulting in a comprehensive dataset where each original logic tree entry is paired with its corresponding FOL notation, ensuring compatibility with automated theorem proving tools like Prover9.

FOLIO-Refinement

The FOLIO-Refinement dataset was created to fine-tune the models used in the self-verification process. This dataset was derived from the original FOLIO dataset by introducing controlled errors into the FOL representations.

Introducing Errors We introduce 5 types of errors into the FOL formulas:

- **Quantifier Misplacement**: Randomly selects a quantifier and a logical operator, then moves the quantifier to immediately follow the operator.
- **Parentheses Manipulation**: Either removes a random pair of parentheses or adds a new pair at random positions.
- **Variable Scope Alteration**: Swaps the positions of two randomly selected quantifiers, potentially altering variable scopes.

```
"Natural Language Facts": [
    "Eli is not jittery.",
    "Patricia is civil."]

"Natural Language Rules": [
    "If someone is southern, then he is neither jittery nor soft."]

"Natural Language Question":
    "Eli is jittery."

"Logical Facts":[
    "0": ["Eli", "soft", "-", "fact 0"],
    "1": ["Patricia", "civil", "+", "fact 1"]
    ]

"Logical Rules": [
    {
    "p": {"fact": [["all", "southern", "+"]], "conj": "none"},
    "q": {"fact": [["all", "jittery", "-"], ["all", "soft", "-"]], "conj": "and"},

    "type": "imp",
    "reasoning": "AIC",
    "class": 4
    }

"Logical Question": ["Eli", "jittery", "+"]

"Answer": "False"
```

Listing 1: A sample of the original LogicNLI dataset

```
"Natural Language Rules": [
    "Eli is not jittery.",
    "Patricia is civil.",
    "If someone is southern, then he is neither jittery nor soft."]

"Natural Language Question":
    "Eli is jittery."

"Logical Rules:[
    "¬Jittery(eli)",
    "Civil(patricia)",
    "(∀x (Southern(x))) → (∀x (¬Jittery(x) ∧ ¬Soft(x)))"]

"Logical Question":[
    "Jittery(eli)"]

"Answer": "False"
```

Listing 2: A sample of the LogicNLI dataset after conversion

- **Quantifier Removal**: Randomly removes a quantifier from the formula.
- **No Error**: Leaves the formula unchanged, to maintain a balance of correct formulas in the dataset.

For each FOL formula in the original dataset, one of these error functions (including the no-error function) was randomly selected and applied.

Creating training examples Each training example in the FOLIO-Refinement dataset consists of:

- A natural language statement from the original FOLIO context
- The type of error introduced (or ”no_error” if unchanged)
- The ”sketch” FOL formula with the introduced error
- The correct FOL formula
- The set of valid predicates for the problem

Generating Prompts For each training example, we created a prompt suitable for fine-tuning LLMs. The prompt includes:

- A system message describing the task and rules for FOL formula generation
- A user message containing the natural language statement, the erroneous sketch, and the set of valid predicates
- The expected output (the correct FOL formula)

This process ensures that the FOLIO-Refinement dataset contains a diverse range of syntactic errors that self-verification models might encounter when processing FOL formulas.

Grammars

In our implementation, we utilize context-free grammars (CFGs) to constrain the generation of First-Order Logic (FOL) formulas. These grammars are written in the GBNF (Graydon’s BNF) format, a variation of the Backus-Naur Form specifically designed for use with language models. For a detailed explanation of the GBNF syntax, refer to the official documentation [8].

Initially, we designed a recursive grammar capable of handling nested formulas of arbitrary depth [Lst. 4]. However, due to limitations in the llama.cpp library, which we use for interfacing with open-source language models, we encountered issues with recursive rules, documented in [20]. To overcome this constraint, we modified our approach by unrolling the grammar to explicitly handle formulas nested up to three levels deep [Lst. 3].

Prompts

When designing our prompts, we followed the implementation of [18]. However, we provide the task description in the system prompt, update the template with the problem, and provide it in the user prompt, rather than including the task description in the user prompt. We do this to follow as closely as possible https://platform.openai.com/docs/guides/prompt-engineering/OpenAI’s prompting guidelines. We also ask the model to generate its output in JSON format, to facilitate parsing its answers to interact with the symbolic solver.

End-to-end reasoning with LLMs In order to directly generate the answer to a reasoning problem, We include in the user prompt [Lst. 6] both the possible choices and some examples of how to reason.

