Unification

Definition 1. Let t and u be any terms. A substitution $\sigma = [s_1/X_1, \ldots, s_m/X_m]$ is a *unifier* for t and u if

$$t\sigma = u\sigma$$

Definition 2. If σ and τ are substitutions then their *composition* $\sigma \circ \tau$ is the substitution given by

$$t(\sigma \circ \tau) = (t\sigma)\tau$$

for all terms t.

If $\sigma = [s_1/X_1, \ldots, s_m/X_m]$ and $\tau = [r_1/Y_1, \ldots, r_k/Y_k]$ then

$$\sigma \circ \tau = [s_1\tau/X_1, \ldots, s_m\tau/X_m, r_1/Y_1, \ldots, r_k/Y_k]$$

E.g., if $\sigma = [\ f(U,V)/X,\ \ g(V)/Y\ ]$ and $\tau = [\ h(Z)/U,\ \ b/V\ ]$ then

$$\sigma \circ \tau = [\ f(h(Z), b)/X,\ \ g(b)/Y,\ \ h(Z)/U,\ \ b/V\ ]$$

Definition 3. If σ_1 and σ_2 are unifiers for t and u then σ_1 is *more general than* σ_2 if there exists a subsitution τ such that

$$\sigma_2 = \sigma_1 \circ \tau$$

A substitution σ is a *most general unifier* (MGU) for the terms t and u if it is a unifier and it is more general than any other unifier for t and u, i.e.

- $t\sigma = u\sigma$, and
- whenever $t\sigma' = u\sigma'$ then $\sigma' = \sigma \circ \tau$ for some substitution τ.

Unification Algorithm

The algorithm takes two terms t and u and either succeeds, if they can be unified, and returns a MGU; or fails otherwise.

- If $t = X$ and $u = Y$ then succeed with $\sigma = [Y/X]$.
- If $t = X$ and $u = g(u_1, \ldots, u_m)$ then succeed with $\sigma = [u/X]$ if X does not occur in u and fail otherwise.
- If $t = f(t_1, \ldots, t_n)$ and $u = Y$ then succeed with $\sigma = [t/Y]$ if Y does not occur in t and fail otherwise.
- If $t = f(t_1, \ldots, t_n)$ and $u = g(u_1, \ldots, u_m)$ where $f \neq g$ then fail.
- If $t = f(t_1, \ldots, t_n)$ and $u = f(u_1, \ldots, u_n)$ then use the algorithm recursively to unify the lists $[t_1, \ldots, t_n]$ and $[u_1, \ldots, u_n]$.

If l and k are lists of terms then the unification of l and k proceeds as follows.

- If $l = [\,]$ and $k = [\,]$ then succeed with the identity substitution.
- If $l = [\,]$ and $k = (u : k')$ then fail.
- If $l = (t : l')$ and $k = [\,]$ then fail.
- If $l = (t : l')$ and $k = (u : k')$ then:
  1. apply unification to t and u, if that succeeds yielding σ_1 then
  2. apply unification recursively to the lists $l'\sigma_1$ and $k'\sigma_1$, if that also succeeds yielding σ_2 then
  3. succeed with $\sigma = \sigma_1 \circ \sigma_2$; otherwise fail.

Theorem 4. For any terms t and u,

- if t and u have a unifier then the algorithm succeeds and returns a MGU for t and u,
- if no unifier exists then the algorithm fails after finitely many steps.

Proof. Omitted. $\square$

Example 5.

$$h(\ f(U,V),\ U,\ g(V)\ )\ \stackrel{?}{=}\ h(\ X,\ g(Z),\ Z\ )$$

1. $f(U,V) \stackrel{?}{=} X$

   $\sigma_1 = [f(U,V)/X]$

2. $U \stackrel{?}{=} g(Z)$

   $\sigma_2 = [g(Z)/U]$

3. $g(V) \stackrel{?}{=} Z$

   $\sigma_3 = [g(V)/Z]$

$$\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3 = [\, f(g(g(V)),V)/X\ ,\ g(g(V))/U\ ,\ g(V)/Z\,]$$

