Logic Programming

Volker Stolz

stolz@ifi.uio.no

Department of Informatics – University of Oslo

Based on slides by G. Schneider, A. Torjusen and Martin Giese, UiO.

Outline

- A bit of history
- Brief overview of the logical paradigm
- Facts, rules, queries and unification
- Lists in Prolog
- Different views of a Prolog program

History of Logic Programming

- Origin in automated theorem proving
- Based on the syntax of first-order logic
- 1930s: “Computation as deduction” paradigm – K. Gödel & J. Herbrand
- 1965: “A Machine-Oriented Logic Based on the Resolution Principle” – Robinson: Resolution, unification and a unification algorithm.
  - Possible to prove theorems of first-order logic
- Early seventies: Logic programs with a restricted form of resolution introduced by R. Kowalski
  - The proof process results in a satisfying substitution.
  - Certain logical formulas can be interpreted as programs

History of Logic Programming (cont.)

- Programming languages for natural language processing - A. Colmerauer & colleagues
- 1971–1973: Prolog - Kowalski and Colmerauer teams working together
- First implementation in Algol-W – Philippe Roussel
- 1983: WAM, Warren Abstract Machine
- Influences of the paradigm:
  - Deductive databases (70's)
  - Japanese Fifth Generation Project (1982-1991)
  - Constraint Logic Programming
  - Parts of Semantic Web reasoning
  - Inductive Logic Programming (machine learning)

Paradigms: Overview

- Procedural/imperative Programming
  - A program execution is regarded as a sequence of operations manipulating a set of registers (programmable calculator)
- Functional Programming
  - A program is regarded as a mathematical function
- Object-Oriented Programming
  - A program execution is regarded as a physical model simulating a real or imaginary part of the world
- Constraint-Oriented/Declarative (Logic) Programming
  - A program is regarded as a set of equations
- Aspect-Oriented, Intensional, ... Programming

Outline

- A bit of history
- Brief overview of the logical paradigm
- Facts, rules, queries and unification
- Lists in Prolog
- Different views of a Prolog program

Declarative Programming

"Program = Logic + Control"

R. Kowalski

- In "traditional" programming
  - Programmer takes care of both aspects
- In declarative programming
  - The programmer only worries about the Logic
  - The interpreter takes care of Control

Declarative Programming

- Logic prog. supports declarative programming
- A declarative program admits two interpretations
  - Procedural interpretation:
    - **How** the computation takes place
    - Concerned with the *method*
    - A program is a description of an algorithm which can be executed
  - Declarative interpretation:
    - **What** is being computed
    - Concerned with the *meaning*
    - A program is viewed as a formula; possible to reason about its correctness without any reference to the underlying computational meaning
- This means that we can write *executable specifications*.

Example

- Find all grandchildren for a specific person X?
- Declarative description (defines the relation):
  - C is grandchild of P if C is child of a child of P
- Imperative description (explains how to find a grandchild):
  - To find a grandchild to X, first find a child to X. Then find a child to this child
- Imperative description II:
  - To find a grandchild to X, first find a parent to a child, then check if this parent is a child to X

Example: Imperative solution

- Let child be a matrix representing the parent relationship (names coded as Nat)
- For finding all the grandchildren of person:

```
read(person);
for i := 1 to nrPeople do
  if child[i, person] then
      for j := 1 to nrPeople do
          if child[j, i] then
              writeln(j);
          fi
      od
  fi
od
```

Example: Declarative solution

- Write child(*x*,*y*) if *x* is child of *y*.
- grandchild(*x*,*y*) if *x* is grandchild of *y*
- Logic (specification):

$\forall x \forall y (\exists z (child(x,z) \wedge child(z,y)) \rightarrow grandchild(x,y))$

- Prolog:

```
grandchild(X,Y) :- child(X,Z), child(Z,Y).
```

- ":-" is the reverse implication (←)
- "," between the two terms child(X,Z) and child(Z,Y) is the logical and

Important features of Logic Prog.

- Support interactive programming
  - User write a program and interact by means of various *queries*
- Predicates may fail or succeed
  - If they succeed, unbound variables are *unified* and may be bound to values
- Predicates do not return values
  - Terms can only be unified with each other
  - arithmetic expressions evaluated on demand
- No functions in Prolog!

