System Description: DeepLLM, Casting Dialog Threads into Logic Programs

Paul Tarau[0000−0001−7192−9421]

University of North Texas
paul.tarau@unt.edu

Abstract. We automate deep step-by step reasoning in an LLM dialog thread by recursively exploring alternatives (OR-nodes) and expanding details (AND-nodes) up to a given depth. Starting from a single succinct task-specific initiator we steer the automated dialog thread to stay focussed on the task by synthesizing a prompt that summarizes the depth-first steps taken so far.
Our algorithm is derived from a simple recursive descent implementation of a Horn Clause interpreter, except that we accommodate our logic engine to fit the natural language reasoning patterns LLMs have been trained on. Semantic similarity to ground-truth facts or oracle advice from another LLM instance is used to restrict the search space and validate the traces of justification steps returned as focussed and trustable answers. At the end, the unique minimal model of a generated Horn Clause program collects the results of the reasoning process.
As applications, we sketch implementations of consequence predictions, causal explanations, recommendation systems and topic-focussed exploration of scientific literature.
Keywords: *automation of LLM dialog threads, recursive task-focused steering of LLM interactions, logic-programming driven LLM reasoning, LLM-based algorithmic information retrieval, context-driven LLM prompt synthesis* .

1 Introduction

Interaction with today's high-end LLMs like ChatGPT, GPT-4 [3,19], Claude-2 [1] and Bard [9] allows the patient and prompt-savvy user to steer the interaction toward fulfillment of a well-specified information seeking goal. The resulting dialog thread can be labor intensive and assumes solid prompt engineering skills to keep the LLM focussed on the task while digging as deep as needed into details.

This raises the obvious question: can we get back the simplicity of a one-shot query and automatically manage the navigation in the answer-space of the dialog thread?

We start by planning out the key steps of our proposed solution. Clearly, we need first an elaboration or refinement process that reduces a given task to a sequence of subtasks. We call this conjunctive elaboration into subtasks an *AND-step*. Next, we will need a dual, disjunctive elaboration, as a generation of alternative ways to make progress on the task. We call this an *OR-step*. To advance into more detail we can rely on a recursive process that alternates these two steps up to the desired depth.

This brings us to the key topic of this paper: an algorithm that extracts a salient set of answers, by zooming into the desired level of detail, from a single, succinct human *prompt. To this end, we automate step-by step reasoning in an LLM dialog thread to explore recursively alternatives (OR-steps) and expand details (AND-steps) up to a given depth.*

Our approach will follow closely the SLD-resolution algorithm for pure Horn Clause logic [14,13]. Restriction to Horn Clauses is motivated by the fact that LLMs are genuinely "constructive" and known not to be comfortable with negation [15,11], limiting one's interest in either classical negation or negation-as-failure under a closed-world assumption as present in ASP systems [8,21] or in Prolog [27].

This makes the use of a conventional logic programming language unnecessary as Python's coroutining generators are expressive enough for succinctly implementing a simplified SLD-resolution algorithm [23]. Another departure from logic programming as we know it, is that we will need to "unformalize" the underlying logic to more easily interoperate with the LLMs. In fact, LLMs do have a limited ability to generate correct logic forms of simple sentences [28]. But their training is based mostly on completion of natural language sentences and they are more in their element with the reasoning steps humans express in natural language.

Thus, instead of trying to force LLMs to use logic formalisms they are not yet comfortable with, we accommodate our logic engine to fit natural language reasoning, goal driven planning, task decomposition and association patterns with minimal task-specific prompt engineering.

This brings us to the key features of our approach:

- SLD-resolution's clause selection via unification is replaced by LLM-driven dynamic clause head creation with an option of focusing by proximity of embeddings to ground truth facts
- as dialog units are sentences, the underlying logic is propositional
- client-side management (via the API) of the LLM's memory is based on the equivalent of a *goal stack*, as used in logic-programming implementations like [22,23], and a *goal trace* recording our steps on the current search path
- instead of variable bindings, answers are traces of justification steps clearly explaining where they are derived from
- when their depth-limit is reached, the items on the goal stack are interpreted as "abducibles", statements that can be hypothetically assumed and then checked against "integrity constraints" [12,6].
- our depth-bounded refinement steps support compilation of the dialog threads to a Horn Clause program to be explored with logic programming solvers
- modular, task specific, customizable prompt engineering primitives are aggregated together for "AND-step" and "OR-step" prompts
- normalized semantic similarity measures of embeddings can be made available when generating probabilistic logic programs
- sentences in authoritative documents or collections seen as "ground-truth facts" can be used to select abducibles via semantic similarity or advice of an LLM-based oracle

Overall, our approach exploits synergies between structured prompt engineering, logic-guided recursion over LLM queries and semantic search in an embeddings vector store.

Applications are built by customizing prompts, LLM models and recursion level, resulting in automatically generated detailed, “hallucination-free” answers, crisper and more accurate than what one can obtain after lengthy interactions with conventional search engines or Chat-GPT style dialog agents.

