On LLM-generated Logic Programs and their Inference Execution Methods

Paul Tarau[0000−0001−7192−9421]

University of North Texas

`paul.tarau@unt.edu`

Large Language Models (LLMs) trained on petabytes of data are highly compressed repositories of a significant proportion of the knowledge accumulated and distilled so far. In this paper we study techniques to elicit this knowledge in the form of several classes of logic programs, including propositional Horn clauses, Dual Horn clauses, relational triplets and Definite Clause Grammars. Exposing this knowledge as logic programs enables sound reasoning methods that can verify alignment of LLM outputs to their intended uses and extend their inference capabilities. We study new execution methods for the generated programs, including soft-unification of abducible facts against LLM-generated content stored in a vector database as well as GPU-based acceleration of minimal model computation that supports inference with large LLM-generated programs.

Keywords: LLM-generated logic programs; LLM-generated Definite Clause Grammars; LLM-generated relation graphs; soft-unification with abducible facts; GPU-supported evaluation of propositional Horn clause programs; visualization of LLM-generated relations.

1 Introduction

While the multi-step dialog model initiated by ChatGPT is now available from a few dozen online or locally run open source and closed source LLMs, it does not cover the need to efficiently extract salient information from an LLMs "parameter-memory" that encapsulates in a heavily compressed form the result of training the model on trillions of documents and multimodal data.

Steps in this direction have been taken, relying on ground-truth involving additional information sources (e.g., collections of reference documents or use of web search queries). Among them, we mention work on improving performance of Retrieval Augmented Generation (RAG) systems [7] by recursively embedding, clustering, and summarizing chunks of text for better hierarchical LLM-assisted summarization [15], multi-agent hybrid LLM and local computation aggregators [3] and deductive verifiers of chain of thoughts reasoning [9].

A more direct approach is recursion on LLM queries, by chaining the LLM's distilled output as input to a next step and casting its content and interrelations in the form of logic programs, to automate and focus this information extraction with minimal human input [18, 20]. Like in the case of typical RAG architectures [7, 15], this process can rely on external ground truth but it can also use new LLM client instances as "oracles" deciding the validity of the synthesized rules or facts.

With focus on automation of this unmediated salient knowledge extraction from the LLM's parameter memory and its encapsulation in the form of synthesized logic programming code, we will extend in this paper the work initiated in [18, 20] with:

- new LLM input-output chaining mechanisms

- new types of generated logic programs

- new relational representations elicited from LLM output steps

- scalable execution mechanisms that accommodate very large logic programs at deeper recursion levels

- soft-unification based execution of LLM-generated logic programs as a principled encapsulation of the RAG retrieval process

The rest of the paper is organized as follows. Section 2 overviews the DeepLLM architecture described in [20] and its new extensions supporting the results in this paper. Section 3 overviews the generation of Horn clause programs with the online DeepLLM app. Section 4 explains the LLM-based generation of Dual Horn clause programs and their uses to explore counterfactual consequences and theory falsification, Section 5 introduces the use of DCG grammars as a representation of LLM-generated answer and follow-up question pairs. Section 6 describes fixpoint and GPU-supported minimal model computation for the generated programs. Section 7 describes relation-extraction and visualization from the minimal models of the LLM-generated propositional programs. Section 8 introduces the soft-unification based encapsulation of the information retrieval against facts extracted from authoritative document collections. Section 9 discusses related work and Section 10 concludes the paper.

2   Recursive exploration of LLM dialog threads

Generative AI, with often human-like language skills is shifting focus from typical search engines to more conversational interactions. Yet, the challenge remains that humans must still process and verify this information, an often tedious task.

Our answer to this, as implemented in the DeepLLM system is to automate the entire process. We start with a simple "initiator goal" and let the LLM dive recursively in its parametric memory and deliver a detailed answer focused on the initiator and the trace of this chain of steps summarized as the short term-memory maintained via its API. This automation also helps to minimize common issues like inaccuracies, made-up information, and biases that are often associated with LLMs.

We refer to [20, 18] for details of implementation of the DeepLLM system, as well as to its open-source code[1] and its online demo[2].

The DeepLLM system's active components (subclasses of the Agent class) are Interactors, Recursors, and Refiners:

- Interactors manage input prompts and task breakdown

- Recursors handle iterative exploration of subtasks

- Refiners enhance clarity and relevance of LLM responses

To validate its reasoning steps, the system also relies on stored knowledge resources:

- Ground truth facts: sentences collected from online sources or local documents

- Vector store: enabler of "semantic search" via embeddings of sentences

Starting from a succinct prompt (typically a nominal phrase or a short sentence describing the task) an Interactor will call the LLM via its API, driven by a Recursor that analyzes the LLM's 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

---

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

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).