Example 6.

$$h(\ f(U,V),\ g(Y),\ X\ ) \stackrel{?}{=} h(\ X,\ g(Z),\ Z\ )$$

1. $f(U,V) \stackrel{?}{=} X$

   $\sigma_1 = [f(U,V)/X]$

2. $g(Y) \stackrel{?}{=} g(Z)$

   3. $Y \stackrel{?}{=} Z$

      $\sigma_3 = [Y/Z]$

   $\sigma_2 = \sigma_3 = [Y/Z]$

4. $X(\sigma_1 \circ \sigma_2) = f(U,V),\ Z(\sigma_1 \circ \sigma_2) = Y$

   $f(U,V) \stackrel{?}{=} Y$

   $\sigma_4 = [f(U,V)/Y]$

$$\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_4 = [\ f(U,V)/X,\ f(U,V)/Z,\ f(U,V)/Y\ ]$$

Example 7.

$$f(\ f(U,V),\ W\ ) \stackrel{?}{=} f(\ W,\ f(g(V),x)\ )$$

1. $f(U,V) \stackrel{?}{=} W$

   $\sigma_1 = [f(U,V)/W]$

2. $W\sigma_1 = f(U,V)$

   $f(U,V) \stackrel{?}{=} f(g(V),x)$

   3. $U \stackrel{?}{=} g(V)$

      $\sigma_3 = [g(V)/U]$

   4. $V \stackrel{?}{=} x$

      $\sigma_4 = [x/V]$

   $\sigma_2 = \sigma_3 \circ \sigma_4 = [g(x)/U,\ x/V]$

$$\sigma = \sigma_1 \circ \sigma_2 = [\ f(g(x),x)/W,\ g(x)/U,\ x/V\ ]$$

Example 8.

$$f(X, X) \stackrel{?}{=} f(g(Y), Y)$$

$$
\begin{aligned}
1.\quad & X \stackrel{?}{=} g(Y) \\
& \sigma_1 = [g(Y)/X] \\
2.\quad & X\sigma_1 = g(Y) \\
& g(Y) \stackrel{?}{=} Y \\
& \textbf{Fail} \text{ (occurs check)}
\end{aligned}
$$

Example 9.

$$h(f(U,V),U,X) \stackrel{?}{=} h(X,g(Z),U)$$

1. $f(U,V) \stackrel{?}{=} X$

   $\sigma_1 = [f(U,V)/X]$

2. $U \stackrel{?}{=} g(Z)$

   $\sigma_2 = [g(Z)/U]$

3. $X(\sigma_1 \circ \sigma_2) = f(g(Z),V),\ U(\sigma_1 \circ \sigma_2) = g(Z)$

   $f(g(Z),V) \stackrel{?}{=} g(Z)$

   Fail

Resolution

Definition 10. Consider a query

$$?\!-\ \phi_1, \ldots, \phi_n.$$

and a clause

$$\theta :\!-\ \psi_1, \ldots, \psi_m.$$

such that none of the variables in the clause appear in the query. Suppose that the ith formula ϕ_i can be unified with the head θ and let σ be the most general unifier of ϕ_i and θ.

We say that the query can be *resolved* against the clause and that the *resolvent* of the two is the query

$$?\!-\ \phi_1\sigma, \ldots, \phi_{i-1}\sigma,\ \psi_1\sigma, \ldots, \psi_m\sigma,\ \phi_{i+1}\sigma, \ldots \phi_n\sigma.$$

Resolution proof

Prolog works backwards from the goal using resolution steps. The proof succeeds if eventually the query becomes empty.

Consider

```
?- likes(colin,Y).
```

The first step is to resolve the query against the clause

```
likes(X,Y):- cat(X), strokes(Y,X), feeds(Y,X).
```

but note that one of the variables in the clause appears in the query and resolution rules this out.

Renaming variables

The strategy used in Prolog is to choose fresh names for the variables in a clause every time the clause is used.

Thus, we actually resolve against, say,

```
likes(X1,Y1):- cat(X1), strokes(Y1,X1), feeds(Y1,X1).
```

The renaming of variables becomes very important if the same clause is used twice in a proof.

