Logic and Problem Solving with Prolog

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

ICLP'2022

Outline

Prolog at 50: Forever Young!

Prolog: Programming in Logic – a (very) short history

- originating in the late 70's, logic based alternative to LISP, ready for AI
- motivations for *Programming in Logic*:
  - Alain Colmerauer: programming grammars for NLP [4, 5]
  - Robert Kowalski: algorithms = logic + control [12, 10]
- from First Order Logic to Prolog in a nutshell:
  - shape of the general case, after elimination of quantifiers:
    `a ; b :- c,d,e.`
  - restricted to a computationally well-behaved subset of predicate logic:
  - *Horn clauses*: `a :- b,c,d.` all variables universally quantified
  - SLD-resolution, using Alan Robinson's unification algorithm
  - multiple answers returned: (improperly) called "non-deterministic" execution
- newer derivatives: Constraint Programming, SAT-solvers, Answer Set Programming: exploit fast execution of propositional logic
- a founding member of the "declarative programming" family
- more about history: 50 years of Prolog, [9], early years: [11]

Prolog: raising again, possibly as part of a new AI-Spring

Programming Language ratings - from the Tiobe index (June 2022)

- 1 Python 12.20 %
- 2 C 11.91 %
- 10 Swift 1.55 %
- 15 Go 1.02 %
- 20 Prolog **0.74** %
- 28 Julia % 0.52%
- 32 Lisp 0.35 %
- 34 Scala 0.30 %
- 38 Haskell 0.21 %
- 44 Scheme 0.19%
- 50 ML 0.17 %

a few years ago: Prolog not on the list (for being behind the first 50)

Prolog: the Basics

A quick Splash into Prolog, with a Graph Coloring Program

```
% color vertices with given colors Cs
color_graph([],_).
color_graph([e(CI,CJ)|Es],Cs):-
  take_one_color(CI,Cs,OtherCs),
  take_one_color(CJ,OtherCs,_), % CJ is anything except CI
  color_graph(Es,Cs).

take_one_color(C,[C|Cs],Cs).
take_one_color(C,[Other|Cs1],[Other|Cs2]):-take_one_color(C,Cs1,Cs2).

% test data: undirected graph  as list of edges
go:-Edges=[
      e(V1,V2),e(V2,V3),e(V1,V3),e(V3,V4),e(V4,V5),
      e(V5,V6),e(V4,V6),e(V2,V5),e(V1,V6)
    ],
    Colors=[r,g,b],
    color_graph(Edges,Colors),
      writeln(Edges),
    fail
;   writeln(done).
```

Running the Program (BTW, all the code for this tutorial is at https://github.com/ptarau/PrologTutorial

```
/*
$ swipl
?- consult('cols.pro').
...
?- go.
[e(r,g),e(g,b),e(r,b),e(b,r),e(r,b),e(b,g),e(r,g),e(g,b),e(r,g)]
[e(r,g),e(g,b),e(r,b),e(b,g),e(g,r),e(r,b),e(g,b),e(g,r),e(r,b)]
[e(r,b),e(b,g),e(r,g),e(g,r),e(r,g),e(g,b),e(r,b),e(b,g),e(r,b)]
[e(r,b),e(b,g),e(r,g),e(g,b),e(b,r),e(r,g),e(b,g),e(b,r),e(r,g)]
...
[e(b,g),e(g,r),e(b,r),e(r,g),e(g,b),e(b,r),e(g,r),e(g,b),e(b,r)]
[e(b,g),e(g,r),e(b,r),e(r,b),e(b,r),e(r,g),e(b,g),e(g,r),e(b,g)]
done
true.
*/
```

Prolog: unification, backtracking, clause selection

```
?- X=a,Y=X. % variables uppercase, constants lower
X = a, Y = a.

?- X=a,X=b.
false.

?- f(X,b)=f(a,Y). % compound terms unify recursively
X = a, Y = b.

% clauses
a(1). a(2). a(3).    % facts for a/1
b(2). b(3). b(4).    % facts for b/1

c(0).
c(X):-a(X),b(X).     % a/1 and b/1 must agree on X

?-c(R).              % the goal at the Prolog REPL
R=0; R=2; R=3.       % the stream of answers
```

Propositional Logic: with pure Prolog and a few CUTs

- truth tables
- naive SAT algorithm
- naive tautology prover

```
truthTable(F):-term_variables(F,Vs),eval(F,R),writeln(Vs:R),fail ; true.

sat(X):-eval(X,1),!.

taut(X):- not(eval(X,0)).

eval(X,X):-var(X),!,bit(X).
eval(X,R):-integer(X),!,R=X.
...
eval((A*B),R):-eval(A,X), eval(B,Y), conj(X,Y,R).
eval((A+B),R):-eval(A,X), eval(B,Y), disj(X,Y,R).
 ...
bit(0).
bit(1).
```

full code at: https://github.com/ptarau/PrologTutorial/blob/main/code/boolean_logic.pro

Examples

```
?- truthTable(A->B->C).
[0,0,0]:1
[0,0,1]:1
[0,1,0]:1
[0,1,1]:1
[1,0,0]:1
[1,0,1]:1
[1,1,0]:0
[1,1,1]:1
true.

?- sat(A * -B + B * -C + C * -A).
A = B, B = 0,
C = 1.

% Peirce's law
?- taut(((X -> Y) -> X) -> X).
true.
```

A favorite Prolog Data Type: Lists

full code at: https://github.com/ptarau/PrologTutorial/blob/main/code/lists.pro

```
% concatenate to lists
append_([],Ys, Ys).
append_([X|Xs],Ys,[X|Zs]):-append_(Xs,Ys,Zs).

% member in terms of append
member__(X,Xs):-append_(_,[X|_],Xs).

% generate subsets of a set
subset_of([],[]).
subset_of([_|Xs],Ys):-subset_of(Xs,Ys).
subset_of([X|Xs],[X|Ys]):-subset_of(Xs,Ys).

% a simple quicksort predicate
qsort([],[]).
qsort([X|Xs],Rs):-
  findall(Y,(member(Y,Xs),Y<X),LittleOnes),
  findall(Y,(member(Y,Xs),Y>=X),BigOnes),
  qsort(LittleOnes,Firsts), qsort(BigOnes,Lasts),
  append(Firsts,[X|Lasts],Rs).
```

Examples

```
?- append_([1,X,3],[Y,5],[A,2,B,4,C]).
X = 2,
Y = 4,
A = 1,
B = 3,
C = 5.

?- member__(X,[1,2]).
X = 1 ;
X = 2.

?- subset_of([1,2,3],Xs).
Xs = [] ;
Xs = [3] ;
Xs = [2] ;
Xs = [2, 3] ;
Xs = [1] ;
Xs = [1, 3] ;
Xs = [1, 2] ;
Xs = [1, 2, 3].

?- qsort([3,2,1,5,3,6,6,2],Sorted).
Sorted = [1, 2, 2, 3, 3, 5, 6, 6].
```

A Glimpse at how Prolog works, actually

Unification: key intuition and a few more examples

- Prolog terms: trees with function symbols or atoms labeling their nodes
- two terms unify if the corresponding trees overlap, when allowing variables to match subtrees
- but, this only happens if equality constrains hold: a variable cannot match two different subtrees!
- also, cycles are forbidden (see unify_with_occurs_check), but this is not enforced by default

```
% recursive descent to unify subterms
?- f(X,g(X,X))=f(h(Z,Z),U),Z=a.
X = h(a,a) ,
Z = a ,
U = g(h(a,a),h(a,a))

% the same variables can occur on both sides
?- [X,Y,Z]=[f(Y,Y),g(Z,Z),h(a,b)].
X = f(g(h(a,b),h(a,b)),g(h(a,b),h(a,b))) ,
Y = g(h(a,b),h(a,b)) ,
Z = h(a,b)
```

A stack-based unification algorithm

actually part of the Minlog interpreter at: https://github.com/ptarau/PrologTutorial/blob/main/code/minlog/minlog.py

```
def unify(x, y, trail):
    ustack = [y,x]
    while ustack:
        x1 = deref(ustack.pop())
        x2 = deref(ustack.pop())
        if x1 == x2: continue
        if isinstance(x1, Var): x1.bind(x2, trail)
        elif isinstance(x2, Var): x2.bind(x1, trail)
        elif not (isinstance(x1, tuple) and isinstance(x2, tuple)): return False
        else:  # assuming x1,x2 are both tuples (i.e., compound terms)
            arity = len(x1)
            if len(x2) != arity: return False
            for i in range(arity - 1, -1, -1): # going right to left here!
                ustack.append(x2[i])
                ustack.append(x1[i])
    return True
```

deref(): dereferencing is just following variable-to-variable links until a nonvariable term or unbound variable is reached

