Substitutions and Unification in Prolog

The key process in the execution of Prolog programs is a matching operation called *unification*. (= in Prolog). When Prolog is attempting to satisfy a goal, clauses are selected if their head unifies with the goal. This effectively accomplishes a "case analysis" similar to Haskell's pattern matching definitions. However, the Prolog pattern matching is much more general than that in Haskell, and Prolog uses all the cases that match not just the first one.

Prolog's pattern matching is based on the concept of a *substitution*. A **substitution** θ is a function, θ: Variables $\rightarrow$ Terms that has the property that $\theta(V) = V$ for all but finitely many variables. The set of variables V so that $\theta(V) \neq V$ is called the **domain** of θ.

For example, θ = {X/f(Y), Z/3} is a substitution with domain(θ) = {X, Z} (we assume $\theta(V) = V$ for all variables V not explicitly shown). Even though it is only directly defined to operate on variables, we think of a substitution as providing a function θ: Terms $\rightarrow$ Terms, where a substitution is applied to a term simply by replacing all the variables in the term as the substitution requires. So for the example substitution $\theta(g(X,Z,X)) = g(f(Y),3,f(Y))$.

A substitution θ is a **unifier** for terms t and t' provided that $\theta(t) = \theta(t')$. Finding a unifier amounts to finding values for variables (if they exist) that solve an equality. There may be several ways to solve such equalities, and to maximize later options we will prefer solutions that leave variables unchanged wherever that is possible — this is called a **most general unifier (mgu)**. In fact, an answer that Prolog provides to a query is nothing more than an mgu!

For example, the terms f(X,Y,g(X)) and f(Z,g(Z),Y) are unified by both θ_1 = {X/Z, Y/g(Z} and θ_2 = {X/3, Y/g(3), Z/3} since

θ_1(f(X,Y,g(X))) = f(Z,g(Z),g(Z)) = θ_1(f(Z,g(Z),Y)), and

θ_2(f(X,Y,g(X))) = f(3,g(3),g(3)) = θ_2(f(Z,g(Z),Y)).

Of these two unifiers, θ_1 is the more general since it leaves the variable Z where θ_2 uses the constant 3, and a value can be chosen for the former to derive the latter. In fact, θ_1 is the mgu. We may later find that a choice of Z=2 satisfies another constraint of interest and still unifies the two original terms, but this can't be done after θ_2 since the commitment to Z=3 has already been made.

Unification refers to the process of finding an mgu and applying it to the terms for which it is derived. In terms of its relationship to logical inference, unification combines two separate steps of substituting equals for equals and instantiating variables into one process. Unification is a "two-way" matching process as values for variables in either term can be determined. In the logic programming context unification: (1) performs basic tests, (2) transmits the values for arguments, and (3) returns the values of results. If a goal does not "match" the head of a clause, that clause is dropped from consideration; when a goal does match the head of a clause, if an argument in a goal is a term, that term may be unified with a variable in the head of a clause, and if the argument in a goal is a variable, that variable may be unified with a term determined by the clause if it succeeds.

In the creation of unifiers, Prolog omits (deliberately, for efficiency reasons) one vital test — the **occur-check**. The problem can arise in many disguised forms, but can be effectively illustrated with a simple example. Suppose that we wish to unify (if possible) two terms, a variable X and some other term τ. This is generally easy to solve, just adopt the binding X/τ (i.e., $\theta(X) = \tau$) in the unifier. However, if X occurs in τ this is wrong and there is no substitution possible. For instance, suppose that τ = 1+X, then $\theta(\tau) = \theta(1+X) = 1+1+X \neq 1+X = \theta(X)$; no matter what is substituted for X, $\theta(X)$ and $\theta(1+X) = 1+\theta(X)$ are different, so unification is impossible. Never-the-less, since Prolog omits making "occur-checks" (claiming that this rarely occurs in practice), it can report false solutions and may go into an infinite loop, as in this example reporting X = 1+1+1+ …

A Unification Algorithm
(with occur-check)

input: two terms T_1 and T_2 to be unified

output: θ the most general unifier (mgu) of T_1 and T_2

algorithm:

initialize substitution θ to empty, the stack to contain the equation $T_1 = T_2$, and the program variable *failure* to false.

while stack not empty and not failure **do**

begin

pop the top equation, X = Y, from stack;

case

- X is a logic variable that *does not occur* in Y (not checked by Prolog):
  substitute Y for each occurrence of X in θ and in the stack, and add X/Y to θ;
- Y is a logic variable that *does not occur* in X (not checked by Prolog):
  substitute X for each occurrence of Y in θ and in the stack, and add Y/X to θ;
- X and Y are identical constants or logic variables:
  continue;
- X is $f(X_1, X_2, \ldots, X_n)$ and Y is $f(Y_1, Y_2, \ldots, Y_n)$ for some f and n>0:
  push $X_k = Y_k$ on the stack for k = 1, 2, … , n;
- otherwise:
  set failure true;

end case

end while

if failure **then** output 'failure' **else** output θ.

Prolog Operational Semantics

Given a logic program Π, one may pose queries, or goal clauses,

?- B_1, ... , B_n (where each B_i is an atomic formula)

and the program Π is searched for instantiations of the variables of the query that establish that the program logically implies the truth of the instantiated query. This is accomplished in the following way, beginning with no bindings for any variables.

I. If the goal list is empty (n=0), then report variable bindings and HALT SUCCEED (optionally continue with pending alternatives if requested); else

II. Goal B_1 is "matched" against (i.e., unified with) the head of every clause in the program and determines variable instantiations when successful.

A. Each successful match signifies an alternative that *may* lead to eventual success, and is saved (along with current variable bindings) for exploration

1. The first clause whose head matches B_1 is selected — say

   A :- C_1, ... , C_m.

   Clauses for other successful matches remain "pending". The matching correspondence (i.e., substitution) for this first match, call it θ, describes the instantiation patterns (or forms) for variables of B_1 and A that make them identical, $\theta(A)=\theta(B_1)$.

2. add the variable instantiations of θ to those already in effect.
3. form a new goal list by replacing B_1 by the matching clause body and incorporate new variable bindings in all the remaining goals — notice that the goal list is reduced only when m=0 (i.e., a goal matches a fact) — namely $\theta(C_1)$, ... , $\theta(C_m)$, $\theta(B_2)$, ... , $\theta(B_n)$
4. goto **I**

B. If no matches are found for B_1, the successful completion of the current partial instantiations is blocked and it fails. The system "backtracks" to the last successful match that still has pending alternatives. The goal list and instantiations which were in effect at that point are restored, and the search continues with the next alternative. If all pending alternatives have been explored, then HALT FAIL.

Roughly speaking, this amounts to an exhaustive left-to-right, depth-first search of all possible ways of applying the program to construct a proof of the existence asked by the query. The individual goals in the list are treated one at a time and we seek to establish conditions sufficient to satisfy them — when all have been satisfied, the existence proof is successfully completed.

Example Prolog search tree