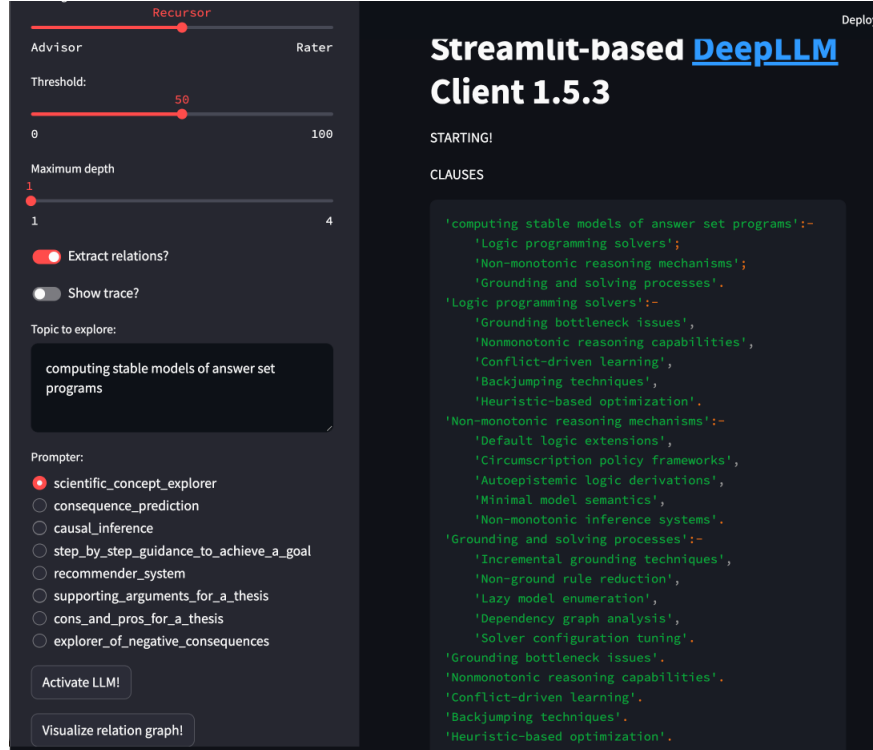# 3 Generating propositional Horn clause programs with the DeepLLM app

Figure 1: DeepLLM app

We refer to [18] for an extensive list of LLM-generated Horn clause programs. We will just briefly describe here the DeeLLM app (see **Fig. 1**) that we will use for generating our logic programs. In the case of the interaction shown in **Fig. 1**, the initiator goal "computing stable models of answer set programs" starts the "scientific concept explorer" option and generates in the right side window a Horn clause program describing successive refinements of the initiator goal.

The DeeLLM app is written with the `Streamlit`[3] webapp generator and offers the choice between GPT4, GPT3.5 or a local LLM, running as a server and supporting an OpenAI compatible API. It then lets the user choose between the Recursor, Advisor and Rater agents, providing for the latter a threshold level slider. The threshold informs the Rater oracle to accept or reject a generated rule head or fact (the higher the threshold the stricter the accept decision). Options to set the maximum recursion depth and activate relation extraction and visualization are also available.

---

[3]`https://streamlit.io/`

The application starts once the user enters the topic to explore, chooses the prompter template and activates the LLM. Besides the output produced in the right window, when run locally, it saves the generated logic program and its computed minimal model as Prolog code files.

4   Generating propositional Dual Horn clause programs

A *Dual Horn clause* is a disjunction of literals with at most one negative literal (or exactly one if it is a definite Dual Horn clause). A *Dual Horn clause Program* is a conjunction of Dual Horn clauses. We represent a Dual Horn clause like $\neg p_0 \vee p_1 \vee \ldots \vee p_n$ in an equivalent implicational form $p_0 \rightarrow p_1 \vee \ldots \vee p_n$, similarly to Prolog's representation of Horn clauses. We adopt a Prolog-like syntax, with $\rightarrow$ represented as "=>" and $\vee$ represented as ";". Note also that "s => false" represents a negated fact the same way as "s :- true" would represent a positively stated fact.

The objective of Dual Horn programs is to describe (constructively) why something is not true i.e., to falsify the initiator goal by back-propagating from its negative (or more generally, undesirable, unwanted, harmful, impossible, etc.,) consequences.

For instance, from a clause like p => q ; r ; s, assuming that p were true, we would infer that at least one of q , r and s should be true. The contrapositive is that if q , r and s are all false, then p should be false as well. Like in the case of SLD-resolution on Horn clauses, this triggers a goal oriented process where successful falsification of all consequences results in falsification of the "counterfactual" hypothesis that initiated the process.

