Natlog: a Lightweight Logic Programming Language with a Neuro-symbolic Touch

Paul Tarau
Dept. of Computer Science and Engineering
University of North Texas
1155 Union Circle, Denton, Texas 76203, USA
paul.tarau@unt.edu

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

We introduce Natlog, a lightweight Logic Programming language, sharing Prolog's unification-driven execution model, but with a simplified syntax and semantics. Our proof-of-concept Natlog implementation is tightly embedded in the Python-based deep-learning ecosystem with focus on content-driven indexing of ground term datasets. As an overriding of our symbolic indexing algorithm, the same function can be delegated to a neural network, serving (possibly generalized) ground facts to Natlog's resolution engine. Our open-source implementation is available as a Python package at `https://pypi.org/project/natlog/`.

Keyphrases: *Python-based logic programming system, embedded logic programming language, ground term fact database indexing, neuro-symbolic logic programming, logic programming language implementation.*

1 Introduction

With renewed interest in neuro-symbolic AI (d'Avila Garcez and Lamb 2020; Lamb et al. 2020; De Raedt et al. 2020; van Bekkum et al. 2021), integration of logic programming languages into deep-learning ecosystems is becoming of paramount importance. Today's deep-learning frameworks like tensorflow[1] and torch[2] and machine-learning frameworks like scikit-learn[3] (Pedregosa et al. 2011) are all as available as Python packages. This hints toward the need for embedding in the same ecosystem an easy to use, syntactically and semantically lightweight logic programming language, that allows data-scientists with limited exposure to logic programming to build neuro-symbolic systems with enhanced reasoning abilities. While convenient bridges exist between Prolog systems and Python, they require familiarity with the intricacies of an API that assumes user awareness of Prolog's internal term representations and their Python equivalents. Moreover, having a logic engine exposed as a Python callable in the inner loop of a classifier or regression learner might involve not just robustness issues under multi-threading and multi-processing[4], but also performance, scalability and system deployment issues, all important for practical applications.

The often very large datasets neural networks require for training and inference correspond

[1] `https://www.tensorflow.org/`
[2] `https://pytorch.org/`
[3] `https://scikit-learn.org/`
[4] as it is the case with the otherwise excellent pysweep, see `https://github.com/damazter/pysweep`

to ground term databases, sometimes in a flat Datalog format as collected from tabular data or as deeper ground terms coming from JSON, Numpy or Pandas data-frames. As Prolog's indexing mechanism conflates indexing of non-ground predicates used as *code* with indexing of *ground fact databases*, it misses opportunities for deep, content-driven indexing of the latter when needed to retrieve relevant facts as efficiently as possible from arbitrarily large datasets.

To address these shortcomings we have developed a proof-of-concept implementation of Natlog[5] with a light, self-explanatory syntax and basic Horn clause semantics, meant to help data scientists not familiar with Prolog or ASP systems to work with a logic reasoning mechanism modeled on *natural language sentences* and easy to embed in the Python-based deep-learning ecosystem.

We start by listing its key features here, with details expanded in the following sections.

- ability to call not only Python functions but also Python generators as if they were just facts in the database
- ability to pretend to be a Python generator returning a stream of answers
- ability to yield answers from any point in the resolution process, not just at its end
- focus on a clear separation of ground term databases and rule processing components
- ability to plug-in a machine-learning subsystem as if it was just a ground dynamic database
- reliance on pure Python datatypes resulting in ability to be significantly accelerated with the pypy JIT compiler
- content-driven indexing, specialized to ground databases
- natural interface to typical dataset formats (.csv, .json, etc.)
- Hilog-equivalent semantics, allowing general terms in function and predicate symbol position

The rest of the paper is organized as follows: Section 2 overviews syntax elements and the Hilog-like semantics of Natlog. Section 3 describes our proof-of-concept system and its integration in the Python ecosystem. Section 4 focuses on the content-driven ground database indexing mechanism. Section 5 describes details of a plug-in neural network overriding the content-driven database indexer. Section 6 discusses related work and section 7 concludes the paper.

Besides familiarity with Horn Clause logic and general ideas about how Prolog is implemented, we assume the reader is familiar with essential Python[6] language constructs and its basic coroutining mechanisms as supported by the `yield` and `yield from` statements.