Sketching When converting NL problems into FOL problems, we specify in the task description [7] some rules to follow when generating FOL formulas. This is because our implementation of Prover9 [] is only able to parse a subset of the full First-Order-Logic grammar. We provide the examples in the user prompt [Lst. 8] already in JSON format, to guide the generation in our desired format.

```
root ::= S

S ::= F | QUANT VAR (S1 | F) | "¬" (S1 | F)
S1 ::= F | QUANT VAR (S2 | F) | "¬" (S2 | F)
S2 ::= F | QUANT VAR (S3 | F) | "¬" (S3 | F)
S3 ::= F | QUANT VAR F | "¬" F

F ::= "¬"? "(" F1 ")" | F1 OP F1 | L
F1 ::= "¬" "(" F2 ")" | "(" F2 ")" | F2 OP F2 | L
F2 ::= "¬" "(" F3 ")" | "(" F3 ")" | F3 OP F3 | L
F3 ::= L

OP ::= "⊕" | "∨" | "∧" | "→" | "↔"
L ::= "¬"? PRED "(" TERMS ")"
TERMS ::= TERM | TERM "," TERMS
TERM ::=  VAR
QUANT ::= "∀" | "∃"
VAR ::= "x" | "y" | "z"
PRED ::= <@\textcolor{blue}{[[PREDICATES]]}@>
```

Listing 3: Context-Free Grammar used to generate valid FOL formulas

```
root ::= S

S ::= F | QUANT VAR (S | F) | "¬" (S | F)

F ::= "¬"? "(" F ")" | F OP F | L

OP ::= "⊕" | "∨" | "∧" | "→" | "↔"
L ::= "¬"? PRED "(" TERMS ")"
TERMS ::= TERM | TERM "," TERMS
TERM ::=  VAR
QUANT ::= "∀" | "∃"
VAR ::= "x" | "y" | "z"
PRED ::= <@\textcolor{blue}{[[PREDICATES]]}@>
```

Listing 4: Recursive variant of the context-free grammar used to generate valid FOL formulas

```
Given a problem statement as contexts, the task is to answer a logical reasoning
question. The response SHALL ALWAYS be a valid JSON, with the following format:
{
"reasoning": [your reasoning],
"answer": [the answer to the logical question]
}
```

Listing 5: Sysytem prompt for generating answers to the problems with CoT

```
Context:
The Blake McFall Company Building is a commercial warehouse listed on the National
Register of Historic Places.
The Blake McFall Company Building was added to the National Register of Historic
Places in 1990.
The Emmet Building is a five-story building in Portland, Oregon.
The Emmet Building was built in 1915.
The Emmet Building is another name for the Blake McFall Company Building.
John works at the Emmet Building.

Question: The Blake McFall Company Building is located in Portland, Oregon.

Options:
A) True
B) False
C) Uncertain
###
{"reasoning": "The Blake McFall Company Building is another name for the
Emmet Building. The Emmet Building is located in Portland, Oregon. Therefore, the
Blake McFall Company Building is located in Portland, Oregon.",
"answer": "A"}
------
Context:
<@\textcolor{blue}{[[CONTEXT]]}@>

Question: <@\textcolor{blue}{[[QUESTION]]}@>

Options:
A) True
B) False
C) Uncertain
###
```

Listing 6: User prompt template for generating answers to the problems with CoT

```
Given a Natural Language Problem, a Natural Language Question and a set of
First-Order-Logic Predicates.
The task is to convert the Problem and Question into First-Order-Logic Rules,
following the provided examples.

For conversion:
1. You SHOULD ONLY USE the predicates provided.
2. You SHOULD USE the following logical operators: ⊕ (either or), ∨ (disjunction),
∧ (conjunction), → (implication), ∀ (universal), ∃ (existential), ¬ (negation),
↔ (equivalence)
3. You SHOULD NEVER USE the following symbols for FOL: "%", "≠"
4. The literals in FOL SHOULD ALWAYS have predicate and entities, e.g.,
"Rounded(x, y)" or "City(guilin)";
5. Expressions such as "y = a ∨ y = b" or "a ∧ b ∧ c" are NOT ALLOWED
6. The FOL rule SHOULD ACCURATELY reflect the meaning of the corresponding NL
statement
7. You SHOULD NEVER have an entity with the same name as a predicate, e.g.,
Chair(x) and Tall(chair) is NOT ALLOWED
8. You SHOULD ALWAYS generate both the FOL rules and the FOL question

Generation Format: you SHOULD ALWAYS generate the response as a valid JSON,
in the following format
"""
{
    "First-Order-Logic Rules": [generated FOL Rules],
    "First-Order-Logic Question": [generated FOL Question]
}
"""
------
```

