UNIVERSITETET
I OSLO

Logic Programming II

Volker Stolz

stolz@ifi.uio.no

Department of Informatics – University of Oslo

Based on slides by Gerardo Schneider, Arild Torjusen and Martin Giese, UiO.

Outline

- Repetition
  - Facts, rules, queries and unification
- Today
  - Lists in Prolog
  - Different views of a Prolog program
  - Arithmetic in Prolog
  - Cut and negation

Facts and rules

- Remember: A declarative program admits two interpretations
  - Declarative interpretation, **What** is being computed.
  - Procedural interpretation, **How** computation takes place
- A Prolog program consists of a sequence of *clauses*
- clauses are *facts* (H) or *rules* (H :- $A_1,\ldots,A_k$)

  person(anne, sofia, martin, 1960)

  child(X,Y) :- person(X,Z,Y,U)
- Declaratively, the rule H:- A_1 , A_2 is read as: "H is implied by the conjunction A_1 , A_2"
- Procedurally, the rule H:- A_1 , A_2 is interpreted as "To answer the query H, answer the conjunctive query A_1 , A_2"

Queries and unification

- We initiate a computation by posing a *query* (|?- A_1,...,A_k)
  | ?- child(paul,Parent))
- For queries without variables we will get a yes/no answer.
- For queries with variables the result is the substitutions for (assignment of) the variables which will make the query true.
- The process of matching a query with facts and rules is called *unification*. The result of the unification is a *substitution*.

Outline

- Repetition
  - Facts, rules, queries and unification
- Today
  - Lists in Prolog
  - Different views of a Prolog program
  - Arithmetic in Prolog
  - Cut and negation

Lists in Prolog

- [] : the empty list
- [a,b,c] : a list with three elements
- [a|[b,c]] : another way of writing [a,b,c]
- [a,b|[c]] : the same
- [X | Y] represents a list with first element X and tail Y
- the member predicate:

```
member(X, [X|Rest]).
member(X, [H | Tail]) :- member(X, Tail).
```

- the append predicate:

```
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

append

```
append([], Ys, Ys).                                   /* 1 */
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).  /* 2 */
```

```
|-? append([a,b],[c,d],Res)
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
                     {X=a, Xs=[b], Ys=[c,d], Res=[a|Zs]}

append([b], [c,d], Zs)
append([X1 | Xs1], Ys1, [X1 | Zs1]) :- append(Xs1, Ys1, Zs1).
                     {X1=b, Xs1=[],Ys1=[c,d],Zs=[b|Zs1]}

append([], [c,d], Zs1)
append([], Ys2, Ys2)
                     {Ys2=[c,d],Zs1=Ys2=[c,d]}

Res = [a|Zs]
    = [a|[b|Zs1]]
    = [a|[b|[c,d]]] = [a,b,c,d] .
```

INF 3110 – 2019

Different views of a Prolog program

- **For testing:**
  | ?- member(wed, [mon, wed, fri]). yes

  | ?- append([a,b],[c,d],[a,b,c,d]) . yes

- **For computing:**
  | ?- member(X, [mon, wed, fri]).
  X = mon ? ; X = wed ?: X = fri ?; no

  | ?- append([a,b],[c,d],Zs) .
  Zs = [a,b,c,d] ? ;

  | ?- append(Xs, Ys, [a,b,c,d]).
  Xs = [],Ys = [a,b,c,d] ? ;
  Xs = [a], Ys = [b,c,d] ? ;
  Xs = [a,b],Ys = [c,d] ? ;
  ...

Outline

- Repetition
  - Facts, rules, queries and unification
- Today
  - Lists in Prolog
  - Different views of a Prolog program
  - Arithmetic in Prolog
  - Cut and negation

Arithmetic in Prolog

- Prolog programs presented so far were *declarative*: they admitted a dual reading as a formula
  - Operations of arithmetic are functional, not relational
- Arithmetic compromises Prolog's declarativeness
  - Solved in constraint logic programming languages

Arithmetic operators

- Built-in data structures:
  - Integers: 1,2,3,... (+, -, *, //)
  - Floating points: 2.3, 3.4456, 5.4e-13,... (+, -, *, /)
- Infix vs prefix notation*
  - 45+35
  - ’+’(45,35)
- It is possible to have user-defined operators with specified priority, associativity, etc

*We will see later how to evaluate expressions

Arithmetic comparison relations

- Prolog allows comparison of ground arithmetic expressions (*gae*, i.e. expressions without variables). *gae*s have *values*
- Built-in comparison relations: <, =<, =:= ("equal"), =\= ("different"), >= and >
- Queries
  - | ?- 6*3 =:= 9*2.
    yes
  - | ?- 8 > 5+3.
    no
  - | ?- 34>=X+4.
    uncaught exception: error(instantiation_error,(>=)/2
- Note difference between
  - = (unifiability relation) 1+1=2 *gives* no, X = 1 *gives* X = 1
  - == (syntactic equality) 1+1 == 2 *gives* no , X == x *gives* no
  - \== (syntactic inequality) 1+1\==2 *gives* yes.
  - =:= (value equality) 1+1 =:= 2 *gives* yes
  - =\= (value inequality) 1+1 =\= 2 *gives* no

Example: ordered lists