By instructing the LLM to infer the negative consequences of the DeepLLM initiator goal, we can obtain a Dual Horn program.

Example 1 *The Dual Horn clauses (recursion level = 0) with heads (starting with consequences of* `'tailgate when driving'`*) in the clause body are:*

```
'tailgate when driving' =>
    'Increased accident risk';
    'Reduced reaction time'.
'Increased accident risk' =>
    'Higher insurance premiums';
    'Severe injury likelihood';
    'Vehicle damage costs';
    'Legal consequences';
    'Emotional trauma impact'.
```

```
'Reduced reaction time' =>
    'Increased accident risk';
    'Delayed braking response';
    'Higher collision likelihood';
    'Compromised driving safety'.
```

The negative facts (unexplored recursion level = 1 goals) are:

```
'Higher insurance premiums' => false.
'Severe injury likelihood' => false.
'Vehicle damage costs' => false.
'Legal consequences' => false.
'Emotional trauma impact '=> false.
```

```
'Delayed braking response' => false.
'Higher collision likelihood' => false.
'Compromised driving safety' => false.
```

Note that "=> false" marks things that we do not want to happen, from where the same backpropagates to the initiator 'tailgate when driving'. Our compilation algorithm[4] will transform this into a definite program placed in module false that can be queried with:

```
?- false:'tailgate when driving'.
true
```

Its successful falsification could then advise a car driving program or person to avoid the aforementioned behavior.

A more interesting exploration (at recursion level=2) of negative consequences in the form of a Dual Horn clause program[5] reveals persuasive counter-arguments to unwise political decisions.

Example 2 *A few unwanted consequences at descent level 2 for 'loosing the FED_s independence':*

```
'loosing the FED_s independence' =>
    'Increased political influence on monetary policy'. % level 0
...
'Increased political influence on monetary policy' =>
    'Politicized interest rates';
    'Short-term economic manipulation';
    'Eroded investor confidence'; % level 1
    'Heightened market volatility';
    'Policy-driven inflation risks'.
 ...
 'Eroded investor confidence' => % level 2
    'Market volatility';
    'Capital flight';
    'Reduced foreign investment'.
```

5 From Self-generated follow-up question-answer chains to DCG grammars

DeepQA[6] (see **Fig. 2**) is a DeepLLM-based application that explores recursively the "mind-stream" of an LLM via a tree of self-generated follow-up questions. Interestingly, by asking the LLM to generate a set of follow-up questions to its own answers creates (especially when the process recurses) a more focused "stream of thoughts", possibly as an emergent property of its "in-context learning" abilities.

After started from an initiator question on a topic of the user's choice, the app explores its tree of *follow-up questions* up to a given depth. As output, it generates a Definite Clause Grammar that can be imported as part of a Prolog program. The DCG, in generation mode, will replicate symbolically the equivalent of the "stream of thoughts" extracted from the LLM interaction, with possible uses of the encapsulated knowledge in Logic Programming applications.

---

[4] https://github.com/ptarau/TypesAndProofs/blob/master/symlp/compile_clauses.pro

[5] full code at https://github.com/ptarau/output_samples/tree/main/deepllm

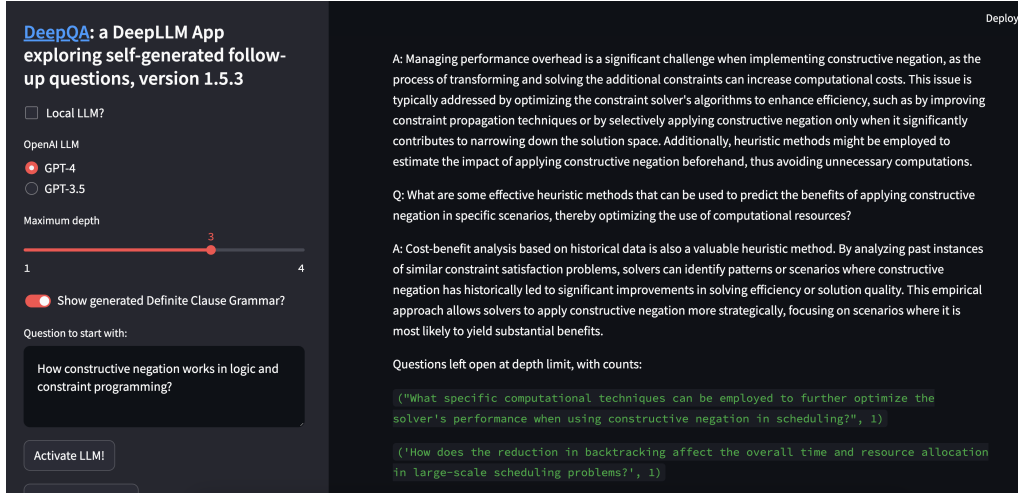[6] https://github.com/ptarau/recursors/tree/main/deepQA

Figure 2: DeepQA with "*How constructive negation works in logic and constraint programming?*"

The synthesized grammar is designed to generate a finite language (by carefully detecting follow-up questions that would induce loops). We also ensure that paths in the question-answer tree are free of repeated answers, which get collected as well, together with questions left open as a result of reaching the user-set depth limit.