Prolog: the two-clause meta-interpreter

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

```
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 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).
```

```
?- metaint([goal(R)]).
R = s(s(s(s(0)))) .
```

Next: Just mimic the 2 clause metainterpreter in Python!

```
def interp(css, goal):
      def step(goals):
        """
        recursively applies unfolding to its goal stack
        backtracking is implemented using "yield"
        """
        def undo(trail):
            ... # unbinds variables on backtracking

        def unfold(g, gs):
          ... # tries to unify goal g with head of a clause

        trail = []
        if goals == ():
            yield goal
        else:
            g, gs = goals
            for newgoals in unfold(g, gs):
                yield from step(newgoals)
                undo(trail)

    yield from step(goal)
```

Fast Implementation: the Warren Abstract Machine (WAM)

- designed by D.H.D. Warren in the early 80'
- uses registers, two stacks (for goals and clause choices), heap and trail
- cited this days as [1], a very good tutorial introduction
- improvements over the years, but basic architecture unchanged
  - simplified WAM, using a transformation to binary clauses [16] [14]
  - alternative designs: stack frames based [19]
  - tabling, first-order semantics for HiLog [13], [3]
  - just-in-time indexing schemes of YAP [6] and SWI-Prolog [18]
- implemented both natively and as a software virtual machine
- an early overview of WAM derivatives: [17]
- a TPLP issue dedicated to Prolog system implementations: [6, 2, 13, 8, 19, 14]
- overview of parallel implementations: [7]

Some Neat Language Constructs in Prolog

A Prolog source-level transformation: Definite Clause Grammars (DCGs)

Prolog's DCG preprocessor transforms a clause defined with "−−>" like

```
a0 --> a1,a2,...,an.
```

into a clause where predicates have two extra arguments expressing a chain of state changes as in

```
a0(S0,Sn):-a1(S0,S1),a2(S1,S2),...,an(Sn-1,Sn).
```

- they can used to compose relations (functions in particular)
- with compound terms (e.g. lists) as arguments they form a Turing-complete embedded language

```
f --> g,h.
f(In,Out):-g(In,Temp),h(Temp,Out).
```

Some extra notation: `{ . . . }` calls to Prolog, `[ . . . ]` wraps terminal symbols

Playing with DCGs - a generation example

```
dalle-->subject,verb,object.

subject-->[a,cat];[a,dog].
verb-->[sitting].
adjective-->[golden];[shiny].
object-->[on,the],adjective,location,[with,a],instrument.
location-->[moon].
instrument-->[violin];[trumpet].

go:-
  dalle(Words,[]),nl,
    member(W,Words), write(W),write(' '),fail.
go.
```

- pick any of the generated sentences
- paste it to: `https://www.craiyon.com/`
- the Dalle-mini program will paint it!

```
?- go.
a cat sitting on the golden moon with a violin
a cat sitting on the golden moon with a trumpet
...
a dog sitting on the shiny moon with a trumpet
```

The imperfect charm of our neuro-symbolic moonshot :-)

Procedural Constructs in Prolog

- Prolog's procedural language constructs are as old as the language itself (a reason some of them will feel a bit rusty)
- CUT ("**!**") is needed to control search and express if-the-else and case constructs
- asserta/1,assertz/1,retract/1,retractall/1,abolish/1,clause/2 create and update the "dynamic database" - basically supporting a self-modifying program
- setarg, nb_setarg support backtrackable or persistent array operations on compound terms playing the role of arrays
- we will not teach them to you, but as you will learn about them just because of that, the advice is to use them with moderation :-)
- but do not fear occasional impurity, especially in library development code (e.g., aggregates) if wrapped up into a declarative looking API

On Prolog Extensions, with a bias for “The Road Not Taken”

Prolog Extensions, with a common theme

the shared theme, at least at implementation level: some form of coroutining

- constraint programming: boolean, finite domains etc.
  - key to implementing them: coroutining via attributed variables
  - goals are suspended until unification triggers resumption
  - another application: lazy streams and in particular lazy lists
- tabling: avoiding re-execution of goals already seen, including with "well-behaved negation as failure" (SLDNF-resolution)
- parallel execution: AND-parallelism, OR-parallelism
- multi-threading, remote execution, coordination
- our next focus: a less well-known but critical extension to keep Prolog competitive with modern coroutining features now prevalent in popular languages: (Python, Javascript, C#, etc.,) $\Rightarrow$ First-class Logic Engines

Coroutining with First-class Logic Engines

we want full reflection of Prolog's multiple-answer generation (our 2-clause meta-intepreter "cheats" when it inherits that from the underlying Prolog)

- a *logic engine* is a Prolog language processor reflected through an API that allows its computations to be controlled interactively from another *logic engine*
- intuition: it 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
- logic engines can create other logic engines as well as external objects
- logic engines can be controlled cooperatively or preemptively

SWI-Prolog code at: `https://github.com/ptarau/PrologTutorial/blob/main/code/engines.pro`

Engines: new_engine/3

new_engine(AnswerPattern, Goal, Engine):

- creates a new instance of the Prolog interpreter, uniquely identified by `Engine`
- shares code with the currently running program
- initialized with `Goal` as a starting point
- `AnswerPattern` : answers returned by the engine will be instances of the pattern

Engines: ask/2, stop/1

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 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 Prolog's backtracking over all answers into a deterministic stream of lazily generated answers

stop(Engine):

- stops the Engine
- `no` is returned for new queries

The `yield` operation: a key co-routining primitive

yield(Term)

- will save the state of the engine and transfer *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 calling `yield/1` or when a computed answer becomes available

A simple application example: throwing an exception

```
throw(E):-yield(exception(E)).
```

SWI-Prolog implementation (also with more on “talking” to the engines) at:
`https://www.swi-prolog.org/pldoc/man?section=engines`

Typical use of the Engine API

1. the *client* creates and initializes a new *engine*
2. the client triggers a new computation in the *engine*:
   - the *client* passes some data and a new goal to the *engine* and issues a `ask/2` operation that passes control to it
   - the *engine* starts a computation from its initial goal or the point where it has been suspended and runs (a copy of) the new goal received from its *client*
   - the *engine* returns (a copy of) the answer, then suspends and returns control to its *client*
3. the *client* interprets the answer and proceeds with its next computation step
4. the process is fully reentrant and the *client* may repeat it from an arbitrary point in its computation

What can we do with first-class engines?

- define the complete set of ISO-Prolog operations at source level
- in fact, one can define the engine operations in Horn clause Prolog - with a bit of black magic (e.g., splitting a term into two variant terms)
- implement (at source level) Erlang-style messaging - with millions of engines
- implement Prolog's dynamic database at source level
- build an algebra for composing engines and their answer streams
- implement "tabling" (a from of dynamic programming that avoids recomputation) at source level

NEW: first-class logic engines have been added to Natlog, a Prolog with a lighter syntax tightly integrated with Python: [15]
use `pip3 install natlog` to install it or get it from
`https://github.com/ptarau/minlog`

Source-level emulation of Prolog built-ins with engines

```
if(Cond,Then,Else):-
   new_engine(Cond,Cond,Engine),
   ask(Engine,Answer), stop(Engine),
   process_answer(Answer,Cond,Then,Else).

process_answer(the(Cond),Cond,Then,_):-call(Then).
process_answer(no,_,_,Else):-call(Else).

not_(G) :- if(G,fail,true).
once_(G) :- if(G,true,fail).
var_(X):-not_(not_(X=1)),not_(not_(X=2)).
nonvar_(X):-not_(var_(X)).

copy_term_(T,CT) :- new_engine(T,true,E), ask(E,the(CT)), stop(E).

findall_(X,G,Xs) :- new_engine(X,G,E), ask(E,Y), collect(E,Y,Xs).

collect(_,no,[]).
collect(E,the(X),[X|Xs]) :- ask(E,Y), collect(E,Y,Xs).
```

