Reflections on Automation, Learnability and Expressiveness in Logic-based Programming Languages

Paul Tarau

University of North Texas
paul.tarau@unt.edu

Abstract. This position paper sketches an analysis of the essential features that logic-based programming languages will need to embrace to compete in a quickly evolving field where learnability and expressiveness of language constructs, seen as aspects of a learner's user experience, have become dominant decision factors for choosing a programming language or paradigm.

Our analysis centers on the main driving force in the evolution of programming languages: automation of coding tasks, a recurring promise of declarative languages, instrumental for developing software artifacts competitively.

In this context we will focus on taking advantage of the close correspondence between logic-based language constructs and their natural language equivalents, the adoption of language constructs enhancing the expressiveness and learnability of logic-based programming languages and their synergistic uses in interacting declaratively with deep learning frameworks.

Keywords: logic-based programming language constructs, automation, expressiveness and learnability, coroutining with logic engines, definite clause grammars as prompt generators, embedding of logic programming and in deep learning ecosystems.

1 Introduction

Driven by the importance of automation and simplification of coding tasks in a logic programming context, the question we plan to answer is:

> What features need to be improved or invented to ensure a lasting ability of logic-based programming languages to compete with languages that have adopted the latest innovations in usability, robustness and easy adoption by newcomers?

Our short answer is that *we need to focus on closeness to natural language, learnability, flexible execution mechanisms and highly expressive language constructs*.

We will elaborate in the next sections on why these features matter, with hints on what language constructs are needed for implementing them.

As an application, we will show the effectiveness of some of our proposed language constructs via definite clause-grammar based prompt generators for today's text-to-text and text-to-image deep learning systems.

2 The Challenges

2.1 It is just Automation (again)

Automation, seen as a replacement of repetitive tasks, has been a persistent theme from which essential computational processes including compilation, partial-evaluation and meta-interpretation have emerged.

Besides competition from today's functional programming languages and proof assistants, all with strong declarative claims, logic-based languages face even stiffer competition from the more radical approach to automation coming from deep learning.

To state it simply, this manifests as replacement of rule-based, symbolic encoding of intelligent behaviors via machine learning, including unsupervised learning among which transformers [9] trained on large language models have achieved outstanding performance in fields ranging from natural language processing to computational chemistry and image processing. For instance, results in natural language processing with prompt-driven generative models like GPT3 [1] or text-to-image generators like DALL.E [5] or Stable Diffusion [10] have outclassed similarly directed symbolic efforts. In fact, it is quite hard to claim that a conventional programming language (including a logic-based one) is as declarative as entering a short prompt sentence describing a picture and getting it back in a few seconds.

We will argue in the next sections that it makes sense for logic-based programming languages to embrace rather than fight these emerging trends.

2.2 The Shifting of the Declarative Umbrella

Logic-based programming languages have shared with their functional counterparts and a few data-base management tools the claim of being "declarative" in the very general sense that the code seen as a specification of *what* needs to be done has clear enough information for the implementation to be able to "automatically" figure out *how* it can be done.

However it is becoming clearer every day that ownership of the declarative umbrella is slowly transitioning to deep neural networks-based machine learning tools that, to state it simply, replace human coding with models directly extracted from labeled and more and more often from raw, unlabeled data. This suggests the usefulness of closely embedding a logic-based language in this fast evolving ecosystem.

2.3 The Importance of Learnability

Learnability is not a crisply definable concept, but it becomes intuitively clear especially as someone gets fluent in several programming languages and paradigms.

Learnability is experienced positively or negatively when teaching or learning a new programming language and also when adopting it as an ideal software development stage for a given project. Good barometers for learnability are the learning curves of newcomers (including children in their early teens), the hurdles they experience and the projects they can achieve in a relatively short period of time. Another one is how well one can pick up the language inductively, simply by looking at coding examples.

When thinking about what background can be assumed in the case of newcomers, irrespectively of age, natural language pops up as a common denominator.

As logic notation originates in natural language there are conspicuous mappings between verbs and predicates and nominal groups as their arguments. Spatial and temporal event streams, in particular visual programming, animations and games relate to logic in more intricate ways and at a more advanced level of mastering a logic-based language.

That hints toward learning methods and language constructs easily mapped syntactically and semantically to natural language equivalents.

2.4 The importance of Expressiveness

As part of their evolution, programming languages learn from each other. Expressiveness enhancements are contagious. More precisely, language constructs that encapsulate formally clear data objects and their typical usage patterns propagate, often crossing heavily defended programming paradigm border walls.