```
ordered([]).
ordered([X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

- Queries
  - | ?- ordered([3,4,67,8]).
    no
  - | ?- ordered([3,4,67, 88]).
    yes
  - | ? - ordered([3,4,X,88]).
    {INSTANTIATION ERROR: 4=<_30 - arg 2}

Evaluation of arithmetic expressions

- We need to introduce a way to evaluate expressions
  - | ?- X=:=3+4.   yields an error
  - | ?- X=3+4.
    X = 3+4
- Evaluation is done using ”is”
  - | ?- X is 3+4.
    X = 7
  - ”is” is a builtin predicate which has been defined as an operator for simpler syntax, we could also write:

    | ?- is(X,3+4).

    X = 7

Example: Factorial

```
factorial(0,1).
factorial(N,F) :-N>0, N1 is N-1,
                 factorial(N1,F1),
                 F is N*F1.
```

- Queries
  - | ?- factorial(5,X).
    X = 120
    Yes

  The following query gives an error however:
  - | ?- factorial(X,120).   ”X>0” is not allowed!
    uncaught exception: error(instantiation_error,(>)/2)

Example: Length of lists

- An intuitive definition **but wrong**

```
length([],0).
length([_ | Ts], N+1) :- length(Ts,N).
```

- Query
  - | ?- length([3,5,56,7],X).

    X = 0+1+1+1+1
    Yes
- What's the problem?

Expressions are not automatically evaluated in Prolog!

Example: Length of lists

- A good definition

```
length([],0).
length([_ | Ts], N) :- length(Ts,M), N is M+1.
```

- Queries
  - ```
    | ?- length([3,5,56,7],X).
    X = 4
    Yes
    ```
  - ```
    | ?- length(X,5).
    X = [_,_,_,_,_]
    yes
    ```

length(X,5)

```
length([],0).
length([_ | Ts], N) :- length(Ts,M), N is M+1.

:- length(X,5)
:- length(Ts,M), 5 is M+1
1. :- 5 is 0+1   Ts/[], M/0     FAIL
2. :- length(Ts1,M1), M is M1+1, 5 is M+1    Ts/[_,Ts1]
2.1 :- M is 0+1, 5 is M+1   Ts1/[], M1/0, Ts/[_,Ts1]
2.1 :- 5 is 1+1    Ts1/[], M1/0, Ts/[_,Ts1], M/1       FAIL
2.2 :- length(Ts2,M2), M1 is M2+1, M is M1+1, 5 is M+1
            Ts1/[], M1/0, Ts/[_,Ts1], Ts1/[_,Ts2]
...
```

INF 3110 – 2019

Outline

- Repetition
  - Facts, rules, queries and unification
- Today
  - Lists in Prolog
  - Different views of a Prolog program
  - Arithmetic in Prolog
  - Cut and negation

cut

- *cut* is a built in system predicate which affects the procedural behaviour of a program
- its main function is to reduce the search space of Prolog computations by dynamically pruning the search tree.
- Ex:

  p(**s1**) :- A1

  ...

  p(**si**) :- B, !, C

  ...

  p(**sk**) :- Ak

- When cut is encountered,
  - all alternative ways of computing B are discarded.
  - all computations of p(**t**) are discarded as backtrackable alternatives.