Listing 7: System Prompt for generating FOL problems

```
Natural Language Problem:
"""
<@\textcolor{blue}{[[EXAMPLE\_NL\_PROBLEM1]]}@>
"""
Natural Language Question:
"""
<@\textcolor{blue}{[[EXAMPLE\_NL\_QUESTION1]]}@>
"""
First-Order-Logic Predicates:
"""
<@\textcolor{blue}{[[EXAMPLE\_PREDICATES1]]}@>
"""
######
{
    "First-Order-Logic Rules": "<@\textcolor{blue}{[[EXAMPLE\_FOL\_RULES1]]}@>",
    "First-Order-Logic Question": "<@\textcolor{blue}{[[EXAMPLE\_FOL\_QUESTION1]]}@>"
}
------
Natural Language Problem:
"""
<@\textcolor{blue}{[[PROBLEM]]}@>
"""
Natural Language Question:
"""
<@\textcolor{blue}{[[QUESTION]]}@>
"""
First-Order-Logic Predicates:
"""
<@\textcolor{blue}{[[PREDICATES]]}@>
"""
######
```

Listing 8: User prompt template for generating a FOL problem with dynamically selected examples

```
Given a Natural Language Statement, a Sketch First Order Logic Statement and a set of
Predicates.
The task is to rewrite the Sketch using only the provided Predicates, and fix errors
if any, following the provided examples.

For fixing the formulas:
1. You SHOULD ONLY USE the predicates provided.
2. You SHOULD USE the following logical operators: ⊕ (either or), ∨ (disjunction),
∧ (conjunction), → (implication), ∀ (universal), ∃ (existential), ¬ (negation),
↔ (equivalence)
3. You SHOULD NEVER USE the following symbols for FOL: "%", "≠"
4. The literals in FOL SHOULD ALWAYS have predicate and entities, e.g.,
"Rounded(x, y)" or "City(guilin)";
expressions such as "y = a ∨ y = b" or "a ∧ b ∧ c" are NOT ALLOWED
5. The FOL rule SHOULD ACCURATELY reflect the meaning of the corresponding
NL statement
6. The FOL rule SHOULD BE AS CLOSE AS POSSIBLE to the provided sketch
7. You SHOULD NEVER have an entity with the same name as a predicate, e.g.,
Chair(x) and Tall(chair) is NOT ALLOWED
------
```

Listing 9: System prompt for correcting parsing errors

```
Natural Language Statement: Mia’s favorite season is not the same as Emma’s.

Sketch First Order Logic Statement: FavoriteSeason(mia, y) ∧ y ≠ FavoriteSeason(emma, y)

Predicates:
"""
Season(x) ::: x is a season
WantlongVacation(x) ::: x wants a long vacation
Love(x,y) ::: y is the favourite season of x
"""
######
Valid FOL Statement: FavoriteSeason(mia, y) ∧ ¬(FavoriteSeason(emma, y))"
------
Natural Language Statement: All restaurants with a rating greater than 9 are listed
in Yelp recommendations.

Sketch First Order Logic Statement: ∀x (Rating(x, y) ∧ y > 9 → Listed(x))

Predicates:
"""
Listed(x) ::: x is listed in Yelp recommendations
RatingGreaterThan9(x) ::: x has a rating greater than 9
NegativeReview(x) ::: x receives many negative reviews
PopularAmongLocalResidents(x) ::: x is popular among local residents
NoTakeOutService(x) ::: x has no takeout service
"""
######
Valid FOL Statement: ∀x (RatingGreaterThan9(x)→ Listed(x))
------
Natural Language Statement: When a person reads a book, that person gains knowledge.

Sketch First Order Logic Statement: ∀x ∀y (ReadBook(x, y) → Gain(Knowledge(x)))

Predicates:
"""
Book(x) ::: x is a book
Knowledge(x) ::: x contains knowledge
ReadBook(x, y) ::: x reads book y
GainKnowledge(x) ::: x gains knowledge
Smarter(x) ::: x becomes smarter
"""
######
Valid FOL Statement: ∀x ∀y (ReadBook(x, y) → GainKnowledge(x))
------
Natural Language Statement: <@\textcolor{blue}{[[NLSTATEMENT]]}@>

Sketch First Order Logic Statement: <@\textcolor{blue}{[[SKETCH]]}@>

Predicates:
"""
<@\textcolor{blue}{[[PREDICATES]]}@>
"""
######
Valid FOL Statement:
```

