Principles of Programming Languages IX

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

Today

- Prolog II

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
  - No side effects
  - E.g.
    - \(? X==joe. Yes X = joe\)
    - \(? a(b,X,c) == a(b,Y,c). Yes X = Y\)

- **\(!=\)** does not unify with operator: \(X != Y\)
  - 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.
    - \(? joe!= fred. Yes\)
    - \(? a(b,X,c) != a(b,Y,c). No\)
    - \(? 4 != 2 + 2 Yes.\)

Other arithmetic operators:

- \(+\), \(-\), \(*\), \(/\)
- \(\text{\#}\) (integer division)
- \(\text{mod}\)
- Comparison after evaluation: \(>, <, \leq, \geq\)

- **\(is\)** is operator: \(X is Expr\)
  - 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.
    - \(? 5 is (3 * 7) // 4 Yes\)
    - \(? X is (1 + 3 * 4) mod 6 Yes X=4\)
    - \(? X is (2 + 3) Yes\)
Prolog: lists

• A sequence of terms of the form
  \[[t_1, t_2, \ldots, t_n]\]
  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 6 elements: \(a, b, c, d, e,\) an empty list, and \(f\)
  - Prolog supports Scheme-style nested lists
    - E.g. \(\[H | T\] = \[a, b, c\]. \(H = a\)
      \(T = \[b, c\]\)
  - You can also use the same notation \(\[\cdot\]\) to construct lists:
    - E.g. \(\[a, b, c\]\)

Prolog: recursion – examples

• Factorial:
  - Declarative Semantics:
    \(\text{factorial}(n) = 1\) if \(n = 0\), else \(\text{factorial}(n) = n \times \text{factorial}(n-1)\)
  - Prolog:
    \[
    \text{factorial}(0, 1). \\
    \text{factorial}(Y, X) :- Y > 0, Y1 is Y - 1, \text{factorial}(Y1, X1), X is Y \times X1.
    \]

Prolog: lists & unification

• Examples:
  - \(\[A, B, C\] = \[a, b, c, d, e, f\]\):
    \(A = a\) \(B = b\) \(C = [c, d, e, f]\)
  - \(\[A, b, c, d, e, f\]\):
    \(A = a\) \(B = b\) \(C = [c, d, e, f]\)
  - \(([X], Y) = ([\text{john}, \text{skates}]. \text{X} = \text{john} \ \text{Y} = \text{skates}\)
  - \([\text{a}, [\text{b}, \text{c}, \text{d}]]\):
    \(H = a\) \(T = [\text{b}, \text{c}, \text{d}]\)
  - \([\text{name}, \text{age}] = ([\text{john}, \text{skates}]. \text{X} = \text{john} \ \text{Y} = \text{skates}\)
  - \([\text{cat}] = ([\text{h}, \text{t}] | \text{Z}]. \text{H} = \text{cat} \ \text{T} = [\text{Z}]\)
  - \([\text{the}, \text{Y}] | \text{Z} = ([\text{X}, \text{h}, \text{are}]. \text{Y} = \text{h} \ \text{Z} = [\text{X}, \text{are}, \text{here}]\)
  - \([\text{H}, \text{T}] = a(b, c(d)). \text{Error}\)
  - \([\text{pun}, \text{age}] = ([\text{name}, \text{age}]. \text{X} = \text{name} \ \text{Y} = \text{age}\)

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:
    \[
    \text{append}([\ ] | X, Y, Z). \\
    \text{append}([H | X], Y, [H | Z]).
    \]

•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:
    \[
    \text{member}([X, Y | T], \ldots). \\
    \text{member}([X | T], \ldots). \\
    \text{member}([\ ] | T), \ldots).
    \]

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. \(\text{a} \times (\text{Y}) = \text{a}(\text{Y})\)
    \(\text{The predicate acts as a recursive subroutine.}\)

• Mutually recursive predicates: recursion might be indirect, involving several rules:
  - E.g. \(\text{a} \times (\text{Y}) = \text{b}(\text{Y}), \text{c}(\text{Y}).\)
    \(\text{The predicates a and c are said to be mutually recursive.}\)

• Non-linear recursion:
  - E.g. \(\text{a}(\text{X}) = \text{b}(\text{X}, \text{Y}), \text{c}(\text{Y,Z}), \text{d}(\text{Z}).\)
    \(\text{The generator we call a recursive proof tree.}\)
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:

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

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?
  functor(some-parameters)  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.

  ```
  family(person(First-name,Last-name,Date(Year,Month,Day),works(Company,Salary)),
  person(First-name,Last-name,Date(Year,Month,Day),works(Company,Salary)),
  [person(First-name,Last-name,Date(Year,Month,Day),unemployed),
  person(First-name,Last-name,Date(Year,Month,Day),unemployed)])).
  ```

- 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).

- 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,_,_),[_,_,_]).
    Name = ann
    Surname = fox;
    Name = pat
    Surname = fox;
    Name = jim
    Surname = fox;
    no
    ```

    ```
    family(person(tom,fox,date(7,may,1950),works(cbc,15200)),
    family(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).
    exists(Person) :-husband(Person);wife(Person);child(Person).
    salary(person(_,_,_,works(_,S)),S).
    salary(person(_,_,_,unemployed),0).
    dateofbirth(person(_,_,_,_),_).

    | ?- exists(person(Name,Surname,_,_)).
    Name = tom
    Surname = fox;
    Name = ann
    Surname = fox;
    Name = pat
    Surname = fox;
    Name = jim
    Surname = fox;
    no
    ```

- Find the names of all the people in database:

```
% 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: 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 alternative 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,
    so no point in looking for alternatives.

Prolog: cut ! – example

• 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

In this example, we changed the procedural
meaning of the program, but not the declarative meaning

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 alternative 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,
    so no point in looking for alternatives.

Prolog: cut! – example

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

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

  | % some relations with !
  | (X,Y) - X < 3. %rule 1
  | (X,Y) - X < 3. %rule 2
  | (X,Y) - X < 6. %rule 3
  | X =< Y,2<Y. %query
  | no

In this example, we changed the procedural
meaning of the program, but not the declarative meaning

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:

  head(X) :- goal1a, goal1b,goal1c,goal1d. %rule 1
  head(X) :- goal2a,goal2b,goal2c,goal2d. %rule 2
  head(X) :- goal3a,goal3b,goal3c,goal3d. %rule 3
  ! %head is a constant
  %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
      dependence on goal1a and goal1b being true as well.

Prolog: structures

• 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!

• 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 ! – 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
  Y = 2
  | ?- f(5,Y). %query

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

% What if we removed the cuts?

f(X,0) :- X < 3. %rule1
f(X,2) :- X < 6.   %rule2
f(X,4)            %rule3

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

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