Examples

- an infinite Fibonacci stream with yield

```
fibo(X) :- new_engine(_, slide_fibo(1,1),E), repeat, ask(E,the(X)).

slide_fibo(X,Y):-Z is X+Y, yield(X), slide_fibo(Y,Z).
```

```
?- fibo(X).
X = 1 ;
X = 1 ;
X = 2 ;
X = 3 ;
...

?- findall_(X,member(X,[1,2,3]),Xs).
Xs = [1, 2, 3].

?- X=1,if(X=1,Y=2,Y=3).
X = 1,
Y = 2.

?- X=42,if(X=1,Y=2,Y=3).
X = 42,
Y = 3.
```

Prolog as a Program Solving Tool

Sudoku (for kids): the board filled out with logic variables

```
s4x4([[
    [S11,S12, S13,S14],
    [S21,S22, S23,S24],

    [S31,S32, S33,S34],
    [S41,S42, S43,S44]
 ],
 [
    [S11,S21, S31,S41],
    [S12,S22, S32,S42],

    [S13,S23, S33,S43],
    [S14,S24, S34,S44]
 ],
 [
    [S11,S12, S21,S22],
    [S13,S14, S23,S24],

    [S31,S32, S41,S42],
    [S33,S34, S43,S44]
 ]]).
```

The program: just a maplist of maplists of permutations

```
sudoku(Xss):-
 s4x4(Xsss),Xsss=[Xss|_],
 maplist(maplist(permute([1,2,3,4])),Xsss).

permute([],[]).
permute([X|Xs],Zs):-permute(Xs,Ys),ins(X,Ys,Zs).

ins(X,Xs,[X|Xs]).
ins(X,[Y|Xs],[Y|Ys]):-ins(X,Xs,Ys).

go:-sudoku(Xss),nl,member(Xs,Xss),write(Xs),nl,fail ; nl.
```

```
?- go.
[1,2,3,4]
[3,4,1,2]
[2,3,4,1]
[4,1,2,3]
...
```

code at https://github.com/ptarau/PrologTutorial/blob/main/code/sudoku4kids.pro

Sudoku, for ages 13 or more - with graph coloring

- the previous program is simple and declarative but very slow
- what can we do, with just Prolog and no constraints, SAT or ASP?
- sudoku on a NxN grid is NP-complete ...
- NP-complete programs have P-time translations to other NP-complete programs, with possibly faster solvers for them (e.g., SAT)
- our solver candidate will be graph coloring
- we will constrain variables to have different colors that their neighbors on rows, columns or square blocks!
- a bit like the “all_different” global constraint propagator
- full code at: `https://github.com/ptarau/PrologTutorial/blob/main/code/sudoku_in_style.pro`
- moreover, we would like, as we are in Prolog, to either solve a given Problem or to generate (all) problems of a given size

Transforming Sudoku to a graph coloring problem

full code at: https://github.com/ptarau/PrologTutorial/blob/main/code/sudoku_in_style.pro

```
% build dif graph
to_dif_graph(B,Difs):-
   functor(B,_,N),
   bagof(XDif,
     I^J^(index_pairs(N,I-J),to_a_dif_graph(N,B,I,J,XDif)), Difs).

% collects all difs for var at I,J in board
to_a_dif_graph(N,B,I,J,X-Difs):-at(B,I-J,X),
   difs(row_dif,N,B,I,J,RowDifs),
   difs(col_dif,N,B,I,J,ColDifs),
   difs(bloc_dif,N,B,I,J,[BlocDifs]),
   app([RowDifs,ColDifs,BlocDifs],Difs).

% collects difs of a row, col or bloc generator
difs(Generator,N,B,I,J,Difs):-
  bagof(V, call(Generator,N,B,I,J,V),Difs).
```

Once translated to graph coloring, Sudoku is just a maplist of color picks!

```
sudoku(B):-
  functor(B,_,N), % N is the size of the board B
  to_dif_graph(B,Difs),  % builds dif graph
  maplist(pick(N),Difs). % constrains graph
```

- pick/2 finds a position X (seen as a “color”) such that X is not in the set Xs of its neighbors in rows, columns or blocks
- note that this set is trimmed down such that each neighbor (a logical variable or a “color” assigned to an integer) is represent only once

```
pick(N,X-Xs):- between(1,N,X), \+ member_const(X,Xs).

member_const(X,[Y|_]):-nonvar(Y),X=:=Y,!.
member_const(X,[_|Xs]):-member_const(X,Xs).
```

- NOTE: more compact Sudoku solvers can be built using finite domain constraints or ASP-systems, but our point here is the fact that one can do it quite efficiently in plain Prolog

Solving a known to be hard Sudoku problem: “Escargot”

```
escargot([
      [1, _, _, _, _, 7, _, 9, _],
      [_, 3, _, _, 2, _, _, _, 8],
      [_, _, 9, 6, _, _, 5, _, _],
      [_, _, 5, 3, _, _, 9, _, _],
      [_, 1, _, _, 8, _, _, _, 2],
      [6, _, _, _, _, 4, _, _, _],
      [3, _, _, _, _, _, _, 1, _],
      [_, 4, _, _, _, _, _, _, 7],
      [_, _, 7, _, _, _, 3, _, _]
  ]).
1 6 2 8 5 7 4 9 3
5 3 4 1 2 9 6 7 8
7 8 9 6 4 3 5 2 1
4 7 5 3 1 2 9 8 6
9 1 3 5 8 6 7 4 2
6 2 8 7 9 4 1 3 5
3 5 6 4 7 8 2 1 9
2 4 1 9 3 5 8 6 7
8 9 7 2 6 1 3 5 4
```

the program can solve or generate problems of given size (tested up to 16x16)

N-Queens as a pure Horn Clause program

- original idea due to Thom Frühwirth
- row, columns and diagonal constraints are expressed directly in term of the lists containing the positions of the queens, $\Rightarrow$no arithmetic involved!
- see proof of correctness in Wlodek Drabent's paper at ICLP'22

```
goal(Queens):-queens([1,2,3,4,5,6,7,8,9,10],Queens).

queens(Qs,Ps):-gen_places(Qs,Ps),place_queens(Qs,Ps,_,_).

gen_places([],[]).
gen_places([_|Qs],[_|Ps]):-gen_places(Qs,Ps).

% place_queen(Queen,Column,Updiagonal,Downdiagonal)
place_queen(I,[I|_],[I|_],[I|_]).
place_queen(I,[_|Cs],[_|Us],[_|Ds]):-place_queen(I,Cs,Us,Ds).

place_queens([],_,_,_).
place_queens([I|Is],Cs,Us,[_|Ds]):-
  place_queens(Is,Cs,[_|Us],Ds),
  place_queen(I,Cs,Us,Ds).
```

Examples

```
?- qs([a,b,c,d],Qs).
Qs = [b, d, a, c] ;
Qs = [c, a, d, b] .

?- numlist(1,24,Ns),qs(Ns,Qs).
Ns = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
Qs = [24,21,23,20,22,8,3,7,19,10,18,6,4,17,1,5,2,16,14,12,9,15,13,11].
```

- it is a pure Prolog program, no CUTs, no arithmetic
- the program simply states that the queens should be distinct on columns and diagonals, once row position is fixed
- input does not need to be numeric, the program can be seen simply as a filter on the set of all permutations of given size
- unusually fast for a pure Prolog program (e.g., N=24)
- still, not comparable with CLP-FD, SAT or ASP equivalents going up to sizes in the hundreds

The final round: a few harder things, done easily

λ-terms with de Bruijn Indices

- *de Bruijn Indices* provide a name-free representation of lambda terms
- terms that can be transformed by a renaming of variables (α-conversion) will share a unique representation
  - variables following lambda abstractions are omitted
  - their occurrences are marked with natural numbers *counting the number of lambdas until the one binding them* on the way up to the "root" of the term
- `(l(a(s(0),l(0))))` represents α-equivalent terms (roughly, same up to a renaming of their variables) e.g.,
  - λz.λy. (z (λx.x))
  - λz.λy. (z (λz.z))
  - ...
- λ-terms are called *closed* if the de Bruijn counters do not reach outside the scope of the lambda binders, when counting one step up for each binder crossed
- otherwise they are called *open*

