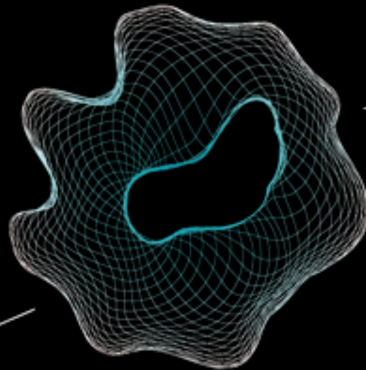
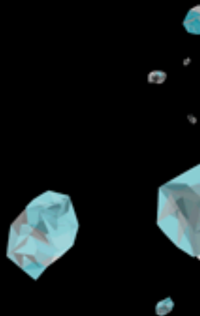
UNIVERSITY OF TWENTE.

Module 8: Programming Paradigms
Logic Programming
part 3: Advanced Prolog and Modern Developments

Sebastiaan Joosten (some slides: Jaco van de Pol)

22 May 2019





- Meta-programming in Prolog
- Constraint Logic Programming
- Modern use of Prolog

Special Predicates for Program Manipulation

call/1 : defined like **call(P) :- P.** **(mixing data / code)**
assert/1 : adds a clause **(self-modifying code!)**
asserta/assertz : adds a clause at the beginning / end
retract/1 : deletes a clause **(self-modifying code!).**

Other built-in predicates, for “self-inspection”:

- **clause(H,B):** find all clauses matching H.

- **var(X):** check if X is (currently) a variable.
- **atom(X):** Check if X is (currently) an atom.
- **T =.. [F,Args]:** Decompose term T into fun F and Args

Writing a Prolog Interpreter in Prolog

- The simplest way:

```
solve(P) :- call(P).
```

- A bit more fine-grained:

```
solve( true ) :- !.
solve( (P, Q) ) :- !,solve(P), solve(Q).
solve( P ) :- clause(P,Body), solve(Body).
```

- **Of course, this only works for Pure Prolog**
- **Why would you do it?**
  - **Modifying search order, bounded depth, etc.**
  - **Print traces, proof search trees, etc.**

Writing a Prolog Interpreter in Prolog (2)

- A simple tracing Prolog meta-interpreter

```
solveTrace(P) :- solve(P,0).

solve(true, N) :- !.
solve( (P,Q), N) :- !,solve(P, N), solve(Q, N).
solve( P, N) :-
 trace(P,N), clause(P,Body), N1 is N+1, solve(Body,N1).

trace(P,N) :- tab(N*2), print(P), nl.
```

- Try it out!

```
?- solveTrace( (member(X,[1,2,3]), member(X,[3,4,5])) ).
```

Fibonacci

```
fib(1,1).
fib(2,1).
fib(N,F) :- N > 2
          , N1 is N-1
          , fib(N1,F1)
          , N2 is N-2
          , fib(N2,F2)
          , F is F1+F2.
```

```
:-dynamic fib/2.

fib(1,1).
fib(2,1).
fib(N,F) :- N > 2
    , N1 is N-1
    , fib(N1,F1)
    , N2 is N-2
    , fib(N2,F2)
    , F is F1+F2
    , asserta((fib(N,F):-!)).
```

What happens if we run (after starting Prolog):
?- fib(N,5)

Overview

- Meta-programming in Prolog
- Constraint Logic Programming
- Modern use of Prolog

CLP: Constraint Logic Programming

- Prolog can not solve equations like:
  - X+5 = 2*Y + 7
  - X+5 is 2*Y + 7
- SWI Prolog extends Prolog with constraint programming

```
?- X+5 #= 2*Y + 7.
X#=2*Y+2.

?- X+5 #= 2*Y + 7, X=99.
false.

?- X+5 #= 2*Y + 7, X=100.
X = 100, Y = 49.
```

CLP(fd) = CLP over Finite Domains

See http://www.pathwayslms.com/swipltuts/clpfd/clpfd.html

```
:- use_module(library(clpfd)).

test1(X, Y) :-
    X in 0..10,
    Y in 4..8,
    X #> Y.

?- test1(X,Y).
X in 5..10,
Y#=<X+ -1,
Y in 4..8.
```

CLP(fd) what does it do?

