Solving Natural Language logical questions with LLM and Z3 solver

cantayxaolong

1 FOL Conversion and Solving

The `solving_fol_single_question` function converts NL premises and questions into FOL formulas using `nl_to_fol`. Premises are optionally reordered (`permute_fol`) to optimize inference, and nested brackets in conclusions are corrected (`fix_nested_fol_brackets`) if enabled. The `solve_fol` function evaluates conclusions against premises, returning an answer ("true", "false", or "No concluse"), premise indices, and a proof string (truncated to 500 characters).

For multiple-choice questions, `extract_choices` retrieves options, and each is evaluated to identify true, false, or inconclusive outcomes. Single questions standardize answers (e.g., "true" to "Yes").

- **FOL Conversion and Solving** The `solving_fol_single_question` function converts NL premises and questions into FOL formulas using `nl_to_fol`, powered by Qwen2.5-Coder-7B. The LLM leverages prompt engineering and few-shot learning to generate precise FOL representations. The `solve_fol` function evaluates conclusions against premises, returning an answer ("true", "false", or "No concluse"), premise indices, and a proof string (truncated to 500 characters).

  For multiple-choice questions, `extract_choices` retrieves options, and each is evaluated to classify outcomes. Single questions standardize answers (e.g., "true" to "Yes").
- **LLM-Based Premise Reordering** The `permute_fol` function uses Qwen2.5-Coder-7B to reorder FOL premises for optimal logical inference. Given NL premises and their initial FOL forms, the LLM analyzes logical dependencies and suggests an order that minimizes inference complexity. Prompt engineering guides the LLM to prioritize premises with foundational predicates, while few-shot examples demonstrate effective reordering patterns. This step is optional and skipped if timeouts occur.
- **LLM-Based Bracket Fixing** The `fix_nested_fol_brackets` function employs Qwen2.5-Coder-7B to correct nested bracket errors in FOL conclusions. The LLM parses FOL formulas, identifies unbalanced or misplaced brackets, and generates syntactically correct versions. Few-shot learning provides examples of valid FOL bracket structures, and prompts ensure adherence to logical syntax rules. This step is optional and controlled by the `fixBracket` parameter.
- **LLM-Based Fallback** When FOL solving fails (e.g., due to parsing errors or timeouts), `solve_fol_problem_fullLM` employs an LLM with prompt engineering and few-shot learning. Structured prompts guide the LLM to parse NL inputs or generate answers directly, while few-shot examples enhance reasoning accuracy. The LLM returns answers, indices, and explanations.
- **Asynchronous Handling with FastAPI** The system uses a main FastAPI server to receive requests and distribute tasks to multiple sub-servers via asynchronous `asyncio` calls. Each sub-server processes a subset of premises or questions, enabling parallel execution. The main server aggregates results, ensuring scalability for large datasets. The `step_change_client` function manages server coordination, balancing load across sub-servers.
- **Timeout Handling** Timeouts are enforced using a `start_time` parameter and `is_timeout` checks. If a task exceeds the time limit, `solving_fol_single_question` returns a `TIMEOUT_RETURN` value, triggering the LLM fallback. Asynchronous tasks are monitored to prevent server overload, with timeouts ensuring no single task monopolizes resources.

2 Implementation Details

The system incorporates:

- **Prompt Engineering**: Carefully designed prompts improve LLM performance in NL parsing and reasoning.

  Prompt for Converting Natural Language to First-Order Logic (FOL)

  You are a world-class expert in Formal Logic and AI prompt engineering. Your task is to convert natural language premises into consistent First-Order Logic (FOL) formulas. Maintain full consistency between all premises. Use standard FOL symbols: $\wedge$ for 'and', $\vee$ for 'or', $\rightarrow$ for 'implies', $\neg$ for 'not', $\forall x$ (e.g., `ForAll(x, ...)`). Keep the variables meaningful (e.g., use 'c' for curriculum, 'f' for faculty, etc.). Use standardized predicate names, e.g., `well_structured(c)`, `enhances_engagement(c)`, `can_enroll_organic_chemistry(student)`. For universal rules, use $\forall$, `ForAll(x, ...)` (e.g., keyword 'everyone', keyword 'If', etc.). For facts, state directly without $\forall$. The question's predicate names must be in premises.

  Prompt for Correcting Parentheses in First-Order Logic (FOL) Statements

  You are an expert in First-Order Logic (FOL) syntax. Given a list of FOL statements with potentially incorrect or missing parentheses, your task is to fix only the parentheses to make each statement logically correct, without changing any meaning or structure other than fixing brackets. Always ensure that operators (like $\wedge$, $\vee$, $\rightarrow$) are properly enclosed. Then add parentheses where needed to correctly group logical operators ($\wedge$, $\vee$) and comparison operators ($=$, $\geq$, $\leq$, $>$, $<$).
- **Few-Shot Learning**: Example-driven context enhances LLM accuracy for FOL conversion and direct solving.

