Birds of a Feather Flock Together: Embedding a Logic-based Programming Language in the Python Ecosystem

Paul Tarau

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

Abstract. Driven by expressiveness commonalities of Python and our Python-based embedded logic-based language Natlog, we design high-level interaction patterns between equivalent language constructs and data types on the two sides. 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.
We show the effectiveness of our design via Natlog apps working as orchestrators for JAX and Pytorch training and inference pipelines and a DCG-driven DALL.E prompt generator.
Caveat emptor: As the paper is about interoperation of language constructs between Python and a Prolog-like logic programming language implemented in it, we assume that the reader is fluent in both. We also assume familiarity with essential features of today's very high-level programming languages and deep learning frameworks and the design decisions behind them.
Keywords: logic-based embedded languages, programming language design, high-level inter-paradigm data exchanges, coroutining with logic engines, logic-based neuro-symbolic symbiosis, logic-based neural network configuration and training.

1 Introduction

Turing-complete programming languages win users based on how easily one can make computers do things, for which we will use here *expressiveness* as an umbrella concept. Declarative programming has been seen over the years as an *expressiveness enhancer*, assuming that telling *what* rather than *how* to do things makes it easier to achieve a desired outcome. With this view of declarative programming in mind, besides competition from today's functional programming languages and proof assistants, all with strong declarative claims, logic-based languages face an even stiffer competition from the more radical approach coming from deep learning.

At a first level this manifests as replacement of rule-based, symbolic encoding of intelligent behaviors via machine learning, including unsupervised learning among which transformers [15] 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 [7] or Stable Diffusion [16] 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.

At the next level, "no-coding" or "low-coding" automation mechanisms that are replacing or radically reducing the work of human coders are emerging [6]. With training performed on code fetched from open-source repositories, the output of these early AI-driven code generation experiments results in fairly readable and useful Python code, and it is actually marketed by github as a significant productivity accelerator.

Thus it is becoming clearer as time goes by, 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 (including that done in declarative languages) with models directly extracted from labeled and more and more often from raw, unlabeled data.

This forces us to question, as lucidly as possible, what contribution logic-based reasoning and more concretely logic-based language design can bring to this fast evolving ecosystem.

We will focus in this paper on an easy answer to this question, while aware that more accurate but also more intricate answers will come out in the near future as part of emerging neuro-symbolic computing research[1].

Let's start by observing that the actual code base enabling most of today's deep learning systems is a convoluted, ungeneralizable mix of unstructured scripting, low level GPU-acceleration and linear-algebra libraries. This suggests that there's plenty of room to enhance the design and implementation of the networks themselves in the declarative frameworks logic-based programming languages provide.

As we will argue in the next sections, this requires revisiting some of the design premises and language constructs that logic-based languages share among them, independently of their execution models.

Several language features, not present when the blueprints for logic-based programming languages were conceived, have evolved in the last 30+ years:

- laziness, seen as on demand execution driven by availability of data or readiness for optimal execution
- ways to automate derivation of executable code from specifications
- syntactic simplifications ensuring readability and accelerating interactive development loops
- movement from declared to inferred types and accommodation of gradual typing as a mix of typed and untyped code

Laziness is instrumental 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 achieve architecture independent execution, all in the same

[1] a fast evolving research field, thoroughly overviewed in [9]

linguistic framework. The presence of such asynchronous execution mechanisms is facilitated by Python's built-in coroutining constructs (e.g., `async` and `yield`).

To achieve comparative expressiveness, and to open the doors for using logic-based languages as orchestrators for deep-learning tools, it makes sense to design a similar component composition framework on top of something as simple as Prolog's Horn Clause logic. At the same time, the fact that logic notation originates in natural language hints toward language constructs mapped syntactically closer to natural language equivalents.

There's also plenty of room to borrow, with focus on expressiveness enhancers often in the inner loop of code development, for which Python stands out as the ideal "lender". Among the language features that one is likely to be impressed with at a first contact with Python [8], 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, set, dictionary comprehensions) exposed with a lightweight and flexible syntax
- on demand execution via coroutining is exposed with a simple and natural syntax
- last but not least, nested parenthesizing is avoided or reduced via indentation.

We will explore in the next sections practical language constructs covering some key features where logic-based languages are left behind. To make our language design proposals experimentally testable, we have embedded in Python a lightweight Prolog-like language, Natlog.