Among potential applications we will overview the following:

– consequence predictions and causal explanations with full justification traces
– recommendation closely focussed on an initial preference seed
– actionable step-by-step advice on practical “how to repair” problems
– topic-focussed scientific literature keyphrase and key concept generation

The rest of the paper is organized as follows: Section 2 sketches the architecture of our implemented system. Section 3 introduces Interactors – designed by aggregating components needed for interacting with LLM APIs. Section 4 describes Recursors – our programs steering the LLMs dialog threads while focusing on the task at hand over multiple levels of nested OR-steps and AND-steps. Section 5 describes Refiners, specializations of Recursors checking against ground-truth facts using semantic distances to abducible facts as well as several LLM-based oracle agents. Section 6 describes our propositional Horn Clause model generator that extracts the set of true facts inferred from the logic program generated by our Recursors and Refiners. Section 7 shows how task-specific applications are built simply by adjusting the AND-step, OR-step and oracle prompts. Section 8 discusses variations on the main theme of the paper and possible future extensions. Section 9 discusses related work and section 10 concludes the paper.

Note : The system has been fully implemented[1] and it is deployed online[2].

2 System Architecture

We start with a quick overview of an implemented system architecture and a sketch of its execution flows, as shown in Figure 1.

Starting from a succinct prompt (typically a nominal phrase describing the task) an Interactor will call the LLM via its API, driven by a Recursor that analyzes the LLM responses and activates new LLM queries as it proceeds to refine the information received up to a given depth. Refiners are Recursor subclasses that rely on semantic search in an Embeddings Store containing ground-truth facts as well as on oracles implemented as specialized Interactors that ask the LLM for advice on deciding the truth of, or the rating of hypotheses. Besides returning a stream of answers, Recursors and Refiners compile their reasoning steps to a propositional Horn Clause program available for inspection by the user or subject for execution and analysis with logic programming tools (in particular, with our Model builder – a fast Propositional Horn Clause theorem prover).

[1] code at `https://github.com/ptarau/recursors`
[2] demo at `https://deepllm.streamlit.app`

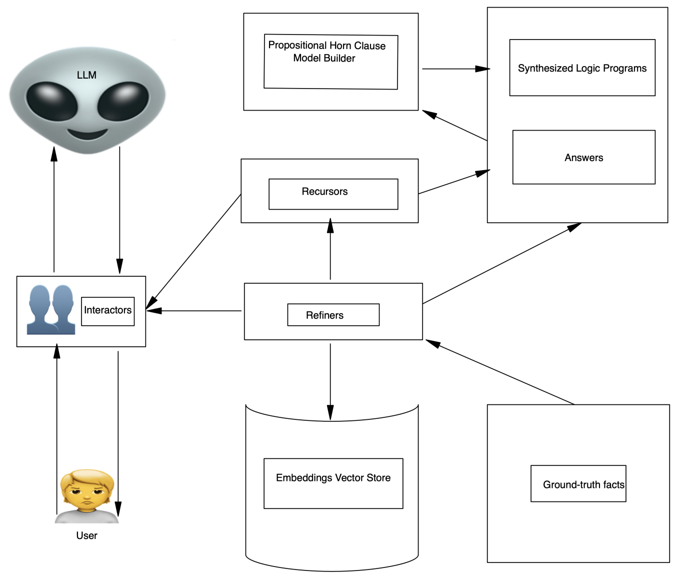


Fig. 1: System Architecture

3 Interactors

Setting up the interaction mechanism with LLMs via an API is a multi-faceted process involving several orthogonal aspects.

We will overview here our interactor Agent class managing the dialog with LLMs centered around OpenAI's `gpt-4` and `gpt-3.5-turbo` models [3,19], but able also to accommodate smaller footprint, locally running LLMs models like LLaMA 2 [25], that provide an OpenAI compatible API.

An interactor is put together by designing Prompter, Tuner, Tracker and Talker components.

Tuner The Tuner is a wrapper around the LLM's API, managing the API parameters and the settings of the interaction.

Prompters Prompters are patterns expressed as Python dictionaries from which substitution of `$`-variables with data provided at various recursion levels will generate actual prompts to be sent to the LLMs.

Here is an overview of our Prompters' key features and use cases:

- a Prompter is a dictionary of prompt templates for aggregated, task specific OR and AND prompt generators or decision oracles
- on top of them we build a collection of task specific parametric prompt templates

– AND and OR prompt templates for a given task are designed together to facilitate their experimental fine-tuning
– prompt templates instantiate one-shot instructions to the LLM that enforce focussed, succinct answers
– possible post-processors (algorithmic or implemented as “verifier” LLM oracles) can be used to discard answers when the LLM disobeys the instructions either in requested syntactic form or in content.

We will show next a few prompt template examples. Note as the LLMs (in this case GPT-3.5 and GPT-4) and their APIs evolve, minor edits might be needed to adjust them to the changes.