- A term like **X in 0..10** adds a constraint.
  - There is only one way to add a constraint:
    No branching happens here.
- Next, prolog simplifies the constraints
- Prolog then determines if the set is still solvable
  - If not, it backtracks.
- Whenever the unifier matches a constrained variable (turns it into an integer), the constraints are checked (to see if the set is solvable)
- Alternatively, you can write **label(Vars)** to force variables in the list **Vars** to be instantiated. At this point, prolog can branch!

CLP(fd)

```
p(X) :- X #< 5, q(X,Y).
q(7,Y) :- expensive_computation(Y).
q(3,Y) :- something_else(Y).
```

- Note that we cannot start with **X < 5** as **X** is still a variable. Prolog will complain: insufficiently instantiated. Without **#<** we would have to write:

```
p(X) :- q(X,Y), X < 5.
q(7,Y) :- expensive_computation(Y).
q(3,Y) :- something_else(Y).
```

- However, this would mean we would first call **expensive_computation/1**

What is fast/slow in CLP(fd)?

- Simplification can deal with linear equations really well: for **#<** and **#=** you are allowed to use variables, **+** , n* and numbers (constants).
- When using *, try not to multiply two variables (except if one is instantiated).

- Non-linear arithmetic is slower in CLP(fd), but often still faster than forcing backtracking: **#\/** for choice, **#min**, **#max**, **#\=** , **all_different** etc. are all potentially slower.

- Order of adding constraints matters less in CLP(fd).
- Main advantage of CLP(fd): avoids branching!

Using CLP in general

- General form of a CLP program:
  Constrain – Generate – Test
  1. Define all the FD-constraints (simplified automatically)
  2. Generate the solutions (e.g. with label/1)
  3. Test for further conditions (arbitrary Prolog code)

SEND + MORE = MONEY

```
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
        Vars = [S,E,N,D,M,O,R,Y],
        Vars ins 0..9,
        all_different(Vars),
        S*1000 + E*100 + N*10 + D +
        M*1000 + O*100 + R*10 + E #=
        M*10000 + O*1000 + N*100 + E*10 + Y,
        M #\= 0, S #\= 0,
        label(Vars).

?- puzzle(X).
X = ([9, 5, 6, 7]+[1, 0, 8, 5]=[1, 0, 6, 5, 2]) ;
```

Fibonacci, backwards!

```
:- use_module(library(clpfd)).

fib(1, 1).
fib(2, 1).
fib(N, F) :- N #> 1
          , N1 #= N – 1
          , N2 #= N – 2
          , F #= F1 + F2
          , F1 #> 0, F2 #> 0
          , fib(N1, F1), fib(N2, F2).

?- fib(N, 8).
N = 6 ;
false.
```

Fibonacci, fast and backwards!

```
:- use_module(library(clpfd)).

n_fib(1, 1, 0).
n_fib(2, 1, 1).
n_fib(N, F, F1) :- N #> 1
    , N1 #= N - 1
    , F #= F1 + F2
    , F1 #> 0, F2 #> 0
    , n_fib(N1, F1, F2).

fib(N, F) :- n_fib(N, F, _).
```

https://m00nlight.github.io/constraint%20logic%20programming/2016/01/01/using-clpfd-to-solve-factorial-and-fibonacci-problems-reversely

Other applications ... abundant

- Solving Sodokus efficiently
  - And many other puzzles
- Planning, scheduling, optimization, allocation problems
  - E.g.: time tables
- Test case generation for automated testing
  - Compute inputs that bring you to a line of code (coverage)

Overview

- Meta-programming in Prolog
- Constraint Logic Programming
- Modern use of Prolog

Prolog: History and Use

- See https://en.wikipedia.org/wiki/Prolog
  http://www.drdobbs.com/parallel/the-practical-application-of-prolog/184405220
- History:
  - Based on work of *Robert Kowalski* (Edinburgh University, 1970)
  - Invented by *Alain Colmerauer* (Marseille, 1972)
  - Used for AI: expert systems, natural language processing, etc.

- Used as the first language prototype Erlang for Ericsson
- Used at Boeing:
  - Generalized Query System for Assembly Instructions
- Used in the IBM Watson project (recall Jeopardy?)
  - DeepQA software, written in Java, C++, and Prolog
  - *"We required a language in which we could conveniently express pattern matching rules over parse trees for natural languages"*