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 [] is a function, []: Variables [] Terms that has the property that [](V) = V for all but finitely many variables. The set of variables V so that [](V) ≠ V is called the domain of [].

For example, [] = {X/f(Y), Z/3} is a substitution with domain([]= {X, Z} (we assume [](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 []: Terms [] 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 [](g(X,Z,X)) = g(f(Y),3,f(Y)).

A substitution [] is a unifier for terms t and t' provided that [](t) = [](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 []_1 = {X/Z, Y/g(Z)} and []_2 = {X/3, Y/g(3), Z/3} since

[]_1(f(X,Y,g(X))) = f(Z,g(Z),g(Z)) = []_1(f(Z,g(Z),Y)), and

[]_2(f(X,Y,g(X))) = f(3,g(3),g(3)) = []_2(f(Z,g(Z),Y)).

Of these two unifiers, []_1 is the more general since it leaves the variable Z where []_2 uses the constant 3, and a value can be chosen for the former to derive the latter. In fact, []_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 []_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 [] This is generally easy to solve, just adopt the binding X[]= (i.e., [X]=[]) in the unifier. However, if X occurs in [] this is wrong and there is no substitution possible. For instance, suppose that [] = [1+X, then [](1+X) = 1+1+X ≠ 1+X = [](X); no matter what is substituted for X, [X] and [](1+X) = 1+X is 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: \( \square \) the most general unifier (mgu) of \( T_1 \) and \( T_2 \)

algorithm:
initialize substitution \( \square \) to empty, the stack to contain the equation \( T_1 = T_2 \), and the program variable \( \text{failure} \) to false.

\[
\text{while stack not empty and not failure do}
\]
begin
pop the top equation, \( X = Y \), from stack;

\text{case}
\[
\begin{align*}
\quad & \text{X is a logic variable that does not occur in } Y \text{ (not checked by Prolog):} \\
& \quad \text{substitute } Y \text{ for each occurrence of } X \text{ in } \square \text{ and in the stack, and add } X/Y \text{ to } \square; \\
\quad & \text{Y is a logic variable that does not occur in } X \text{ (not checked by Prolog):} \\
& \quad \text{substitute } X \text{ for each occurrence of } Y \text{ in } \square \text{ and in the stack, and add } Y/X \text{ to } \square; \\
\quad & \text{X and } Y \text{ are identical constants or logic variables:} \\
& \quad \text{continue;}
\end{align*}
\]
\[
\begin{align*}
\quad & \text{X is } f(X_1, X_2, \ldots, X_n) \text{ and } Y \text{ is } f(Y_1, Y_2, \ldots, Y_n) \text{ for some } f \text{ and } n>0: \\
& \quad \text{push } X_k = Y_k \text{ on the stack for } k = 1, 2, \ldots, n; \\
\quad & \text{otherwise:} \\
& \quad \text{set failure true;}
\end{align*}
\]
end case
end while

if failure then output 'failure' else output \( \square \).
Prolog Operational Semantics

Given a logic program \([\text{p}]\), one may pose queries, or goal clauses,
\[- B_1, \ldots , B_n \] (where each \(B_i\) is an atomic formula)
and the program \([\text{p}]\) 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

\[ \text{A :- C}_1, \ldots , \text{C}_m. \]

Clauses for other successful matches remain “pending”. The matching
 correspondence (i.e., substitution) for this first match, call it \([A]\), describes the
instantiation patterns (or forms) for variables of \(B_1\) and \(A\) that make them
identical, \(\boxed{[A]} = \boxed{(B_1)}\).

2. add the variable instantiations of \([A]\) 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
   \(\boxed{(C}_1, \ldots , \boxed{C}_m), \boxed{(B}_2, \ldots , \boxed{B}_n)\)

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

\( \delta: a([], Bs, Bs). \)
\( \beta: a([A \mid As], Bs, [A \mid Cs]) :- a(As, Bs, Cs). \)

? a(X,Y,[1,2|X]) , a(Y,X,[2,1|X]).
\( a([1,2],[],[2,1]) \)
\( a(As1,Y,[2,1|As1]) , a(Y,[1|As1],[2,1,1|As1]) \)
\( a([2,1],[1],[2,1,1]) \)
\( a(As2,Y,[1,2|As2]) , a(Y,[1,2|As2],[2,1,1,2|As2]) \)
\( a([1],[1],[1]) \)
\( a([],[],[1]) \)

\( X = [] \)
\( Y = [1,2] \)

\( X = [1|As1] \)
\( Y = [2,1] \)

\( As1 = [] \)
\( Y = [2,1] \)

\( As1 = [2|As2] \)
\( As2 = [] \)
\( Y = [1,2] \)

\( ETC. \)

Example Prolog search tree

\( a([1,2],[1],[2,1,1]) \)

\( a([2,1],[1],[2,1,1]) \)

\( a([1],[1],[1]) \)

\( a([],[],[1]) \)

SUCCESS

\( X = [1] \)
\( Y = [2,1] \)