Example 1 *AND-OR prompt patterns for causal reasoning*

```
causal_prompter = dict(
    name='causal',
    and_p="""We need causal explanations in this context: "$context"
       Generate 3-5 explanations of 2-4 words each for the causes of "$g".
       Itemize your answer, one reason for "$g" per line.
       No explanations needed, just the 2-4 words noun phrase,
       nothing else.
       Your answer should not contain ":" or "Cause".
        """,
    or_p="""We need causal explanations in this context: "$context"
       Generate 2-3 alternative explanations citing facts that might
       cause "$g".
       Itemize your answer, one noun phrase per line.
       No explanations needed, just the noun phrase, nothing else.
       Avoid starting your sentence with the word "Alternative".
       Your answer should not contain ":" .
       Your answer should avoid the words "Causes" and "causes" ."""
)
```

Note that the $context *and current goal* $g *parameters will specialize the pattern for each of the uses of its* and_p *and* or_p *components in the recursive descent process. Note also the “petty” avoidance remarks in the prompt that we had to use to ensure that the answer returned by the LLM matches the expected output structure, given that after parsing, it has to provide the inputs of the next recursive step.*

Example 2 *Oracle pattern used to filter hypotheses generated by a Recursor*

```
decision_prompter = dict(
    name='oracle',
    decider_p="""
    You play the role of an oracle that decides if "$g" is relevant for
    our interest in "$context".
    Your answer should be "True" or "False" expressing agreement or
    disagreement with the relevance of "$g".
    """
)
```

The pattern is used to decide about adequacy of a given subtask or alternative in a given context. With a similar rater oracle we request ratings on a scale of 100 if we want to generate a probabilistic logic program to be analyzed with tools like Problog [5,18].

Tracker The Tracker is managing API messages, contexts and API costs. It ensures that answers to questions already answered by the LLM are cached and reused to save costs and ensure full determinism and replicability. It also handles the on-demand migrations from an Interactor's short-term memory to its long-term memory. While the short-term memory is kept small enough to fit in the LLMs message size, both memories are dictionaries used to retrieve available cached answers. As a special case, Trackers also enable spilling of the full content of the short-term memory to the long-term memory when a fresh dialog thread is needed for a change of topic or focus.

Talker The Talker is a component managing the overall interaction with the LLM. It implements the Interactor's high level `ask` method that encapsulates the details of applying the appropriate prompt template to a given question, activates mechanisms to trim the context to a size acceptable to the LLM, activates conversion of the content of the short-term memory to the message format the LLM expects. It also activates possible application-specific post-processing of the LLM's answers and manages the retrying of the completion request if the API is temporarily unresponsive.

4 Recursors

Recursors implement the central idea of this paper: automatically focusing a dialog thread with an LLM, while exploring a given topic in depth.

4.1 Synthesizing the Logic Program on recursive descent

Starting from a succinct initiator goal, the Recursor performs a recursive descent guided by task specific Prompters. At each step, the OR-prompter asks the LLM to generate alternative ways to fulfill the current goal. Then, for each alternative, the LLM is asked to expand it to a sequence of task specific subgoals, guided by the AND-prompter. The AND and the OR prompter templates are activated with information about the current context and the current goal. The context is a linearization of a chronologically ordered trace that accumulates the previously expanded goals. The presence of this context, serving as the *short term dialog memory* of the otherwise stateless LLM API, steers the generative process to stay focussed on the task.

At a given step, the effect of the OR-prompter expanding the head h in a series of alternatives $a_1, \ldots, a_n$ can be described as a set of binary Horn Clauses of the form:
$h\ :-\ a_1.$
$h\ :-\ a_2.$
$\ldots$
$h\ :-\ a_m.$
more concisely expressed with a disjunctive body as:

$h\ :-\ a_1;\ a_2\ ;\ \ldots\ ;\ a_m.$
On the other other hand, the result of an AND-prompter can be described as a set of Horn clauses of the form:

$h\ :-\ b_1,\ b_2,\ \ldots,\ b_n.$

When a depth limit is reached, the remaining unexplored goals on the goal stack are considered as *abducibles* [12,6], i.e., hypotheses to be assumed until integrity constraints might invalidate them, a process that, like in Logic Programming, results in backtracking to explore other possibilities. Should some of them fail, the presence of the OR-nodes at each recursive step ensures that plenty of choices remain available, despite possible failures.

A simple way to select abducibles is to check the semantic distance of their embeddings to embeddings in a set of *ground-truth facts*. For efficiency, a few nearest neighbors of each abducible are fetched from the vector embeddings store (see subsection 5.1) and their average distance to the ground truth is used to decide if the abducible is assumed as a hypothesis. A summary of the sentences extracted from the ground-truth facts can be used as an explanation supporting the abducible. This can be seen as an instant constraints-driven filtering operation that results in eagerly omitting the assumption of the irrelevant abducibles as hypotheses.

Besides returning a stream of answers, we also generate a propositional Horn Clause program to be further explored with logic programming tools.

At the end, a minimal model [13] of the remaining rules can be obtained with a SAT solver, although our implementation prefers a fast direct algorithm (see section 6), given that Horn Clause satisfiability is polynomial [7].

4.2 The Unfolder

Our implementation of the depth limited recursive descent encapsulates the unfolding of AND-nodes and OR-nodes. An Unfolder instance contains two Interactor Agents, one for each node type, initialized with their jointly designed prompter dictionary described in section 3. The agents are activated with the `ask_and` and `ask_or` methods and are also responsible for persisting past LLM interactions in appropriately named unique disk caches. By alternating the creation of AND-nodes and OR-nodes we will reshape the resulting dialog thread as a propositional Horn Clause program.

4.3 The AndOrExplorer

The process of building the propositional Horn Clause program is encapsulated in the AndOrExplorer class, that handles:

- the invention of clause heads by an OR-node induced by a given goal
- the invention of clause bodies by an AND-node induced by the clause head.

