Principles of Programming Languages (II)

Matteo Pradella

April 2016

Introduction to Prolog

1. Created around 1972 by Alain Colmerauer with Philippe Roussel, based on Robert Kowalski's **procedural interpretation of Horn clauses**.
2. decidable subset: **Datalog**, a query and rule language for deductive databases
3. its failure as a mainstream language traditionally due to the following:
4. Prolog usage in Fifth Generation Computer Systems project (FGCS)
5. FCGS was an initiative by Japan's Ministry of International Trade and Industry, begun in 1982, to create a "fifth generation computer"

Prolog in the real world

1. Prolog is not used much nowadays
2. still, for some activities it can be **very useful** - probably it was just a bad idea to use it as a system programming language. There are implementations usable for parts of the logic of complex applications e.g. written in Java
3. first of all: **rapid prototyping**. E.g. Prolog was used at Ericsson for implementing the first Erlang interpreter
4. other usages: some academic intrusion detection systems; event handling middlewares; some Nokia phones with the defunct MeeGoo; IBM's Watson is written in Java, C++, and Prolog.

Prolog and logic: Horn clauses

1. in general, a logic program has the following form:

$$\forall X_1 \ldots X_m \bigwedge_i (\phi^i \Leftarrow \theta^i_1 \wedge \theta^i_2 \wedge \ldots \wedge \theta^i_n)$$

2. note that $\phi \Leftarrow (\theta_1 \vee \theta_2) \wedge \theta_3$
3. is equivalent to $(\phi \Leftarrow \theta' \wedge \theta_3) \wedge (\theta' \Leftarrow \theta_1) \wedge (\theta' \Leftarrow \theta_2)$
4. we start from a **goal** φ, and follow the implications "backward", until we reach a solution or we **fail** (we could also loop. . .)

A first example

1. A **logic program**:
   1. $\forall X, L \; find(cons(X, L), X)$
   2. $\forall X, Y, L \; (find(cons(Y, L), X) \Leftarrow find(L, X))$
2. i.e. (being $(A \Leftarrow B) \iff (\neg A \Rightarrow \neg B)$):
   1. $\forall X, L \; find(cons(X, L), X)$
   2. $\forall X, Y, L \; (\neg find(cons(Y, L), X) \Rightarrow \neg find(L, X))$

A first example (cont.)

1. example **query**: $find([1, 2, 3], X)$?
2. from a logic point of view we are stating $false \Leftarrow find([1, 2, 3], X)$, i.e. we try to find a **counterexample**
3. it is the same as trying to satisfy $\forall X \; \neg find([1, 2, 3], X)$
4. first of all: Prolog is based on the **closed world assumption**: if something is not stated or deducible, it is false
5. we use the 1st clause, with $X = 1$, and $L = [2, 3]$:
6. $find(cons(1, [2, 3]), 1)$, so we have a contradiction with $X = 1$.

A first example (cont.)

1. another **query**: $\mathit{false} \Leftarrow \mathit{find}([1,2,3],2)$, i.e. $\neg \mathit{find}([1,2,3],2)$
2. 2nd clause: $\neg \mathit{find}(\mathit{cons}(1,[2,3]),2) \Rightarrow \neg \mathit{find}([2,3],2)$
3. *Modus Ponens*: $\neg \mathit{find}([2,3],2)$
4. but this is contradicts an instance of the 1st clause: $\mathit{find}(\mathit{cons}(2,[3]),2)$.
5. hence, $\neg \mathit{find}([1,2,3],2)$ does not hold.

A first example (cont.)

1. using the same technique, we obtain the following results:
2. $find(X, 5)$? gives $X = cons(5, V)$
3. $find([1, 2, X, 5], 5)$? gives $X = 5$
4. $find([1, 2, X, 5], Y)$? gives $Y = 1$
5. in practice, Prolog uses a lot of **extra-logical** commands and techniques, so it is often useful to consider its **procedural** semantics

The Prolog Language: Syntax

1. **Constants**: usual numbers, "strings"
2. but also **atoms**: test void = := 'this-name' 'hello world' []
3. **Variables**: must start with either a capital letter (Variable, VAR, X) or _ (stands for "don't care", usually called **anonymous** in Prolog jargon)
4. **compound terms** represent everything else, from procedure to data structures (**homoiconicity**)