Example 3 *Definite Clause Grammar generated by initiator question* 'How constructive negation works in logic and constraint programming?'*:*

```
% DCG GRAMMAR RULES:

q0-->q0_,a0_,q1.
q0-->q0_,a1_,q2.
q1-->q1_,a3_,q4.
...
q12-->q12_,a38_.
```

For instance, the first rule rewrites the initiator q0 *into:*

- *the terminal* q0_ *that will produce the actual text of the question*

- *the terminal* a0_ *that will produce the actual text of the answer to* q0_

- *the non-terminal* q1 *continuing the generation process with one of the follow-up questions generated by the LLM*

```
% QUESTION TERMINALS:

q0_-->['Q: How constructive negation works in logic and constraint programming?'].
q1_-->['Q: Can you provide an example of how constructive negation might refine
    the solution space in a practical constraint programming problem?'].
q2_-->['Q: How does constructive negation differ from classical negation in terms
    of computational efficiency and outcome in constraint satisfaction problems?'].
    ...
```

```
% ANSWER TERMINALS:

a0_-->['A: Constructive negation in logic and constraint programming is a method
    used to handle negation in a way that allows for the derivation of new
    constraints from negative information. Instead of simply rejecting solutions
    that do not satisfy a certain condition, constructive negation works by
    deducing what must be true if a given condition is false. This is particularly
    useful in constraint programming where constraints define what is possible
    rather than what is not. By applying constructive negation, the system can
    infer additional constraints that must be met for the negation to hold,
    effectively refining the solution space.'].
    ...
```

When reaching the user-specified recursion depth, the unanswered follow-up questions are collected as "open questions" in the predicate `opens/2` *with the second argument indicating the number of times (over all branches of the tree) the question has been generated. In the case of an LLM with a very large parametric memory (e.g., GPT4, Claude 3 or Gemini) values above 1 are unlikely, while with smaller LLMs (e.g., Vicuna) repeated follow-up questions can happen more often.*

```
% OPEN QUESTIONS:

opens('What specific computational techniques can be employed to further optimize
    the solver_s performance when using constructive negation in scheduling?',1).
opens('How does the reduction in backtracking affect the overall time and
            resource allocation in large-scale scheduling problems?',1).
...
```

Starting the DCG in generation mode from its `q0` *initiator goal is achieved as follows:*

```
% entry point to generate the language covered by the DCG grammar
go:-q0(Xs,[]),nl,member(X,Xs),write(X),nl,nl,fail.
```

One can also use DeepQA to quickly assess the strength of an LLM before committing to it. For instance, when used with a much weaker than GPT4 local LLM (enabled with Vicuna 7B by default) one will see shorter, more out of focus results, with a lot of repeated questions and answers collected by DeepQA in corresponding bins.

The full Prolog code discussed in thus example is available online[7] as well the DeepQA app[8].

6 Computing minimal models of LLM-generated logic programs

6.1 Minimal model computation with a GPU-friendly Torch-based Linear Algebra Algorithm

At deeper recursion levels, the generated logic programs, providing a symbolic representation of an LLM's parameter memory can quickly reach millions of clauses, ready to reason with.

To take advantage of the significant acceleration provided by GPUs we have implemented a `torch`-based linear algebraic minimal model computation algorithm[9] along the lines of [13].

---

[7]`https://github.com/ptarau/output_samples/tree/main/deepqa`
[8]`https://deep-auto-quests.streamlit.app/`
[9]`https://github.com/ptarau/recursors/blob/main/tenslogic/proptens.py`

The implementation is centered around

```python
def tp(M, v):
    """
    one step fixpoint operator
    """
    r = M @ v
    return (r >= 1.0).to(torch.float32)
```

that advances one step of the fixpoint computation with a matrix multiplication "@" and

```python
def tp_n(M, v0):
    """
    iterated fixpoint operator
    """
    oldv = v0
    n = M.shape[0]
    for i in range(n):
        newv = tp(M, oldv)
        if torch.allclose(newv, oldv):
            return newv
        oldv = newv
```

that proceeds until a fixpoint is detected using `torch.allclose`.