The AndOrExplorer implements its recursive descent by relying directly on the Python-stack and emulating Prolog's backtracking via Python's `yield`-based corouting mechanism. It returns the trace of expanded goals (and invented clause heads) for each successful “proof step”, assuming all facts at the depth limit as abducibles, subject to future validation by independent Oracle Agents.

The clause invention step is sketched by the following Python code snippet:

```python
def new_clause(self, g, trace, topgoal):
    or_context = to_context(trace, topgoal)
    hs = self.unf.ask_or(g, or_context)    # invent the clause heads
    and_context = to_context((g, trace), topgoal)
    for h in hs:
        bs = self.unf.ask_and(h, and_context)  # invent their bodies
        yield (h, bs)
```

The `or_context` is built from the generic OR-pattern instantiated to the specifics of the step in the `trace` of the goals expanded so far on this resolution branch. Note that `topgoal` is also passed to the context builder to help focus on the original goal that has started the recursive descent. It is responsible for the generation of the list of clause heads `hs`. For each clause head `h` in `hs`, a clause body `bs` is generated by the AND-node prompter. Finally each clause is yielded as a pair `(h, bs)`.

Besides its `resume`, `persist` and `costs` methods the `AndOrExplorer` defines also an `appraise` method meant to be overridden by its refiner subclasses.

4.4 The SVO Relation Extractor

The relation Extractor class invokes the LLM to decompose facts inferred from the Horn Clause program into <Subject, Verb, Object> triplets, usable for knowledge representation tasks. It also shows them in the `streamlit` Web app[3] using the `pyvis` graph visualizer.

4.5 The Logic Programming connection

The recursive descent algorithm is implemented as a generator of answers (traces of steps included) to the initiator goal, in a way similar to Prolog’s SLD-resolution algorithm operating on Horn Clauses. In fact, its Python implementation has been derived as a simplification of the Natlog [23] Horn Clause interpreter, where clause selection via the unification algorithms is replaced by synthesis of a clause head by an OR-node. Then, given the clause head, the body of the clause is generated by the LLM as an AND-node expansion of the synthesized clause head.

Instead of the `true` or `fail` answer generated by a Prolog system running the propositional Horn Clause program, the complete trace of goals generated by the LLM and “solved” by our recursor is returned as an “explanation” of the “reasoning steps” taken in the process. In fact, the resulting Horn Clauses are also ”compiled” on the fly to a Prolog program that could be independently explored with a Prolog, Datalog or ASP system. However, given the presence of loops (as the LLM might come back in the recursive process to things it has already seen), we use instead of Prolog a low polynomial-time model builder that is insensitive to the presence of loops [7].

[3] `https://deepllm.streamlit.app/`

5 Refiners

Refiners are extensions of Recursors evaluating AND-nodes and OR-nodes against ground-truth facts in the embeddings store or via *decision* or *rating* LLM-oracles (see subsection 5.3).

In the first case, normalized semantic distances between embeddings of a goal hitting a depth limit and ground-truth facts are used. If close enough to a ground-truth fact, the "abducible" goal becomes a clause head and the ground-truth fact becomes the body of a newly generated clause. If not close enough to any ground-truth fact, the goal becomes the head of a clause marked for failure when compiled to the Prolog program by having as body the atom `fail`.

Alternatively, abducibles can be evaluated by an oracle – another LLM instance that judges their relevance to the task in the current context, for instance against embeddings of ground truth statements stored in a vector store.

5.1 The Embeddings Vector Store

We build a simple `numpy`-based vector store supporting efficiently ground-truth collections of fact embeddings obtained via the LLM's API. The ground-truth facts store is then used to find the K nearest neighbors of a given query and also to return cosine distances, usable as probabilities to decide what hypotheses can be assumed during the recursive descent as "abducible" facts, subject to filtering via evaluation of integrity constraints. Adopting scalable vector stores or databases can support the use of a knowledge base possibly derived from very large document collections.

5.2 Filtering with semantic distance to ground-truth facts

By using the ground-truth facts in our embeddings store the simplest way to appraise if a given goal is "on topic" is to compute its semantic distance to its nearest neighbor in the store, as shown in the following code snippet:

```
    def appraise(self, g, _trace, _topgoal):
        rates, neighbors = self.store.qa(g, top_k=1)
        rate, neighbor = float(rates[0]), neighbors[0]
        return rate > self.threshold:
```

The method `qa` that queries the store passes the goal `g` and the request for a single nearest neighbor `top_k`.

A more elaborate technique relies on k nearest neighbors fetched from the store that would collectively "champion" the goal if their (weighted) average semantic distance to the goal is below a threshold, fixed in advance or dynamically computed or machine-learned form past appraisals.

The mechanism can also be extended to continuously check for staying close in terms of semantic distance to the ground-truth facts.

Alternatively, the semantic distances (interpreted as probabilities) can be used to annotate clause heads as part of a probabilistic logic program to be evaluated later.

Oracles can also be used to implement *continual appraising*: at each step they can check reasonable closeness to ground truth and task at hand. In particular, they can mark confidence level for each rated step and then select overall highest only.

5.3 Refining decisions with LLM-based oracles

In the absence of a set of ground-truth facts relevant to a given initiator goal, the LLM itself can be asked to make True/False decisions or generate ratings.