λ-terms of and their (simple) types

- λ-terms are binary-unary trees, with leaves in unary arithmetic marking distances to their lambda binders (de Bruijn notation)
- the constructors are
  - `l/1` = lambda node
  - `a/2` = application node
  - `s/1` = de Bruijn index counter (in successor arithmetic)
  - `0` = de Bruijn index zero
- the types of the lambda terms are binary trees:
  - type constructor `->`
  - variables in leaf positions

code at `https://github.com/ptarau/PrologTutorial/blob/main/code/type_inference.pro`

Generating all λ-terms of a given size (possibly open)

- we use successor arithmetic: 0, s(0),s(s(0)) ...
- possibly open term: de Bruijn indices might point higher then our lambda binders
- size definition: a/2 = 2 units, l/1 = 1 unit. s/1 = 1 unit, 0 = 0 units

```
genLambda(s(S),X):-genLambda(X,S,0).

genLambda(X,N1,N2):-nth_elem(X,N1,N2).
genLambda(l(A),s(N1),N2):-genLambda(A,N1,N2).
genLambda(a(A,B),s(s(N1)),N3):-
  genLambda(A,N1,N2),
  genLambda(B,N2,N3).

nth_elem(0,N,N).
nth_elem(s(X),s(N1),N2):-nth_elem(X,N1,N2).
```

Examples

```
?- genLambda(s(s(s(0))),Term).
Term = s(s(0)) ;
Term = l(s(0)) ;
Term = l(l(0)) ;
Term = a(0, 0) ;
false.

?- genLambda(s(s(s(s(0)))),Term).
Term = s(s(s(0))) ;
Term = l(s(s(0))) ;
Term = l(l(s(0))) ;
Term = l(l(l(0))) ;
Term = l(a(0, 0)) ;
Term = a(0, s(0)) ;
Term = a(0, l(0)) ;
Term = a(s(0), 0) ;
Term = a(l(0), 0) ;
false.
```

Generating closed terms

- a list, initially empty of variables is built
- each lambda binder pushes a variable to it
- each leaf is constrained to correspond, via its de Bruijn index to a variable
- we use the list to count binders - but it will later hold types that we infer

```
genClosed(s(S),X):-genClosed(X,[],S,0).

genClosed(X,Vs,N1,N2):-nth_elem_on(X,Vs,N1,N2).
genClosed(l(A),Vs,s(N1),N2):-genClosed(A,[_|Vs],N1,N2).
genClosed(a(A,B),Vs,s(s(N1)),N3):-
  genClosed(A,Vs,N1,N2),
  genClosed(B,Vs,N2,N3).

nth_elem_on(0,[_|_],N,N).
nth_elem_on(s(X),[_|Vs],s(N1),N2):-nth_elem_on(X,Vs,N1,N2).
```

Example

```
?- genClosed(s(s(s(0))),Term).
Term = l(l(0)) .

?- genClosed(s(s(s(s(0)))),Term).
Term = l(l(s(0))) ;
Term = l(l(l(0))) ;
Term = l(a(0, 0)) .

?- genClosed(s(s(s(s(s(0))))),Term).
Term = l(l(l(s(0)))) ;
Term = l(l(l(l(0)))) ;
Term = l(l(a(0, 0))) ;
Term = l(a(0, l(0))) ;
Term = l(a(l(0), 0)) ;
Term = a(l(0), l(0)) .
```

Type Inference for λ-terms

- in an application (`a/2` node): `X->Y` and `X` reduce to `Y`
- all variables of a lambda binder `l/1` should have the same type
- $\Rightarrow$ `unify_with_occurs_check` will unify them, avoiding cycles!

```
type_of(X,T):-type_of(X,T,[]).

type_of(I,V,Vs):-
  nth_elem_of(I,Vs,V0),
  unify_with_occurs_check(V,V0).
type_of(l(A),(X->Y),Vs):-
  type_of(A,Y,[X|Vs]).
type_of(a(A,B),Y,Vs):-
  type_of(A,(X->Y),Vs),
  type_of(B,X,Vs).

nth_elem_of(0,[X|_],X).
nth_elem_of(s(I),[_|Xs],X):-nth_elem_of(I,Xs,X).
```

this is unusually easy in Prolog, but it might take a few hundred lines in your favorite other language ...

Generating simply-typable de Bruijn terms of a given size

- types mimic function application (i.e., β-reduction of lambda terms)
- we refine our program generating closed terms by imposing constraints on the variables introduced by lambda binders
- de Bruijn indices pointing to the same variable should agree on types
- unification with "occurs-check": circular types are not allowed!

```
genTypable(s(S),X,T):-genTypable(X,T,[],S,0).

genTypable(X,V,Vs,N1,N2):-genIndex(X,Vs,V,N1,N2).
genTypable(l(A),(X->Xs),Vs,s(N1),N2):-genTypable(A,Xs,[X|Vs],N1,N2).
genTypable(a(A,B),Xs,Vs,s(s(N1)),N3):-
  genTypable(A,(X->Xs),Vs,N1,N2),
  genTypable(B,X,Vs,N2,N3).

genIndex(0,[V|_],V0,N,N):-unify_with_occurs_check(V0,V).
genIndex(s(X),[_|Vs],V,s(N1),N2):-genIndex(X,Vs,V,N1,N2).
```

Examples

```
?- genTyped(4).
l(l(l(s(0))))
A->B->C->B

l(l(l(l(0))))
A->B->C->D->D

l(a(0,l(0)))
((A->A)->B)->B

l(a(l(0),0))
A->A

a(l(0),l(0))
A->A
```

Writing a theorem prover for Intuitionistic Propositional Logic

code at `https://github.com/ptarau/PrologTutorial/blob/main/code/prover.pro`

- intuitionistic logic is key to today's proof assistants
- there's a correspondence between computations and proofs: the *Curry-Howard isomorphism*
- in its simplest form, it connects the *implicational fragment of propositional intuitionistic logic* with types in the *simply typed lambda calculus*
- type inference computes the type of a λ-term (a theorem in IPC!)
- a theorem prover solves the harder (PSPACE-complete) inverse problem of finding a proof given a formula representing a type
- a simply-typed λ-term can be synthesized as a result of a successful proof for the implicational subset
- next we will implement a more general prover, covering all interesting IPC operators

Roy Dyckhoff's **G4ip** sequent calculus

termination ensured with "multiset ordering", no loop checking is needed!

$\Gamma, p \Rightarrow p \quad Ax \quad (p \text{ an atom})$

$\Gamma, \bot \Rightarrow \Delta \quad L\bot$

$$\frac{\Gamma \Rightarrow \varphi \quad \Gamma \Rightarrow \psi}{\Gamma \Rightarrow \varphi \wedge \psi}\ R\wedge$$

$$\frac{\Gamma, \varphi, \psi \Rightarrow \Delta}{\Gamma, \varphi \wedge \psi \Rightarrow \Delta}\ L\wedge$$

$$\frac{\Gamma \Rightarrow \varphi_i}{\Gamma \Rightarrow \varphi_0 \vee \varphi_1}\ R\vee\ (i = 0, 1)$$

$$\frac{\Gamma, \varphi \Rightarrow \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \varphi \vee \psi \Rightarrow \Delta}\ L\vee$$

$$\frac{\Gamma, \varphi \Rightarrow \psi}{\Gamma \Rightarrow \varphi \to \psi}\ R\to$$

$$\frac{\Gamma, p, \varphi \Rightarrow \Delta}{\Gamma, p, p \to \varphi \Rightarrow \Delta}\ Lp\to\ (p \text{ an atom})$$

$$\frac{\Gamma, \varphi \to (\psi \to \gamma) \Rightarrow \Delta}{\Gamma, \varphi \wedge \psi \to \gamma \Rightarrow \Delta}\ L\wedge\to$$

$$\frac{\Gamma, \varphi \to \gamma, \psi \to \gamma \Rightarrow \Delta}{\Gamma, \varphi \vee \psi \to \gamma \Rightarrow \Delta}\ L\vee\to$$

$$\frac{\Gamma, \psi \to \gamma \Rightarrow \varphi \to \psi \quad \gamma, \Gamma \Rightarrow \Delta}{\Gamma, (\varphi \to \psi) \to \gamma \Rightarrow \Delta}\ L\to\to$$