2 A Natural Syntax with Hilog Semantics

Our proof-of-concept Natlog implementation relies on Horn Clause logic (in a syntactically lighter form) and a flexible but simple mechanism to delegate to Python everything else.

2.1 A syntax warm-up, by examples

Natlog terms are represented as immutable nested tuples. A parser and scanner for a simplified Prolog term syntax is used to turn terms into nested Python tuples. Surface syntax of facts, as read from strings, is just whitespace separated words (with tuples parenthesized) and sentences ended

[5] `https://github.com/ptarau/pypro`
[6] `https://www.python.org/`

with "." or "?". Like in Prolog, variables are capitalized, unless quoted. With terms represented as *immutable nested tuples*, we adopt a simplified Horn Clause syntax, that removes the need for parenthesizing at clause level and the use of parenthesized tuples for deeper terms.

Example 1
Computing the transitive-closure of a relation.

```
cat is feline.
tiger is feline.
mouse is rodent.
feline is mammal.
rodent is mammal.
snake is reptile.
mammal is animal.
reptile is animal.

tc A Rel C : A Rel B, tc1 B Rel C.

tc1 B _Rel B.
tc1 B Rel C : tc B Rel C.
```

To query it, at the Python prompt one can type:

```
>>> n=natlog(file_name="natprogs/tc.nat") # load program
>>> n.query("tc Who is animal ?") # execute query
```

It will answer with the transitive closure of the "`is`" relation:

```
GOAL PARSED: (('tc', 0, 'is', 'animal'),)
ANSWER: ('tc', 'cat', 'is', 'animal')
ANSWER: ('tc', 'tiger', 'is', 'animal')
...
ANSWER: ('tc', 'reptile', 'is', 'animal')
```

The computed answers are also available as a Python *generator* via the method `solve(...)`, and an interactive top-level is available via the method `repl()`.

To match the usual Prolog semantics, lists can be represented as iterated 2-tuples, with creation of long constant lists delegated to Python.

Example 2
A simple program generating all permutations of a list.

```
perm () ().
perm (X Xs) Zs : perm Xs Ys, ins X Ys Zs.

ins X Xs (X Xs).
ins X (Y Xs) (Y Ys) : ins X Xs Ys.
```

The interpreter can handle function and generator calls to Python using a simple prefix operator syntax. It can also call facts in a ground term database and return an answer from an arbitrary position in the code, as summarized by the following prefix annotations to these functions:

- `#f A B .. Z`: call f(A,B,C,..,Z) for its side effects, with no result returned.
- `` `f A B .. Z R``: call Python function f(A,B,C,..,Z) and unify R with its result

- `''f A B .. Z R`: call Python generator f(A,B,C,..,Z) and unify R with its multiple yields, one at a time
- `^f A B .. Z`: yield term `(f A B ..)` as an answer from any point in the resolution process
- `~P A B .. Z`: unify `(P A B .. Z)` with matching facts in the ground fact database

2.2 The Terms-as-Nested-Tuples Transformation

Replacing Prolog's terms with tuples (immutable arrays in Python) lifts the semantics of Natlog to that of Hilog (Chen et al. 1989). In fact, there's an injective term algebra morphism emphasizing that semantics is still that of first-order Horn Clause logic, described in full detail in (Chen et al. 1989).

We will give the gist of it here, as an injective embedding of the basic Horn Clause subset into our tuples-based representation, which results also in a way to downgrade function symbols to atomic constants, by lifting all parts of a compound term as marked with single functions symbol, not occurring in the term algebra, say "$". Then, we can omit the special symbol as implicit, with the immutable nested tuples representation of our terms. As we will show later, this enables simple, content-driven deep indexing of ground term databases and neuro-symbolic plug-ins.