The program contains readers of Horn clause programs represented in as `.json` files. It can handle medium size programs (a few thousand clauses), as despite the GPU acceleration, complexity is still dominated by $O(N^3)$ matrix products.

We will show here a small test program running the minimal model computation. After defining:

```python
top = "true"
bot = "false"

vs = (p, q, r, s) = "pqrs"
```

We represent the program as pair made of the head of the clause and the list of atoms in its body:

```python
prog = [
(         p, [q]),
(         p, [r]),
(         q, [r, s]),
(         r, [top]),
(         bot, [q])
]
```

We can then compute the model with:

```python
>>> print(compute_model(prog))
['p', 'r']
```

Future work using torch sparse tensors[10], to ensure scalability for very large generated programs is planned along the lines of [11].

---

[10]`https://pytorch.org/docs/stable/sparse.html`

6.2 Fixpoint-based minimal model computation

It is not unusual to have loops in the propositional Horn Clause program connecting the LLM generated items by our recursors and refiners that would create problems with Prolog's depth-first execution model. As using a SAT-solver would be an overkill in this case, given that Horn Clause and Dual Horn clause formula satisfiability is known to be polynomial, we have implemented a simple low-polynomial complexity [6] propositional satisfiability checker and model builder[11].

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 [1], irrespectively to unrelated contradictions elsewhere in the program.

7 Generating relation triplets for knowledge graphs

Our DeepLLM app offers an option to generate from the minimal model of the program a relation graph (see **Fig. 3**) consisting of implication links (marked with ":") to which it adds generalization links (marked with "is").

Implication links are extracted directly from the logic program while generalization links, serving as additional explanations, are generated by the LLM via an additional request.

Several other types of relation graphs can be generated depending on the planned reasoning method.

One of them is extraction of <subject, verb, object> (SVO) triplets obtained by prompting the LLM to split a complex sentence in simpler ones and extract from each simple sentence an SVO triplet.

Another is a hybrid method, combining relations extracted by using dependency grammars [21], embeddings-based similarity relations, Wordnet-based and LLM-generated hypernyms and meronyms.

8 Reasoning with soft unification on noisy facts

The minimal models of LLM-generated Horn clause programs encapsulate facts and their consequences elicited from DeepLLM's initiator queries in the form of natural language sentences. When writing a logic program that performs symbolic reasoning relying on a ground fact database of such sentences, an interesting form of *abductive reasoning* emerges. When hitting an undefined ground sentence, intended as a query to match database facts we can rely on vector embedding of the sentences and proximity of the query and the facts in the vector space as a "good enough" match, provided that the semantic distance between them is below a given threshold. We will next describe a proof of concept of this strategy that we illustrate on a small quotation dataset consisting of a few sentences[12].