As Python has been an early adopter of such expressiveness enhancers, it makes sense to consider for adoption some of its language features that one is likely to be impressed even at a first contact, as some of the following:

- ease of defining finite functions (dictionaries, mutable and immutable sequences and sets), all exposed as first class citizens
- aggregation operations (list, set, dictionary comprehensions) exposed with a lightweight and flexible syntax
- coroutining (via the yield statement or async annotations) exposed with a simple and natural syntax
- nested parenthesizing and heavy punctuation avoided or reduced via indentation

Prolog shares some of those but it is usually via separate libraries or semantically more intricate definitions (e.g., setof with care about how variables are quantified as an implementation of set comprehensions). We will explore in the next sections some language constructs covering features where logic-based languages are left behind.

3 A random Walk in the Space of Solutions

We will next have a glimpse at a "gradient descent" in the space of possible solutions to these challenges with hints about suggested language design and language construct improvements that apply specifically to Prolog and Prolog-like languages and to lesser extent, also to their ASP or Datalog cousins.

3.1 The Testbed: Natlog, a Lightweight Prolog-dialect Embedded in Python

Our Python-based Natlog system has been originally introduced in [8], to which we refer to for syntax, semantics and low level implementation details. It is currently evolving as a fresh implementation[1], and it will be used as a testbed for the key ideas of this paper.

[1] at `https://github.com/ptarau/natlog`, ready to install with "`pip3 install natlog`"

Prolog's semantics, but with a lighter syntax While fixing semantics as the usual SLD-resolution, we can keep the syntax and the pragmatics of a logic-based language as close as possible to natural language[2].

We have sketched an attempt to that in the Natlog system's syntax, that we will adopt here. As a hint of its syntactic simplifications, here is a short extract from the usual family program in Natlog syntax:

```
sibling of X S: parent of X P, parent of S P, distinct S X.

grand parent of X GP: parent of X P, parent of P GP.
```

```
ancestor of X A : parent of X  P, parent or ancestor P A.

parent or ancestor P P.
parent or ancestor P A : ancestor of P A.
```

3.2 A Quick Tour of a few Low-Hanging Expressiveness Lifters

Expressiveness is the relevant distinguishing factor between Turing-complete languages. It can be seen as a pillar of code development automation as clear and compact notation entails that more is delegated to the machine.

A finite function API Finite functions (tuples, lists, dictionaries, sets) are instrumental in getting things done with focus on the problem to solve rather than its representation in the language.

In Natlog they are directly borrowed from Python and in systems like SWI-Prolog dictionaries are a built-in datatype. They can be easily emulated in Prolog but often with a different complexity than if natively implemented.

In an immutable form as well as enabled with backtrackable and non-backtrackable updates, finite functions implemented as dynamic arrays and hash-maps can offer a less procedural and more expressive alternative to reliance on Prolog's `assert` and `retract` family of built-ins.

Built-ins as functions or generators Reversible code like in Prolog's classic `append/3` examples or the use of DCGs in both parsing and generation are nice and unique language features derived from the underlying SLD-resolution semantics, but trying to lend reversibility and more generally multi-mode uses to built-ins is often a source of perplexity. Keeping built-ins uniform and predictable, while not giving up on flexibility, can be achieved by restricting them to a few clearly specified uses:

– functions with no meaningful return like `print`, denoted in Natlog by prefixing their Python calls with "`#`".

[2] but *not closer*, as unnecessary verbosity can hinder expressiveness

– functions of N inputs returning a single output as the last argument of the corresponding predicate with $N+1$ arguments, denoted in Natlog by prefixing their calls with a backquote symbol "`". Note that this syntax, more generally, also covers Python's *callables* and in particular class objects acting as instance constructors.
– generators with N inputs yielding a series of output values on backtracking by binding the $N+1$-th argument of the corresponding predicate, denoted in Natlog by prefixing their call with two backquotes " ``".

This simplification (as implemented in Natlog) would also make type checking easier and enable type inference to propagate from the built-ins to predicates sharing their arguments as a convenient mechanism to implement gradual typing.

4 A Step on "The Road Not Taken": First Class Logic Engines

While constraint solvers and related coroutining primitives are present in most widely used logic-based languages, first class logic engines, seen as on-demand reflection of the full execution mechanism, as implemented in BinProlog [7], have been adopted only in SWI Prolog relative recently[3]. Interestingly, similar constructs have been present as far as in [2], where they were efficiently implemented at abstract machine level.