Outline

- A bit of history
- Brief overview of the logical paradigm
- Facts, rules, queries and unification
- Lists in Prolog
- Different views of a Prolog program

Running Prolog at IFI

```
[stolz ~]$ gprolog
GNU Prolog 1.4.4 (64 bits)
Compiled Jan 15 2014, 20:38:19 with gcc
By Daniel Diaz
Copyright (C) 1999-2013 Daniel Diaz
| ?- [lists].
compiling /ifi/asgard/a03/stolz/lists.pl for byte code...
/ifi/asgard/a03/stolz/lists.pl compiled, 2 lines read - 535 bytes written,
   14 ms

(1 ms) yes
| ?- my_append([1,2,3],[4,5,6],Xs).

Xs = [1,2,3,4,5,6]

yes

…

| ?- halt.
```

Some programming principles

- We program by creating a (formal) world which we explore. Two phases:
  1. Describe the formal world.
  2. Ask questions about it (the machine answers)
- The description of the problem is done through
  - Facts: Basic truths in the world.
  - Rules: Describe how to divide the problem into simpler subproblems ("subgoals"). (Facts and Rules are both called Clauses)
  - Queries: Prolog answers questions ("queries") by using facts and rules

Clauses: Facts

- Facts:
  - isPrime(7), greaterThan(3,1), sum(2,3,5), brother(kain,abel)
- Example: Family relations
  - Persons have a name, a mother, a father and a birthday. *person(a,b,c,d)* denotes a person with name *a*, mother *b,* father *c,* and year of birth *d.*
- Represented by facts:

```
person(anne, sofia, martin, 1960).
person(john, sofia, george, 1965).
person(paul, sofia, martin, 1962).
person(maria, anne, mike, 1989).
```

- Constants: words starting with lower-case letters
- ("anne", "sofia") and numbers.
- Relations: words starting with lower-case letters
- ("person")

Queries

```
person(anne, sofia, martin, 1960).
person(john, sofia, george, 1965).
person(paul, sofia, martin, 1962).
person(maria, anne, mike, 1989).

|?- person(anne, sofia, martin, 1960).
yes
| ?- person(berta, sofia, martin, 1961).
no
```

- Prolog works in a closed world: what is true is what it knows, i.e. what is defined in the database - There is no don't know answer!

Queries with variables

- Variable: a word starting with upper-case letters or with '_' ("Year" and "Child" in the example below)
- How are the variables used?
  - Prolog searches in the knowledge base until it finds something that "fits" (unification) and gives it as a result
  - The matching substitution(s) is returned.

```
| ?- person(anne, sofia, martin, Year).
Year = 1960
yes

| ?- person(Child, anne, mike, Year).
Child = maria
Year = 1989
yes
```

Unification

- Unification: instantiating variables so that terms become syntactically equal.
  (solving equation between terms)
- Used to match a query with facts/rules (Cf. Sec.15.3. for a more formal exposition).
- To unify two terms s ≣ t
  - f(s1,s2...) ≣ f(t1,t2...) → s1 ≣ t1 and s2 ≣ t2...
  - f(s1,s2) ≣ f(t1) → fail (different arity)
  - f(...) ≣ g(...) → fail (different head)
  - X ≣ t → instantiate X with t
  - X ≣ Y → replace all Y by X.

Unification Example

- Unify $g(X,a) \equiv g(f(Y),Y)$
- $X \equiv f(Y)$ and $a \equiv Y$
- instantiate X/f(Y) and Y/a
- together X/f(a) and Y/a
- makes both terms equal to g(f(a),a)
- Syntactic equality! 1+2 and 3 don't unify!

Unification in Prolog

- Example
  - fact: *child(anne,sofia)*
  - query: *child(X,sofia)*
  - unification: *X:= anne .*
- Example
  - rule: grandchild(X,Z) :- child (X,Y), child(Y,Z)
  - query: grandchild(anne,G)
  - unification: X/anne, Z/G
  - solve: child(anne,Y), child(Y,G)

Composite queries

- Composite queries may be done using comma (,) and semicolon (;)
  - Comma represents the logical and
  - Semicolon represents the logical or

```
| ?- person(paul, martin, Father, Year);
     person(paul, Mother, martin, Year).
Mother = sofia
Year = 1962
yes
```