---

[11]https://github.com/ptarau/recursors/blob/main/deepllm/horn_prover.py
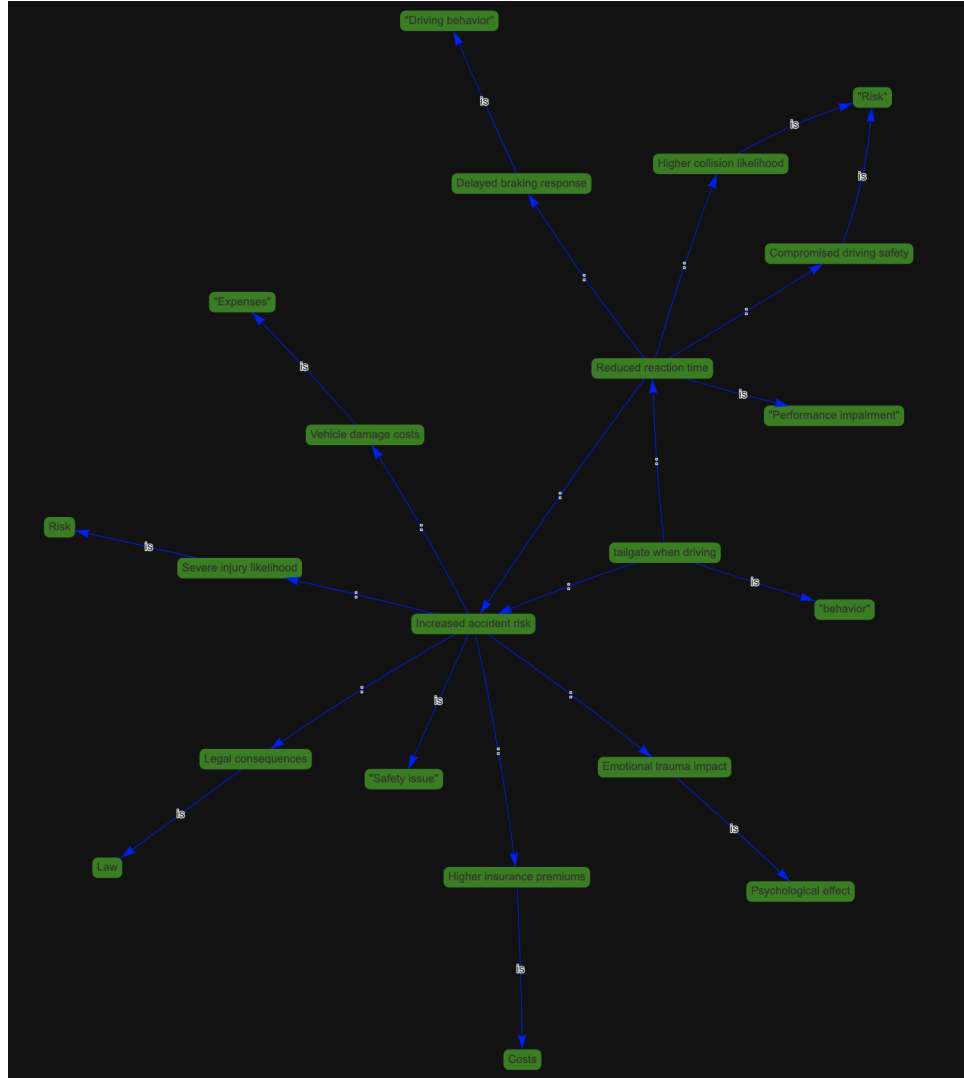[12]https://github.com/ptarau/natlog/blob/main/docs/quotes.txt

Figure 3: Relation graph for "*tailgate when driving*"

We have implemented `Softlog`[13] as an extension to the `Natlog` system [17, 19], a Python-based Prolog dialect with a simpler syntax and a lightweight Python interface. It works simply by overloading `Natlog`'s built-in ground fact database's unification method with a form of *soft-unification* [4, 2, 10], implemented as follows:

```python
def unify_with_fact(self, goal, trail):

    # q = query goal to be matched
    # k = number of knns to be returned
    # d = minimum knn distance
    # v = variable to be unified with the matches

    q, k, d, v = goal
    d = float(d) / 100
    _, answers = self.emb.knn_query(q, k)
    for sent, dist in answers:
        if dist <= d:
            self.abduced_clauses[(q, sent)] = dist
            yield unify(v, sent, trail)
```

The following `Natlog` script is then used to query a small set of sentences serving as `Softlog`'s ground database. Note that the "~" symbol is Natlog's convention for marking calls to a ground (soft-)database.

```
knn 3.
threshold 70.

quest  Quest Answer:
  knn K, % K is the number passed to the K Nearest Neighbors query
  threshold D,
  ~ Quest K D Answer.
```

We implement soft unification queries as K closest neighbors (KNN) computations against embeddings in our `sentence_store`[14]. We use Sentence Transformers [12] to compute embeddings and store them locally in an efficient and scalable vector database. As usual in `Natlog`, the Python iterator returning multiple KNN matches is mapped to Prolog's backtracking with multiple answers returned as alternative bindings to a result variable.

```
?- quest 'What happens if you do not know where you go' X?
ANSWER: {'X': 'If you don t know where you are going
             you will end up somewhere else said Yogi Berra.'}
ANSWER: {'X': 'If you don t know where you are going
             any road will get you there said Lewis Carroll.'}
```

When a query (`Q A`) binds `A` to an answer extracted from the vector store, a binary clause (`Q :- A`) and its supporting fact (`A :- true`) are inserted into the dictionary of *abduced clauses*. If we add them to the program that triggered the generation of the clauses, we obtain a self-contained standard logic program that returns exactly the same answers as its `Softlog` counterpart. Alternatively, the computed distances can be normalized as probabilities, to annotate clauses used in a Probabilistic Logic Programming language like Problog [5].