One can think about First Class Logic Engines as a way to ensure the *full meta-level reflection* of the execution algorithm. As a result, they enable on-demand computations in an engine rather than the usual eager execution mechanism of Prolog.

We will spend more time on them as we see them as "the path not taken" that can bring significant expressiveness benefits to logic-based languages, similarly to the way Python's `yield` primitive supports creation of user-defined generators and other compositional asynchronous programming constructs. To obtain the full reflection of Natlog's multiple-answer generation mechanism, we will make fresh instances of the interpreter first-class objects.

4.1 A First-class Logic Engines API

A *logic engine* is a Natlog language processor reflected through an API that allows its computations to be controlled interactively from another *logic engine*.

This is very much the same thing as a programmer controlling Prolog's interactive toplevel loop: launch a new goal, ask for a new answer, interpret it, react to it. The exception is that it is not the programmer, but it is the program that does it! We will next summarize the execution mechanism of Natlog's first class logic engines.

The predicate "`eng AnswerPattern Goal Engine`" creates a new instance of the Natlog interpreter, uniquely identified by `Engine` that shares its code with the currently running program. It is initialized with `Goal` as a starting point. `AnswerPattern` ensures that answers returned by the engine will be instances of the pattern.

The predicate "`ask Engine AnswerInstance`" tries to harvest the answer computed from `Goal`, as an instance of `AnswerPattern`. If an answer is found, it is returned as `(the AnswerInstance)`, otherwise the atom `no` is returned. It is used to retrieve

[3] `https://www.swi-prolog.org/pldoc/man?section=engines`

successive answers generated by an engine, on demand. It is also responsible for actually triggering computations in the engine.

One can see this as transforming Natlog's backtracking over all answers into a deterministic stream of lazily generated answers.

Finally, the predicate "`stop Engine`" stops the Engine, reclaiming the resources it has used and ensures that `no` is returned for all future queries to the engine.

Natlog's yield operation: a key coroutining primitive Besides these predicates exposing a logic engine as a first class object, the annotation "`^Term`" extends our coroutining mechanism by allowing answers to be *yielded from arbitrary places* in the computation. It is implemented simply by using Python's `yield` operation. As implemented in Python, engines can be seen as a special case of generators that yield one answer at a time, on demand.

4.2 Things that we can do with First Class Logic Engines

We will sketch here a few expressiveness improvements First Class Logic Engines can bring to a logic-based programming language,

Source-level emulation of some key built-ins with engines We can emulate at source level some key Prolog built-ins in terms of engine operations, shown here with Natlog's simplified syntax.

```
if_ C Y N : eng C C E, ask E R, stop E, pick_ R C Y N.

pick_ (the C)  C Y _N : call Y.
pick_ no _C _Y N : call N.

not_ G : if_ G (fail) (true).
once_ G : if_ G (true) (fail).

findall_ X G Xs : eng X G E, ask E Y, collect_all_ E Y Xs.

collect_all_ _ no ().
collect_all_ E (the X) (X Xs) : ask E Y, collect_all_ E Y Xs.
```

An infinite Fibonacci stream with yield Like in a non-strict functional language, one can create an infinite recursive loop from which values are yielded as the computation advances:

```
fibo N Xs : eng X (slide_fibo 1 1) E,  take N E Xs.

slide_fibo X Y :  with X + Y as Z,  ^X, slide_fibo Y Z.
```

Note that the infinite loop's results, when seen from the outside, show up as a stream of answers as if produced on backtracking. With help of the library predicate `take`, we extract the first 5 (seen as a Python dictionary with name "`X`" of the variable as a key and the nested tuple representation of Natlog's list as a value), as follows:

```
?- fibo 5 Xs?
ANSWER: {'Xs': (1, (1, (2, (3, (5, ()))))))}
```

5 Borrowing some Magic: Logic Grammars as Prompt Generators

With magic wands on a lease from text-to-text generators like GPT3 [1] and text-to-image generators like DALL-E [5] or Stable Diffusion [10] we can introduce Definite Clause Grammars (DCGs) as prompt generators for such systems.

As examples of the natural synergy between declarative constructs of a logic-based language and the declarative features of today's deep learning systems, we will next overview Natlog applications for text-to-text and text-to-image generation. We refer to the Natlog code[4] and its Python companion[5] for full implementation details.

5.1 Prompt engineering by extending GPT3's text completion

GPT3 is basically a text completion engine, which, when given an initial segment of a sentence or paragraph as a *prompt*, it will complete it, often with highly coherent and informative results.