The LLM-based True/False Decider The following code snippet delegates the steering to focus on a given context (in this case the initiator goal that has started the recursive descent). In this case, the `appraise` method instantiates the oracle prompt pattern shown in subsection 3.

```
def appraise(self, g, trace, topgoal):
    advice = just_ask(self.oracle, g=g, context=topgoal)
    return 'True.' == advice
```

More elaborate refiners can use the depth-first path `trace` or an LLM-generated summary thereof instead of `topgoal`.

The LLM-based Rater The Rater queries the LLM asking for a score on the 0 to 100 scale that is next converted into a probability. Like the Decider oracle, it uses the goal at hand and its context. For both oracles, the prompter can be configured to ask for an explanation sentence or paragraph to be adopted as ground-truth in case of favorable True/False decision or high enough confidence level.

5.4 Toward Trustable Generative AI

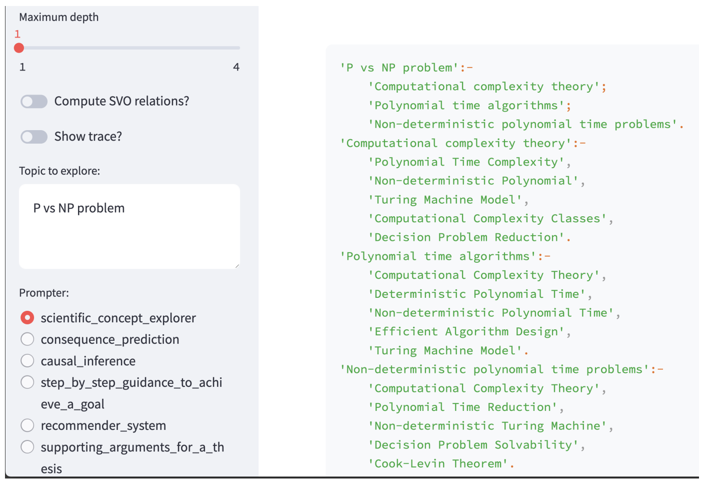
Both oracle types, in concert with the focussed reasoning steps enforced by casting AND/OR steps into a propositional Horn clause program provide a principled approach toward trustable generative AI, an often expressed requirement for wide adoption of today's LLMs in medical, educational, defense and several other business applications as well as likely subjects of upcoming government regulations [10].

6 The Model builder: a Propositional Horn Clause Satisfiability solver

It is not unusual to have loops in the propositional Horn Clause program connecting the LLM generated items by our recursors and refiners. As that would create problems with Prolog's depth-first execution model, we implement a simple low-polynomial complexity propositional satisfiability checker and model builder along the lines of [7].

The model builder works by propagating truth from facts to rules until a fix point is reached. Given a Horn Clause $h:-b_1,b_2,...,b_n$, when all b_i are known to be true (i.e., in the model), h is also added to the model. If integrity constraints (Horn clauses of the form $false : -b_1, b_2, ..., b_n$) have also been generated by the oracle agents monitoring our refiners, in the advent that all $b_1, b_2, ..., b_n$ end up in the model, $b_1, b_2, ..., b_n$ implying $false$ signals a contradiction and thus unsatisfiability of the Horn formula associated to the generated program. However as the items generated by our recursive process are not necessarily expressing logically connected facts (e.g., they might be just semantic similarity driven associations), turning on or off this draconian discarding of the model is left as an option for the application developer. Also, the application developer can chose to stop as soon as a proof of the original goal emerges, in a way similar to goal-driven ASP-solvers like [2], irrespectively to unrelated contradictions elsewhere in the program.

7 Applications


Maximum depth
1
1 4

Compute SVO relations?

Show trace?

Topic to explore:

P vs NP problem

Prompter:

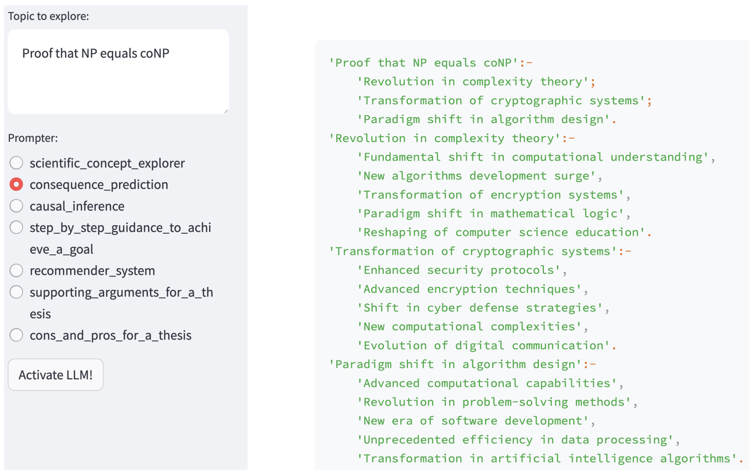
[✓ scientific_concept_explorer]
consequence_prediction
causal_inference
step_by_step_guidance_to_achieve_a_goal
recommender_system
supporting_arguments_for_a_thesis

```
'P vs NP problem':-
    'Computational complexity theory';
    'Polynomial time algorithms';
    'Non-deterministic polynomial time problems'.
'Computational complexity theory':-
    'Polynomial Time Complexity',
    'Non-deterministic Polynomial',
    'Turing Machine Model',
    'Computational Complexity Classes',
    'Decision Problem Reduction'.
'Polynomial time algorithms':-
    'Computational Complexity Theory',
    'Deterministic Polynomial Time',
    'Non-deterministic Polynomial Time',
    'Efficient Algorithm Design',
    'Turing Machine Model'.
'Non-deterministic polynomial time problems':-
    'Computational Complexity Theory',
    'Polynomial Time Reduction',
    'Non-deterministic Turing Machine',
    'Decision Problem Solvability',
    'Cook-Levin Theorem'.
```

