Data structures in PROLOG

Lecture 2

Summary

- Esercise solutions
- "is" Predicate
- Term Definition
- Unification
- Natural numbers
- Lists
- Esercises

Prolog: is predicate

`A is B`

is a system predicate, true when the *evaluation* of the expression B returns a value, that is **assigned** to the variable A.

The evaluation of B is done using system operators.

Predicates defined by `is` are NOT invertible:

`?5 is X+Y.`

does not return the values of X and Y making the atom true.

`?X is 3+4.`

succeeds and returns `X=7`.

Prolog: programs with `is`

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

Terms

The set $\textsc{Term}$ of *terms* is inductively defined as:

1. Every constant symbol is a term;
2. Every variable symbol is a term;
3. If $t_1 \ldots t_n$ are terms and f is an n-ary, $f(t_1, \ldots, t_n)$ is a term (called *functional term*).

Examples: x, c, $f(x, y + c)$,...

Atoms and clauses are defined as before.

Unification recap: Substitutions

A *substitution* is a function from the set of variables $\textsc{Var}$ to the set of terms $\textsc{Term}$:
$\sigma : Var \mapsto Term$.

Given t, $t\sigma$ is defined (without function symbols) as follows:

- if c is a constant symbol, $c\sigma = c$;
- if x is a variable symbol, $x\sigma = \sigma(x)$;
- if f is a function symbol of arity n, $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$.

The substitution σ of a variable x by a term t is denoted by $x = t$ (or x/t).

Unification

An expression s is *more general* than an expression t, if t is an instance of s, but not viceversa.

Example: $p(a, X)$ is more general than $p(a, b)$.

A *unifier* of two expressions is the substitution, that makes them identical (when applied to them).

Example: $\{X = b\}$ is a unifier of $p(a, X)$ and $p(a, b)$.

Most general unifier

Intuitively, the *most general unifier* of two expressions, is the unifier that gives the most general instance of the two expressions.

Example: $\{X = b, Y = b, Z = a\}$ e $\{X = Y, Z = a\}$ are both unifiers of $p(a, X)$ and $p(Z, Y)$,

but $\{X = Y, Z = a\}$ is more general than $\{X = b, Y = b, Z = a\}$.

This unifier is unique up to variable renaming and is called mgu (most general unifier).

Unification (review)

1. $t_i = s_i$ identical variables or constants: skip to the next pair.
2. t_i variable: if t_i occurs in s_i then failure, otherwise $t_i = s_i$ is added to the unifier and all the occurrences of t_i are replaced by s_i.
3. s_i variable: as the previous one.
4. let t_i $f(tt_1, \ldots, tt_n)$ and s_i $g(ss_1, \ldots, ss_m)$ if $\neg(f = g) \vee \neg(n = m)$ then failure, otherwise unify $< tt_1, ss_1 >, \ldots < tt_n, ss_n >$.

Unification algorithm (full)

Input: C a set of pairs $< t_1, s_2 >$ where t_i, s_i are terms
Output: most general unifier θ, if exists, otherwise false
begin
θ := $\{\}$; success := true;
while not empty(C) and success **do**
begin
choose $< t_i, s_i >$ in C;
if $t_i = s_i$ **then** C:=C/$\{< t_i, s_i >\}$
else if var(t_i)
then if occurs(t_i, s_i)
then success:=false;
else begin
θ := subst(θ, t_i, s_i) $\cup$ $\{t_i = s_i\}$;
C:=subst(rest(C), t_i, s_i)
end
else if var(s_i)
then if occurs(s_i, t_i)
then success:=false;

else begin
$\theta :=$ subst$(\theta, s_i, t_i) \cup \{s_i = t_i\}$;
C:=subst(rest(C)$, s_i, t_i$)
end
else if $t_i = f(tt_1, \ldots, tt_n)$ **and**
$s_i = g(ss_1, \ldots, ss_m)$ **and**
$f = g \wedge n = m$
then C:= rest(C) $\cup\{< tt_1, ss_1 >, \ldots < tt_n, ss_n >\}$
else success := false
end;
if not success **then** output false **else** output true, θ
end

Unification: examples

$p(f(X,Y),a,g(b,W))$ unifies with $p(Z,X,g(b,Y))$.

