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

Paul Tarau

University of North Texas

September 22, 2021

ICLP'2021

Motivations

- a lightweight Logic Programming language can provide inference services to the Python-based deep-learning ecosystem
- besides seasoned logic programmers, the implementation should be able to serve data scientists unfamiliar with the usual logic programming tools
- $\Rightarrow$ for that, a few things need to be made simpler (e.g., syntax, 2-way interoperation with Python)
- we also want to facilitate for logic programmers work with large datasets and interaction with the deep-learning ecosystem
- $\Rightarrow$ we need a natural framework to explore new neuro-symbolic interaction mechanisms
- $\Rightarrow$ Natlog

A (more) natural syntax, by examples

- a transitive closure computation

```
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.
```

- the usual permutation generator

```
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.
```

A quick look at the interpreter

- terms are immutable Python nested tuples
- goals are unfolded against heads of “prototype“ clauses
- on unification success, bodies of clauses are “relocated“ by replacing their variables with fresh ones
- variables point to term chunks from an environment implemented as a Python list
- variables or compound terms are allowed in predicate positions (Hilog semantics)
- code at: `https://github.com/ptarau/pypro/blob/master/natlog/natlog.py`
- to install and possibly embed in applications: pip3 install natlog

Integration in the Python Ecosystem

- calling a Python function for its result and/or side effects
- calling Python generators and having their yields collected into a logic variable as if they were alternative bindings obtained on backtracking
- pretending to be a Python generator:

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

- ability to yield 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 # stop after the first 42
```

The program will yield from the infinite stream generated by "`worm`", the result:

```
ooooooooooooooooooooooooooooooooooooooooooo
```

Reasons for a Content-driven Ground Database Indexer

- traditional Prolog implementations conflate code-indexing and ground database indexing
- this looked like a good thing when code and data where comparable in size (e.g., WAM)
- typical use cases for Machine Learning (ML) involve much larger datasets than the code handling them !
- $\Rightarrow$ a logic programming language in an ML ecosystem needs a "content-driven" indexing mechanism, besides the usual first or multi-argument indexing of today's Prolog systems

The Indexing Mechanism for Ground Term Databases

- when adding a fact to the ground database (a nested tuple with atomic constants occurring as leaves), we index it using the *set of 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
- 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
- $\Rightarrow$ given a query (possibly containing variables), we compute all its ground matches with the database
- we filter out non-unifiable "false positives" as part of the usual LD-resolution mechanisms

A few Optimizations

- selecting the constant with the fewest occurrences in the database to provide the set to start with
- as tuples are immutable, the query term does not need to be copied (or equivalently, heap-represented)
- specializing Unification against ground terms (e.g., no occurs-check is needed !)
- 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 "Path-to-a-constant Indexing Mechanism"
  - paths to constants are represented as ("hashable" in Python) tuples associated to a ground term
    - for ground term: `(f a (g (f b) c))` the path is: ⇒ `(0 f) (1 a) (2 0 g) (2 1 0 f) (2 1 1 b) (2 2 c)`

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.

The Neural Ground Term Database

- the code skeleton for the neural ground term database is at `https://github.com/ptarau/pypro/blob/master/natlog/ndb.py`
- implemented as the `ndb` class below:

```
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
- (e.g., a multi-layer perceptron neural network)
- it yields, when used in inference mode the set of ground clauses likely to match the query

A Neuro-Symbolic Natlog Program: training mode

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* (a bitvector of fixed size) 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 Neuro-Symbolic Natlog Program: inference mode

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 of facts in the database and return them as matches

Natlog program calling a database of properties of chemical elements

- the program: 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

```
 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
```

x

- The Python program running the Natlog code and the neural-net:

```
def ndb_chem() :
  nd = neural_natlog(
    file_name="natprogs/elements.nat",
    db_name="natprogs/elements.tsv"
  )
  nd.query("gases Num Element ?")
```

- it will print out the atoms that occur as gases at normal temperature
- answers are computed as candidates provided by the *neural* indexer and then validated by a *symbolic* unification step:

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

Conclusions

- Natlog's 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
- syntactic closeness to natural-language sentences facilitates 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
- our neural-net plugin mechanism identifies a new way to experiment with integrating deep-learning and logic-based inferences while validating correctness of the results of neural inferences symbolically