Thus, to get from GPT3 the intended output (e.g., answer to a question, elations extracted from a sentence, building analogies, etc.) one needs to rewrite the original input into a prompt that fits GPT3's text completion model.

We will use here Natlog's syntactically lighter Definite Clause Grammars, with one or more terminal symbols prefixed by "`@`" and "`=>`" replacing Prolog's "`-->`". A prompt generator with ability to be specialized for several "kinds" of prompts is described by the DCG rule:

```
prompt Kind QuestText => prefix Kind, sent QuestText, suffix Kind.
```

The predicate `sent` takes a question sentence originating from a user's input and maps it into a DCG non-terminal transforming cons-list `Ws1` into cons-list `Ws2`:

```
sent QuestText Ws1 Ws2 :
   `split QuestText List, to_cons_list List Ws, append Ws Ws2 Ws1.
```

The predicate `query` takes the DCG-generated `prompt` derived from user question `Q` and converts it back to a string passed to GPT'3 completion API by a call to the function `complete`, implemented in Python, with its answer returned in variable `A`.

```
query Kind Q A:
   prompt Kind Q Ps (), to_list Ps List, `join List P, `complete P A.
```

Next we will describe specializations to question/answering, relation extraction and analogy invention. An easy way to transform a question answering task into a completion task is to emulate a hypothetical conversation:

[4] see `https://github.com/ptarau/natlog/blob/main/apps/natgpt/chat.nat`
[5] see `https://github.com/ptarau/natlog/blob/main/apps/natgpt/chat.py`

```
prefix question =>   @ 'If' you would ask me.
suffix question =>  @ 'I' would say that.
```

Extraction of subject-verb-object phrases can be mapped to completion tasks as in:

```
prefix relation =>  @ 'If' you would ask me what are the subject
                       and the verb and the object in .
suffix relation =>
   @  'I' would say subject is.
```

For analogy invention we will need to create a custom trigger as follows:

```
trigger X Y Z =>
   @ given that X relates to Y by analogy
     'I' would briefly say that Z relates to.

analogy X Y Z A:
   trigger X Y Z Ps (), to_list Ps List, `join List P, `complete P A.
```

We will next show interaction examples for all these use cases.

First, question answering:

```
?- query question 'why is logic programming declarative' R?
ANSWER: {'R': 'logic programming is declarative because it expresses the
logic of a problem without describing its control flow. This means that
the programmer does not need to specify the order in which the operations
should  be performed, as the logic programming language will determine
the most efficient way to solve the problem.'}
```

Next, relation extraction. Note that after some preprocessing, the extracted triplets can be used as building blocks for knowledge graphs.

```
?- query relation 'the quick brown fox jumps over the lazy dog' R?
ANSWER: {'R':'"quick brown fox",verb is "jumps" and object is "lazy dog"'}
```

Finally, some examples of analogical reasoning that show GPT3 finding the missing component and explaining its reasoning.

```
?- analogy car wheel bird A?
ANSWER: {'A': 'wing by analogy. This is because both car and wheel
are used for  transportation, while bird and wing are used for flight.'}

?- analogy car driver airplane A?
ANSWER: {'A': 'pilot by analogy. The pilot is responsible for the safe
operation  of the airplane, just as the driver is responsible for the
safe operation of the car.'}
```

5.2 Text-to-image with DALL.E

To declaratively specify the content of an image to DALL.E [5] or Stable Diffusion [10], Natlog's Definite Clause Grammars work as easy to customize prompt generators for such systems.

As the same OpenAI API (with a slightly different Python call) can be used for text-to-image generation (followed by displaying the generate picture in the user's default browser), the interaction with Python is expressed succinctly by the predicate `paint` that receives as `Prompt` the description of the intended picture from the user.

```
paint Prompt: `paint Prompt URL, #print URL, #browse URL.
```

The query to visualize in the user's browser such a DCG-generated prompts is:

```
?- paint '<text description of intended image>'.
```

with an example of output shown in Fig. 1

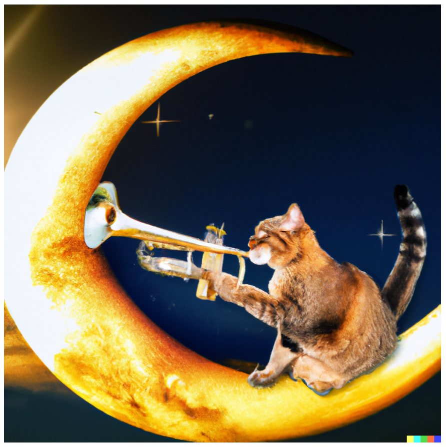

Fig. 1: `paint 'photo of a cat playing on the shiny moon with a trumpet'.`

The Natlog DCG, in generation mode, will iterate over possible styles and content elements of a desired painting as in the following example:

```
image => style, subject, verb, object.

style => @photorealistic rendering.
style => @a dreamy 'Marc' 'Chagall' style picture.
style => @an action video game graphics style image.

subject => @of, adjective, noun.
noun => @robot.
adjective => @shiny.
verb => @walking.
object => location, @with,  instrument.
location => @on planet 'Mars'.

instrument => @high hills and a blue purse.
instrument => @a sombrero hat.
```