Example 3
Transformation to nested single function-symbol terms:
`f(A,g(a,B),B)` $\Rightarrow$ `$(f,A,$(g,a,B),B).`

Let $T^{\$}$ be the term algebra $\mathbb{T}$ extended with the constant symbol "$". We define a function $hl : \mathbb{T} \to \mathbb{T}^{\$}$ as follows:

1. if c is a constant, then $hl(c) = c$
2. if v is a variable, then $hl(v) = v$
3. if $x = f(x_1,\ldots,x_n)$ then $hl(x) = \$(f,hl(x_1),\ldots,hl(x_n))$

Following (Tarau 2021), we denote $\odot$ the clause unfolding operation that, when iterated, computes the result of LD-resolution, Prolog's specialization of SLD-resolution (Kowalski and Emden 1976; Lloyd 1984). The transformation $hl : \mathbb{T} \to \mathbb{T}^{\$}$ is injective and it has a left inverse hl^{-1}. The following relations hold:

$$hl(C_1 \odot C_2) = hl(C_1) \odot hl(C_2) \tag{1}$$

$$C_1 \odot C_2 = hl^{-1}(hl(C_1) \odot hl(C_2)) \tag{2}$$

where C_1 and C_2 are Horn Clauses. Observe that $\odot$ commutes with unification and consequently hl and hl^{-1} commute with LD-resolution. The result follows by induction on the structure of $\mathbb{T}$ and the structure of $\mathbb{T}^{\$}$ for its inverse hl^{-1}.

As all terms have only "$" in function symbol positions, by omitting them, one can see these terms as multi-way trees with variable or constant labels marking leaf nodes. Abandoning explicitly marked function and predicate symbols in favor of nested tuples with constants and variables in leaf-positions preserves the semantics of Horn Clause resolution, while it can also accommodate the more general Hilog semantics, should one want to place compound terms in the first position in a nested tuple.

3 The Proof-of-concept Python implementation

We are now ready to overview the details of our proof-of-concept Natlog implementation. A simple regular expression-based *Scanner*[7] returns tokens matching the usual Prolog atomic terms. The recursive descent *Parser*[8] handles our Horn clause syntax and the parenthesized Python-like nested tuple notation for compound terms.

3.1 Representing Terms with Native Python Types

Python is a language that can be often 30-40 times slower than **C** for equivalent code[9]. At the same time it can be significantly accelerated by using the pypy JIT compiler[10], provided that one relies as much as possible on Python's native types that would be mapped by the JIT compiler into equivalent **C** types.

As our terms are immutable and unification operations will dominate the inner-loop of the interpreter, we opt for an *environment* represented as a Python list (in fact, a dynamic mutable array), in which variables, represented as integer indices, will place their bindings to other variables or immutable terms. Thus, we use a simple, structure-sharing representation for our terms, without having to manage our own heap. As Python's native `int` type is borrowed to represent variables inside our nested tuples, we represent actual integers with a class `Int` defined as a wrapper over the native `int` type.

Unification[11] (with an option for occurs-check) and trailing are implemented as usual, the unification algorithm receiving, besides the two terms, the mutable environment and the trail, as parameters.

The interpreter, kept as simple as possible in this proof-of-concept implementation, is an iteration of the clause unfolding operation, similar to the fast Java-based system described in (Tarau 2018), with special care for a natural embedding in the Python ecosystem.

Our data-type choices result in an order of magnitude speed-up[12] with the pypy JIT compiler (e.g., from `35.0755` seconds to `2.9985` seconds on a 10 Queens, all solutions program).

3.2 Quick Overview of the Natlog Interpreter

The Natlog interpreter[13] yields its answers directly from the stream of answers generated by the `step` function which receives a list of goals to reduce. The interpreter is written as declaratively as possible, in a purely functional style, with its key steps implemented as nested inner functions.

It's code skeleton looks like this:

```
# unfolds repeatedly; when done, yields an answer
def interp(css, goals , transformer, db=None):
  def step(goals):
    # reduces goals and yields answer when no more goals
    ...
```

[7] `https://github.com/ptarau/pypro/blob/master/natlog/scanner.py`
[8] `https://github.com/ptarau/pypro/blob/master/natlog/parser.py`
[9] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python3-gcc.html`
[10] `https://www.pypy.org/`
[11] `https://github.com/ptarau/pypro/blob/master/natlog/unify.py`
[12] `https://github.com/ptarau/pypro/blob/master/small_bm.py`
[13] `https://github.com/ptarau/pypro/blob/master/natlog/natlog.py`

```
...
...
goal = goals[0]
vs = list(vars_of(goal))
yield from step((goal, ()))
```

The work is done by the inner `step` function and its inner functions, all accessing the list of clauses `css`, the list of `goals`, a `transformer` overriding Python's `eval` function for possible filtering out of insecure functions or tracing, as well as an optional ground term fact database `db`.