---

[13] https://github.com/ptarau/natlog/tree/main/softlog
[14] https://github.com/ptarau/sentence_store

```
ABDUCED CLAUSES:

'What happens if you do not know where you go' :
    'If you don t know where you are going you will end up somewhere else
    said Yogi Berra'. % distance=0.5874345302581787
'What happens if you do not know where you go' :
    'If you don t know where you are going any road will get you there
    said Lewis Carroll'. % distance=0.6724047660827637
```

Note that by contrast to the usual exact unification based answers, `Softlog` works quite well when the query is *close enough* to a matching entry in the sentence store, a reasonable assumption when the facts have been generated from multiple LLM runs and several ground truth resources.

```
?- quest 'What did Wilde say about temptation' X?
ANSWER: {'X': 'I can resist anything except temptation said Oscar Wilde.'}

?- quest 'What did Alice say about following advice' X?
ANSWER: {'X': 'I give myself very good advice but I very seldom follow it
               said Lewis Carroll.'}
```

Given the nature of semantic search, surname is enough to find Oscar Wilde and as `Alice` associates with the author Lewis Carroll, soft unification will fetch it from the sentence store.

9   Related Work

By contrast to "neuro-symbolic" AI [14], 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.

Our recursive descent algorithm shares the goal of extracting more accurate information from the LLM interaction with work on "Chain of Thought" prompting of LLMs [22, 9] and with step by step [8] refinement of the dialog threads. Our approach shares with tools like LangChain [3] 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.

We have not found any references to the use of Dual Horn clauses in logic programming but it is a well known result [16]) that their complexity in the propositional case is polynomial, similarly to their of Horn clause counterparts. This fact makes them also good generation targets for LLM-extracted knowledge processing.

We have not found anything similar to generating question-answer-follow-up question chains, although it is common practice for chatbots to suggest (a choice between) follow-up questions[15].

Our torch-based model-computation algorithm follows closely the matrix-computation logic of [13], our contribution being its succinct and efficient GPU-friendly implementation.

Interest in several forms of soft-unification has been active [4, 2, 10] as differentiable substitute of symbolic unification in neuro-symbolic systems. By contrast, our focus in this paper is flexible information retrieval of LLM-generated natural language content, for which high quality embeddings were available either from LLM APIs or local resources like the torch-based sentence-transformers [12].

---

[15]including the author's own `https://auto-quest.streamlit.app/`

10 Conclusion

It is now undeniable that Generative AI is a major disruptor not just of industrial fields ranging from search engines, automation of software development and robotics to medical and legal advisory systems, but also a disruptor of research fields, including symbolic AI as we know it and machine Learning itself. In particular, results produced by dominant ML or NLP techniques as well as work on integration of neural and symbolic systems have become replaceable by much simpler applications centered around LLM queries and RAG systems. In fact, by concentrating the knowledge encapsulated in its parametric memory into a single declarative interface, Generative AI can replace complex, labor-intensive software functionality with a simple LLM API call or a question in one's favorite natural language.

This motivates our effort to "join the disruption" and explore several new ways to elicit the knowledge encapsulated in the LLMs' parametric memory as logic programs, together with an investigation of their optimal inference execution methods. We have not just exposed as logic programs the several kinds of knowledge snippets extracted by recursive automation LLM dialog threads , but we have also devised efficient inference execution mechanisms for them.

We hope that this effort has revealed some natural synergies between Generative AI systems and logic programming tools, ready to fill gaps like the lack of rigorous reasoning abilities of the LLMs, their lack of alignment to the user's intents and their known deficiencies on factuality.

References

[1] Joaquin Arias, Manuel Carro, Elmer Salazar, Kyle Marple & Gopal Gupta (2018): *Constraint Answer Set Programming without Grounding*. Theory and Practice of Logic Programming 18(3-4), pp. 337–354, doi:10.1017/S1471068418000285.

[2] Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini & Michael Spranger (2022): *Logic Tensor Networks*. Artificial Intelligence 303, p. 103649, doi:10.1016/j.artint.2021.103649. Available at `https://www.sciencedirect.com/science/article/pii/S0004370221002009`.

[3] Harrison Chase (2022): *LangChain*. Available at `https://github.com/hwchase17/langchain`. Https://www.langchain.com/.

[4] Nuri Cingillioglu & Alessandra Russo (2020): *Learning Invariants through Soft Unification*, doi:10.48550/arXiv.1909.07328. arXiv:1909.07328.

[5] Luc De Raedt, Angelika Kimmig & Hannu Toivonen (2007): *ProbLog: A Probabilistic Prolog and Its Application in Link Discovery*. In: IJCAI, 7, pp. 2462–2467, doi:10.5555/1625275.1625673.

[6] William F. Dowling & Jean H. Gallier (1984): *Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae*. J. Log. Program. 1(3), pp. 267–284, doi:10.1016/0743-1066(84)90014-1. Available at `10.1016/0743-1066(84)90014-1`.

[7] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel et al. (2020): *Retrieval-augmented generation for knowledge-intensive nlp tasks*. Advances in Neural Information Processing Systems 33, pp. 9459–9474.