Clauses: Rules

- Let child(X,Y) represent "X is a child of Y":

```
person(anne, sofia, martin, 1960).
person(john, sofia, george, 1965).
person(paul, sofia, martin, 1962).
person(maria, anne, mike, 1989).
child(X,Y) :- person(X,Z,Y,U).
child(X,Y) :- person(X,Y,Z,U).     %  :- is read "if"

| ?- child(paul,martin).
yes

| ?- child(paul,Parent).

Parent = martin ? ;
Parent = sofia ? ;
no
```

Clauses: Rules

- Begin uninteresting variables with _:

```
child(X,Y) :- person(X,_,Y,_).
```

The two _ are different!

```
child(X,Y) :- person(X,Y,_Z,_U).
```

Scope of variables

- The scope of the occurrence of a variable is the rule where it appears
  - All the occurrences of a variable in a rule are bound to each other
  - Two different rules are completely independent
- The names of variables are arbitrary, but try to avoid misleading names

Finding the answer to queries

```
child(X,Y) :- person(X,Y,Z,U).
child(X,Y) :- person(X,Z,Y,U).
```

| ?- child(paul, martin).

We can use two different rules:

person(paul,martin,Z,U).

There is no corresponding fact

person(paul,Z,martin,U).

It matches person(paul,sofia,martin,1962).

prolog answers

yes

Finding the answer to queries

```
child(X,Y) :- person(X,Y,Z,U).
child(X,Y) :- person(X,Z,Y,U).
```

| ?- child(paul, Parent).

Two possibilities:

person(paul,Parent,Z,U).

Matches with person(paul,sofia,martin,1962)

The unification will give Parent = sofia.

person(paul,Z,Parent,U).

Matches person(paul,sofia,martin,1962)

The unification will give Parent = martin.

Rules with more than one condition

```
siblings(X,Y) :- child(X,Z), child(Y,Z), X \== Y.
```

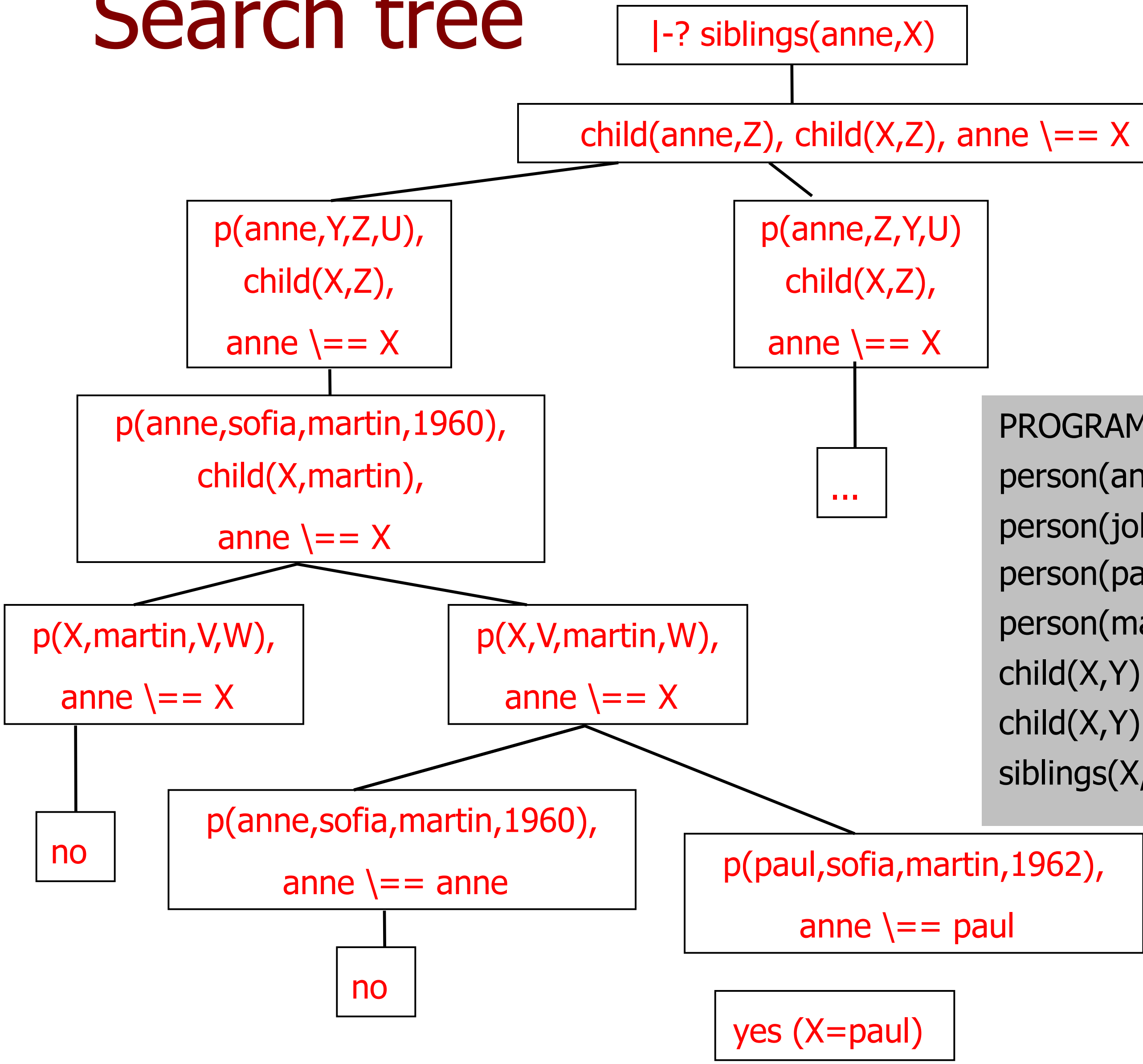
- Comma is the logical *and*, so all the conditions must be satisfied.
- X \== Y means that X and Y are syntactically unequal (e.g. siblings(anne,anne) will yield ”no”)