The paper is organized as follows. Section 2 introduces and motivates the key design ideas behind the Natlog embedded logic-based language. Section 3 overviews Natlog's coroutining mechanisms using First-Class Logic Engines and their applications. Section 4 shows uses of Natlog scripts as orchestrators for designing, training and testing deep-learning systems in the JAX and Pytorch ecosystems as well as a use case of Natlog's logic grammars as prompt generators for text-to-image large scale model neural networks. Section 5 discusses related work and section 6 concludes the paper.

2 Key Design Ideas behind Natlog, a Lightweight Prolog-dialect Embedded in Python

Our Natlog system has been originally introduced in [12], to which we refer to for syntax, semantics and implementation details. It is currently evolving as a fresh implementation[2], and it will be used as a testbed for the key ideas of this paper.

While keeping Natlog's semantics as close as possible to Prolog, we have brought its syntax a step closer to natural language. In particular, we are not requiring predicate symbols to wrap parenthesized arguments or predicate symbols to be constants. As a hint of its syntactic simplifications, here is a short extract from the usual "family" program in Natlog syntax.

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

```
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.
```

2.1 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. At the same time, expressiveness enhancers need to be kept as simple as possible, with their users experience in mind.

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 any Prolog system, 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 offer a more flexible and semantically simpler 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 multi-mode uses to built-ins obscures code and hinders debugging. Keeping built-ins uniform and predictable, while not giving up on flexibility, can be achieved by restricting them to:

- functions with no meaningful return like `print`, denoted in Natlog by prefixing their Python calls with `#`.
- 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 `‘`. Note that this syntax, more generally, also covers Python's *callables* and in particular class names 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 `‘‘`.

This simplification, as implemented in Natlog, also makes type checking easier and enables type inference to propagate from the built-ins to predicates sharing their arguments. Gradual typing via propagation of the types of uniquely-moded, function-style built-ins is also a reducer of "brittleness", seen as how easy is to break things without warnings from the compiler or run-time system.

Interoperation with Python, as seen from a few Natlog library predicates Interaction with Python's finite functions happens via function calls (with similar Python names) as in:

```
to_dict Xs D : `to_dict Xs D.
from_dict D Xs : `from_dict D Xs.
```

```
arg I T X : `arg T I X.
setarg I T X : #setarg T I X.
```

or generator calls as in:

```
in_dict D K_V : ``in_dict D K_V.
argx I T X: `len T L, ``range 0 L I, `arg T I X.
```

Given Natlog's expected practical uses as a Python package, even when inside Natlog's REPL, answers are shown as the corresponding Python objects, given the one-to-one correspondence between terms and nested tuples and between variable binding and dictionaries.

```
?- to_dict ((one 1) (two 2) (three 3)) D, arg two D X?
ANSWER: {'D': {'one': 1, 'two': 2, 'three': 3}, 'X': 2}

?- eq (a b c) T, argx I T X?
ANSWER: {'T': ('a', 'b', 'c'), 'I': 0, 'X': 'a'}
ANSWER: {'T': ('a', 'b', 'c'), 'I': 1, 'X': 'b'}
ANSWER: {'T': ('a', 'b', 'c'), 'I': 2, 'X': 'c'}
```

Python's built-in generators can be also exposed as backtracking operations, mimicking SWI-Prolog's between/3:

```
between A B X : with B + 1 as SB, ``range A SB X.
```

Natlog uses cons-lists like `(1 (2 (3 ())))` for the usual, unification-based list operations. A few built-in predicates support their conversion to/from Python tuples or lists:

```
to_tuple Xs T : `from_cons_list_as_tuple Xs T.
to_list Xs T : `from_cons_list Xs T.
to_cons_list T Xs : `to_cons_list T Xs.
```

Reflecting metaprogramming constructs In function and generator calls, Python's `eval` is used to map the Natlog name of a function or generator to its Python definition. However, to conveniently access object and class attributes, Natlog implements `setprop` and `getprop` relying directly on corresponding Python built-ins.

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

Similary, 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.
```

This has a surprisingly simple definition, a testimony that *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)
```

As the reader familiar with Python will notice, a method "`m`" is simply an attribute of an object "`o`", directly callable once it has been retrieved from its name "`f`".