3.2.1 The Unfolding Step

Goals are reduced by unifying the current goal against candidate clauses and extending, on success, the list of goals with new goals derived from the body of a matching clause.

```
  # unfolds a goal using matching clauses
  def unfold(g, gs):
    for cs in css:
      h, bs = cs
      h=relocate(h) # create fresh head
      if not unifyWithEnv(h, g, vs, trail=trail, ocheck=False):
        undo()
        continue  # FAILURE
      else:
        bs1 = relocate(bs) # create fresh body
        bsgs = gs
        for b1 in reversed(bs1) :
          bsgs=(b1,bsgs)
        yield bsgs  # SUCCESS
```

Like in (Tarau 2018) we *relocate* the variable references in the head of the clause prototype for its tentative unification with the goal, and propagate that to its body on success, while undoing them on failure.

3.3 The Python Calls

Example 4
The code supporting a simple call to Python (e.g., "print"), with no return expected looks as follows.

```
 def python_call(g,goals):
    f=eval(g[0])
    args=to_python(g[1:])
    f(*args)
```

More general calls, receiving the value returned by a Python function are implemented with the convention that the result is unified with the last argument of the term. The same convention is followed when calling a Python generator that yields a (possibly infinite) stream of results. The `step` function is called recursively (in this case in a continuation passing style), but turning it into a *trampoline* that eliminates stack usage is quite easy along the lines of (Tarau 2021).

Example 5
A call that unifies the last argument of a term with a value yield from a generator after passing to it its first arguments, assumed ground looks as follows:

```
    def gen_call(g,goals) :
     gen=transformer(g[0])
     g=g[1:]
     v=g[-1]
     args=to_python(g[:-1])
     for r in gen(*args) :
       r=from_python(r)
       if unifyWithEnv(v, r, vs, trail=trail, ocheck=False):
         yield from step(goals) # recursive call here
       undo()
```

Note the `to_python` function that creates a fully dereferenced nested tuple representation of a term and the forwarding of yields from the `step` function, in a continuation passing style.

3.4 Key Features of the Python Embedding

Calling a Python function is exposed by prefixing the function name with `#`.

Example 6
A simple function call.

```
goal X : b X, #print 'printing b =' X, c X.
```

Similarly, generators can be called and have their yields collected into a logic variable as if they were alternative bindings obtained on backtracking:

Example 7
A small program exposing Python generators in Natlog code.

```
good l.
good o.
goal X : ``iter hello X, good X.
goal X : `` range 1000 1005 X.
```

The query “`goal Answer?`” first prints out the characters `'l','l','o'` and then natural numbers from `1000` to `1004`.

Example 8
Pretending to be a Python generator. We assume that the string “`prog`” contains the permutation program in Example 2. Running the query:

```
  n=natlog(text=prog)
  for answer in n.solve("perm (a (b (c ()))) P?"):
    print(answer[2])
```

we obtain a stream of answers yield by our interpreter pretending to be a Python generator:

```
('a', ('b', ('c', ())))
('b', ('a', ('c', ())))
('b', ('c', ('a', ())))
('a', ('c', ('b', ())))
('c', ('a', ('b', ())))
('c', ('b', ('a', ())))
```

Note that in this proof-of-concept implementation list constructors are simply nested tuples of length 2 and components of the answer tuples yield by the interpreter can be accessed with the usual array index notation as in `answer[2]`, selecting the argument P of the tuple `(perm _  P)`.

We have also enabled *yielding an answer from any point in the resolution process*, as a feature enabled by coroutining with first class Prolog engines as described in (Tarau 2012) and also implemented in SWI=Prolog[14].

Example 9
Notation for yielding an answer from an arbitrary point in a program.

```
n = natlog(text="worm : ^o, worm.")
for i , answer in enumerate(n.solve("worm ?")):
  print(answer[0])
  if i >= 42 : break
```

The program will yield, under Python's control of how many answers it wants from the infinite stream generated by "`worm`", the result:

```
ooooooooooooooooooooooooooooooooooooooooooo
```

We separate the Horn Clause-equivalent rule language used for *code* from the *ground database* used to access datasets for Machine Learning applications. This allows us to import .csv, .tsv and .json files as if they were collections of facts in our ground term database.