Syntax for terms

1. standard syntax is e.g. f(g(3), X),
2. the name of the applied function (f or g in the term before), is also called **functor**
3. usually the **arity** of a functor is indicated like this: f/2, g/1
4. there are shortcuts for infix operators: 2+3/4 stands for +(2,/(3,4))

Lists

1. are terms like the others
2. the **empty list** is [], while . is **cons**
3. e.g. `.(1,.(2,.(3,[])))`
4. special syntax: the very common `[1,2,3]`
5. | is used to access **car** and **cdr**: the expression [X | L] represents a pair in a list, where X is **car**, while L is **cdr**

Strings vs atoms

1. **strings** are a bit uncommon: a string is a list; a character is represented by a number holding its ASCII code
    1. e.g. "hello" stands for [104, 101, 108, 108, 111]
    2. (but: there are utilities for managing unicode strings in modern implementations)
2. atoms are like **symbols** in Scheme
3. atoms are **not** strings: anAtom, 'this is another atom', "this is a string"

Programs

1. we saw the basic idea with logic formulae and Horn clauses
2. syntax: $\Leftarrow$ is written :-, $\wedge$ is comma, $\vee$ is semicolon
3. going back to our simple example:

   ```
   % a very first purely logical example
   find([X|_], X).
   find([Y|L], X) :- find(L, X).
   ```

4. we usually save it to a file (containing **facts**), then load it in the Prolog system (with [ex].)
5. at this point, we can perform **queries**, e.g. `find([1,2,3], 2).`

Our example, rewritten

1. this is an equivalent variant of the program (with ;)
   `find([Y|L], X) :- Y = X ; find(L, X).`
2. Prolog reads the clauses top to bottom, from left to right
3. the **head** of the clause is matched with the **procedure call**, by using an algorithm called **unification**
4. clauses are also called **sentences**, having a head and a **body**

Procedural semantics

1. To execute a goal, the system searches for the **first clause** whose **head** matches with the **goal**
   1. matching is performed through **unification** [Robinson 1965] (see next)
   2. the matching clause is **activated** by executing each of the **goals in its body**, from left to right.
2. If the system **fails** to find a match for a goal, it **backtracks**, i.e. it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause
   1. next it tries to find a **subsequent clause** which also matches the goal.

Unification (=)

1. An uninstantiated **variable** can be unified with an **atom**, a **term**, or another uninstantiated **variable** (it becomes an *alias*).
    1. e.g. X = f(3), X = bob, X = Y
2. Two **atoms** can only be unified if they are **identical**.
    1. e.g. 3 = 3, bob = bob, 3 $\neq$ bob
3. A **term** can be unified with another term if the **top function symbols** and arities of the terms are identical, and if all the **parameters** can be unified (recursion).
    1. e.g. f(g(3),Y) = f(Z,bob), 3+X $\neq$ Z-Y, [X,2|Y] = [1,2,3,4]

Occur check

1. unification does not have an **occur check**, i.e. when unifying a variable against a term the system does not check whether the variable occurs in the term: e.g. `X = f(X)`
2. when the variable occurs in the term, unification should fail, but in Prolog the unification succeeds, producing a **circular term**.
3. The absence of the occur check is not a bug or design oversight, but a **design decision**:
4. unification **against a variable** should be thought of as the **basic operation** of Prolog, but this can take constant time only without occur check.

An example

1. as we saw before, there is no clear role of **input** and **output** in a procedure call
2. consider the following example:

```
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
```

3. in this case `concatenate` is usually called with the first two parameters for input, and the last one for the result
4. e.g.

```
?- concatenate([1,2,3],[4,5,6],X).
X = [1, 2, 3, 4, 5, 6].
```

5. Note 1: it is common to use the last parameter for the result
6. Note 2: `concatenate` is called **append** in the library

Parameters and result

1. we may also use it in this way:

```
?- concatenate(X,[2,Y],[1,1,1,2,3]).
X = [1, 1, 1],
Y = 3 .
```

2. or this:

```
?- concatenate(X,[2,Y],[1,1,2,3]).
X = [1, 1],
Y = 3 ;      % are there other solutions?
false.       % no
```

Parameters and result (cont.)