These predicates are part of the Natlog library code in file `lib.nat`[3] that can be included as part of a Natlog script with help of the Python function `lconsult`[4].

Reflecting the type system As the following examples show, Python's "`type`" built-in can be used to reflect and inspect natlog's types.

```
?- `type (a b) T?
ANSWER: {'T': <class 'tuple'>}
?- `type a T1, `type b T2, eq T1 T2.
ANSWER: {'T1': <class 'str'>, 'T2': <class 'str'>}
?- `type X V, eq X a, `type X C?
ANSWER: {'X': 'a', 'V': <class 'natlog.unify.Var'>, 'C': <class 'str'>}
```

Note in the last example, that after unification, the type of a variable is dereferenced to the type of its binding.

The two-clause meta-interpreter We next exemplify reflection in Natlog via a simple two-clause metainterpreter and then point out an essential limitation that we would need to overcome by introducing First Class Logic Engines.

The meta-interpreter `metaint/1` uses a (difference)-list view of Horn clauses.

[3] at `https://github.com/ptarau/natlog/blob/main/natlog/natprogs/lib.nat`
[4] in file `https://github.com/ptarau/natlog/blob/main/natlog/natlog.py`

```
metaint ().          % no more goals left, succeed
metaint (G Gs) :     % unify the first goal with the head of a clause
   cls (G Bs) Gs,    % build a new list of goals from the body of the
                     % clause extended with the remaining goals as tail
   metaint Bs.       % interpret the extended body
```

- clauses are represented as facts of the form `cls/2`
- the first argument representing the head of the clause + a list of body goals
- clauses (seen as "difference lists") are terminated with a variable, also the second argument of `cls/2`.

```
cls ((add 0 X X) Tail) Tail.
cls ((add (s X) Y (s Z)) ((add X Y Z) Tail)) Tail.
cls ((goal R) ((add (s (s 0)) (s (s 0)) R) Tail)) Tail.
```

The following example shows it in action:

```
?- metaint ((goal R) ()).
ANSWER: {'R': ('s', ('s', ('s', ('s', 0))))}
```

Note however that we are cheating here[5], as we borrow backtracking and the fresh instance creation for clauses from the underlying runtime system. We will next delve into the details of Natlog's co-routining mechanism, which, while expressed as a small set of strong metaprogramming primitives, is operationally quite close to Python's own `yield` built-in and its coroutining uses.

3 Natlog's First Class Logic Engines

Constraint solvers bring to logic-based languages an automated coroutining mechanism when they suspend computations until more data is available and propagate constraints to the inner loops of SLD-derivations with impressive performance gains including on NP-complete problems. However this implicit coroutining mechanism does not expose the interpreter itself as a first-class object.

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.

[5] as it is also the case with metainterpreters written in Prolog

3.1 A First-class Logic Engines API

To obtain the full reflection of Natlog's multiple-answer generation, we need to provide these mechanisms more directly by making a fresh instance of the interpreter a first-class object.

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!

The execution mechanism of interoperating logic engines can be summarized as follows.
The predicate `eng AnswerPattern Goal Engine` works as follows:

- creates a new instance of the Natlog interpreter, uniquely identified by `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.

The predicate `ask Engine AnswerInstance` works as follows:

- 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 successive answers generated by an Engine, on demand
- it is 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.

The predicate `stop(Engine)` works as follows:

- stops the Engine, reclaiming the resources it has used
- ensures that `no` is returned for all future queries

In Natlog these are implemented as built-in operations.

3.2 The coroutining Mechanism Implemented by the Engine API

The yield operation: a key co-routining primitive The annotation "`^Term`" extends our coroutining mechanism by allowing answers to be *yield from arbitrary places* in the computation. It works as follows:

- it saves the state of the engine and transfers *control* and a *result* `Term` to its client
- the client will receive a copy of `Term` simply by using its `ask/2` operation
- an engine returns control to its client either by initiating a yield operation or when a computed answer becomes available.

When implemented in Python, engines can be seen simply as a special case of generators that yield one answer at a time, on demand.

3.3 Things that we can do with First Class Logic Engines

We will outline here, with help from a few examples, a few expressiveness improvements First Class Logic Engines can bring to a logic-based programming language.

