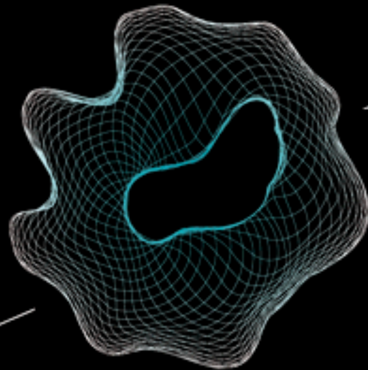
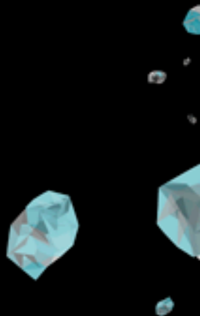
UNIVERSITY OF TWENTE.

Module 8: Programming Paradigms
Logic Programming
part 2: arithmetic & control, unification & resolution

Sebastiaan Joosten (slides in part: Jaco van de Pol)

May 16 2019





The paradigm of logic programming, so far

- Allows a very flexible format, no declarations needed
  - Trivial syntax
- No pre-determined computation order
  - Freely mix “forward” and “backward” computations.
  - That is: Prolog works with **relations**, not with **functions**
- Backtracking and unificiation provide
  a powerful built-in problem solver:
  - Very convenient to solve puzzles with few lines of code
- Beware: Prolog uses a ***closed world assumption***
- Beware: Order of rules is relevant for termination

Contents

- Theoretical background of Pure Prolog
  - Unification (pattern matching)
  - Resolution (search tree, backtracking)
- Extensions of Prolog
  - Control and Negation
  - Program manipulation (self-modifying code)
  - Arithmetic operations
- More on lists
  - Practical exercise: Tree-sort

Syntax: Don’t confuse Data and Code

- Terms (structured data elements)
  - Variables: X, Y, _, _G99971
  - Atoms: a, b, c, pietje, 12345
  - Functions: f(a,b,X), f(g(X),h(Y,Y)), X*Y+Z, [X,Y|Z], state(atdoor,onbox,atwindow,has)
- Literals (propositions that can be true/false):
  - Predicates over terms: p, q(X), r(f(X),g(X,Y)), member(X,[X|L])
- Clauses (rules, facts, queries):
  - h :- b1, …, bm   where h and b1,...,bm are literals

- Why does data and code look alike so much?
  - On purpose: Prolog programs can manipulate themselves
  - It is trivial to write a Prolog meta-interpreter in Prolog

Clauses: facts, rules, queries

- A ***clause*** is of the form **h :- b1, …, bn.**
  - Here h, b1, …, bn are called ***literals***.
  - Facts, rules and queries are all clauses:
  - *Read 1:* h follows from b1 … bn
  - *Read 2:* b1 … bn implies h

- A ***rule*** has a non-empty head and a body:
  - **h** is the ***head*** of the rule
  - **b1, …, bn** is the ***body*** of the rule
- A ***fact*** is a clause with an empty body:
  - **h.**
- A ***query*** is a clause with an empty head:
  - **:- b1, ... , bn.** Sometimes also written as: **?- b1, ..., bn**.

Contents

- Theoretical background of Pure Prolog
  - Unification (pattern matching)
  - Resolution (search tree, backtracking)
- Extensions of Prolog
  - Control and Negation
  - Program manipulation (self-modifying code)
  - Arithmetic operations
- More on lists
  - Practical exercise: Tree-sort

Substitution and Unification (definitions)

- A ***substitution*** is a mapping from variables to terms
  s = [X/f(a), Y/f(X)]
- We write term[subst] for the result
  of "simultaneous substitution":
  f(g(X),Y) **[X/f(Y), Y/f(a)]** results in **f(g(f(Y)), f(a))**
- Term **t1** is ***more general than*** term **t2**,
  if one can instantiate t1 to t2. That is:
  for some substitution s, we have **t1[s] = t2**
- Substitution s1 is ***more general than*** s2,
  if for some s3, **[s1] o [s3] = [s2].**
- A ***unifier*** of terms **t1** and **t2** is a substitution **s**,
  such that **t1[s] = t2[s]**

- ***Unification*** computes a ***most general unifier (mgu).***

Unification (examples)

- The MGU is unique modulo renaming:
  - **X=Y**: **[X/Y]** or **[Y/X]**
- Other examples:
  - **X=f(Y): [X/f(Y)]**
  - **[X,Y|L] = [1,2,3,4]: [X/1, Y/2, L/[3,4]]**
  - **f(X)=g(Y): no mgu**