[8] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever & Karl Cobbe (2023): *Let's Verify Step by Step*, doi:10.48550/arXiv.2305.20050. arXiv:2305.20050.

[9] Zhan Ling, Yunhao Fang, Xuanlin Li, Zhiao Huang, Mingu Lee, Roland Memisevic & Hao Su (2023): *Deductive Verification of Chain-of-Thought Reasoning*, doi:10.48550/arXiv.2306.03872v3. arXiv:2306.03872.

[10] Jaron Maene & Luc De Raedt (2023): *Soft-Unification in Deep Probabilistic Logic*.

[11] Tuan Quoc Nguyen, Katsumi Inoue & Chiaki Sakama (2022): *Enhancing linear algebraic computation of logic programs using sparse representation*. New Generation Computing 40(1), pp. 225–254, doi:10.1007/s00354-021-00142-2.

[12] Nils Reimers & Iryna Gurevych (2019): *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. In Kentaro Inui, Jing Jiang, Vincent Ng & Xiaojun Wan, editors: *EMNLP/IJCNLP (1)*, Association for Computational Linguistics, pp. 3980–3990, doi:10.18653/v1/D19-1410. Available at `http://dblp.uni-trier.de/db/conf/emnlp/emnlp2019-1.html#ReimersG19`.

[13] Chiaki Sakama, Katsumi Inoue & Taisuke Sato (2017): *Linear Algebraic Characterization of Logic Programs*. In Gang Li, Yong Ge, Zili Zhang, Zhi Jin & Michael Blumenstein, editors: *Knowledge Science, Engineering and Management*, Springer International Publishing, Cham, pp. 520–533, doi:10.1007/978-3-319-63558-3_44.

[14] Md Kamruzzaman Sarker, Lu Zhou, Aaron Eberhart & Pascal Hitzler (2021): *Neuro-Symbolic Artificial Intelligence: Current Trends*, doi:10.48550/ARXIV.2105.05330. Available at `https://arxiv.org/abs/2105.05330`.

[15] Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie & Christopher D. Manning (2024): *RAPTOR: Recursive Abstractive Processing for Tree-Organized Retrieval*, doi:10.48550/arXiv.2401.18059. arXiv:2401.18059.

[16] Thomas J. Schaefer (1978): *The complexity of satisfiability problems*. In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, Association for Computing Machinery, New York, NY, USA, p. 216?226, doi:10.1145/800133.804350. Available at `https://doi.org/10.1145/800133.804350`.

[17] Paul Tarau (2021): *Natlog: a Lightweight Logic Programming Language with a Neuro-symbolic Touch*. In Andrea Formisano, Yanhong Annie Liu, Bart Bogaerts, Alex Brik, Veronica Dahl, Carmine Dodaro, Paul Fodor, Gian Luca Pozzato, Joost Vennekens & Neng-Fa Zhou, editors: Proceedings 37th International Conference on *Logic Programming (Technical Communications)* , *20-27th September 2021*, doi:10.4204/EPTCS.345.27.

[18] Paul Tarau (2023): *Full Automation of Goal-driven LLM Dialog Threads with And-Or Recursors and Refiner Oracles*:arXiv:2306.14077. doi:10.48550/arXiv.2306.14077. arXiv:2306.14077.

[19] Paul Tarau (2023): *Natlog: Embedding Logic Programming into the Python Deep-Learning Ecosystem*. In Enrico Pontelli, Stefania Costantini, Carmine Dodaro, Sarah Gaggl, Roberta Calegari, Artur D'Avila Garcez, Francesco Fabiano, Alessandra Mileo, Alessandra Russo & Francesca Toni, editors: Proceedings 39th International Conference on *Logic Programming*, Imperial College London, UK, 9th July 2023 - 15th July 2023, *Electronic Proceedings in Theoretical Computer Science* 385, Open Publishing Association, pp. 141–154, doi:10.4204/EPTCS.385.15.

[20] Paul Tarau (2024): *System Description: DeepLLM, Casting Dialog Threads into Logic Programs*. In Jeremy Gibbons & Dale Miller, editors: *Functional and Logic Programming*, Springer Nature Singapore, Singapore, pp. 117–134, doi:10.1007/978-981-97-2300-3_7.

[21] Paul Tarau & Eduardo Blanco (2021): *Interactive Text Graph Mining with a Prolog-Based Dialog Engine*. Theory Pract. Log. Program. 21(2), pp. 244–263, doi:10.1017/S1471068420000137.

[22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le & Denny Zhou (2022): *Chain of Thought Prompting Elicits Reasoning in Large Language Models*. CoRR abs/2201.11903, doi:10.48550/arXiv.2201.11903. arXiv:2201.11903.