- cut gives more control to the programmer, but compromises the declarative reading of the Prolog programs and makes it difficult to see what will happen in the computation.

rsiblings example

- Recall the rsiblings rule.

```
rsiblings(X,Y) :-   child(X,Parent1),
                    child(Y,Parent1),
                  X \== Y,
                    child(X,Parent2),
                    child(Y,Parent2),
                    Parent1 \== Parent2.
```

- | ?- rsiblings(anne,X).
- X = paul ? ;
- X = paul ? ;
- no

rsiblings with cut

- With cut

```
rsiblings(X,Y) :- child(X,Parent1),
                  !,
                  child(Y,Parent1),
                  X \== Y,
                  child(X,Parent2),
                  child(Y,Parent2),
                  Parent1 \== Parent2.

| ?- rsiblings(anne,X).

X = paul ? ;

no
| ?- rsiblings(X,anne).

no
```

rsiblings with cut, next try...

```
rsiblings(X,Y) :- child(X,Parent1),
                  child(Y,Parent1),
                  X \== Y,
                  !,
                  child(X,Parent2),
                  child(Y,Parent2),
                  Parent1 \== Parent2.
```

```
| ?- rsiblings(anne,X).

X = paul

yes
| ?- rsiblings(X,anne).

X = paul

yes
```

But what if anne has more than one sibling?

Cut destroys declarativity

Cut makes it possible to control program execution
-> Added efficiency.

On the other hand:

- Programs become hard to understand.
- Need to document in which ways predicates can be called.
- Compromises the original intension of the language.

Negation as failure

- Negation can be defined by cut.

```
not(X) :- X, ! , fail .
not(_) .
```

- The built-in negation operator is \+

```
| ?- \+ person(haakon,sonja,harald,1973) .
yes
```

- The query \+ A succeeds if and only if the query A fails.
- Corresponds to our “normal” notion of negation if the negated query always terminates and is ground. Consider negation of non-ground term X=1:

```
\+ (X=1)
no
```

IO in Prolog

- Various predicates for input/output.
  - print(f(a)) prints out a term.
  - display('Hello World') prints a string.

```
print_list([]) :- print(nothing).
print_list([X]):- write('only '), print(X).
print_list([X|Ys]) :- print(X), print_list_help(Ys).

print_list_help([]).
print_list_help([X|Xs]) :- write(' and '),print(X),
    print_list_help(Xs).
```

Problem with IO

- The problem: does not work with backtracking:

```
io_problem :- print(one), fail.
io_problem :- print(two).
```

- Will print onetwo

```
io_problem :- fail, print(one).
io_problem :- print(two).
```

- Will print two

- even though conjunction should be commutative.

Outline

- Repetition
  - Facts, rules, queries and unification
  - Lists in Prolog
  - Different views of a Prolog program
- Today
  - Arithmetic in Prolog
  - Cut and negation

Problems with Prolog

- No types
- No (standardized) module system
- Non-declarative arithmetic
- Need to use cut
- Cut makes automated optimization hard
- IO disaggrees with backtracking

More Logic PLs

- Mercury
- Higher-order logic programming, Lambda-Prolog
  - Like Prolog, but lambda terms instead of first order
  - Higher-order unification
  - *Not* a functional language!
- Curry: http://www-ps.informatik.uni-kiel.de/currywiki/start
- Constraint Logic Programming languages
  - Prolog just gathers instantiations for variables.
  - Instead, gather constraints that need to be satisfied.

  E.g. X > 3, X < 6, X \== 5

  System infers instantiation X=4