Example 10
Calls to predicates in the ground database look as ordinary calls except that they are prefixed with a tilde character.

```
~my_database_predicate A B ... Z
```

Natlog will handle them using specialized indexing and unification algorithms.

4 The Content-driven Ground Database Indexer

Traditional Prolog implementations conflate code-indexing and ground database indexing. In fact, clever just-in-time indexing mechanisms (as in YAP and SWI-Prolog) (Costa 2009) or trie-based indexing used for tabling in XSB-Prolog (Swift and Warren 2012) are a testimony of the implementors' ingenuity for covering both code-centered and data-centered indexing in a uniform way. Still, while traditional Prolog indexing makes sense when facts are just base cases of recursive predicates, is is likely to be suboptimal for a database of a few million possibly deep ground facts. In particular, this is the case when small code snippets need to interact with with very large datasets in Machine Learning applications, especially when the underlying logic engine is implemented as comparatively slower Python code.

Moreover, when indexing arbitrarily complex ground terms (e.g., when fetching a JSON file with a deeply nested structure), one might want to focus on the *set of constants* occurring in a given data component, devise based on them a *content-driven* indexing mechanism, and then delegate refining the correct matches to a specialized unification algorithm.

Natlog's content-driven indexing mechanism[15] is specialized to ground terms. It is kept separate from the logic engine and it can be overridden by more elaborate indexing mechanisms, including neural-networks in inference-mode.

[14] `https://www.swi-prolog.org/pldoc/man?section=engines`
[15] `https://github.com/ptarau/pypro/blob/master/natlog/db.py`

4.1 The Indexing Mechanism

When adding a fact to the ground database represented as nested tuple, with atomic constants occurring as leaves, we index it for all constants occurring in it. We use for that a Python dictionary that associates to each constant the set of clauses in which the constant occurs.

Given a query (possibly containing variables) we compute all its ground matches with the database, knowing that *if a constant occurs in the query, it must also occur in a ground term that unifies with it*, as the ground term has no variables in any position that would match the constant otherwise.

We start with the set of clauses where the first constant occurs. Then we reduce it progressively by intersecting it with the sets of clauses in which subsequent constants in the query occur, as shown in the following Python code snippet:

```
def ground_match_of(self,query):
  # find all constants in query
  constants=const_of(query)
  if not constants :
    # match against all clauses self.css, no help from indexing
    return set(range(len(self.css)))
  # pick a copy of the first set where c occurs
  first_constant=next(iter(constants))
  matches=self.index[first_constant].copy()
  # shrink it by intersecting with sets  where other constants occur
  for x in constants:
    matches &= self.index[x]
  # these are all possible ground matches - return them
  return matches
```

Simple optimizations include selecting the constant with the fewest occurrences to provide the set to start with.

4.2 Specializing Unification against Ground Terms

The indexing mechanism relies on the following facts about unifying against a ground term:

- unification against a ground term is sound without occurs-check, given that variables occur only on one side
- if a constant occurs in the (possibly non-ground) query, it must also occur in a ground term that unifies with it, as the ground term has no variables that would match the constant otherwise
- as tuples are immutable, the query term does not need to be copied (or equivalently, heap-represented) and bindings for each attempt to match a ground term in the database can be discarded on failure, simply by throwing away the temporary environment, with no trailing needed

The indexing mechanism returns a set of ground fact candidates but does not guaranty unifiability, which depends not only on the constant set occurring in ground terms but also on their tree structure. Consequently, we rely on the usual unification step of LD-resolution to refine the set of terms that passed the indexing test and filter out false positives.

4.3 Extensions

4.3.1 The Path-to-a-constant Indexing Mechanism

To approximate more closely unification against a ground term, a simple extension to our indexing algorithm is to use the path to the location of each constant, represented as an immutable tuple (to ensure that it is "hashable" – a requirement for keys in Python's dictionaries).

Example 11
Paths to constants represented as tuples in a nested tuple representing a ground term.

```
term:   (f a (g (f b) c))
paths:
  (0 f)
  (1 a)
  (2 0 g)
  (2 1 0 f)
  (2 1 1 b)
  (2 2 c)
```

A more compact (but less selective) indexing mechanism would be to use another immutable (e.g., a `frozenset`) as a key, if one wants to trade some time for space and reduce the size of the index.

