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.
    
    ```prolog
    ? 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.
    
    ```prolog
    ? joe \neq \) fred. Yes
    ? a(b,X,c) \neq a(b,Y,c). No
    ```
Prolog: unification operators

• **==** 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

• \( \backslash \equiv \) *not already instantiated to* operator: \( X \backslash \equiv Y \)
  - 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 \( \equiv \) hello. Yes
    - | a(b,X,c) \( \equiv \) a(b,Y,c). Yes
    - | 1 +2 \( \equiv \) 3 Yes

• \( =\equiv \) *is already instantiated to* operator: \( X =\equiv Y \)
  - Semantically: not-identical test after evaluating terms
  - E.g.
    - | 4 =\equiv 2 + 2 No
Prolog: unification operators

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

- **is operator:**
  - X is Expr \ is(X,Expr) \n  - Semantically: evaluate second term and test if it is equal to X
  - succeeds as 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 “\(\mid\)” into \([\text{Head} \mid \text{Tail}]\) where Head is the first item as an object and Tail is the rest of the list (as a list)
  – E.g.  
    \[
    \text{?- } [H \mid T] = [a, b, c].
    \]
    
    H = a
    T = [b, c]

• You can also use the same notation “\(\mid\)” to construct lists:
  – E.g.  
    \[
    \text{?- } L = [a \mid [b, c]].
    \]
    
    L = [a, b, c]
Prolog: lists & unification

- **Examples:**
  - \([A, B | C]\) = \([a, b, c, d, e, f]\). \(A = a\) \(B = b\) \(C = [c, d, e, f]\)
  - \([A, b | C]\) = \([a, B, c, d, e, f]\). \(A = a\) \(B = b\) \(C = [c, d, e, f]\)
  - \([X, Y]\) = \([john, skates]\). \(X = john\) \(Y = skates\)
  - \([\text{cat}]\) = \([H|T]\). \(H = \text{cat}\) \(T = []\)
  - \([[\text{the}, Y]|Z]\) = \([[X, hare],[is, here]]\). \(Y = \text{hare}\) \(Z = [[\text{is, here}]\) \(X = \text{the}\)
  - \([H|T]\) = \(a(b, c(d))\). Error
  - \([n(X, Y),a(1)]\) = \([\text{Name}, \text{Age}]\). \(X = _G17\) \(Y = _G18\)
    \(\text{Name} = n(_G17, _G18)\) \(\text{Age} = a(1)\)
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:**
  
  \[ n! \equiv n(n - 1) \cdots 2 \cdot 1. \]

  - **Declarative Semantics:**
    
    Factorial is 1 if \( n = 0 \), else Factorial is \( n \times \text{factorial} (n-1) \)

  - **Prolog:**
    
    ```prolog
    factorial(0,1).
    factorial(Y,X) :- Y>0, Y1 is Y-1, factorial(Y1,X1), X is Y*X1.
    ```

    ![Factorial Calculation Diagram](diagram.png)
Prolog: recursion – examples

• **Appending lists:**
  – Declarative Semantics:
    Append 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
    ```prolog
    append([],X,X).
    append([H | X], Y, [H | Z]) :- 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:
    ```prolog
    member(X,[X|T]).
    member(X,[Y|T]):-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:**
    
    \[
    \text{above}(X,Y) :- \text{on}(X,Y). \quad (1) \\
    \text{above}(X,Z) :- \text{above}(X,Y), \text{above}(Y,Z). \quad (2) \\
    \text{on}(a,b). \quad (3) \\
    \text{on}(b,c). \quad (4) \\
    \text{on}(c,d). \quad (5) \\
    \]

```
|?- above(a,b).
Yes
|?- above(a,d).
Yes
|?- above(b,b).
Infinite recursion! trace it to see why.
|?- 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:

| a | b | c | d |

% Second attempt

\(\text{above}(X,Y) :- \text{on}(X,Y)\). \hspace{1cm} (1)
\(\text{above}(X,Z) :- \text{on}(X,Y), \text{above}(Y,Z)\). \hspace{1cm} (2)
\text{on}(a,b). \hspace{1cm} (3)
\text{on}(b,c). \hspace{1cm} (4)
\text{on}(c,d). \hspace{1cm} (5)

\(?- \text{above}(a,d).
\)
Yes
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).

__Diagram__

```
    above
    /   \
above, on
    |   |
above
    |   |
above
    |   |
above, on
    |   |
above
    |   |
  above
    |   |
    ...
```
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?
  \textit{functor}(\text{some-parameters}) \quad \text{e.g.}\ \text{woman}(\text{marry})

• We can construct complex data structures using nested \textit{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:

```prolog
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
Family database – 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);
               wife(Person);child(Person).
salary(person(_,_,_,works(_,S)),S).
salary(person(_,_,_,unemployed),0).
dateofbirth(person(_,_,Date,_),Date).

% 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
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 =< X and X < 6 then Y = 2
  if 6 =< X then Y = 4

% In Prolog
f(X,0) :- X < 3. %rule 1
f(X,2) :- 3 =< X, X < 6. %rule 2
f(X,4) :- 6 =< X. %rule 3

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

| ?- f(1,Y),2<Y. %query
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 !

\[
\begin{align*}
  f(X,0) & :- X < 3,!. \quad \% rule 1 \\
  f(X,2) & :- 3 \leq X, X < 6,!. \quad \% rule 2 \\
  f(X,4) & :- 6 \leq X. \quad \% rule 3
\end{align*}
\]

\[
\begin{array}{c}
| \equiv f(1,Y),2 < Y. \quad \% query \\
no
\end{array}
\]

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

- **Double-step cont’d:**
  
  % same relations with !
  
  f(X,0) :- X < 3,!.  %rule1
  f(X,2) :- 3 =< X, X < 6,!.  %rule2
  f(X,4) :- 6 =< X.  %rule3
  | ?- f(5,Y).  %query
  
  Y = 2

*Can we come up with a more efficient version?*

f(X,0) :- X < 3,!.  %rule1
f(X,2) :- X < 6,!.  %rule2
f(X,4).  %rule3
| ?- f(5,Y).  %query

Y = 2

% What if we removed the cuts?  |  | ?- f(1,Y). % query
f(X,0) :- X < 3.  %rule1  |  Y = 0;  % right answer
f(X,2) :- X < 6.  %rule2  |  Y = 2;  % wrong answer, why?
f(X,4).  %rule3  |  Y = 4;  % wrong answer, why?

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