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

X = paul ? ;
X = john ? ;
X = paul ? ;

no
```

Search tree



PROGRAM:

```
person(anne, sofia, martin, 1960).
person(john, sofia, george, 1965).
person(paul, sofia, martin, 1962).
person(maria, anne, mike, 1989).
child(X,Y) :- person(X,Z,Y,U).
child(X,Y) :- person(X,Y,Z,U).
siblings(X,Y) :- child(X,Z), child(Y,Z), X \== Y.
```

More rules

- Let rsiblings(X,Y) represent that X and Y have the same parents (father and mother)

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

More rules

- Let hsiblings(X,Y) represent that X and Y have exactly one parent in common

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

Some queries

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

X = paul ? ;
X = paul ? ;
no

| ?- hsiblings(anne,X).

X = john ? ;
no
```

Recursive rules

- Let descendant(X,Y) represent that X is a descendant of Y

```
descendant(X,Y) :- child(X,Y).
descendant(X,Y) :- child(X,Z), descendant(Z,Y).
```

- NB! Order of rule definitions:
  - Non-recursive rule first
  - Recursive goal at the end.

Recursive rules - Queries

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

X = sofia ? ;
X = martin ? ;
no

| ?- descendant(X, sofia).

X = anne ? ;
X = john ? ;
X = paul ? ;
X = maria ? ;
no
```

Outline

- A bit of history
- Brief overview of the logical paradigm
- Facts, rules, queries and unification
- Lists in Prolog
- Different views of a Prolog program

Lists in Prolog

- Basic idea: same as in ML.
- Conceptually, a list is either:
  - nil, the empty list
  - cons(hd,tl), the list with head hd and a tail tl
- A list of prime numbers:

  cons(2,cons(3,cons(5,cons(7,nil))))
- BUT: use special syntax [] and [hd | tl]

  [2 | [3 | [5 | [7 | []]]]]

Prettier Syntax for Lists

- [] : the empty list
- [a,b,c] : a list with three elements, same as
  [a | [b | [c | []]]]
- [a,b|X] : another way of writing
- [a | [b | X]]

Unification: just like always...

- [a, b, c] ≣ [A | B] will be unified as
- A/a and B/[b, c]

Unification on lists

- [a,b,c] unifies with [Head | Tail]
  Result: Head=a and Tail=[b,c]
- [a] unifies with [H | T]
  Result: H=a and T=[ ]
- [a,b,c] unifies with [a | T]
  Result: T=[b,c]
- [a,b,c] does **not** unify with [b | T]
- [] does **not** unify with [H | T]
- [] unifies with []

Unification on lists: Example

- Assume the following fact: p([H | T], H, T).
- Query:

```
| ?- p([a,b,c], X, Y).