It is easy to see that all the properties of unification against a ground term mentioned in subsection 4.2 extend to the case of a path-to-constant indexing mechanism.

For very large fact databases containing deep ground terms, one might want to refine indexing by using as a key the exact path locating the constant which labels a leaf in the multi-way tree representation of a term.

4.3.2 The Ground term Database as an Associative Memory

Another extension is the possible case of ground clauses as "associative memory" elements. This fits LD-resolution semantics as content-driven ground indexing of the heads would yield, besides bindings, clause bodies to be further explored via the usual LD-resolution steps.

5 Using a Neural Network Plug-in as a Content-Driven Ground Term Database Indexer

A neural-net based equivalent of our content-driven indexing algorithm is obtained by overriding its database constructor with a neural-net trained database `ndb()` as shown below:

```
class neural_natlog(natlog):
  def db_init(self):
    self.db=ndb() # neural database equivalent
```

Otherwise, the interface remains unchanged, the LD-resolution engine being oblivious to working with the "symbolic" or "neural" ground-fact database.

We keep the dependencies of Natlog to only two Python packages, `numpy` and `scikit-learn`, and these dependencies are only activated for `neural_natlog`.

The code skeleton for the neural ground term database[16] is implemented as the `ndb` class below:

[16] `https://github.com/ptarau/pypro/blob/master/natlog/ndb.py`

```
class ndb(db) :
  def load(self,fname,learner=neural_learner):
    # overrides database loading mechanism to fit learner
    ...

  def ground_match_of(self,query_tuple):
    # overrides database matching with learned predictions
   ...
```

The overridden `load(...)` method will fit a scikit-learn machine learning algorithm[17] (by default a simple multi-layer perceptron neural network), to yield, when used in inference mode by the method `ground_match_of(...)`, the set of ground clauses likely to match the query. As in the case of the content-driven ground term indexer, we will create an association between the set of constants occurring in the query and the set of ground facts containing them in the database.

Our design will keep in mind the need to return more than one answer from the neural indexer, as multiple facts can match a given query. As usual in Machine Learning terminology, we denote **X** the input from which the algorithm will need to learn the expected output, denoted **y**. Our model will be a *classifier* that associates to each constant the set of clauses it occurs in, to mimic the ground fact database indexer that it overrides.

The training mode, happening in the `load(...)` method, proceeds as follows:

1. load the dataset from a Natlog, .csv, .json file
2. have the superclass "`db`" create the index associating to each constant the set of facts it occurs in
3. create a `numpy` diagonal matrix with one row for each constant (our **X** array)
4. compute a *OneHot encoding*[18] for the set of clauses associated to each constant (our **y** array)
5. call the `fit` method of the the sklearn classifier (a neural net by default, but swappable to any other, e.g., Random Forest, Stochastic Gradient Descent, etc.) with the **X,y** training set

The inference mode, happening in the `ground_match_of(...)` method proceeds as follows:

1. compute the set of all constants in the query that occur in the database
2. compute their OneHot encoding
3. use the classifier's `predict` method to return a bitset encoding the predicted matches
4. decode the bitset to integer indices in the database and return them as matches

Example 12
Natlog program calling a database of properties of chemical elements (note the ˜ prefix in the first clause).

```
data Num Sym Neut Prot Elec Period Group Phase Type Isos Shells :
   ˜ Num Sym Neut Prot Elec Period Group Phase Type Isos Shells.

an_el Num El    : data Num El '45' '35' '35' '4' '17' liq 'Halogen' '19' '4'.
gases Num El : data Num El  _1   _2   _3  _4   _5 gas  _6        _7   _8.
```

The ground database loaded from the tab-separated (.tsv) file `elements.tsv`:

[17] (abbreviated `sklearn` when imported as a package)
[18] basically a bitvector of fixed size

```
1 H 0 1 1 1 1 gas Nonmetal 3 1
2 He 2 2 2 1 18 gas Noble Gas 5 1
3 Li 4 3 3 2 1 solid Alkali Metal 5 2
...
84 Po 126 84 84 6 16 solid Metalloid 34 6
85 At 125 85 85 6 17 solid Noble Gas 21 6
86 Rn 136 86 86 6 18 gas Alkali Metal 20 6
```

