Natlog: Embedding Logic Programming into the Python Deep-Learning Ecosystem

Paul Tarau

University of North Texas

July 11, 2023

ICLP’2023
there are deep *family resemblances* between Prolog and Python
they suggest and enable a smooth embedding in Python of a lightweight Prolog dialect ⇒ *Natlog*¹
the resulting symbiosis:
- Prolog benefits from the much wider Python deep learning ecosystem
- a Logic Programming language enables neuro-symbolic inference and better deep learning system orchestration
- Natlog’s simplified syntax brings an easy to learn logic programming language to the ML practitioners

¹https://github.com/ptarau/natlog, *install*: “pip3 install natlog”
Our focus on the Python ⇔ Prolog family resemblances

- Python’s generators ⇔ Prolog’s backtracking
- Python’s nested tuples ⇔ Prolog’s terms
- Python’s coroutines ⇔ Prolog’s first-class logic engines
- Python’s reflection ⇔ Prolog’s meta-interpretation
- Other, more minor:
  - list, set, dict comprehensions ⇔ findall, setof, bagof
  - list and tuple syntax similarity
  - high-level I/O for compound objects
  - interactive REPLs
  - Automatic memory management (including symbol GC)
Natlog: a Prolog with a lightweight syntax, embedded in Python

- 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 $P A$.

- terms are represented as nested tuples, all Python datatypes are directly reflected
- except variables: a lightweight class `Var` with a single value slot
- Natlog benefits from Python’s memory management and no data conversion is needed
- Natlog is not slow: 227K LIPS when running under `pypy3`
High-level, intuitive data exchanges

All “callables” (functions, classes, instances defining a \_\_call\_\_ method in Python) 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.
Reflecting metaprogramming constructs

- to conveniently access object and class attributes, Natlog implements `setprop` and `getprop`

```
setprop O K V : #setattr O K V.
getprop O K V : `getattr O K V.
```

- **elegant metaprogramming constructs on the two sides make language interoperation unusually easy**

```
def meth_call(o, f, xs):
    m = getattr(o, f)
    return m(*xs)
```

- Method calls are supported via the Python function `meth_call` as in the following stack manipulation API:

```
stack S : `list S.  % note the use of the callable empty list constructor
push S X : #meth_call S append (X).
pop S X : `meth_call S pop () X.
```
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!
- first class logic engines 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
Natlog’s First Class Logic Engines API

`eng AnswerPattern Goal Engine:`
- creates a new instance of the Natlog interpreter, returned as `Engine`
- shares code with the currently running program
- it is initialized with `Goal` as a starting point, but not started
- `AnswerPattern` ensures that answers returned by the engine will be instances of the pattern.

`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 retrieves successive answers generated by an Engine, on demand
- it is responsible for actually triggering computations in the engine

`stop Engine:`
- stops the Engine, reclaiming the resources it has used
- ensures that `no` is returned for all future queries
The ^ operation: “ejecting” answers from infinite loops

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

\[
\text{fibo } N \ Xs : \text{eng } X \ (\text{slide}_{-}\text{fibo } 1 \ 1) \ E, \ \text{take } N \ E \ Xs.
\]

\[
\text{slide}_{-}\text{fibo } X \ Y : \text{with } X + Y \text{ as } Z, \ ^X, \text{slide}_{-}\text{fibo } Y \ Z.
\]

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 \text{take}, we extract the first 5:

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

Note that answers of an Engine can be “ejected” at any point in the computation (here with the “^X” notation in \text{slide}_{-}\text{fibo})
The `trust` Engine operation

- when the special atom `trust` is yielded, the goal that follows it replaces the goal of the engine, with all backtracking below that point discarded and all memory consumed so far made recoverable.
- ⇒ infinite loops can work in constant space, even in the absence of last call optimization.
- `loop/2` shows how to generate an infinite sequence of natural numbers:

```prolog
loop N N.
loop N X : with N + 1 as M, ^trust loop M X.
```

? - loop 0 X?