1. or also in this way:

```
?- concatenate(X,Y,[1,1,1,2,3]).
X = [1, 1, 1, 2, 3],
Y = [] ;
X = [1, 1, 1, 2],
Y = [ 3 ] ;
X = [1, 1, 1],
Y = [2, 3] ;
X = [1, 1],
Y = [1, 2, 3] ;
X = [ 1 ],
Y = [1, 1, 2, 3] ;
X = [],
Y = [1, 1, 1, 2, 3].
```

An exercise: non-deterministic Push-down Automata (NPDA)

1. it is very easy to simulate a NPDA with Prolog
2. main idea: define a predicate *config* which represents the **configuration** of the automaton at a given step of the computation
3. **config** has three parameters
    1. the first one is the current **state**
    2. the second one is the current **stack**
    3. the third one is the part of the **input string** that we still have to read
4. it is natural to use **lists** for representing the stack and the input string
5. e.g. we start with `config(q0, [z0], [a,a,a,b,b,b])` if we are considering an automaton for $a^n b^n$

NPDA: solution

1. we check acceptance, when the input string is over:

```
config(State, _, []) :- final(State).
```

2. standard move:

```
config(State, [Top|Rest], [C|String]) :-
  delta(State, C, Top, NewState, Push),
  append(Push, Rest, NewStack),
  config(NewState, NewStack, String).
```

3. ϵ-moves are just a variant:

```
config(State, [Top|Rest], String) :-
  delta(State, epsilon, Top, NewState, Push),
  append(Push, Rest, NewStack),
  config(NewState, NewStack, String).
```

4. `run(Input) :- initial(Q), config(Q, [z0], Input).`

using NPDA

1. we want to try with a nondeterministic language, so we take $L = \{a^n b^n\} \cup \{a^n b^{2n}\}$

```
delta(q0, a, z0, q1, [a, z0]).  % a^n b^2n
delta(q1, a, a,  q1, [a, a]).
delta(q1, b, a,  q2, [a]).
delta(q2, b, a,  q3, []).
delta(q3, b, a,  q2, [a]).
delta(q3, epsilon, z0, q4, []).

delta(q0,  a, z0, q11, [a, z0]). % a^n b^n
delta(q11, a, a,  q11, [a, a]).
delta(q11, b, a,  q21, []).
delta(q21, b, a,  q21, []).
delta(q21, epsilon, z0, q4, []).

initial(q0).     final(q4).
```

using NPDA (cont.)

1. example queries:

```
?- run([a,a,a,b,b,b]).
true .

?- run([a,a,b,b,b,b]).
true .

?- run([a,a,b,b,b]).
false.

?- run([a,a,b|X]).
X = [b, b, b, epsilon] ;
X = [b, b, b] ;
X = [b, epsilon] ;
X = [b] ;
false.
```

Numerical expressions

1. Prolog supports numerical expressions, but they are treated differently form "mainstream" languages, as they are not usually evaluated
2. A bad example, exponentiation:

```
pow(_,0,1).
pow(X,1,X).
pow(X,N,R) :- pow(X,N-1,R1), R = X*R1.     % two errors!
```

3. this **loops** for $N > 1$, because it calls $pow(X, N-1, ...)$, $pow(X, N-1-1, ...)$, ...

Numerical expressions (cont.)

1. There are a couple of useful predicates for managing numbers: $=:=$ and *is*, that **evaluate** numeric expressions
2. e.g. the query `X is 2/3` returns `X` $= 0.6666666666666666$
3. the query $3 + 1 =:= 6 - 2$ returns true

Numerical expressions (cont.)

