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

Paul Tarau

Department of Computer Science and Engineering
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 uses for interoperation with popular deep learning frameworks.
Keywords: features of logic-based programming languages, declarative language constructs, automation, expressiveness and learnability, neural-symbolic interoperation, coroutining with logic engines

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 adopting 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 mater, with hints on what language constructs are needed for implementing them.

2 The Challenges

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.

This manifests in the most obvious way as replacement of rule-based, symbolic encoding of intelligent behaviors via machine learning, including unsupervised learning among which transformers [6] 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 [2] or prompt-to-image generators like DALL.E [3] or Stable Diffusion [7] have outclassed similarly directed symbolic efforts. It is 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 dozen of seconds.

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

3 A Random Walk in the Space of Solutions

3.1 The Importance of Learnability

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. A good barometer for it 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.

For instance, as logic notation originates in natural language, there are conspicuous mappings between verbs and predicates and between nominal groups as predicate 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.

3.2 The importance of Expressiveness

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.

Among the language features that one is likely to be impressed with at a first contact with Python, the following stand out:

- easiness of defining finite functions (dictionaries, mutable and immutable sequences and sets), all exposed as as first class citizens

– aggregation operations (list, tuple, set, dictionary comprehensions) exposed with a lightweight and flexible syntax
– coroutining (via the yield statement or async annotations) is exposed with a simple and natural syntax
– nested parenthesizing 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 hint next toward bringing the others as constructs in a logic-based language.

3.3 The Importance of Being Declarative (but Definitely not Earnest!)

The following programming language features, not present in Prolog, have evolved in the last 30+ years:

– laziness, seen as on demand execution driven by availability of data or readiness for optimal execution
– refinements in declarative programming, ways to automate derivation of executable code from “elegant” specifications
– syntactic simplifications ensuring readability and accelerating code entry during development
– movement from declared to inferred types and accommodation of gradual typing as a mix of typed and untyped code

Declarative coding as well as laziness seen as asynchronous on-demand evaluation of declared constructs have progressively emerged as effective enhancers of the expressiveness of programming languages adopting them.

Laziness is instrumental also in functional-programming inspired, highly declarative deep learning frameworks like JAX [1]. In this case, laziness acts compositionally, with declaratively specified array operations cooperating with neural network design and compilation steps to architecture independent execution, all in the same linguistic framework.

Arguably, in the presence of similar asynchronous execution mechanisms, one could design a more natural component composition framework on top of Prolog’s Horn Clause logic.

3.4 First Class Logic Engines

Execution eagerness (as present in the usual SLD-resolution implementations) can be alleviated with:

– constraint solvers that suspend computation in a given branch until variables are bound to enough data for progress
– first class engines that suspend computation until answers are asked for - an ‘answers on demand” mechanism

While constraint solvers are present in most widely used logic-based languages, first class logic engines, originating in BinProlog [4] have been only adopted in SWI prolog relative recently[1].

A *first class logic engine* is a 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!

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.

The execution mechanism of interoperating logic engines can be summarized as follows.

- we create a new instance of the interpreter that shares code with the currently running program
- the engine is initialized with a given goal as its starting point
- when queried, an engine iterates over the (possibly infinite stream of solutions), one-by-one
- an engine can also yield partial answers from arbitrary points in the execution path

3.5 The "always on" Connection between Natural Language and Logic

Logical thinking exists outside of logic and in particular outside a specific logic-based programming language. However, statements in logic formalisms closely follow natural language, and with some syntax twisting acumen, translating from one to another is an effortless process, a reason why often mathematicians express connectives, conditionals and quantifiers as natural language constructs.

The Natlog Experiment: Simplifying the basic Language Constructs 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.

We have sketched an attempt to that in the Python-based Natlog system [5].

As a hint of its syntactic simplifications, here is a short extract from the usual family program in Natlog syntax:

```
mother of X M: parent of X M, female M.

father of X M: parent of X M, male M.

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

Recursion, wrapped in a natural feeling syntax, computes "`ancestor of`" as transitive closure of the "`parent of`" relation, given as a set of facts.

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

```
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.
```

Impressing with the Magic: Introducing Logic Grammars as Prompt Generators With magic wands on a lease from the text-to-image generators like DALL-E [3] or Stable Diffusion [7] we can introduce Definite Clause Grammars (DCGs) as prompt generators for such systems.

Again we will use here Natlog's lighter DCG syntax, with "=>" standing for Prolog's "-->" and @X standing for Prolog's [X] used for terminal symbols.

```
@X (X Xs) Xs.

dall_e => @photo, @of, subject, verb, object.

subject => @a, @cat.
subject => @a, @dog.

verb => @playing.

adjective => @golden.
adjective => @shiny.
```

```
object => @on, @the, adjective, location, @with, @a, instrument.

location => @moon.

instrument => @violin.
instrument => @trumpet.

go: dall_e Words (), `to_tuple Words Ws, #writeln Ws, fail.
go.
```

```
QUERY: go.
photo of a cat playing on the golden moon with a violin
photo of a cat playing on the golden moon with a trumpet
photo of a cat playing on the shiny moon with a violin
photo of a cat playing on the shiny moon with a trumpet
...
photo of a dog playing on the shiny moon with a trumpet
```

Such techniques can help with accepting more easily the rougher edges of logic programming, as learners can be motivated when expressiveness brings long chains of causal affects between perceptually distinct channels like image and text.

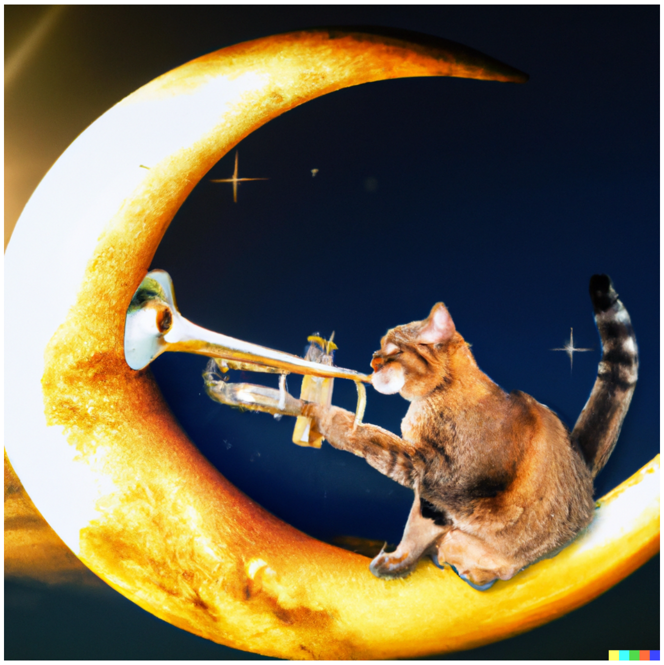

Fig. 1: Result of a DCG-generated Dall.e-prompt

4 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 implementationally speaking "low-hanging fruit" solutions to the challenges, with some emphasis on coroutining methods and neuro-symbolic interoperation mechanisms.

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 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.

References

1. Bradbury, J., Frostig, R., Hawkins, P., Johnson, M.J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., Zhang, Q.: JAX: composable transformations of Python+NumPy programs (2018-2022), http://github.com/google/jax
2. 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
3. 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
4. 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
5. 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)
6. 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
7. Vision, C.M., at LMU Munich, L.R.G.: JAX: composable transformations of Python+NumPy programs (2018-2022), https://github.com/CompVis/stable-diffusion