The Theorem Prover for **IPC**

- we derive the prover directly from Roy Dyckhoff's G4ip calculus
- code at `https://github.com/ptarau/PrologTutorial/blob/main/code/prover.pro`

```
iprover(true,_):-!.
iprover(A,Vs):-memberchk(A,Vs),!.
iprover(_,Vs):-memberchk(false,Vs),!.
iprover(~A,Vs):-!,iprover(false,[A|Vs]).
iprover(A<->B,Vs):-!,iprover(B,[A|Vs]),iprover(A,[B|Vs]).
iprover((A->B),Vs):-!,iprover(B,[A|Vs]).
iprover((B<-A),Vs):-!,iprover(B,[A|Vs]).
iprover(A & B,Vs):-!,iprover(A,Vs),iprover(B,Vs).
iprover(G,Vs1):- % atomic or disj or false
  select(Red,Vs1,Vs2),
  iprover_reduce(Red,G,Vs2,Vs3),
  !,
  iprover(G,Vs3).
iprover(A v B, Vs):-(iprover(A,Vs) ; iprover(B,Vs)),!.
```

The Prover, continued

- we delegate details to helper predicates: `iprover_reduce/4` and `iprover_impl/4`.

```
iprover_reduce(true,_,Vs1,Vs2):-!,iprover_impl(false,false,Vs1,Vs2).
iprover_reduce(~A,_,Vs1,Vs2):-!,iprover_impl(A,false,Vs1,Vs2).
iprover_reduce((A->B),_,Vs1,Vs2):-!,iprover_impl(A,B,Vs1,Vs2).
iprover_reduce((B<-A),_,Vs1,Vs2):-!,iprover_impl(A,B,Vs1,Vs2).
iprover_reduce((A & B),_,Vs,[A,B|Vs]):-!.
iprover_reduce((A<->B),_,Vs,[(A->B),(B->A)|Vs]):-!.
iprover_reduce((A v B),G,Vs,[B|Vs]):-iprover(G,[A|Vs]).
```

```
iprover_impl(true,B,Vs,[B|Vs]):-!.
iprover_impl(~C,B,Vs,[B|Vs]):-!,iprover((C->false),Vs).
iprover_impl((C->D),B,Vs,[B|Vs]):-!,iprover((C->D),[(D->B)|Vs]).
iprover_impl((D<-C),B,Vs,[B|Vs]):-!,iprover((C->D),[(D->B)|Vs]).
iprover_impl((C & D),B,Vs,[(C->(D->B))|Vs]):-!.
iprover_impl((C v D),B,Vs,[(C->B),(D->B)|Vs]):-!.
iprover_impl((C<->D),B,Vs,[((C->D)->((D->C)->B))|Vs]):-!.
iprover_impl(A,B,Vs,[B|Vs]):-memberchk(A,Vs).
```

- Classical Logic “for free”, via Glivenko’s theorem:

```
cprover(T):-iprover( ~ ~T).
```

Examples

```
?- iprover(p v ~p).
false.

?- cprover(p v ~p).
true.

?- iprover(~ ~ ~ p <-> ~ p).
true.

?- iprover(~ ~ p <-> p).
false.

?- cprover(~ ~ p <-> p).
true.

?- iprover((h<-b<-c<-d) <-> (h<-b&c&d)).
true.
```

A simple Prolog-based Neuro-symbolic Computing Technique

On training Neural Networks as Theorem Provers

- we have used Prolog to derive a generator for lambda terms and their types
- we can generate a dataset for training neural networks, turning them into reliable theorem provers, for the harder inverse problem: given a formula in the implicational subset of IPC, find a proof term for it!
- an easier problem: restrict to the linear subset of implicational IPC
- open problems, future work:
  - can this be extended to full fragments of IPC or LL?
  - would the similar success rates apply to large, random generated formulas?
  - how would the NNs perform on larger, human-made formulas?

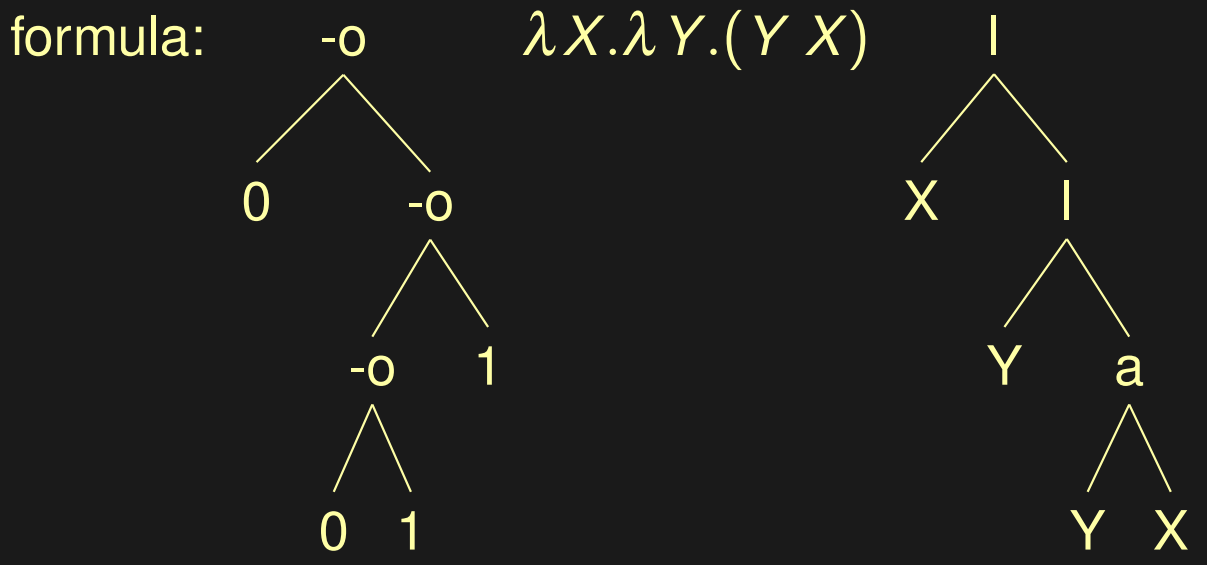
Machine Learning (ML) with Deep Neural Networks (NNs)

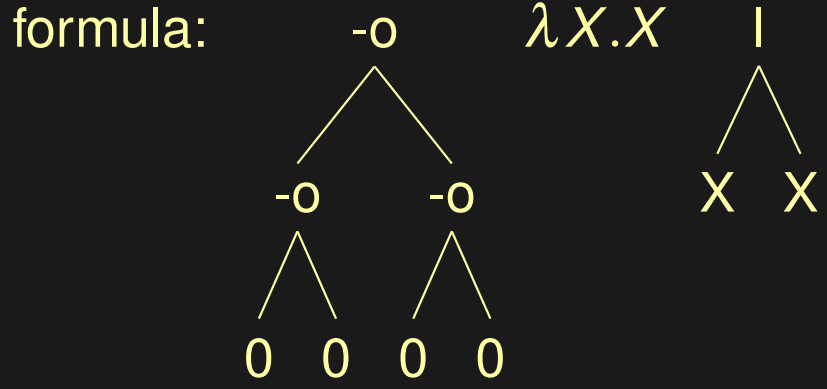
- the key ML concepts to watch for:
  - "honesty": split the dataset into: training, validation and (independent) test sets
  - things to avoid:
    - overfitting (works on training, fails on validation and testing data)
    - unlikely to work well on random (high Kolmogorov complexity) data
- the key NN general concepts to watch for:
  - NNs are *trainable universal approximators* for a given function
  - $L_{t+1} = \sigma(A * L_t + b)$ where L_t is a layer at step t, A is a matrix containing trainable parameters, b is a bias vector and σ is a non-linear function (logistic sigmoid, tanh, RELU(x)=max(0,x), etc.)
  - differentiable functions: gradients computed on backpropagation
  - an intuition behind why deep NNs are needed: each layer abstracts away statistically relevant patterns that are fed to the next layer
  - often, to ensure generalization, information is deliberately lost

Training the Neural Networks as Theorem Provers via the Curry-Howard Isomorphism

- formulas/types and proofs/lambda terms are both trees
- $\Rightarrow$ we can represent them as prefix strings
- what type of neural networks to use?
  - with trees as prefix string: $\Rightarrow$ "seq2seq" recurrent NNs
  - LSTM (long short term memory) NNs : good to handle long distance dependencies in the prefix forms
  - transformers might be better, but they need very large amounts of data and heavy GPU training
- an interesting special case: linear implication with theorems corresponding to lambda terms in which every binder binds exactly one variable
- "$->$" becomes "$-\circ$" (the "lollipop")