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 function `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, ()))))))}
```

The `trust` operation When the special atom `trust` is yielded, the continuation that follows it replaces the goal of the engine, with all backtracking below that point discarded and all memory consumed so far made recoverable. As a practical consequence, infinite loops can work in constant space, even in the absence of last call optimization.

Using it, the predicate `loop` generates an infinite sequence of natural numbers.

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

```
? - loop 0 X?
ANSWER: {'X': 0}
ANSWER: {'X': 1}
...
```

A more interesting use case is throwing and catching an exception:

```
throw E : ^(exception E), fail.

catch Goal Catcher Recovery:
   eng (just Goal) Goal Engine,
   in Engine Answer,
   maybe_caught Answer Catcher Goal Recovery.

maybe_caught (exception C) C _ Recovery : call Recovery, ^trust true.
maybe_caught (exception C) Catcher _ _ : distinct C Catcher, throw C.
maybe_caught (just G)  _ G _ .
```

Source-level emulation of some key Natlog built-ins with engines While not of a practical importance given that often these are handled by the implementation language of the run-time system, they show the significant expressiveness lift first-class logic engines can bring on top of a Horn Clauses-only sublanguage of Natlog.

```
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.

copy_term_ T CT : eng T (true) E, ask E (the CT), stop E.
```

As a more convoluted example, one can test if a variable is free by ensuring that it can be bound to two distinct values, while using double negation to ensure that such bindings do not actually persist.

```
var_ X : not_ (not_ (eq X 1)), not_ (not_ (eq X 2)).

nonvar_ X: not_ (var_ X).
```

4 Natlog as an Orchestrator for Neuro-Symbolic Deep Learning Systems

We will show here how the use of Natlog as an embedded logic-based scripting language can simplify the design and the execution of neural networks as well as making their internal logic easily understandable.

4.1 Interoperation with JAX and Pytorch

Interoperation with JAX A natural partner to logic-based languages when interacting with deep learning systems is a declaratively designed neural-network like Google and DeepMind's JAX [1].

JAX 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.

The interaction with a logic-based programming language is facilitated by the lazy functional syntax of JAX (seen as an embedded sublanguage in Python).

After defining a set of data loaders, initialization functions and basic neural network layers, both learning and inference can be expressed in JAX as a composition of functions (more exactly as a *future* consisting of such a composition to be eventually activated after a compilation step).

In a Python-based logic language like Natlog, this orchestration process can be 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. JAX's high-level, referentially transparent matrix operations can be reflected safely as Natlog predicates with the result of the underlying N-argument function unified with their N+1-th extra argument. As such, they can be passed as bindings of logic variables between clause heads and clause bodies in an easy to understand, goal-driven design and execution model.

JAX's equivalents of Natlog's compound terms called *pytrees* hold arbitrary aggregates of data and can be differentiated with JAX's `grad` operator as a single unit.

Hyperparameter optimization searches can be naturally expressed as constraint-driven optimization processes.

The synergy between the declarative neural framework and the declarative logic orchestrator can also help with identifying the complex causal chains needed for debugging, optimizing the network architecture as well as with the explainability of both the design and the execution of the resulting neuro-symbolic system.

The following Natlog code snippet[6] generates a "deep xor' dataset, known to be unusually challenging even for deep neural nets.

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

The predicate `iter` recurses N times over the truth table of `xor` to obtain the truth table of size 2^N of X_1 *xor* X_2 *xor* ...X_n that we will use as our synthetic dataset.

```
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->2x-1 maps {0,1} into {-1,1} to facilitate
   with XX - 1 as X1,   % the work of the networks Linear Layers
   iter_op J Op (X1 E1) E2 R2 R3.
```

The dataset will be passed to JAX after conversion to tuples of tuples.

[6] see `https://github.com/ptarau/natlog/blob/main/apps/deepnat/natjax.nat` for more implementation details

```
dataset N Op Xss (Ys):
  findall (X Y) ( iter N Op X Y) XssYsList,
  to_pairs XssYsList XssList YsList,
  to_tuple XssList Xss,
  to_tuple YsList Ys.

to_pairs ()()()  . % note Prolog's [X|Y] becoming (X Y) in Natlog
to_pairs ((Xs Y)  Zss) (Xs Xss) (Y Ys) : to_pairs Zss Xss Ys.
```

We also generate, depending on the number of variables `N`, an appropriate list of hidden-layer sizes (via the predicate `hidden_sizes`) for a Multi-Layer Perceptron that we design in JAX to be trained on our synthetic dataset. We use the "`'`" marker to indicate Python calls to functions like `train_model` and `test_model` implemented in a companion Python file[7].