Fig. 2: Exploring the P vs. NP problem

A good hint on deciding which recursor or refiner is the most appropriate for developing an application, is the closeness of its atomic steps to processes of human problem solving that are similar to logic inferences, e.g., by sharing a similar underlying boolean algebra, lattice or preordered set structure. Besides causal reasoning or consequence prediction most goal-oriented tasks (e.g., planning) fit this structure. It is also good to be aware that when exploring the causes or the consequences of an initial state of the world, technological development, military, political or judicial decision, it is likely that the LLM will generate a richer model than if it explores names of movies, books or songs in a recommender system, where titles are often overlapping semantically with unrelated embeddings. In the former, restricting the model with a stricter oracle can even out spurious facts. In the latter, being aware that the LLMs will work better on very well known movies or books than when asking for recommendations similar to a relatively new or niche product, can guide the scope of the application.

When developing an application that, starting from a keyphrase of a scientific paper or the name of a scientific domain (e.g., as shown in Fig. 2 with the system exploring the P vs. NP problem), an oracle set up to filter out less specific concepts from encompassing more general domain can help with the return of more salient results. In this case a second oracle, filtering out generic methodological boilerplate concepts, shared by virtually all scientific domains will be also useful to give more focussed results. Similar refinements can also be used when focusing on predicting consequences of a (likely counterfactual) result (e.g., as shown in Fig. 3).


Topic to explore:

Proof that NP equals coNP

Prompter:

- [ ] scientific_concept_explorer
- [✓] consequence_prediction
- [ ] causal_inference
- [ ] step_by_step_guidance_to_achieve_a_goal
- [ ] recommender_system
- [ ] supporting_arguments_for_a_thesis
- [ ] cons_and_pros_for_a_thesis

```
'Proof that NP equals coNP':-
    'Revolution in complexity theory';
    'Transformation of cryptographic systems';
    'Paradigm shift in algorithm design'.
'Revolution in complexity theory':-
    'Fundamental shift in computational understanding',
    'New algorithms development surge',
    'Transformation of encryption systems',
    'Paradigm shift in mathematical logic',
    'Reshaping of computer science education'.
'Transformation of cryptographic systems':-
    'Enhanced security protocols',
    'Advanced encryption techniques',
    'Shift in cyber defense strategies',
    'New computational complexities',
    'Evolution of digital communication'.
'Paradigm shift in algorithm design':-
    'Advanced computational capabilities',
    'Revolution in problem-solving methods',
    'New era of software development',
    'Unprecedented efficiency in data processing',
    'Transformation in artificial intelligence algorithms'.
```

Fig. 3: Exploring consequences of the NP=coNP hypothesis

In the case of requesting expert advice on practical common tasks (e.g., actionable step-by-step advice on how to repair something), the oracle can filter out advice to contact the manufacturer or seek the advice of an expert nearby, when the point is to receive the actual steps need to solve the problem. This can also be implemented by a set of negative ground facts for which a refiner can try to maximize average semantic distances or a post-processor that rejects choices containing words or keyphrases in a blacklist.

Another kind of application of significant practical value is to use a set of generated models to benchmark newcomer LLMs' performance against established best in their class like, at the time of writing this paper, OpenAI's GPT-4. This can be achieved with something as simple as the Jaccard distance between the inferred models at a given depth and it can be fine-tuned to the specific task the LLMs are planned to be used for. Related to this, when transferring from strong LLMs like GPT-4 to weaker ones, it can be useful to train the Reinforcement Learning loop rewards to be based on how many hits the weaker LLM gets when recursing on a relatively small collection of “critically important topics” automatically collected from the stronger LLM, thus providing a novel, potentially very effective transfer learning mechanism.

The generated Prolog programs, models and execution traces are available online at: `https://github.com/ptarau/recursors/tree/main/examples`.

8 Discussion on Limitations and Variations on the Theme

8.1 Limitations

We will next overview some of the limitations we have experienced when testing (the current implementation of) our recursors and refiners when on several target applications.

First, let us note that recursors are obviously not needed when one-shot detailed descriptions of common processes (e.g., cooking recipes, tell a joke, write a haiku) are available directly from the LLM. They are also unnecessary for help from the LLM to write a news story, a bio, an essay, a resume or an add, where interactive fine-tuning of the LLM’s in-context learning by the user is a clear requirement.

The mapping between the recursively generated items and propositional logic does not apply equally to all tasks and the inference steps work differently when the LLM is used simply as an associative-memory connecting concepts interesting as brainstorming incentives to humans, but not meant to be logically focussed on a dominant topic or task.

Indicators like semantic similarity are not relevant for recommendation systems over items consisting of titles of movies, books or songs where their distributional semantics is dominated by the more common uses of the title’s actual word phrases.

Strictly enforcing integrity constraints generated by oracles looking at local contexts often results, when propagated through the inference process, in unsatisfiability (and thus an empty model). Note however that this limitation can be alleviated by accepting a partial model supporting the initial goal, even if the resulting logic program might be inconsistent, an option supported by our model generator described in section 6.