Formulas depicted as trees, together with their proof terms





seq2seq Neural Networks

- sequence as input, train to guess sequence as output
- used originally for translation of natural languages, with training on large parallel corpora
- notable alternatives: *transformers*, trained to predict masked words in a sentence as well as predict next sentence in a text
- *unsupervised* - just feeding them very large text data
- examples: BERT, GPT-3 - impressive performance on several NLP tasks (e.g., GPT-3 generating fake news)
- newer variants, possibly more in interesting: **tree2tree**, **dag2dag** and several types of **graph neural networks** (e.g., convolutional, attention, spectral, torch geometric)

LSTM seq2seq Neural Networks

- recurrent neural networks keep track of dependencies within sequences
- feedback from values at time t is fed into computations at time $t+1$
- long short-term memory (LSTM) is a recurrent neural network (RNN) architecture
- it can not only process single data points (such as images), but also entire sequences of data (such as text, speech or video)
- LSTM NNs have feedback connections $\Rightarrow$ LSTM avoids vanishing or exploding gradient problems by also feeding *unchanged* values to the next layer

Evaluating the Performance of our Neural Networks as Theorem Provers

- the experiments with training the neural networks using the linear and intuitionistic theorem dataset are available at: `https://github.com/ptarau/neuralgs`
- the $<$ *formula*, *proof term* $>$ generators are available at: `https://github.com/ptarau/TypesAndProofs`
- the generated datasets are available at: `http://www.cse.unt.edu/~tarau/datasets/`

next, we will illustrate with some plots that our `seq2seq` LSTM recurrent neural network trained on encodings of theorems and their proof-terms performs unusually well

Accuracy of the LSTM seq2seq neural network on our formula/proof term dataset for the linear logic subset

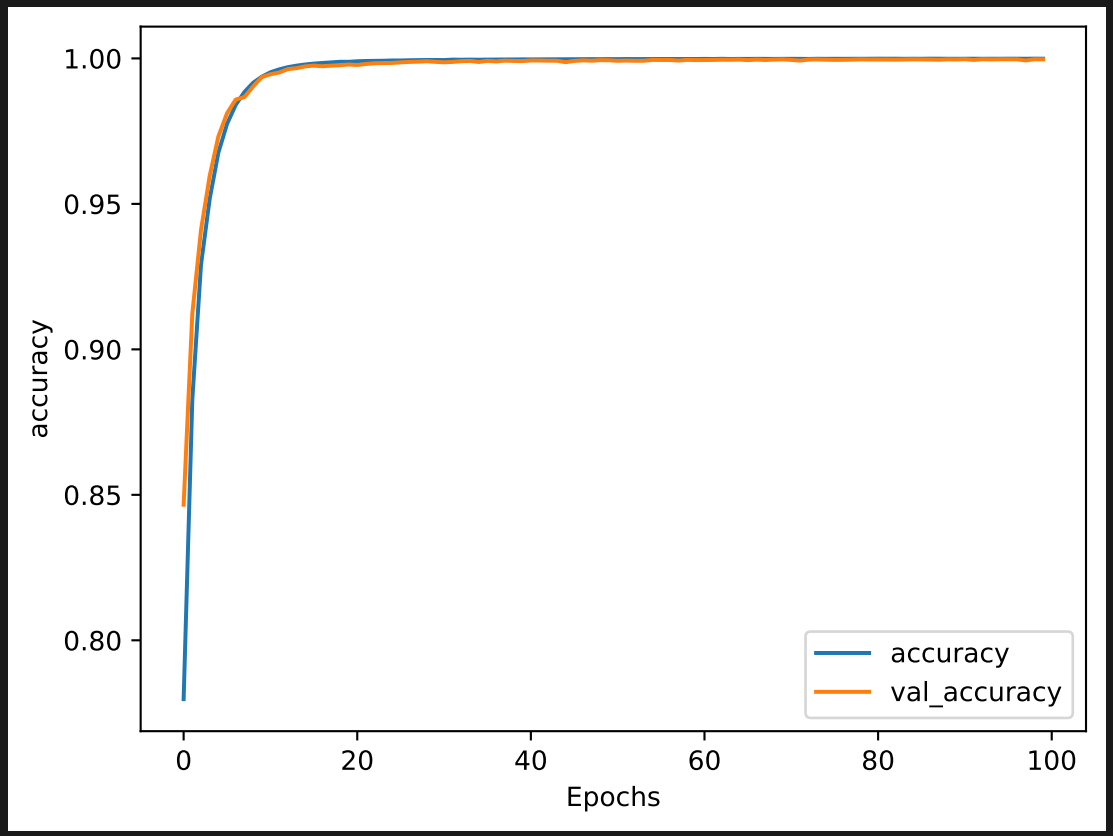


Figure: Accuracy curve for 100 epochs

Loss curve of the LSTM seq2seq neural network on our formula/proof term dataset for the linear logic subset

