Principles of Programming Languages IX

Wael Aboelsaadat
wael@cs.toronto.edu
http://www.dgp.toronto.edu/~wael/324.html

Today

- Prolog II
## Prolog: unification operators

**• = unify with operator:** \(X = Y\)
- Semantically: unifiable test
- Succeeds as long as \(X\) and \(Y\) can be unified
- \(X\) may or may not be instantiated. \(Y\) may or may not be instantiated
- As a side effect, \(X\) and \(Y\) become bound together (refer to the same object)
- E.g.
  - \(? X=joe. Yes X = joe\)
  - \(? a(b,X,c) = a(b,Y,c). Yes X = Y\)

**• \(\neq\) does not unify with operator:** \(X \neq Y\)
- Semantically: not-unifiable test
- Succeeds as long as \(X\) and \(Y\) cannot be unified
- Both \(X\) and \(Y\) must be instantiated but may have uninstantiated elements.
- No side effects
- E.g.
  - \(? \text{Joe} \neq \text{Fred}. Yes\)
  - \(? a(b,X,c) \neq a(b,Y,c). No\)

**• \(==\) is already instantiated to operator:** \(X == Y\)
- Semantically: identical test
- Succeeds as long as \(X\) and \(Y\) are already instantiated to the same object
- Any variable inside \(X\) and \(Y\) must be the same
- No side effects
- E.g.
  - \(? 4 == 2 +2 No\)
  - \(? a(b,X,c) == a(b,Y,c). No\)
  - \(? a(b,X,c) == a(b,X,c). Yes\)

**• \(=:=\) is already instantiated to operator:** \(X =:= Y\)
- Semantically: identical test after evaluating terms
- E.g.
  - \(? 4 =:= 2 + 2 Yes.\)
  - \(? a(b,X,c) =:= a(b,Y,c). Error, a cannot be evaluated\)
Prolog: unification operators

- `\==` not already instantiated to operator: `X \== Y`
  - Semantically: not-identical test
  - Succeeds as long as X and Y are not already instantiated to the same object
  - No side effects
  - E.g.
    - `? A \== hello. Yes`
    - `? a(b,X,c) \== a(b,Y,c). Yes`
    - `? 1 +2 \== 3 Yes`

- `=\=` is already instantiated to operator: `X =:= Y`
  - Semantically: not-identical test after evaluating terms
  - E.g.
    - `? 4 =\= 2 + 2 No`

Other arithmetic operators:
- Add, subtract, multiply, division: `+ * /`
- Integer division: `//`
- Modulus: `mod`
- Comparison after evaluation: `> >= <= <`

- `is` operator: `X is Expr is(X,Expr)`
  - Semantically: evaluate second term and test if it is equal to X
  - Succeeds a long as X and the arithmetic evaluation of Expr can be unified
  - X may or may not be instantiated
  - Expr must not contain any uninstantiated variables
  - As a side effect, X is instantiated to the arithmetic evaluation of Expr
  - E.g.
    - `? 5 is ( ( 3 * 7 ) + 1 ) // 4 Yes`
    - `? X is ( ( 3 * 4 ) +10) mod 6 X=4`
    - `? is(2+3,5). No`
    - `? is(5,2+3). Yes`
Prolog: lists

- A sequence of terms of the form
  \[ t_1, t_2, t_3, t_4, ..., t_n \]
  where term \( t_i \) is the \( i \)th element of the list

- \[ \] is the ‘empty list’. It is an atom not a list.

- Example: \[ a, b, c, [ d, e, [], f ] \]
  - A list with 4 elements: a, b, c, and a list with 4 elements:d, e, an empty list, and f
  - Prolog supports Scheme-style nested lists

- Can break apart lists using “\|” into \[ Head | Tail \] where Head is the first item as an object and Tail is the rest of the list (as a list)
  - E.g. \?- [H | T] = [a, b, c].
    \[ H = a \]
    \[ T = [b,c] \]

- You can also use the same notation “\|” to construct lists:
  - E.g. \?- L = [a | [b, c]].
    \[ L = [a, b, c] \]

Prolog: lists & unification

- Examples:
  - \[ [A, B | C] = [a, b, c, d, e, f]. \] \( A = a \) \( B = b \) \( C = [c, d, e, f] \)
  - \[ [A, b | C] = [a, B, c, d, e, f]. \] \( A = a \) \( B = b \) \( C = [c, d, e, f] \)
  - \[ [X,Y] = [john, skates]. \] \( X=\text{john} \) \( Y=\text{skates} \)
  - \[ [\text{cat}] = [H|T]. \] \( H = \text{cat} \) \( T = [] \)
  - \[ [[the,Y]|Z] = [[X,hare ],[is,here ]]. \] \( Y = \text{hare} \) \( Z = [[\text{is, here}]] \) \( X = \text{the} \)
  - \[ [H|T] = a(b, c(d)). \] Error
  - \[ [n(X,Y),a(1)] = [\text{Name, Age}]. \] \( X = _\text{G17} \) \( \text{Name} = n(_\text{G17}, _\text{G18}) \) \( Y = _\text{G18} \) \( \text{Age} = a(1) \)
Prolog: recursion