   X=a
   Y=[b,c]
   yes
```

Unification on lists: Example

- Assume the following fact: p([H | T], H, T).
- Query:

```
| ?- p([a], X, Y).
 X=a
 Y=[]
 yes

 | ?- p([], X, Y).
 no
```

Find an element in a list

- Check if the first element is the one we are searching for.
- If not, we look for the element in the rest of the list.
- Either we find X or the list becomes empty.

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

member(2,[1,2,3]) ? -> member(2,[2,3]) ? -> yes

Append two lists

- We will define a relation to concatenate two lists Xs and Ys into a third list Zs:

| ?- append([1, 2, 3], [4,5], Result). Should give

Result = [1,2,3,4,5].

- Prolog program:

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

Functions?

- There are no functions in Prolog, but relations
  - Functions are a particular case of relations
  - This allows using Prolog programs in multiple ways
- A function f:A → B can be represented in Prolog as a relation relf(a,b)
  - relf(a,b) may be understood as f(a)=b
- So, in append(List1, List2, Result).
  - List1 and List2 may be seen as input parameters
  - Result is the output parameter
- Compare with ML:
  - ML: fun fst(x::xs) = x
  - Prolog: fst([X|Xs],X) .

    | ?- fst([1,2,3],X). X = 1 ? ;

Outline

- A bit of history
- Brief overview of the logical paradigm
- Facts, rules, queries and unification
- Lists in Prolog
- Different views of a Prolog program

Anonymous variables

- When we are not interested in the value of a certain parameter, we may use '_'
- Example: In the program

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

we are not interested in the Head parameter

(nor in the Rest parameter).

- We can write it as follows:

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

Multiple uses of a Prolog program (1)

- Some Prolog programs may be used both for testing and for computing

- Example: member(X, Xs) means X is a member of the list Xs

```
member(X, [X | _]).
member(X, [_ | Xs]):- member(X,Xs).
```

Multiple uses of a Prolog program (1)

- For testing:

```
| ?- member(wed, [mon, wed, fri]).
yes
```

- For computing:

```
| ?- member(X, [mon, wed, fri]).
X = mon ?
X = wed ?
X = fri ?
no
```

Multiple uses of a Prolog program (2)

- It's possible to use the same program to concatenate two lists and to split a list in all possible ways

- Example: append(Xs,Ys,Zs)

- To concatenate two lists:

```
| ?- append([first, second, third], [fourth, fifth], Zs).
```

```
Zs = [first, second, third, fourth, fifth].
```

Multiple uses of a Prolog program (2)

- To split a list in all possible ways:

| ?- append(Xs, Ys, [first, second, third, fourth, fifth]).

Xs = [] Ys = [first,second,third,fourth,fifth] ?

Xs = [first] Ys = [second,third,fourth,fifth] ?

Xs = [first,second] Ys = [third,fourth,fifth] ?

Xs = [first,second,third] Ys = [fourth,fifth] ?

Xs = [first,second,third,fourth] Ys = [fifth] ?

Xs = [first,second,third,fourth,fifth] Ys = [] ?

Further reading

- Mitchell's book – Chapter 15
- Even further reading: Sterling and E. Shapiro: *The Art of Prolog*, 1994. MIT Press Series.

Mitchell's chap 15 – an overview.

15.1 History of logic programming

15.2 Brief overview of the logic programming paradigm

15.3 Equations solved by unification of atomic actions.
The formal basis for unification and the unification algorithm.

15.4 Clauses as parts of procedure declarations – Deals with Clauses = Rules and Facts and how they are computed.
1 Simple Clauses - The point is to make a relationship between logic programming and imperative programming.
2 Computation process
3 Clauses

15.5 Prolog's approach to programming
More about how computations take place. Multiple uses of prolog programs (testing vs. computing). Several examples.

15.6 Arithmetic in prolog

15.7 Control, ambivalent syntax and meta-variables.

15.8 Assessment of prolog.

15. 9 Bibliography

15.10 Summary

Prolog

"There is no question that Prolog is essentially a theorem prover à la Robinson. Our contribution was to transform that theorem prover into a programming language"

Colmerauer & Roussel (1996)