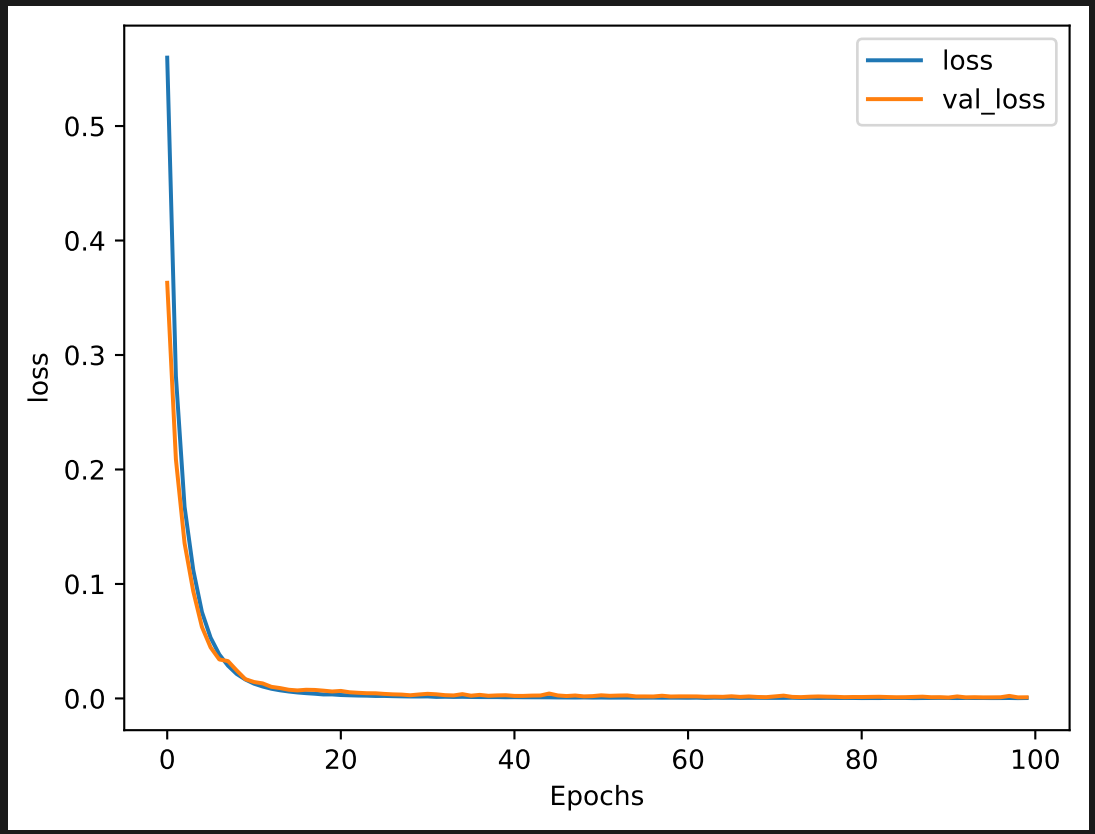


Figure: Loss curve for 100 epochs

Accuracy for linear subset + unprovable formulas

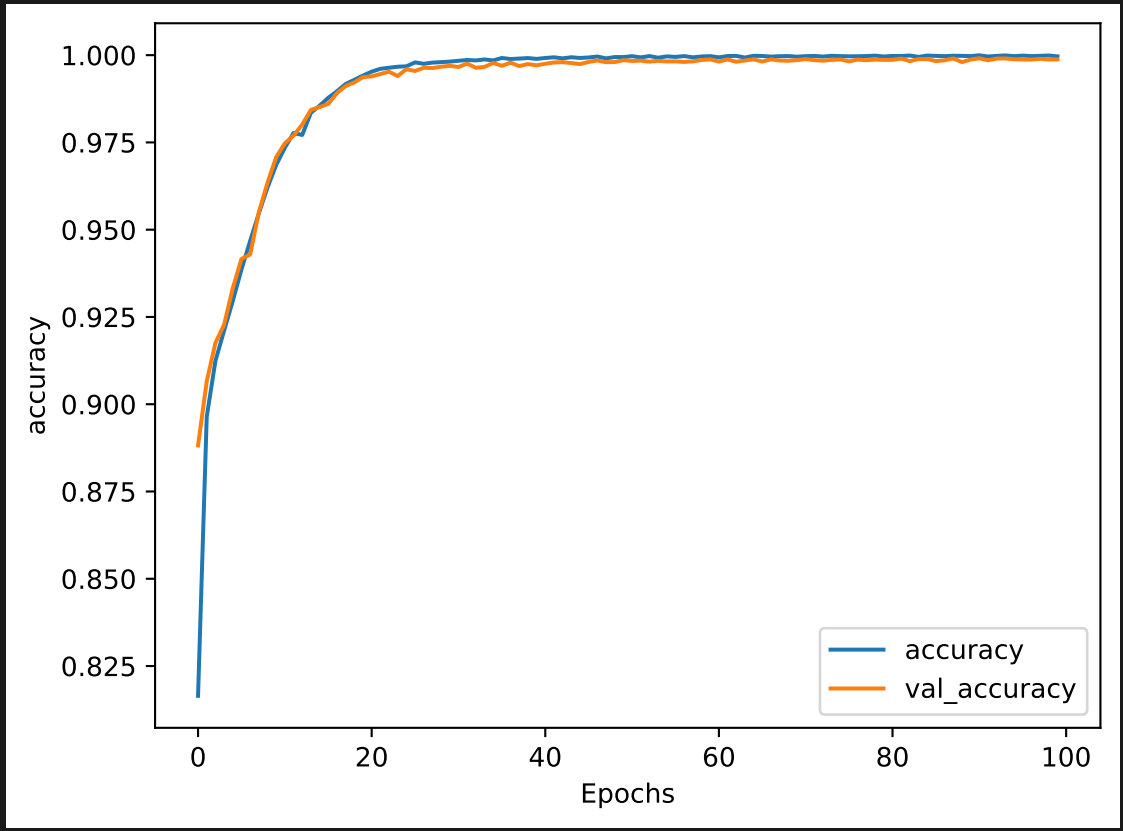


Figure: Accuracy curve for 100 epochs

Loss for linear subset + unprovable formulas

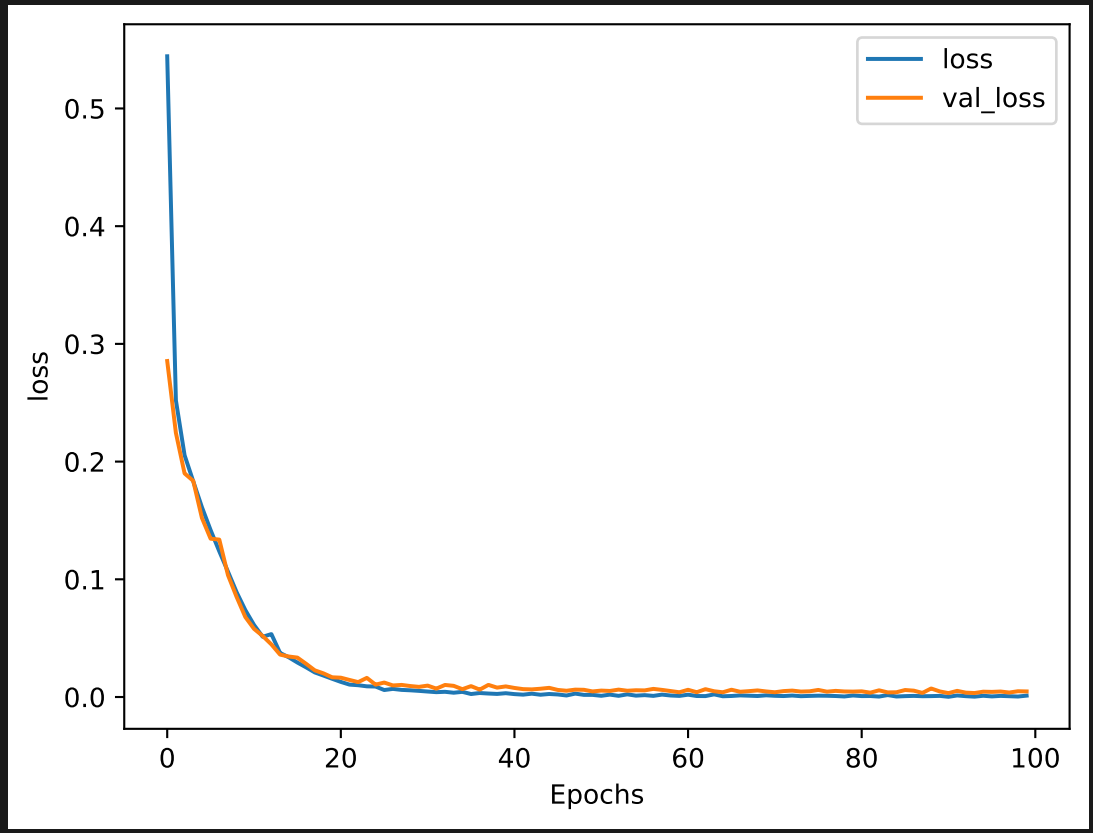


Figure: Loss curve for 100 epochs

Can we train Neural Network as Provers for a PSPACE-complete Logic?