- **Recursively defined predicate:** if a predicate symbol occurs both in the head and body of a rule, then the rule is recursive.
  - E.g. \( a(X) :- b(X,Y), a(Y) \).
    This predicate acts like a recursive subroutine.

- **Mutually recursive predicates:** recursion might be indirect, involving several rules.
  - E.g. \( a(X) :- b(X,Y), c(Y) \).
    \( c(Y) :- d(Y,Z), a(Z) \).
    The predicates \( a \) and \( c \) are said to be mutually recursive.

- **Non-linear recursion:**
  - E.g. \( a(X) :- b(X,Y), a(Y), c(Y,Z), a(Z) \).
    This generates what we call a recursive proof tree.

Prolog: recursion – examples

- **Factorial:**
  - **Declarative Semantics:**
    Factorial is 1 if \( n = 0 \), else Factorial is \( n \times \text{factorial}(n-1) \)
  - **Prolog:**
    \[
    \text{factorial}(0,1).
    \text{factorial}(Y,X) :- Y>0, Y1 is Y-1, \text{factorial}(Y1,X1), X is Y\times X1.
    \]
    ![Recursive proof tree for factorial](image)
Prolog: recursion – examples

• Appending lists:
  – Declarative Semantics:
    Appending an empty list to a non-empty list is the non-empty list
    else work on one list by removing its elements and adding it to the other list.
  – Prolog
    \[\text{append}([], X, X).\]
    \[\text{append}([H | X], Y, [H | Z]) :- \text{append}(X, Y, Z).\]

• Member of a list:
  – Declarative Semantics:
    X is a member of a list if X is equal to the first element, or a member of any
    sublist of that list
  – Prolog:
    \[\text{member}(X, [X|T]).\]
    \[\text{member}(X, [Y|T]): - \text{member}(X, T).\]

Prolog: recursion – examples

• Blocks:
  – Declarative Semantics:
    Block X is above block Y if X is placed on top of Y, or X is placed on top of some
    block Z that is above Y.
  – Prolog:
    \[\% \text{ first attempt}\]
    \[\text{above}(X, Y) :- \text{on}(X, Y).\] (1)
    \[\text{above}(X, Z) :- \text{above}(X, Y), \text{above}(Y, Z).\] (2)
    \[\text{on}(a, b).\] (3)
    \[\text{on}(b, c).\] (4)
    \[\text{on}(c, d).\] (5)
    |?- \text{above}(a, b).
    Yes
    |?- \text{above}(a, d).
    Yes
    |?- \text{above}(b, b).
    Infinite recursion! trace it to see why.
    |?- \text{above}(c, a).
    Infinite recursion! trace it to see why.
Prolog: recursion – examples

- Blocks:
  - Declarative Semantics:
    Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.
  - Prolog:

```prolog
% Second attempt
above(X,Y) :- on(X,Y).                           (1)
above(X,Z) :- on(X,Y), above(Y,Z).  (2)
on(a,b).       (3)
on(b,c).        (4)
on(c,d).        (5)
|?- above(a,d).
Yes
```

Prolog: recursion – examples

- Note that sometimes changing the order of rules and/or rule premises can cause problems for Prolog

- Example: above(X,Z) :- on(X,Y), above(Y,Z). (1)
above(X,Y) :- on(X,Y).                           (2)
on(a,b).       (3)
on(b,c).        (4)
on(c,d).        (5)
|?- above(a,d).
above
| rule (1)
avbove, on
| above
| rule (1)
above, on
| above
| rule (1)
above, on
| ...
### Prolog: recursion – examples

- **Infinite recursion:**
  - E.g. \( p : - p. \)
    
    `% declaratively perfectly correct, but procedurally causes infinite loop`

- **What to do about infinite recursion?**
  - Rewrite the rules and facts *(most widely used technique)*
  - Define a second non-recursive version *(similar to a base case)*
  - Use `!` to stop the unification *(more about this later).*

### Prolog: complex types - structures

- **Recall: what’s a function term?**
  
  \[ \text{functor(some-parameters)} \quad \text{e.g. woman(marry)} \]