The Python program[19] running the Natlog code and the neural-net classifier:

```
def ndb_chem() :
  nd = neural_natlog(
    file_name="natprogs/elements.nat",
    db_name="natprogs/elements.tsv"
  )
  print('RULES')
  print(nd)
  print('DB FACTS')
  print(nd.db)
  nd.query("an_el Num Element ?")
  nd.query("gases Num Element ?")
```

will print out (after listing its data and code), as the result of the second query, the atoms that occur as gases at normal temperature ranges, all computed as candidates provided by the *neural* indexer and then validated by a *symbolic* unification step:

```
GOAL PARSED: (('gases', 0, 1),)
ANSWER: ('gases', '1', 'H')
ANSWER: ('gases', '2', 'He')
...
ANSWER: ('gases', '54', 'Xe')
ANSWER: ('gases', '86', 'Rn')
```

6 Related Work

A similar syntactic departure in functional programming from the function symbol followed by its parenthesized arguments was present as far as in LISP and has persisted in Miranda and Haskell. It has also propagated to Micro-Prolog (Clark and McCabe 1984). The syntax and Hilog-like semantics of Natlog is similar to that of the Java-based iProlog[20] system (Tarau 2018). Derived from "first principles" (starting with a Prolog meta-interpreter), iProlog implements LD-resolution with help of a goal-stack, itself similar to the effect of the binarization transformation described in (Tarau 2012). Contrary to the emphasis on performance in the case of iProlog, which despite being an interpreter and implemented in Java, is only 2-3 times slower than C-based compiled Prolog implementations, our focus in Natlog was its seamless integration with the Python machine-learning ecosystem. This task was largely facilitated by the presence of Python's `eval` function, providing easy bi-directional calls. To keep things simple, Natlog adopts a structure-sharing execution algorithm, while iProlog is essentially, like the WAM (Aït-Kaci 1991), a structure-copying system.

[19] tests.py
[20] `https://github.com/ptarau/iProlog`

Work on deep indexing of Prolog terms (Ramesh et al. 2001) used an automaton based on paths tracing function symbols in the term tree. That is similar to the path-to-the-constant indexing extension discussed in this paper, except for the simplifications that we obtain by focusing on ground terms and the expression of our algorithm exclusively in terms of set intersection operations. In (Tarau 2018) a more elaborate indexing mechanism relying on constant symbols occurring in a term is described, which also accommodates non-ground clauses. The mechanism adopted in Natlog can be seen as its specialization restricted to ground terms.

More radical departures from the traditional function-symbol + arguments representation of Prolog terms is explored in (Tarau 2021) via a series of unification-oblivious program transformations, that generalize the assumptions on which Hilog (Chen et al. 1989) and (in a syntactically different wrapping) Natlog rely.

It is not unusual for logic programming languages to adopt a tight integration with Python as is the case with the ASP system clingo[21] (Cabalar et al. 2020) or with Problog (Manhaeve et al. 2018), where neural machine learning is integrated with a probabilistic logic programming system.

We refer to (d'Avila Garcez and Lamb 2020) for an overview of of the fast-growing attempts to neuro-symbolic integration and to (Lamb et al. 2020) for an overview specialized to graph-neural networks, especially suitable for relational inference on unstructured data. In (De Raedt et al. 2020) a large number of neuro-symbolic systems are surveyed, along multiple features, among which we mention the focus on the expressiveness of the logic ranging from propositional to first order predicate logic and its probabilistic or deterministic nature. An ontology of design patterns for neuro-symbolic systems is explored in (van Bekkum et al. 2021), to which our neural ground fact database indexer would be a possibly new addition.

7 Conclusions

We have described informally (but as informatively as possible) the Natlog logic programming language and its proof-of-concept Python implementation. We will highlight here the novel ideas that it brings with focus on its possible practical contributions to embedding logic programming tools and techniques in deep-learning applications.

The tight integration with Python's generators and coroutining mechanisms enables extending machine-learning applications with an easy to grasp logic programming subsystem. Our departure from traditional Prolog's predicate and term notation puts forward a more readable syntax together with a more flexible Hilog-like semantics. Its closeness to natural-language sentences is likely to be an incentive for adoption by data-scientists not familiar with logic programming.