This generates text ready to be passed via the OpenAI Python APIs to DALL.E:

```
?- image Words (), `to_tuple Words Ws, #writeln Ws, nl, fail.
photorealistic rendering of shiny robot walking on planet Mars
  with high hills and a blue purse
photorealistic rendering of shiny robot walking on planet Mars
  with a sombrero hat
.....
```

Besides the expected dependence on the `style` component (photorealistic vs. Chagall-style), as an illustration of GPT3's stereotyping bias, female and respectively male features would be derived from the generated robot pictures depending on the `purse` vs. `sombrero hat` picked by the DCG, as can be seen in the generated images[6].

6 Related Work

An introduction to Natlog, its initial proof-of-concept implementation and its content-driven indexing mechanism are covered in [8], but the language constructs and application discussed in this paper are all part of a fresh, "from scratch" implementation. Interoperation with Python has been also used in Janus [6] connecting Python and XSB-Prolog via their foreign language interfaces and systems like DeepProblog [3], in the latter as a facilitator for neuro-symbolic computations.

OpenAI's own `GPT 3.5`-based `ChatGPT`[7] automates the mapping of more queries (e.g., questions, code generation, dialog sessions, etc.) using an extensive Reinforcement Learning With Human Advice process [4]. By contrast, our DCG-supported approach relies exclusively on the pure GPT3 text-completion API on top of which we engineer task-specific prompts.

7 Conclusion

We have informally overviewed automation, learnability and expressiveness challenges faced by logic-based programming languages in the context of today's competitive landscape of alternatives from other programming paradigms as well as from neural net-based machine learning frameworks. We have also sketched solutions to the challenges, with some emphasis on coroutining methods and neuro-symbolic interoperation mechanisms. We have illustrated the surprising synergies that emerge when joining declarative logic programming constructs and declarative prompt-driven interactions with Large Language Models based deep learning systems.

Acknowledgments

These reflections have been inspired by the live and deep-probing Prolog'50 discussions lead by Bob Kowalski and Veronica Dahl with focus on logical thinking and logic-based

[6] at `https://github.com/ptarau/natlog/tree/main/apps/natgpt/pics`
[7] https://chat.openai.com/chat

programming as well as on approaches to make logic-based programming accessible to newcomers, including use cases for a first-contact introduction to computing. I am thankful to the participants of these meetings for sharing their thoughts on both the last 50 years and the next 50 years of logic programming. Finally, many thanks go to the reviewers of the paper for their careful reading and constructive suggestions that helped clarify and substantiate key concepts covered in the paper.

References

1. 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://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf
2. Hermenegildo, M.V.: An abstract machine for restricted AND-parallel execution of logic programs. In: Proceedings on Third international conference on logic programming. pp. 25–39. Springer-Verlag New York, Inc., New York, NY, USA (1986)
3. Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., De Raedt, L.: Deepproblog: Neural probabilistic logic programming. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31, pp. 3749–3759. Curran Associates, Inc. (2018), http://papers.nips.cc/paper/7632-deepproblog-neural-probabilistic-logic-programming.pdf
4. 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
5. Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., Sutskever, I.: Zero-shot text-to-image generation (2021). https://doi.org/10.48550/ARXIV.2102.12092, https://arxiv.org/abs/2102.12092
6. Swift, T.: The Janus System: Multi-paradigm Programming in Prolog and Python . in current volume (2023)
7. Tarau, P.: The BinProlog Experience: Architecture and Implementation Choices for Continuation Passing Prolog and First-Class Logic Engines. Theory and Practice of Logic Programming **12**(1-2), 97–126 (2012). https://doi.org/10.1007/978-3-642-60085-2“˙2
8. 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)
9. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L.u., Polosukhin, I.: Attention is all you need. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017), https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
10. Vision, C.M., at LMU Munich, L.R.G.: Stable Diffusion (2018-2022), https://github.com/CompVis/stable-diffusion