Listing 10: User prompt template for correcting parsing errors

```
Given a set of First-Order-Logic Rules, a First-Order-Logic Question and an
error message.
The task is to identify the reason for the error, the method to fix it, and
ultimatelly return the fixed formulas, following the provided examples.

For fixing the formulas:
1. You SHOULD ONLY USE the predicates provided.
2. You SHOULD USE the following logical operators: ⊕ (either or), ∨ (disjunction),
∧ (conjunction), → (implication), ∀ (universal), ∃ (existential), ¬ (negation),
↔ (equivalence)
3. You SHOULD NEVER USE the following symbols for FOL: "%", "≠"
4. The literals in FOL SHOULD ALWAYS have predicate and entities, e.g.,
"Rounded(x, y)" or "City(guilin)";
expressions such as "y = a ∨ y = b" or "a ∧ b ∧ c" are NOT ALLOWED
5. The FOL rule SHOULD ACCURATELY reflect the meaning of the corresponding
NL statement
6. You SHOULD NEVER have an entity with the same name as a predicate, e.g.,
Chair(x) and Tall(chair) is NOT ALLOWED

If you cannot identify the reason for the error, rewrite the original problem as the
correct one.

Generation Format: you SHOULD ALWAYS generate the response as a valid JSON, in the
following format:
"""
\{
"Error Reason": [the reason why the error is there],
"Solution": [how to fix the error],

"Correct Program":
    \{
    "First-Order-Logic Rules": [the correct FOL Rules],
    "First-Order-Logic Question": [the correct FOL Question]
    \}
\}
"""
------
```

Listing 11: System prompt for correcting execution errors

```
\{
"First-Order-Logic Rules": "ExtendedPlay(inside) ∧ LeadSingle(showyourlove) ∧ LeadSingleOf
    (showyourlove, inside) ::: The lead single of the extended play Inside
is Show Your Love.",

"First-Order-Logic Question": "HasMusicVideo(showyourlove) ::: There is a music video
for Show Your Love."
\}
Error Message:
"Fatal error:  The following symbols are used with multiple arities: LeadSingle/2,
LeadSingle/1"
######
\{
"Error Reason": "In one of the logic Rules, the predicate LeadSingle appears before
LeadSingleOf. Since a predicate cannot be a subset of another, this leads to an
error.",

"Solution": "Change LeadSingleOf to EPLeadSingle wherever it appears, so that it’s
not a subset of any other predicate.",
"Correct Program":
    \{
    "First-Order-Logic Rules": "ExtendedPlay(inside) ∧ LeadSingle(showyourlove) ∧
    EPLeadSingle(showyourlove, inside) ::: The lead single of the extended play Inside
    is Show Your Love.",

    "First-Order-Logic Question": "HasMusicVideo(showyourlove) ::: There is a music
    video for Show Your Love."
    \}
\}
------
<@\textcolor{blue}{[[PROGRAM]]}@>
Error Message:
<@\textcolor{blue}{[[ERROR MESSAGE]]}@>
######
```

Listing 12: User prompt template for correcting execution errors

```
  "### Predicates:
- JapaneseCompany(x) ::: x is a Japanese game company.
- Created(x, y) ::: x created y.
- Game(x) ::: x is a game.
- Top10(x) ::: x is in the Top 10 list.
- SoldMoreThanMillion(x) ::: x sold more than one million copies.

### Premises:
1. ∃x (JapaneseCompany(x) ∧ Created(x, legendOfZelda))
2. ∀x (Top10(x) → ∃y (JapaneseCompany(y) ∧ Created(y, x)))
3. ∀x (SoldMoreThanMillion(x) → Top10(x))
4. SoldMoreThanMillion(legendOfZelda

### Conclusion:
¬Top10(legendOfZelda)

<@\textcolor{red}{\#\#\# Analysis:}@>
<@\textcolor{red}{To determine if the conclusion is true, false, or uncertain, we need to
    analyze the}@>
<@\textcolor{red}{premises:}@>

<@\textcolor{red}{1. From premise 4, we know that the Legend of Zelda sold more than one
    million copies}@>
<@\textcolor{red}{2. From premise 3, we know that if a game sells more than one million
    copies, it will}@>
<@\textcolor{red}{be selected into the Top 10 list.}@>
<@\textcolor{red}{3. Therefore, from premises 3 and 4, we can infer that the Legend of
    Zelda is in the}@>
<@\textcolor{red}{Top 10 list (Top10(legendOfZelda)).}@>

<@\textcolor{red}{Given this inference, the statement \"The Legend of Zelda is not in the
    Top 10}@>
<@\textcolor{red}{list\" (¬Top10(legendOfZelda)) contradicts the derived conclusion that
    the Legend of}@>
<@\textcolor{red}{Zelda is in the Top 10 list.}@>

<@\textcolor{red}{\#\#\# Conclusion:}@>
<@\textcolor{red}{The statement \"The Legend of Zelda is not in the Top 10 list\" is **
    false**.""\#\#\# }@>
```

Listing 13: GPT-4o Response when using LogicLM. All parts in red were not requested in the prompt, and were generated spontaneously by the model