- Computing the MGU can be rather complicated:
  - **f(g(X),Y,Y) = f(A,g(B),A): [A/g(B), Y/g(B), X/B]**

- The MGU can become exponentially long:
  - **(((a*Z)*Y)*X)*W = W*(X*(Y*(Z*a)))**
  - **[Z/a, Y/a*a, X/(a*a)*(a*a), W / ((a*a)*(a*a))*((a*a)*(a*a))]**

Example run of unification algorithm (1)

- **Start point:** [f(g(X),Y,Y) = f(A,g(B),A)]
- **Decompose:** [g(X)=A, Y=g(B), Y=A]
- **Swap:** [A=g(X), Y=g(B), Y=A]
- **Eliminate:** [A=g(X), Y=g(B), Y=g(X)]
- **Eliminate:** [A=g(X), Y=g(B), g(B)=g(X)]
- **Decompose:** [A=g(X), Y=g(B), B=X]
- **Eliminate:** [A=g(X), Y=g(X), B=X]
- **Finished!**

Unification Algorithm (linear: Martelli-Montanari)

A good page on unification:

https://en.wikipedia.org/wiki/Unification_(computer_science)

The algorithms starts with a set of equations [s1=t1 , … , sn=tn] and results in either no, or a substitution [x1 = r1 , ... , xm = rm].

The algorithm repeats the following steps, in arbitrary order, until termination:

- **Delete**: equation {t=t} can be deleted
- **Decompose**: replace {f(s1..sn)=f(t1..tn)} by {s1=t1, … , sn=tn}
- **Conflict**: an equation {f(s1..sn)=g(t1..tm)} leads to **no,** if $f \neq g$ or $m \neq n$
- **Swap**: replace {f(t1…tn)=X} by {X=f(s1…sn)}
- **Occurs Check**: equation {X=f(t1..tn)} leads to **no,** if $X \in vars(t1..tn)$
- **Eliminate**: *otherwise*, apply [X=f(t1..tn)] to ***all other equations***

Example run of unification algorithm (2)

- **Starting point**: [f(g(X),g(Y),Y) = f(A,A,A)]
- **Decompose**: [g(X)=A, g(Y)=A, Y=A]
- **Swap**: [A=g(X), g(Y)=A, Y=A]
- **Eliminate**: [A=g(X), g(Y)=g(X), Y=g(X)]
- **Decompose**: [A=g(X), Y=X, Y=g(X)]
- **Eliminate**: [A=g(X), Y=X, X=g(X)]
- **Occurs Check**: no mgu!

Contents

- Theoretical background of Pure Prolog
  - Unification (pattern matching)
  - Resolution (search tree, backtracking)
- Extensions of Prolog
  - Control and Negation
  - Program manipulation (self-modifying code)
  - Arithmetic operations
- More on lists
  - Slow-sort, Quick-sort
  - Practical exercise: Tree-sort

Working of Prolog

- Computation
  = Unification + Resolution steps + Depth-First Search

A single resolution step:

- Goal (query): ***?- p(f(X),Y), q(X), r(Y).***
- Program Rule: ***p(A,f(B)) :- s(A,g(B)), t.***
- Unification: **[A/f(X), Y/f(B)]**
- New subgoal: ***?- s(f(X),g(B)), t, q(X), r(f(B)).***

Prolog search order:

- First subgoal is solved first (***left to right***)
- First rule in the program is tried first (***top to bottom***)
- Backtracking happens on a failure (dead end)
- ***Depth-First Search*** traversal through computation tree

Prolog resolution procedure (simplified)

Prolog will repeatedly execute the following steps:

1. Take the first subgoal of the query
2. Find the first/next matching rule with this subgoal
3. Rename the variables in the rule “apart from” the goal
4. Compute a substitution that unifies the subgoal and the rule head
5. Apply the (most general) substitution to the goal ***and*** to the rule
6. Replace the first subgoal with the body of the rule

Together, this is called a single *resolution step*.

When Prolog gets stuck, it automatically backtracks to the next matching rule, undoing intermediate subsitutions

Possible outcomes of “running” a query?

1. When the goal is empty, the query has succeeded.
   - The substitutions together provide a concrete answer
2. When the goal is not-empty, and no rule is applicable, the query has failed.
3. The computation may go on forever.
   In that case the program is “wrong”.
   - Either the search space contains loops
   - Or the search space is infinite
4. The final possibility in practice is that the search space is so large that your computer gets exhausted.