ANSWER: {'X': 0}
ANSWER: {'X': 1}
...

Paul Tarau  University of North Texas  Natlog: LP in a Deep-Learning Ecosystem  July 11, 2023  10 / 17
a JAX example: deep xor in Natlog

```
xor 0 0 0.
xor 0 1 1.
xor 1 0 1.
xor 1 1 0.
```

iter recurses $N$ times over the truth table of xor to obtain the truth table of size $2^N$ of $X_1 \text{ xor } X_2 \text{ xor } ... X_n$ that we will use as our synthetic dataset for an MLP network

```
iter N Op X Y: iter_op N Op () E 0 Y, to_tuple E X.
```

```
iter_op 0 _Op E E R R.
iter_op I Op E1 E2 R1 R3 :
  when I > 0, with I - 1 as J,
  Op X R1 R2,
  with $X + X$ as $XX$,  % $x \rightarrow 2x-1$ maps $\{0,1\}$ into $\{-1,1\}$ to facilitate
  with $XX - 1$ as $X1$,  % the work of the network's Linear Layers
  iter_op J Op (X1 E1) E2 R2 R3.
```
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:

\[
\text{prompt } \text{Kind } \text{QuestText} \Rightarrow \text{prefix } \text{Kind, sent } \text{QuestText, suffix } \text{Kind}.
\]

\[
\text{sent takes a question sentence and maps it into a DCG non-terminal by transforming cons-list } \text{Ws1} \text{ into cons-list } \text{Ws2}:
\]

\[
\text{sent } \text{QuestText } \text{Ws1} \text{ Ws2} : \text{\textbackslash \texttt{\textasciitilde split} QuestText List, to_cons_list List Ws, append Ws Ws2 Ws1}.
\]

\[
\text{query takes the DCG-generated prompt derived from user question } \text{Q} \text{ and converts it back to a string passed to GPT’3 completion API}
\]

\[
\text{query } \text{Kind } \text{Q } \text{A}: \text{prompt } \text{Kind } \text{Q } \text{Ps } ()\text{,to_list Ps List, \textbackslash \texttt{\textasciitilde join} List P, \textbackslash \texttt{\textasciitilde complete} P } \text{A}.
\]
Examples

?- query question 'how are transformers used in GPT' R?
ANSWER: {'R': 'transformers are used in GPT (Generative Pre-trained Transformer) models to generate text from a given prompt. The transformer architecture is used to learn the context of the input text and generate a response based on the context. GPT models are used in many natural language processing tasks such as question answering, machine translation, summarization, and text generation.'}

?- 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".'}

?- query relation 'high interest rates try to desperately contain inflation' R.
ANSWER: {'R': '"high interest rates", verb is "try to desperately contain", and object is "inflation".'}

?- 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.'}
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.
verb => @walking.
adjective => @shiny.

object => location, @with, instrument.

location => @on planet 'Mars'.
instrument => @high hills and a blue purse.
instrument => @a sombrero hat.

API:

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

and the image pops-up in the user’s browser.
Two pictures, with the usual bias, even for robots

Figure: paint photorealistic rendering of shiny robot walking on planet Mars: 1) with a sombrero hat and 2) with high hills and a blue purse
The same two, but with a shift in style

Figure: paint a dreamy Marc Chagall style picture of shiny robot walking on planet Mars: 1) with a sombrero hat and 2) with high hills and a blue purse
Natlog is built taking advantage of “family resemblances” between elegant language constructs shared by Python and Prolog:
- generators and backtracking,
- nested tuples and terms
- reflection and meta-interpretation
- coroutines and first-class logic engines

Natlog enables logic-based language constructs to access the full power of the Python ecosystem:
- a logic-base language is a good orchestrator for deep-learning applications
- there are synergies in “prompt engineering” for text-to-text and ’text-to-image’ Generative AI

next in line: Full Automation of Goal-driven LLM Dialog Threads with And-Or Recursors and Refiner Oracles
- turning GPT-4 and friends into “virtual logic engines”:
  - code at https://github.com/ptarau/recursors