```
run N Op Seed Epochs Loss Acc LossT AccT:
   dataset N Op Xss Ys,
   split_dataset Xss Ys X Xt Y Yt,
   hidden_sizes N Sizes,
   #print hidden sizes are Sizes,
   `train_model X Y Sizes Epochs (Model LossFun),
   `test_model Model LossFun X Y (Loss Acc),
   `test_model Model LossFun Xt Yt (LossT AccT).
```

Similar calls to pass the dataset to Python and to split it into "train" and "test" subsets, as well as seamless interaction with objects like JAX-arrays (seen by Natlog as constant symbols) are shown in the Natlog definition of `split_dataset`.

```
split_dataset Xss Ys Xtr Xt Ytr Yt:
   `array Xss X,
   `array Ys Y0,
   `transpose Y0 Y,
   `split X Y 0 0.1 (Xtr Xt Ytr Yt),
   show_sizes (X Y Xtr Xt Ytr Yt).
```

Something as simple as `‘array Xss X` converts a Natlog tuple of tuples to a two-dimensional JAX array, relying on the fact that JAX's underlying `numpy` matrix library does such operations automatically. Note that embedding of Natlog in Python makes such data exchanges $O(1)$ in time and space as no data-conversions need to be performed. Note also that the same applies to compound terms that correspond one-to-one to immutable nested tuples in Python and in particular to `pytrees`, instrumental in creating more advanced neural nets in the JAX ecosystem.

Interoperation with Pytorch In the `Pytorch` ecosystem a combination of object-orientation and callable models encapsulate the underlying complexities of dataset management, neural network architecture choices, automatic differentiation, backpropaga-

[7] `https://github.com/ptarau/natlog/blob/main/apps/deepnat/natjax.py`

tion and optimization steps. However, we can follow a similar encapsulation of architectural components as we have shown for JAX, and delegate to Python the details of building and initializing the network layers[8].

Accessing the namespaces of the Python packages To ensure unimpeded access to relevant Python objects that can be as many as a few hundred for packages like JAX or Pytorch, we have devised a namespace sharing mechanism with the Python side. For objects visible in the original namespace of the "`natlog`" package this can be achieved by calling eval on the name of the function or generator. However, when Natlog itself is imported as a package (as in the JAX and Pytorch apps relying on it) we will have to collect to a dictionary the names visible in the Python client importing Natlog in the app. To this end, we use the Python function `share_syms` to extend the `shared` dictionary that is then passed to the `Natlog` class constructor and made available to the run-time system to map strings naming functions, classes or objects to their definitions.

```
def share_syms(shared,names_to_avoid):
    for n, f in globals().items():
          if n not in names_to_avoid:
              shared[n] = f
    return shared
```

4.2 Logic Grammars as 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 magic wands on a lease from generators like DALL-E [7] or Stable Diffusion [16], Natlog's Definite Clause Grammars can work as easy to customize prompt generators for such systems.

We will use here Natlog's lightweight 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.
```

[8] see `https://github.com/ptarau/natlog/blob/main/apps/deepnat/nattorch.nat` and `https://github.com/ptarau/natlog/blob/main/apps/deepnat/nattorch.py` for the details of the implementation

```
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.
```

This generates text ready to be passed via their Python APIs to DALL.E or Stable Diffusion with the output for "*photo of a cat playing on the shiny moon with a trumpet*" shown in Fig. 1.

```
?- 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 golden moon with a violin
photo of a dog playing on the golden moon with a trumpet
photo of a dog playing on the shiny moon with a violin
photo of a dog playing on the shiny moon with a trumpet
```