- **We can construct complex data structures using nested function terms and lists.**
  - Represents a statement about the world

- **Example 2:**
  - A family consist of 2 persons, and 0 or more children. Each person is either employed for some salary or unemployed.

  \[
  \text{family(person(First-name,Last-name,date(Day,Month,Year),works(Company,Salary)),}
  
  \text{person(First-name,Last-name,date(Day,Month,Year),works(Company,Salary)),}
  
  \text{[person(First-name,Last-name,date(Day,Month,Year),unemployed),}
  
  \text{person(First-name,Last-name,date(Day,Month,Year),unemployed))}).}
  \]
Prolog: structures – example 2

• Family database:

family(
  person(tom,fox,date(7,may,1950),works(cbc,15200)),
  person(ann,fox,date(9,may,1951), works(ctv,25700)),
  [person(pat,fox,date(5,may,1973),unemployed),
   person(jim,fox,date(5,may,1973),unemployed)]).

%To find if there is a married woman that
% have at least three children:
| ?- family(_,person(Name,Surname,_,_),[_,_,_|_]).
  no

%To find if there is a family of three children,
| ?- family(_,_,[_,_,_]).
  no

%To find all the fox families
| ?- family(person(X,fox,Y,Z),T,W).
  X = tom
  Y = date(7,may,1950)
  Z = works(cbc,15200)
  T = person(ann,fox,date(9,may,1951), works(ctv,25700))
  W = [person(pat,fox,date(5,may,1973),unemployed),
       person(jim,fox,date(5,may,1973),unemployed)];
  no

Prolog: structures – example 2 – cont’d:

family(
  person(tom,fox,date(7,may,1950),works(cbc,15200)),
  person(ann,fox,date(9,may,1951), works(ctv,25700)),
  [person(pat,fox,date(5,may,1973),unemployed),
   person(jim,fox,date(5,may,1973),unemployed)]).

% Let us add more useful rules
husband(X) :- family(X,_,_).
wife(X)  :- family(_,X,_).
child(X) :- family(_,Children),
           member(X,Children).
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
exists(Person) :-husband(Person);
                wifed(Person);child(Person).
salary(person(_,works(_),S).
salary(person(_,unemployed),0).
dateofbirth(person(_,Date),Date).

%To find if there is a married woman that
% have at least three children:
| ?- family(_,person(Name,Surname,_,_),[_,_,_|_]).
  no

% Find the names of all the people in database
| ?- exists(person(Name,Surname,_,_)).
  Name = tom
  Surname = fox;
  Name = ann
  Surname = fox;
  Name = pat
  Surname = fox;
  Name = jim
  Surname = fox;
  no

% Find all children born in 1973
| ?- child(X),dateofbirth(X,date(_,1973)).
  X = person(pat,fox,date(5,may,1973),
         unemployed);
  X = person(jim,fox,date(5,may,1973),
         unemployed);
  no
Prolog: structures – example 2

• Family database – cont’d:

% To find the names of unemployed people who were born before 1975
?- exists(person(Name, Surname, date(_, _, Year), unemployed)), Year < 1975.
Name = pat
Surname = fox
Year = 1973;
Name = jim
Surname = fox
Year = 1973;
no

% To find people born before 1951 whose salary is less than 80000
?- exists(Person), dateofbirth(Person, date(_, _, Year)), Year < 1951,
salary(Person, Salary), Salary < 80000.
Person = person(tom, fox, date(7, may, 1950), works(cbc, 15200))
Year = 1950
Salary = 15200;
no

Prolog: structures – example 2

• Family database – cont’d:

% Let us add a rule to add the salaries
total([], 0).
total([Person|List], Sum) :- salary(Person, S), total(List, Rest), Sum is S + Rest.

% To find the total income of family
?- family(Husband, Wife, Children), total([Husband, Wife|Children], Income).
Husband = person(tom, fox, date(7, may, 1950), works(cbc, 15200))
Wife = person(ann, fox, date(9, may, 1951), works(ctv, 25700))
Children = [person(pat, fox, date(5, may, 1973), unemployed),
    person(jim, fox, date(5, may, 1973), unemployed)]
Income = 40900;

% To retrieve the nth child of a family, we need to define how to get the nth element of a list
nth_member(1, [X|L], X).
nth_member(N, [Y|L], X) :- N1 is N – 1, nth_member(N1, L, X).

% Now, let us define how to get the nth child. Note: I left children clause for you to define
?- nthchild(N, Family, Child) :- children(Family, ChildList), nth_member(N, ChildList, Child).
**Prolog: cut !**

- Cut is a special mechanism that can be used to tell Prolog which previous choices it need not consider again when it backtracks.