The evaluation process by example (1)

Recall part of the program on the royal family:

```
child(X,Y) :- mother(Y,X).
child(X,Y) :- father(Y,X).
sister(X,Y) :- child(X,P), child(Y,P), not(X=Y).
```

Let us start with the query: **sister(beatrix,X).**

The (our) interpretation is: ***Which X is beatrix a sister of?***

How does Prolog work this out?

Resolution step (1)

1. The first subgoal is the whole query **?- sister(beatrix,X).**
2. The only matching clause that can be used to prove this is:
   sister(X,Y) :- child(X,P), child(Y,P), not(X=Y).
3. "Standardizing the variables apart" means that variables in the clause are renamed, so they don't occur in the query:
   sister(Z,Y) :- child(Z,P), child(Y,P), not(Z=Y).
4. *Unifying* the goal **sister(beatrix,X)** and the head of the rule **sister(Z,Y)** will yield the substitution: **[Z/beatrix, X/Y]**
5. Applying the subsitution leads to: **sister(beatrix,Y).**
6. Replacing the subgoal by the body of the new rule yields:
   ?- child(beatrix,P), child(Y,P), not(beatrix=Y).

This is our new goal after the first resolution step

Resolution step (2)

1. The current subgoal is:
   ?- child(beatrix,P), child(Y,P), not(beatrix=Y).
2. There are two matching clauses:
   child(X,Y) :- mother(Y,X).
   child(X,Y) :- father(Y,X).
3. Renaming the *first one* apart yields:
   child(X,Z) :- mother(Z,X). **(*) here we could choose father**
4. Unification with the first subgoal yields**:** **[X/beatrix, Z/P]**
5. Applying this unifier yields: **child(beatrix,P)**
6. Replacing the first subgoal with the body of the rule, yields:
   ?- mother(P,beatrix), child(Y,P), not(beatrix=Y)

This is our new goal after the second resolution step

Further resolution steps, and one backtrack

- Resolution proceeds as follows:

?- mother(P,beatrix), child(Y,P), not(beatrix=Y)

Now we use the fact: **mother(juliana,beatrix),** substituting **[P/juliana]**

?- child(Y,juliana), not(beatrix=Y).

?- mother(juliana,Y), not(beatrix=Y). **(**) we could also choose father**

Now use the "fact": **mother(juliana,beatrix).** **(***) we could choose margriet**

?- not(beatrix=beatrix). (after substituting [Y/beatrix])

Oops, this fails! Indeed, beatrix is not her own sister

- We have to undo some steps, to the previous backtrack point (***)
- This will also undo the substitution [Y/beatrix]

:- not(beatrix=margriet). **(**after substituting **[Y/margriet])**

This succeeds!!

Reconstructing the answer after “success”

- We have now explored the “leftmost branch of the resolution tree” for the query **?- sister(beatrix,X)**
- We found that this branch ends with the empty query, which means that the query has been proved. For which value of X?
- Look at the substitutions for X along the branch from top to bottom: first we had **X/Y** and later **Y/margriet**.
- We conclude: **X=margriet**. So, *beatrix is a sister of margriet* (in our interpretation)

- You may now instruct Prolog to continue the search by typing **;**
- So, backtracking continues and other solutions are found.

- **Excercise**: How many solutions are reported, which, why?

Termination

A search-tree need not be finite (containing loops).

This means that a search may not terminate.

Order of clauses in a program matter.

For instance, suppose that clause C1 succeeds and clause C2 leads to infinite computation:

c1.	**c1.**
c2 :- c2.	**c2 :- c2.**
p :- c1.	**p :- c2.**
p :- c2.	**p :- c1.**

If C1 comes before C2, the solution to **?- p** will be found,

If C2 comes before C1, the solution to **?- p** will not be found.

The order of predicates in clauses and in queries also matters.

Resolution step (once more, if unclear)

- Let us have a goal:
  ?- g1, … , gm.
- We want to resolve the first subgoal with a rule
  h :- b1, ..., bn.
- First rename variables away:
  h’ :- b1’, ..., bn’.
- Compute the most general unifier
  s = mgu(g1,h’).
- If successful, continue with the new goal:
  ?- b1’[s], ..., bn’[s], g2[s],...,gm[s]

Last illustration of backtrack process

Program: member(X,[X|L]).
member(Y,[Z|K]) :- member(Y,K).