There’s increased sensitivity to prompts during deeper recursive thought-to-thought steps. In this case, careful prompt engineering is needed as recursors can easily induce “butterfly effects” - small variations in wording of the prompt can drastically change the resulting model.

8.2 Future Work Directions

Once the key idea of the paper for steering LLMs to generate a stream of items focussed on the initiator goal is implemented, several “variations on the theme” can be tried out relatively easily by overriding methods in Prompters, Recursors and Refiners.

Granularity Refiners At a higher granularity one can work with sentences/statements instead of noun phrases, sometime a more natural match to the underlying propositional logic.

At a lower granularity, one might want to use SVO triplets that LLMs are quite good at decomposing a sentence into. The generated SVO triplets can then serve as building blocks for Description Logics or Datalog programs.

Question generators Answers to LLM-generated how+wh-questions for a given goal can be used as expansions to new goals simply by re-engineering our AND-OR Prompters.

Generalizers Inductive Logic Programming techniques can be used to generalize the resulting propositional or triplet clauses by sharing common SVO fragments, possibly in combination with Prolog-rules describing the ground-truth background knowledge.

Diversifiers and Harmonizers To further restrict unwanted "hallucinatory" generation twists *diversifiers* for OR-nodes and *harmonizers* for AND-nodes can be expressed as additional integrity constraints. A diversifier will work by ensuring no two OR-nodes are too close semantically while a harmonizer will ensure that no two AND-nodes are too far semantically. Both could be implemented with help of semantic distances in the embeddings vector store.

Extend Implementation Techniques One can implement recursors by relying on bare completion-only LLMs (e.g., GPT3) as their usual question-answering fine-tunings (e.g., ChatGPT) can be emulated with minor prompt engineering efforts.

To limit the scope of decider and rater oracles, one can preprocess the ground-truth facts into k-means clusters, and restrict oracle search to the cluster closest to the initiator goal.

Extend the power of the underlying logic language It is possible to use SAT-based ASP solvers [21] or Prolog-based CASP systems [2] to take advantage of failed LLM returns rejected by our oracles, to enhance the ability of the LLMs to reason with negative information in a principled way as well as with negative ground-truth facts meant to avoid extending into semantically close but distinct domains during the recursive descent.

9 Related Work

The major disruption brought by the often near-human quality of generative AI [3,19,25] is quickly changing the landscape of query-driven information retrieval, moving the emphasis from traditional search engines to human-friendly dialog threads. However, the effectiveness of actionable information extraction is often hindered by the slower partner in this interaction – the human that needs to understand, evaluate and validate each step. In this context, our work emphasizes the full automation of this retrieval process, starting from a succinct query term. Thus, we are back to the one-shot simplicity of "short question by human $\rightarrow$ arbitrarily deep, elaborated answer from the AI", steered to stay focussed on the actual query. As a side effect of this automation, our approach preempts most of the usual problems with hallucinations, lack of factuality and bias that LLMs are often blamed for.

Our recursive descent algorithm shares with work on "Chain of Thought" prompting of LLMs [26,17] and with step by step [16] refinement of the dialog threads the goal of extracting more accurate information from the interaction. However our process aims to fully automate the dialog thread while also ensuring validation of the results with help of ground-truth watching oracles and independent LLM-based agents. Our approach shares with tools like LangChain [4] the idea of piping together multiple instances of LLMs, computational units, prompt templates and custom agents, except that we fully automate the process without the need to manually stitch together the components.

By contrast to "neuro-symbolic" AI [20], where the neural architecture is closely intermixed with symbolic steps, in our approach the neural processing is encapsulated in the LLMs and accessed via a declarative, high-level API. This reduces the semantic gap between the neural and symbolic sides as their communication happens at a much higher, fully automated and directly explainable level.

In [28] LLMs are cleverly used to generate Prolog code snippets with an enhanced CASP [2] semantics. This allows hand-building of useful applications like a conversational AutoConcierge bot, recommending local restaurants. By contrast, our method is generic, and "no code" applications consist simply in queries with possible minor prompt engineering, as we adapt logic programming to think in terms familiar to LLMs, rather than adapting LLMs to generate application specific code snippets.

Also within Logic Programming, in relation with probabilistic approaches [5], our abducibles acquire probabilities from normalized vector distances to ground truths, usable to automate generation of probabilistic logic programs, thus sharing objectives with typical neuro-symbolic approaches like [18]. Finally, we share with [24] the idea to use a custom logic solver (an Intuitionistic Propositional Theorem prover, in that case) to synthesize abducibles (the Propositional Horn Clause model generator in our case).

10 Conclusion

We have automated deep step-by step reasoning in LLM dialog threads up to a given depth, by recursively descending from a single succinct initiator phrase, while staying focussed on the task at hand. In the process, we have made LLMs function as de facto logic programming engines that mimic Horn Clause resolution. However, instead of trying to parse sentences into logic formulas, we have accommodated our logic engine to fit the natural language reasoning patterns LLMs have been trained on.