1. Here is the fixed version:

```
pow(_,0,1).
pow(X,1,X).
pow(X,N,R) :- N1 is N-1, pow(X,N1,R1), R is X*R1.
```

2. A performance issue: if we try e.g. `pow(2,16,8)`, this yields "ERROR: Out of local stack"

Extra-logical constructs: **cut**

1. efficient logic programming is not trivial: it is often **hard** to understand the computing steps taken by the system
2. for this reason, there is a particular **extra-logical** construct that the programmer can use to improve the search for the goal
3. this construct is called **cut**, written !
4. its name comes from the fact that it can be used to **prune** branches in the depth-first search performed by the system
5. hence, we are discarding all the backtrack information that was stored during the run

Back to our example

1. Here is the version improved with cuts

```
pow(_,0,1) :- !.
pow(X,1,X) :- !.
pow(X,N,R) :- N1 is N-1, pow(X,N1,R1), R is X*R1.
```

Another example: *Quicksort*

1. first, a way to **partition** lists:

```
part([X|L],Y,[X|L1],L2) :- X =< Y, !, part(L,Y,L1,L2).
part([X|L],Y,L1,[X|L2]) :- X > Y,  !, part(L,Y,L1,L2).
part([],_,[],[]).
```

2. and now quicksort:

```
qsort0([],[]).
qsort0([H|T],Sorted) :- part(T,H,L1,L2), !,
                        qsort0(L1,Sorted1),
                        qsort0(L2,Sorted2),
                        append(Sorted1,[H|Sorted2],Sorted).
```

Higher-order and meta predicates

1. For practical reasons, there are a number of "meta" and "higher order" predicates (we already saw **!**)
2. A useful one is **call**, which is used to perform a **query** in a program
3. `call(G1, A1, A2, ...)` adds arguments $A1$, $A2$, ... to goal $G1$ and performs the resulting query
4. e.g.

```
?- call(plus(1), 2, X).
X = 3.   % i.e. plus(1,2,X).
```

Higher-order and meta predicates: an example

1. here is an implementation of `map` (usually called `maplist` in the library)

```
map(_, [], []).
map(C, [X|Xs], [Y|Ys]) :- call(C, X, Y), map(C, Xs, Ys).
```

2. e.g. if we define `test(N,R):- R is N*N.`

```
?- map(test,[1,2,3,4],X).
X = [1, 4, 9, 16].
```

Call and Negation

1. `fail` is a goal that *always fails*
2. using `call`, !, and `fail` we can define `negation`:

   ```
   not(X) :- call(X), !, fail.
   not(X).
   ```
3. e.g.

   ```
   ?- not(member(6,[1,2,3])).
   true.
   ```
4. Note: `not` is already defined in the language: it is the prefix operator \+
   1. like in: `\+ member(6,[1,2,3]).`

Destructuring terms

1. From the unification algorithm, we know that we cannot match $X(Y) = f(3)$, but sometimes we need something analogous
2. we can do it by using the predicate =.. which is used to **decompose** a term
3. e.g. the query f(2,g(4)) =.. X binds X to the list [f, 2, g(4)]
4. so in the previous case we can do f(3) =.. [X,Y]

Example: a symbolic differentiator

1. basic rules

```
d(U+V,X,DU+DV) :- !, d(U,X,DU), d(V,X,DV).
d(U-V,X,DU-DV) :- !, d(U,X,DU), d(V,X,DV).
d(U*V,X,DU*V+U*DV) :- !, d(U,X,DU), d(V,X,DV).
d(U^N,X,N*U^N1*DU) :- !, integer(N), N1 is N-1, d(U,X,DU).
d(-U,X,-DU) :- !, d(U,X,DU).
```

2. terminating rules

```
d(X,X,1) :- !.
d(C,_,0) :- atomic(C), !.
```

3. **atomic** holds with atoms and numbers

a differentiator (cont.)

1. terminating rules (cont.)

```
d(sin(X),X,cos(X)) :- !.
d(cos(X),X,-sin(X)) :- !.
d(exp(X),X,exp(X)) :- !.
d(log(X),X,1/X) :- !.
```

2. chain rule

```
d(F_G,X,DF*DG) :- F_G=..[_,G], !, d(F_G,G,DF), d(G,X,DG).
```

3. note that: $1+2/3 =..$ X binds X to $[+,1,2/3]$

a differentiator (cont.)

1. now, let us try it:

```
?- d(2*sin(cos(x+cos(x))), x, V).
V = 0*sin(cos(x+cos(x)))+2* (cos(cos(x+cos(x)))*
         (-sin(x+cos(x))* (1+ -sin(x)))).
```

Legal stuff