$p(f(X,Y),a,g(b,W))$ does not unify with $p(Z,f(a),g(b,Y))$.

$p(f(X,Y),a,g(b,W))$ does not unify with $p(X,a,g(b,Y))$.

Unification

Check whether the following pairs of expressions are unifiable, and write the mgu if they unify.
Otherwise, change the second term so that they unify. X, Y, Z are variables and a, b are constants.

(1u) $f(cons(car(X), cdr(Y)), Z, X)$ and $f(Z, Z, cons(car(X), cdr(a)))$
(2u) $f(g(x, a), g(b, a))$ and $f(y, y)$
(3u) $P(g(x, a), f(b, a))$ and $P(g(f(b, y), y), f(z, y))$
(4u) `P([X|[a,b]])` and `P([a|[a|[Xs]]])`

A program for the class timetable

```
course(ai,timetab(tu,10,12),
       teacher(nardi,d),room(diag,b2)).
course(ai,timetab(we,10,12),
       teacher(nardi,d),room(diag,b2)).
course(ai,timetab(fr,10,12),
       teacher(nardi,d),room(diag,b2)).
...
```

A program for the class timetable

```
teaches(Tea,Course) :- course(Course,Timetab,Tea,Room).
length(Course,Len) :-
    course(Course,timetab(Day,Start,End),Tea,Room),
    plus(Start,Len,End).
hasClass(Tea,Day) :-
    course(Course,timetab(Day,Start,End),Tea,Room).
busy(Room,Day,Time) :-
    course(Course,timetab(Day,Start,End),Tea,Room),
    Start =< Time, Time =< End.
```

Natural numbers

```
natural_number(0).
natural_number(s(X)) :- natural_number(X).

plus1(0,X,X) :- natural_number(X).
plus1(s(X),Y,s(Z)):- plus1(X,Y,Z).

lesseq1(0,X) :- natural_number(X).
lesseq1(s(X),s(Y)) :- lesseq1(X,Y).
```

Lists

- `[a,b,c,d]` is a 4 element list;
- `[a | X]` is a list whose first element is a and the rest of the list is denoted by the variable `X`;
- `[Y | X]` is a list whose first element is denoted by the variable `Y` and is the rest of the list; è denotato dalla variable `X`.

Remember that a list of atoms is defined as follows:

- nil is a list;
- if a is an atom and L is a list $cons(a, L)$ is a list

Therefore:

`[a | X]` is the same as $cons(a, X)$

Lists

```
/* member1(X,L) is true when X is an element of L */

member1(X,[X|Xs]).
member1(X,[Y|Ys]) :- member1(X,Ys).

/* append1(X,Y,Z) is true when Z is the
   concatenation of  X and Y */

    append1([],Ys,Ys).
    append1([X|Xs],Ys,[X|Zs]) :- append1(Xs,Ys,Zs).
```

Other programs using lists

```
/* prefix(L1,L) is true when L1 is a prefix of L */

    prefix([],_Ys).
    prefix([X|Xs],[X|Ys]) :- prefix(Xs,Ys).

/* reverse(L1,L2) is true when L2 is the
reverse of L1 (same elements in reversed order */

    reverse1([],[]).
    reverse1([X|Xs],Zs) :- reverse1(Xs,Ys),
                           append1(Ys,[X],Zs).
```

Programs using lists and numbers

```
len([],0).
len([_X|Xs],s(N)) :- len(Xs,N).

len([],0).
len([_X|Xs],N) :- len(Xs,N1), N is N1 + 1.
```

Sorting lists

```
sort1(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).

permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys),
                          permutation(Ys,Zs).
permutation([],[]).

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

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

Home exercises

1. build the search tree for:
   `?- member(c,[a,c,b]).`
   `?- plus1(Y,X,s(s(s(s(s(0)))))).` and
   `?- reverse([a,b,c],X).`
2. Write the PROLOG programs `times`, `power`, `factorial`, `minimum` using the definitions given for natural numbers.
3. Write the PROLOG programs `suffix`,`subset`,`intersection` using lists to represent sets.
4. Write a PROLOG program for a depth-first visit of possibly cyclic graphs, represented through the relation `arc(X,Y)`
5. Write a PROLOG program implementing insertion sort on lists.