Fig. 1: Result of our DCG-generated DALL.E-prompt

5 Related Work

An introduction to Natlog, its initial proof-of-concept implementation and its content-driven indexing mechanism are covered in [12], but the language constructs and ap-

plication discussed in this paper are all part of a fresh, "from scratch" implementation. We have implemented similar First-Class Logic Engines in the BinProlog [11] and Jinni Prolog systems [10] as far as 20+ years ago, but, with the exception of their adoption in SWI-Prolog[9] [17] they have not made it into other widely used Prolog systems. This has, in part, motivated their addition to Natlog, given also their strong similarity with Python's own coroutining mechanisms. For some advanced applications of engines, used for adding lazy stream constructs to SWI-Prolog, we refer to [13]. The use of coroutining in languages like C#, JavaScript and Python has also been used in the Yield Prolog system [14] as a facilitator for implementing backtracking, similarly to our implementation. Contrary to Natlog that adopts its own surface syntax and reflection-based interaction with the host language, Yield Prolog requires idiomatic use of the syntax of the target language, making it significantly more cumbersome to work with. On-demand computation and integration with functional programming constructs is also present in the Curry language [4] which has been implemented via compilation to Prolog or Haskell. We also acknowledge here borrowing some of Curry's lightweight syntactic integration of logic and functional constructs. Interoperation with Python has been also used in Problog [3] and DeepProblog [5], in the latter as a facilitator for neuro-symbolic computations. A comprehensive overview of neuro-symbolic reasoning, including logic-based, term-rewriting and graph-based is given in [9]. While in [12] we describe, as an example of neuro-symbolic interaction the use of a neural network as an alternative multi-argument indexer for Natlog, in this paper our focus is on the use of Natlog as an orchestrator for putting together and training deep learning systems.

6 Conclusion

Motivated by expressiveness challenges faced by logic-based programming languages in the context of today's competitive landscape of alternative paradigms as well as from neural net-based machine learning frameworks, we have sketched some implementationally speaking "low-hanging" solutions to them, with emphasis on coroutining methods and neuro-symbolic interoperation mechanisms. The main contributions of this work, likely to be reusable when bridging other Prolog systems to deep-learning ecosystems, are techniques that facilitate interoperation in the presence of high-level language constructs like finite functions, generators, on-demand computations, backtracking, nested immutable data types and strong reflection and metaprogramming features. We have also overviewed several use cases showing the practical expressiveness of the Natlog-Python symbiosis, with focus on using a logic-based language as an orchestrator for declaratively designed deep-learning applications.

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

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

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. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In: IJCAI. vol. 7, pp. 2462–2467 (2007)
4. Hanus, M.: Functional Logic Programming: From Theory to Curry. In: Programming Logics (2013)
5. Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., De Raedt, L.: Deepproblog: Neural probabilistic logic programming. Advances in Neural Information Processing Systems **31** (2018)
6. Raina, A.: The Rise of Low-Code/No-Code Application Platforms (March 2022), https://collabnix.com/
7. 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
8. van Rossum, G.: The Python Language Reference (2017), https://scicomp.ethz.ch/public/manual/Python/3.6.0/reference.pdf
9. Sarker, M.K., Zhou, L., Eberhart, A., Hitzler, P.: Neuro-symbolic artificial intelligence: Current trends (2021). https://doi.org/10.48550/ARXIV.2105.05330, https://arxiv.org/abs/2105.05330
10. Tarau, P.: Inference and Computation Mobility with Jinni. In: Apt, K., Marek, V., Truszczynski, M. (eds.) The Logic Programming Paradigm: a 25 Year Perspective. pp. 33–48. Springer, Berlin Heidelberg (1999), iSBN 3-540-65463-1
11. Tarau, P.: Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In: Lloyd, J. (ed.) Computational Logic–CL 2000: First International Conference. London, UK (Jul 2000), lNCS 1861, Springer-Verlag
12. 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)
13. Tarau, P., Wielemaker, J., Schrijvers, T.: Lazy Stream Programming in Prolog. In: Bogaerts, B., Erdem, E., Fodor, P., Formisano, A., Ianni, G., Inclezan, D., Vidal, G., Villanueva, A., Vos, M.D., Yang, F. (eds.) Technical Communications of ICLP 2019, EPTCS 309 (2019), http://arxiv.org/abs/1907.11354
14. Thmpson, J.: Yield Prolog (2019), https://yieldprolog.sourceforge.net/
15. 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
16. Vision, C.M., at LMU Munich, L.R.G.: JAX: composable transformations of Python+NumPy programs (2018-2022), https://github.com/CompVis/stable-diffusion
17. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**, 67–96 (1 2012). https://doi.org/10.1017/S1471068411000494