Accuracy for implicational IPC

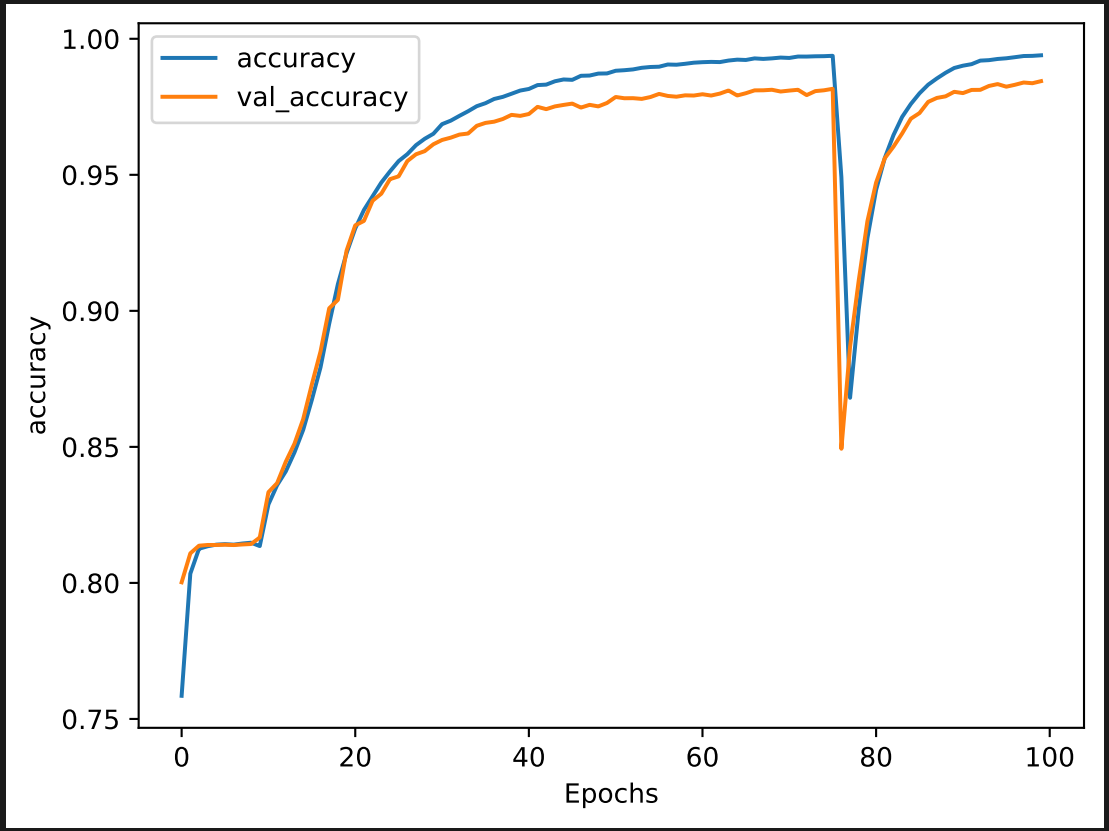


Figure: Accuracy curve for 100 epochs

Loss for implicational IPC

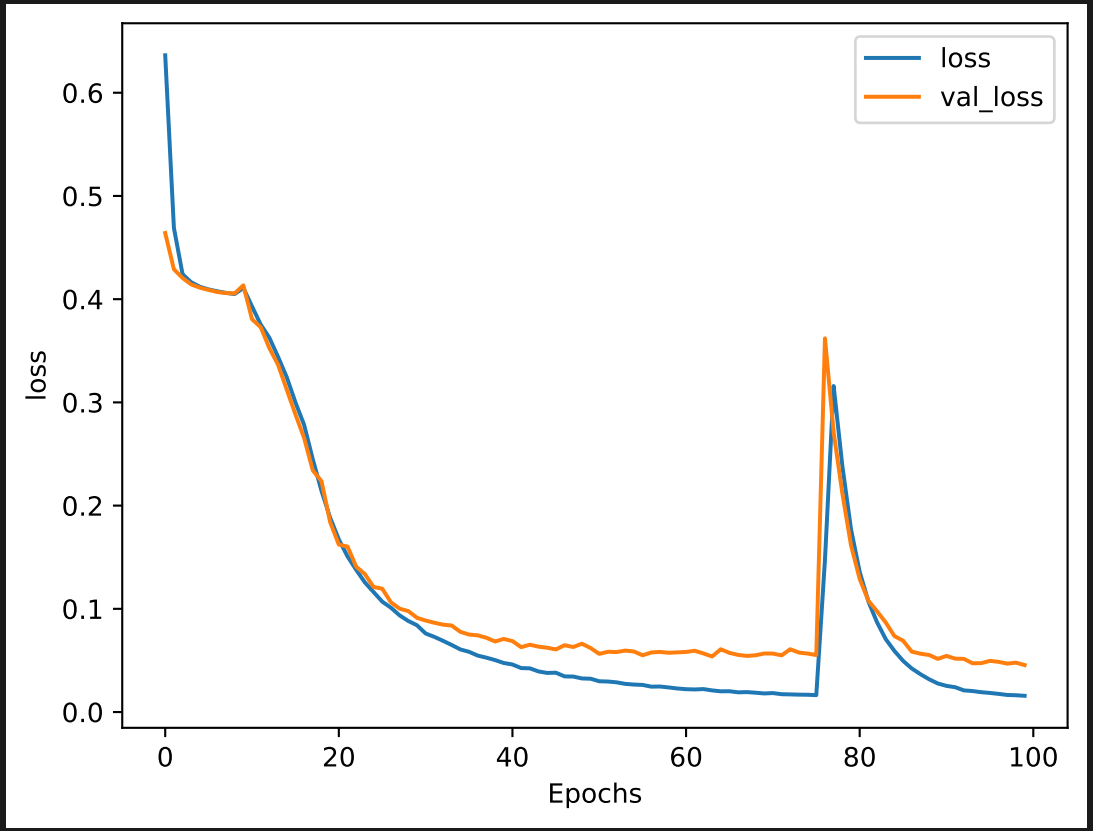


Figure: Loss curve for 100 epochs

Conclusions and Suggestions for Next Learning Steps

- we have overviewed some key strengths of Prolog with focus on problem solving while staying close to its pure, Horn Clause logic kernel
- a highlight: by starting from a two line meta-interpreter, we have captured the necessary step-by-step transformations that one needs to implement in a procedural language that mimics it
- suggested further explorations:
  - constraint solving extensions
  - tabling and its uses for elegant dynamic programming algorithms
  - probabilistic logic programming
- hot research topic: neuro-symbolic extensions to Prolog and synergies with ML in general and NLP in particular

Some Key Prolog Resources

- Marius Triska's excellent online Prolog book `https://www.metalevel.at/prolog`
- The Art of Prolog, a great classic Prolog book: `http://cliplab.org/~logalg/doc/The_Art_of_Prolog.pdf`
- SWI-Prolog's excellent eco-system of libraries and extensions `https://www.swi-prolog.org/`
- a solid, industrial strength commercial system: `https://sicstus.sics.se/`
- some nice, open source Prolog Systems
  - Ciao Prolog: `https://ciao-lang.org/`
  - XSB Prolog: `http://xsb.sourceforge.net/`
  - ECLIPSe Prolog + CP system: `https://eclipseclp.org/`
  - GNU Prolog: `http://www.gprolog.org/`
- and much more: `https://en.wikipedia.org/wiki/Prolog`

Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (1991). https://doi.org/10.7551/mitpress/7160.001.0001

Carlson, M., Mildner, P.: SICStus Prolog – The first 25 years. Theory and Practice of Logic Programming **12**, 35–66 (1 2012). https://doi.org/10.1017/S1471068411000482

Chen, W., Kifer, M., Warren, D.: HiLog: A first-order semantics for higher-order logic programming constructs. In: Lusk, E., Overbeek, R. (eds.) 1st North American Conf. Logic Programming. pp. 1090–1114. MIT Press, Cleveland, OH (1989)

Colmerauer, A.: Metamorphosis grammars. In: Bolc, L. (ed.) Natural Language Communication with Computers. Springer-Verlag (1978), previously in Technical Report, Groupe d'Intelligence Artificielle, Marseille, France, 1975

Colmerauer, A., Kanoui, H., Van Caneghem, M.: Etude et réalisation d'un système Prolog. Tech. rep., G.I.A. Université Aix-Marseille (May 1979)

Costa, V.S., Rocha, R., Damas, L.: The YAP Prolog system. Theory and Practice of Logic Programming **12**, 5–34 (1 2012). https://doi.org/10.1017/S1471068411000512

Dovier, A., Formisano, A., Gupta, G., Hermenegildo, M.V., Pontelli, E., Rocha, R.: Parallel logic programming: A sequel (2021), https://arxiv.org/abs/2111.11218

Hermenegildo, M.V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J.F., Puebla, G.: An overview of Ciao and its design philosophy. Theory and Practice of Logic Programming **12**, 219–252 (1 2012). https://doi.org/10.1017/S1471068411000457

Korner, P., Leuschel, M., Barbosa, J., Costa, V.S., Dahl, V., Hermenegildo, M.V., Morales, J.F., Wielemaker, J., Diaz, D., Abreu, S., Ciatto, G.: Fifty years of prolog and beyond (2022), https://arxiv.org/abs/2201.10816

Kowalski, R.: Logic for Problem Solving. Elsevier North-Holland, Amsterdam (1979)

Kowalski, R.: The early years of logic programming. CACM **31**(1), 38–43 (1988)

Kowalski, R., Emden, M.V.: The Semantics of Predicate Logic as a Programming Language. JACM **23**(4), 733–743 (Oct 1976). https://doi.org/10.1145/321250.321253

Swift, T., Warren, David, S.: XSB: Extending Prolog with Tabled Logic Programming. Theory and Practice of Logic Programming **12**, 157–187 (1 2012). https://doi.org/10.1017/S1471068411000500


Tarau, P.: The BinProlog Experience: Architecture and Implementation Choices for Continuation Passing Prolog and First-Class Logic Engines. Theory and Practice of Logic Programming **12**(1-2), 97–126 (2012). https://doi.org/10.1007/978-3-642-60085-2_2


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)


Tarau, P., Boyer, M.: Elementary Logic Programs. In: Deransart, P., Maluszyński, J. (eds.) Proceedings of Programming Language Implementation and Logic Programming. pp. 159–173. No. 456 in Lecture Notes in Computer Science, Springer, Berlin Heidelberg (Aug 1990)


Van Roy, P.: 1983-1993: The Wonder Years of Sequential Prolog Implementation. Journal of Logic Programming **19**(20), 385–441 (1994)


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


Zhou, N.F.: The language features and architecture of B-Prolog. Theory and Practice of Logic Programming **12**, 189–218 (1 2012). https://doi.org/10.1017/S1471068411000445

Questions?