- The cut is written as ! and is inserted between goals as a pseudo-goal, for example:

  ```prolog
  head(X) :- goal1a, goal1b, !, goal1c, goal1d. % rule 1
  head(X) :- goal2a, goal2b, !, goal2c. % rule 2
  head(X) :- goal3a, goal3b, goal3c, goal3d. % rule 3
  | ?- head(someconstant). % query
  % Without the !, interpreter will try to unify the query with the 3 rules
  % because their heads match the query and it will stop only after the
  % 3rd. ! is a signal to the interpreter not to try unification on rules 2&3
  % after it is done with rule 1 if rule succeeds.
  ```

- **Is the location of the !, in the same clause, significant?**
  - Yes, consider above example, placing ! after goal1b means that only satisfying goal1a and goal1b is sufficient for your solution and you do not need to unify on any of the goals in rules 2&3.
  - Note that this does not mean that rule1 will succeed, because that is dependent on goal1c and goal1d being true as well.

---

**Prolog: cut ! – when to use?**

- **Common uses of the !:**
  - Tell the Prolog system that it has found the *right rule* for a particular goal:
    
    *If you get this far, you have picked the correct rule for this goal.*

  - Tell the Prolog system to fail a particular goal immediately without trying for alternate solutions:
    
    *If you get to here, you should stop trying to satisfy the goal.*

  - Terminate the generation of alternative solutions:
    
    *If you get to here, you have found the only solution to this problem, no point in looking for alternatives.*
Prolog: cut ! - examples

- Double-step function:
  
  if $X < 3$ then $Y = 0$
  if $3 \leq X$ and $X < 6$ then $Y = 2$
  if $6 \leq X$ then $Y = 4$

% In Prolog

$f(X,0) :- X < 3.$  %rule 1
$f(X,2) :- 3 \leq X, X < 6.$  %rule 2
$f(X,4) :- 6 \leq X.$  %rule 3

| ?- f(4,Y).
  Y=2

| ?- f(1,Y),2<Y.
  no

Prolog: cut ! – example

- Double-step function - cont’d:
  
  if $X < 3$ then $Y = 0$
  if $3 \leq X$ and $X < 6$ then $Y = 2$
  if $6 \leq X$ then $Y = 4$

What do we know about this function that Prolog doesn’t?

% same relations with !

$f(X,0) :- X < 3,!.  %rule 1$
$f(X,2) :- 3 \leq X, X < 6,!.  %rule 2$
$f(X,4) :- 6 \leq X.  %rule 3$

| ?- f(1,Y),2<Y.
  no

In this example, we changed the procedural
meaning of the program, but not the declarative meaning
Prolog: cut! – example

- Double-step cont’d:
  
  \[
  \begin{align*}
  &f(X,0) : - X < 3,!. \quad \% rule1 \\
  &f(X,2) : - X \leq 3, X < 6,!. \quad \% rule2 \\
  &f(X,4) : - X \geq 6. \quad \% rule3
  \end{align*}
  \]

  \[
  \begin{align*}
  \mid \text{?- f(5,Y).} &\quad \% query \\
  Y = 2 
  \end{align*}
  \]

  Can we come up with a more efficient version?

  \[
  \begin{align*}
  &f(X,0) : - X < 3. \quad \% rule1 \\
  &f(X,2) : - X < 6. \quad \% rule2 \\
  &f(X,4). \quad \% rule3
  \end{align*}
  \]

  \[
  \begin{align*}
  \mid \text{?- f(5,Y).} &\quad \% query \\
  Y = 2
  \end{align*}
  \]

  \[
  \begin{align*}
  &f(X,0) : - X < 3,!. \quad \% rule1 \\
  &f(X,2) : - X \leq 3, X < 6,!. \quad \% rule2 \\
  &f(X,4). \quad \% rule3
  \end{align*}
  \]

  \[
  \begin{align*}
  \mid \text{?- f(5,Y).} &\quad \% query \\
  Y = 2
  \end{align*}
  \]

  % What if we removed the cuts?

  \[
  \begin{align*}
  &f(X,0) : - X < 3. \quad \% rule1 \\
  &f(X,2) : - X < 6. \quad \% rule2 \\
  &f(X,4). \quad \% rule3
  \end{align*}
  \]

  \[
  \begin{align*}
  \mid \text{?- f(1,Y).} &\quad \% query \\
  Y = 0; \quad \% right answer \\
  Y = 2; \quad \% wrong answer, why? \\
  Y = 4; \quad \% wrong answer, why?
  \end{align*}
  \]

  Here, we changed the procedural and also the declarative meaning.