The content-driven indexing against ground term fact databases is new and it is a potentially useful addition to Prolog and Datalog systems, especially in its extended path-to-the-constant form. As we have shown in a Python code snippet, it is also easily implementable.

Our neural-net plugin mechanism offers a simple yet practical way to integrate deep-learning and logic-based inferences. It is also a new way to approach neuro-symbolic programming by delegating to the machine learning ecosystem a simple subtask at which it is good, while validating correctness of its results symbolically as part of Prolog's well-known LD-resolution execution mechanism.

[21] `https://potassco.org/clingo/`

References

AÏT-KACI, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press.

CABALAR, P., FANDINNO, J., GAREA, J., ROMERO, J., AND SCHAUB, T. 2020. eclingo : A solver for epistemic logic programs. *Theory Pract. Log. Program. 20,* 6, 834–847.

CHEN, W., KIFER, M., AND WARREN, D. 1989. HiLog: A first-order semantics for higher-order logic programming constructs. In *1st North American Conf. Logic Programming*, E. Lusk and R. Overbeek, Eds. MIT Press, Cleveland, OH, 1090–1114.

CLARK, K. L. AND MCCABE, F. G. 1984. *Micro-Prolog - programming in logic*. Prentice Hall international series in computer science. Prentice Hall.

COSTA, V. S. 2009. On just in time indexing of dynamic predicates in prolog. In *Progress in Artificial Intelligence, 14th Portuguese Conference on Artificial Intelligence, EPIA 2009, Aveiro, Portugal, October 12-15, 2009. Proceedings*, L. S. Lopes, N. Lau, P. Mariano, and L. M. Rocha, Eds. Lecture Notes in Computer Science, vol. 5816. Springer, 126–137.

D'AVILA GARCEZ, A. AND LAMB, L. C. 2020. Neurosymbolic ai: The 3rd wave. *arXiv e-prints*, arXiv–2012.

DE RAEDT, L., DUMANČIĆ, S., MANHAEVE, R., AND MARRA, G. 2020. From statistical relational to neuro-symbolic artificial intelligence. *arXiv preprint arXiv:2003.08316*.

KOWALSKI, R. AND EMDEN, M. V. 1976. The Semantics of Predicate Logic as a Programming Language. *JACM 23,* 4 (Oct.), 733–743.

LAMB, L., GARCEZ, A., GORI, M., PRATES, M., AVELAR, P., AND VARDI, M. 2020. Graph neural networks meet neural-symbolic computing: A survey and perspective. *arXiv preprint arXiv:2003.00330*.

LLOYD, J. W. 1984. *Foundations of Logic Programming, 1st Edition*. Springer.

MANHAEVE, R., DUMANCIC, S., KIMMIG, A., DEMEESTER, T., AND DE RAEDT, L. 2018. Deepproblog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 3749–3759.

PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND DUCHESNAY, E. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12*, 2825–2830.

RAMESH, R., RAMAKRISHNAN, I., AND SEKAR, R. 2001. Automata-driven efficient subterm unification. *Theoretical Computer Science 254,* 1, 187–223.

SWIFT, T. AND WARREN, DAVID, S. 2012. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming 12*, 157–187.

TARAU, P. 2012. 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.

TARAU, P. 2018. A Hitchhiker's Guide to Reinventing a Prolog Machine. In *Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017)*, R. Rocha, T. C. Son, C. Mears, and N. Saeedloei, Eds. OpenAccess Series in Informatics (OASIcs), vol. 58. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:16.

TARAU, P. 2021. A Family of Unification-Oblivious Program Transformations and Their Applications. In *Practical Aspects of Declarative Languages - 23rd International Symposium, PADL 2021, Copenhagen, Denmark, January 18-19, 2021, Proceedings*, J. F. Morales and D. A. Orchard, Eds. Lecture Notes in Computer Science, vol. 12548. Springer, 3–19.

VAN BEKKUM, M., DE BOER, M., VAN HARMELEN, F., MEYER-VITALI, A., AND TEIJE, A. T. 2021. Modular design patterns for hybrid learning and reasoning systems: a taxonomy, patterns and use cases. *arXiv preprint arXiv:2102.11965*.