?- member(X,[a,b]), member(X,[b,c]).

1. X=a, L=[b]
   1. ?- member(a,[b,c]). Y=a, Z=b, K=[c]
   2. ?- member(a,[c]). Y=a, Z=c, K=[]
   3. ?- member(a,[]). ???
   4. Failure => backtrack
2. Y=X, Z=a, K=[b]
   1. :- member(X,[b]),member(X,[b,c]). X=b, L=[]
   2. :- member(b,[b,c]). X=b, L=[c]
   3. [] (empty clause)
   4. Success => compute answer

Contents

- Theoretical background of Pure Prolog
  - Unification (pattern matching)
  - Resolution (search tree, backtracking)
- Extensions of Prolog
  - Control and Negation
  - Program manipulation (self-modifying code)
  - Arithmetic operations
- More on lists
  - Slow-sort, Quick-sort
  - Practical exercise: Tree-sort

The operator not

- The operator **not** has a query Q as an argument.
- To evaluate **not(Q)**, first **Q** itself is evaluated.
  - If **Q** succeeds, **not(Q)** fails, ***bindings will be lost***.
  - If **Q** fails, **not(Q)** succeeds ***without any bindings***.

- Remember the clause
  sister(X,Y) :- child(X,P), female(X), child(Y,P), not(X=Y).
- Suppose it is changed into
  sister(X,Y) :- not(X=Y), child(X,P), female(X), child(Y,P).
- Now a query like **sister(beatrix,X)** fails! (***why?***)

- **Lesson:** When using **not(Q)**, all variables of **Q** must be bound at the time that **not(Q)** is going to be evaluated.

Typical power of Prolog so far

- Beware: Order of rules relevant for termination
- Beware: Prolog uses a ***closed world assumption***
- Beware: Prolog uses ***negation-as-failure***.
  - This is really ***different*** from logical negation.

Some useful predicates, also known as “meta-programming”

fail : Always fails.

! : Always true, but it prevents backtracking
commit-operator, also known as “cut”
since it prunes the search tree)

Actually, negation by failure is defined as follows:

```
not(P):- P, !, fail.
not(P).
```

Recall another “control structure”:

(p ; q) : p or q (not exclusive or, mind backtracking)

Commitment symbol (!)

Instead of backtracking over **!**, the predicate in which **!** occurred, fails.

```
b.
a :- b, c.
a :- d.
c :- d, !, e.
c :- b.
d.
```

What happens on the query **?- a**.

First **b** succeeds, then **c** gets evaluated.

For **?- c**, first **d** succeeds, then **!** succeeds as well,

Suppose now, **e** fails. Now there will be NO backtracking over **!**

So since **e** fails, the call of **c** in **a** fails.

Backtracking for **a** continues; first back to **b,** finally to **d.** **SUCCESS**

Example of using commit !

- Typical use of **!**
  - “Good” use: efficiency – indicates that backtracking is not useful (pruning)
  - “OK” use: case distinction, “otherwise” – choices should be disjoint
  - “Bad” use: unintended change of behaviour, clutter the program
- **Drawback: procedural programming, not logic programming.**
- **Example: case-switch**

```
go :- write(“Enter a, b or c”), read(X), do(X).
do(a) :- !, ….
do(b) :- !, ….
do(c) :- !, ….
do(_) :- write(‘Wrong input’).
```

- Without the ! symbols, the message ‘Wrong Input’ would appear during backtracking, even if the input is correct.

If-then-else, otherwise

```
p :- q , !, r.
p :- s.
```

p is defined as: "if q then r else s"

```
max(X,Y,X) :- X>=Y, ! .
max(X,Y,Y).                    % otherwise
```

Add an element without duplication (difficult without !)

```
add(X,L,L) :- member(X,L),! .
add(X,L,[X|L]).
```

Exercise: write a **member/1** that finds an element once.

