A Healthy Diet for Python: devouring a Logic-based Language, organically

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
paul.tarau@unt.edu

Abstract. We identify family resemblances that enable high-level interaction patterns via equivalent language constructs and data types between Python and our Python-based embedded logic-based language Natlog. By directly connecting generators and backtracking, nested tuples and terms, reflection and meta-interpretation, coroutines and first-class logic engines, we enable logic-based language constructs to access the full power of the Python ecosystem.

Keywords: logic-based embedded languages, high-level inter-paradigm data exchanges, coroutining with logic engines

Despite of inhabiting different programming paradigms, Python and Prolog share non-trivial "family resemblances" that might still not be obvious even to fluent speakers of both languages. We will overview them in the context of an unusually lightweight embedding of a Prolog-like logic programming language, Natlog[1] [5].

High-level data exchanges Natlog borrows "as is" all Python's finite functions (sequences, sets, dicts) in mutable or immutable form. It operates on them directly or via Python function and generator calls.

Callables (functions, classes, instances defining a `__call__` method) are invoked from Natlog as in:

```
?- `len (a b c) L.
ANSWER: {'L': 3}
```

Generators are reflected in Natlog as alternative answers on backtracking.

```
?- ``range 1 4 X.
ANSWER: {'X': 1}
ANSWER: {'X': 2}
ANSWER: {'X': 3}
```

When Natlog is called from Python, variable assignments are yielded as Python `dict` objects. Python namespaces are reflected in Natlog via the `globals` dictionary, giving access to Python function, class and variable names.

[1] `https://github.com/ptarau/natlog`

Terms as nested tuples Instead of defining a custom class, Natlog represents terms as immutable nested Python tuples, with a one-to-one mapping of leaves (integers, strings etc.) between the two sides, with the exception of Logic variables that are instances of a lightweight class with a single mutable value field. This lets Natlog inherit Python's automated memory management and allows Natlog to work directly with arbitrary Python datatypes, including matrices and tensors from deep learning frameworks.

Generators and backtracking Natlog's backtracking is easily expressed in terms of Python's generators and coroutining constructs that suspend and resume execution after yielding values in recursively nested calls. At the same time, Natlog's multiple answers are yielded to Python via a generator interface. This tight integration of control structures is lifted to new heights with Natlog's first class logic engines.

Coroutining with yield and first-class logic engines A *first class logic engine* [4] 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. In Natlog, we implement first class logic engines by exposing the interpreter to itself as a Python coroutine that transfers its answers one at a time via Python's `yield` operation.

We will next overview two applications, showing the effectiveness of Natlog as an embedded language.

Natlog's DCGs as DALL.E prompt generators Being embedded in the Python ecosystem offers an opportunity to combine the expressiveness of Definite Clause Grammars (DCGs) and Python's API calls to neural text-prompt-to-image generators. With them Natlog generates custom prompts that are passed to the DALL.E [3] Python API, which returns the generated images[2].

Natlog as an orchestrator for JAX and Pytorch deep learning frameworks Google and OpenMind's JAX [1] deep learning framework is referentially transparent (no destructive assignments) and fully compositional, via a lazy application of matrix/tensor operations, automatic gradient computations and a just-in-time compilation function also represented as a first-class language construct.

In a Python-based logic language like Natlog, defining the network architecture and running it is expressed as a set of Horn Clauses with logic variables bound to immutable JAX objects transferring inputs and outputs between neural predicates representing the network layers[3].

[2] `https://github.com/ptarau/natlog/blob/main/natlog/natprogs/dall_e.nat`

[3] `https://github.com/ptarau/natlog/blob/main/apps/deepnat/`

JAX's high-level, referentially transparent tensor operations can be reflected safely as Natlog predicates by just having the result of the underlying N-argument function unified with the predicate's N+1-th extra argument.

In the `Pytorch` ecosystem [2] a combination of object-orientation and callable models encapsulate the underlying complexities of dataset management, neural network architecture choices, automatic differentiation, backpropagation and optimization steps. In our implementation we follow a similar encapsulation of architectural components as in JAX, while delegating to Python more of the details of building, initializing and running the network layers.

Conclusion We have sketched how a few high-level interaction patterns via equivalent language constructs and data types open the doors for mutually beneficial interoperation between a logic-based language and the Python ecosystem.

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. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 32. Curran Associates, Inc. (2019), `https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-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)