Semantic similarity to ground-truth facts and oracle advice from another LLM instance has been used to restrict the search space and validate the traces of justification steps returned as answers. This has resulted in focussed, controllable output, enabling deep investigations into details of specific scientific domains as well as expert-level causal reasoning or consequence predictions. As such, our approach streamlines key use cases of LLMs as focussed information seeking tools and enables practical application back-ends simply by customizing prompt templates.

By casting the LLM dialog into a precise logic frame and by filtering the LLMs reasoning via semantic closeness to ground-truth facts, we have devised a method to extract hallucination-free focussed knowledge expressed as a logic program that encapsulates trustable AI in a clearly expressed and easily verifiable framework.

References

1. Anthropic: Claude-2 (2023), https://claude.ai/chats/, [Accessed 2023-10-02]
2. Arias, J., Carro, M., Salazar, E., Marple, K., Gupta, G.: Constraint answer set programming without grounding (2018)
3. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems. vol. 33, pp. 1877–1901. Curran Associates, Inc. (2020). https://doi.org/https://doi.org/10.48550/arXiv.2005.14165
4. Chase, H.: LangChain (Oct 2022), https://github.com/hwchase17/langchain
5. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In: IJCAI. vol. 7, pp. 2462–2467 (2007). https://doi.org/10.5555/1625275.1625673
6. Denecker, M., Kakas, A.: Abduction in logic programming. In: Computational Logic: Logic Programming and Beyond. pp. 402–36. Springer (2002)
7. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional horn formulae. J. Log. Program. **1**(3), 267–284 (1984), https://doi.org/10.1016/0743-1066(84)90014-1
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes). pp. 1070–1080. MIT Press (1988)
9. GoogleAI: Bard (2023), https://bard.google.com/, [Accessed 2023-06-23]
10. Hacker, P., Engel, A., Mauer, M.: Regulating ChatGPT and Other Large Generative AI Models. pp. 1112–1123. FAccT '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3593013.3594067
11. Hossain, M.M., Holman, L., Kakileti, A., Kao, T.I., Brito, N.R., Mathews, A.A., Blanco, E.: A Question-Answer Driven Approach to Reveal Affirmative Interpretations from Verbal Negations (2022)
12. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming. J. Log. Comput. **2**(6), 719–770 (1992), http://dblp.uni-trier.de/db/journals/logcom/logcom2.html#KakasKT92
13. Kowalski, R., Emden, M.V.: The Semantics of Predicate Logic as a Programming Language. JACM **23**(4), 733–743 (Oct 1976). https://doi.org/10.1145/321250.321253
14. Kowalski, R.A.: Predicate logic as programming language. In: Rosenfeld, J.L. (ed.) Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974. pp. 569–574. North-Holland (1974)
15. Levy, M.G.: Chatbots Don't Know What Stuff Isn't (may 2023), https://www.quantamagazine.org/ai-like-chatgpt-are-no-good-at-not-20230512/
16. Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., Cobbe, K.: Let's verify step by step (2023)
17. Ling, Z., Fang, Y., Li, X., Huang, Z., Lee, M., Memisevic, R., Su, H.: Deductive verification of chain-of-thought reasoning (2023)
18. Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., De Raedt, L.: Deepproblog: Neural probabilistic logic programming. Advances in Neural Information Processing Systems **31** (2018)

19. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., Lowe, R.: Training language models to follow instructions with human feedback (2022). https://doi.org/10.48550/ARXIV.2203.02155, https://arxiv.org/abs/2203.02155
20. Sarker, M.K., Zhou, L., Eberhart, A., Hitzler, P.: Neuro-symbolic artificial intelligence: Current trends (2021). https://doi.org/10.48550/ARXIV.2105.05330, https://arxiv.org/abs/2105.05330
21. Schaub, T., Woltran, S.: Special Issue on Answer Set Programming. KI **32**(2-3), 101–103 (2018). https://doi.org/10.1007/s13218-018-0554-8, https://doi.org/10.1007/s13218-018-0554-8
22. Tarau, P.: A Hitchhiker's Guide to Reinventing a Prolog Machine. In: Rocha, R., Son, T.C., Mears, C., Saeedloei, N. (eds.) Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017). OpenAccess Series in Informatics (OASIcs), vol. 58, pp. 10:1–10:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/OASIcs.ICLP.2017.10, http://drops.dagstuhl.de/opus/volltexte/2018/8453
23. Tarau, P.: Natlog: a Lightweight Logic Programming Language with a Neuro-symbolic Touch. In: Formisano, A., Liu, Y.A., Bogaerts, B., Brik, A., Dahl, V., Dodaro, C., Fodor, P., Pozzato, G.L., Vennekens, J., Zhou, N.F. (eds.) Proceedings 37th International Conference on Logic Programming (Technical Communications) , 20-27th September 2021 (2021). https://doi.org/10.4204/EPTCS.345.27
24. Tarau, P.: Abductive Reasoning in Intuitionistic Propositional Logic via Theorem Synthesis. Theory and Practice of Logic Programming **22**(5), 693–707 (2022). https://doi.org/10.1017/S1471068422000254
25. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Roziere, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., Lample, G.: Llama: Open and efficient foundation language models (2023)
26. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models (2023)
27. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**, 67–96 (1 2012). https://doi.org/10.1017/S1471068411000494
28. Zeng, Y., Rajasekharan, A., Padalkar, P., Basu, K., Arias, J., Gupta, G.: Automated interactive domain-specific conversational agents that understand human dialogs (2023)