Examples with !

```
member(X,[X,_]) :- ! .
member(X,[_,L]) :- member(X,L).
```

Consider the following program:

```
p(1).
p(2):- !.
p(3).
```

Predict what happens to the following queries:

```
?- p(X).
?- p(X), p(Y).
?- p(X), !, p(Y).
?- p(3).
```

Contents

- Theoretical background of Pure Prolog
  - Unification (pattern matching)
  - Resolution (search tree, backtracking)

- Extensions of Prolog
  - Control and Negation
  - Program manipulation (self-modifying code)
  - Arithmetic operations

- More on lists
  - Slow-sort, Quick-sort
  - Practical exercise: Tree-sort

Program manipulation (special predicates)

consult/1 : reads a program file.
make/0 : re-reads all program files (after modification)

call/1 : defined like **call(P) :- P.**
mixing data with 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!).**

NB: a predicate, read in with consult ***and*** manipulated with assert/retract, must be declared dynamic:
:- dynamic(name/arity).

- Theoretical background of Pure Prolog
  - Unification (pattern matching)
  - Resolution (search tree, backtracking)

- Extensions of Prolog
  - Control and Negation
  - Program manipulation (self-modifying code)
  - Arithmetic operations

- More on lists
  - Slow-sort, Quick-sort
  - Practical exercise: Tree-sort

Arithmetic (1)

- How can we perform calculations in Prolog?
- The equality predicate in Prolog is **=** , indicating unification.
  - The query **X=1+1** succeeds with X=1+1.
- If we want to compute X=2 as a result,
  we have to use the predicate **is**:
  - **X is 1+1** succeeds with X=2.
- **is** expects its second argument to be an arithmetical expression,
  evaluates it, and then behaves like **=**.
- If the right hand contains an unbound variable,
  you will get a runtime error.
- OK **:- X is 10*10, Y is X*X.**
  Result: X=100, Y=10000
- Wrong **:- Y is X*X, X is 10*10.**
  Result: **ERROR: is/2: argument is not sufficiently instantiated**

Arithmetic (2)

- **Example: the factorial function.**
- **The predicate fac(X,Y) means Y = X!**

- **fac(0,1) :- !.** **% 0! = 1**
- **fac(X,Y) :- X1 is X-1 ,fac(X1,Z),Y is X*Z.** **% x! = x.(x-1)!**

- **The first argument is the input, the second the output.**
- **fac(3,X) succeeds with X=6, but fac(X,6) gives an error. (Why?)**

- **Drawback: arithmetic is not purely logical,**
  - **reversible computing is destroyed…**
  - **You must be aware of the computation order...**

Fibonacci, slow and smart

- Slow Fibonacci:

```
fib(1,1) :- ! .
fib(2,1) :- ! .
fib(X,Y) :-
   X1 is X-1, fib(X1,Y1),
   X2 is X-2, fib(X2,Y2),
   Y is Y1 + Y2.
```

Notes:

- Why is the fib-program above not type safe?
  - **number(X), X1 is X-1 (test if X is a number)**
- How to write an efficient Fibonacci?

Fibonacci, slow and smart

- Smart Fibonacci

```
fib(1,1) :- ! .
fib(2,1) :- ! .
fib(X,Y) :-
  X1 is X-1, fib(X1,Y1),
  X2 is X-2, fib(X2,Y2),
  Y is Y1 + Y2,
  asserta(fib(X,Y)).
```

Arithmetic (3)

- There is a third way to do Arithmetic (as of 2001)
- 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.
```

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]) ;
```

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)

- 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)
- Way more flexible than mathematical programming

Contents

- Theoretical background of Pure Prolog
  - Unification (pattern matching)
  - Resolution (search tree, backtracking)

- Extensions of Prolog
  - Control and Negation
  - Program manipulation (self-modifying code)
  - Arithmetic operations

- More on lists
  - Practical exercise: Tree-sort

Binary trees

Practical exercise of today, signoff block 5.

A binary tree is either empty, or
it consists of a number and two subtrees.

1. How can binary trees represented in Prolog?
   - Build a tree T with 7 nodes in Prolog, using **init(T)**

2. Write a predicate that determines the sum of all numbers in a tree:
   - sum(Tree,Num)

Solution

```
init(X):-
   T1=tree(2,empty,empty),
   T2=tree(5,empty,empty),
   T3=tree(11,T1,T2),
   X=tree(21,T3,T3).

sum(empty,0).
sum(tree(N,T1,T2),S) :-
   sum(T1,N1),
   sum(T2,N2),
   S is N+N1+N2.

:- init(X), sum(X,Y).
```

Practical Exercise 3

- Test if a tree is sorted
- Insert/delete from a sorted tree
- Transform a list to a sorted tree
- Transform a sorted tree to a sorted list

- Compose everything to get a sorting algorithm

- Optional:
  - Keep the trees balanced?
  - Printing the trees in a graphically appealing way?