Few-shot for fixing order of NL and FOL premises fonttitle

Example Input:
"NL-premises": ["Alex has completed safety orientation.", "Alex has a membership duration of 8 months.", "Alex has paid annual fees on time.", "If a person has a valid membership card and has completed safety orientation, they can use equipment.", "If a person can use equipment and has a trainer, they can book training.", "If a person's membership duration is at least 6 months, they are eligible for a trainer.", "If a person has paid the annual fee, they have a valid membership."], "FOL-premises": ["membership_duration(Alex) = 8", "safety_orientation(Alex)", "ForAll(x, (valid_membership(x) $\wedge$ safety_orientation(x)) $\rightarrow$ use_equipment(x))", "paid_annual_fee(Alex)", "ForAll(x, paid_annual_fee(x) $\rightarrow$ valid_membership(x))", "ForAll(x, (use_equipment(x) $\wedge$ has_trainer(x)) $\rightarrow$ book_training(x))", "ForAll(x, (membership_duration(x) $\geq$ 6) $\rightarrow$ eligible_trainer(x))",]
Example Output:
[(0,1), (1,0), (2,3), (3,2), (4,5), (5,6), (6,4)]

- **Asynchronous Processing**: FastAPI sub-servers handle tasks concurrently, with `asyncio` managing request distribution.
- **Timeout Handling**: Robust timeout checks prevent resource exhaustion, with fallback to LLM solving on failure.
- **Error Handling**: Exceptions trigger LLM-based solving, ensuring system reliability.

3 Converting FOL string into Polish notation

3.1 Expression Structure and Operator Hierarchy

Logical expressions consist of atomic units (variables, constants, and function terms) combined using logical operators. Common operators include:

- Negation ($\neg$)
- Conjunction ($\wedge$)
- Disjunction ($\vee$)
- Implication ($\rightarrow$)
- Bi-implication ($\leftrightarrow$)

These operators follow a precedence hierarchy, typically:

$$\neg > \wedge > \vee > \rightarrow = \leftrightarrow$$

To accurately convert expressions, this hierarchy must be respected, ensuring that higher-precedence operations are applied before lower-precedence ones, unless overridden by parentheses.

3.2 Recursive Parsing and Polish Notation

The transformation process begins by decomposing the expression using recursive descent parsing. This method evaluates the structure of the expression by:

1. Recognizing subexpressions enclosed in parentheses,
2. Recursively processing each subexpression,
3. Applying the correct precedence rules at each level,
4. Converting subtrees of the expression into prefix form.

Each operation becomes the root of a subtree in a conceptual parse tree, and operands become its children. In prefix notation, the operator appears before its operands, following the order dictated by the parse tree.

For example, the expression:

$$A \wedge (B \vee \neg C)$$

is converted into:

$$\wedge \ A \ \vee \ B \ \neg \ C$$

3.3 Handling Variables, Constants, and Functions

In practical logical formulas, elements can be:

- **Variables:** Symbols like A, x, or y representing logical atoms or domain elements.
- **Constants:** Logical constants such as `True` or `False`.
- **Function Symbols:** Symbols like $P(x)$ or $f(x, y)$ representing propositional or first-order predicates.

Accurate transformation requires distinguishing between these categories, particularly function applications, which may appear syntactically similar to parentheses used for grouping. This is typically achieved through lexical pattern recognition and context-sensitive parsing strategies.

3.4 Regular Expressions and Lexical Analysis

To tokenize the expression—splitting it into operators, variables, constants, and delimiters—regular expressions are employed. The use of regular expressions facilitates:

- Efficient identification of valid symbols and operators,
- Detection of nested or grouped expressions,
- Preprocessing to remove whitespace and standardize formatting.

Tokenization serves as the first step before recursive evaluation, enabling syntactic clarity and error detection.

4 Integration with Z3 Solver

Once the logical expression is transformed into Polish Notation, it becomes significantly easier to map it to data structures compatible with the Z3 solver. Z3 expects expressions to be constructed programmatically using function calls that mirror logical operations (e.g., `And`, `Or`, `Implies`).

Each operator in the prefix expression corresponds to a constructor function in Z3's API. By evaluating the prefix structure recursively, one can instantiate Z3 expressions in a consistent and type-safe manner. Variables and function symbols must also be declared appropriately in the Z3 environment.