The resolvent is

```
?- cat(colin), strokes(Y,colin), feeds(Y,colin).
```

The first subgoal resolves immediatedly against a unit clause to give

```
?- strokes(Y,colin), feeds(Y,colin).
```

We can resolve against the clause

```
strokes(Y2,X2):- cat(X2), human(Y2), likes(Y2,X2).
```

to obtain

```
?- cat(colin), human(Y), likes(Y,colin), feeds(Y,colin).
```

The subgoals now can be resolved one-by-one against unit clauses leaving the empty query $\square$ and we are done.

SLD-Resolution

To organize an automatic search for a proof, one must choose:

1. (a) at each step, the subgoal to be considered,
   (b) the program clause to resolve it with;
2. the overall strategy to be used.

The execution of Prolog programs is based on the scheme:

1. (a) select the first subgoal in the list,
   (b) resolve with the program clauses in the order they are listed;
2. use a depth-first search strategy (we'll see this shortly).

This scheme is known as *SLD-resolution* which is short for "Selection driven Linear resolution for Definite clauses".

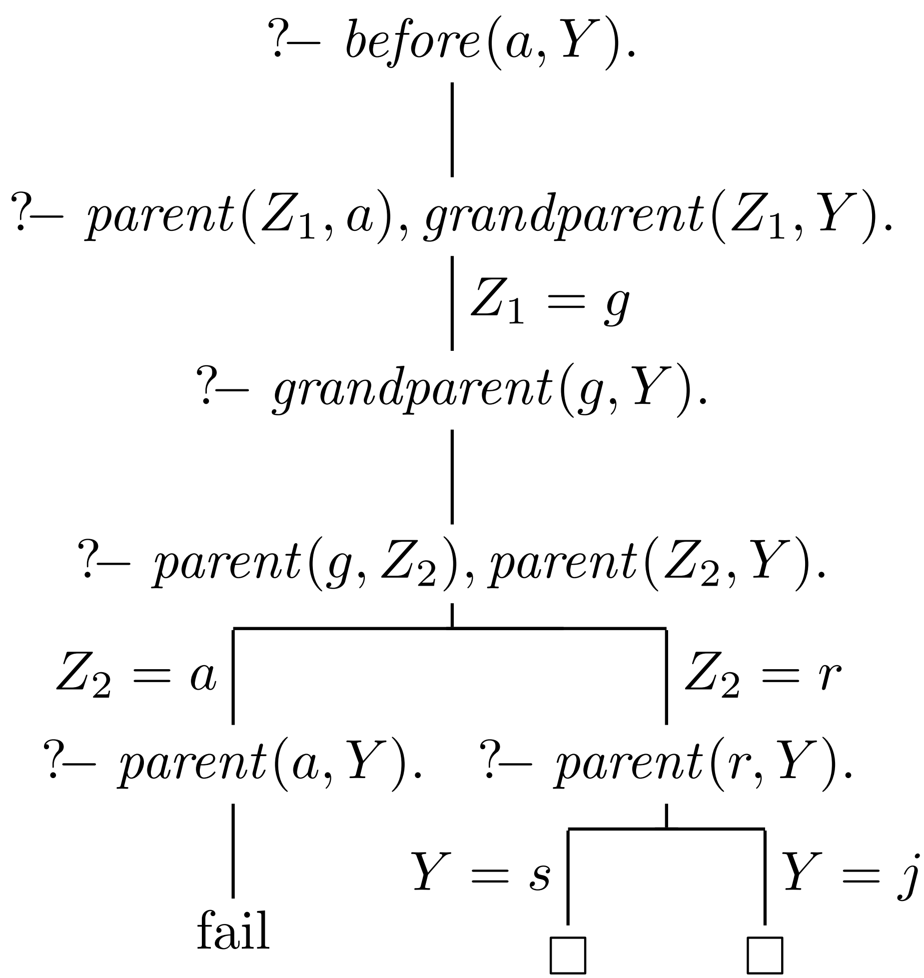
Example 11. Consider the program

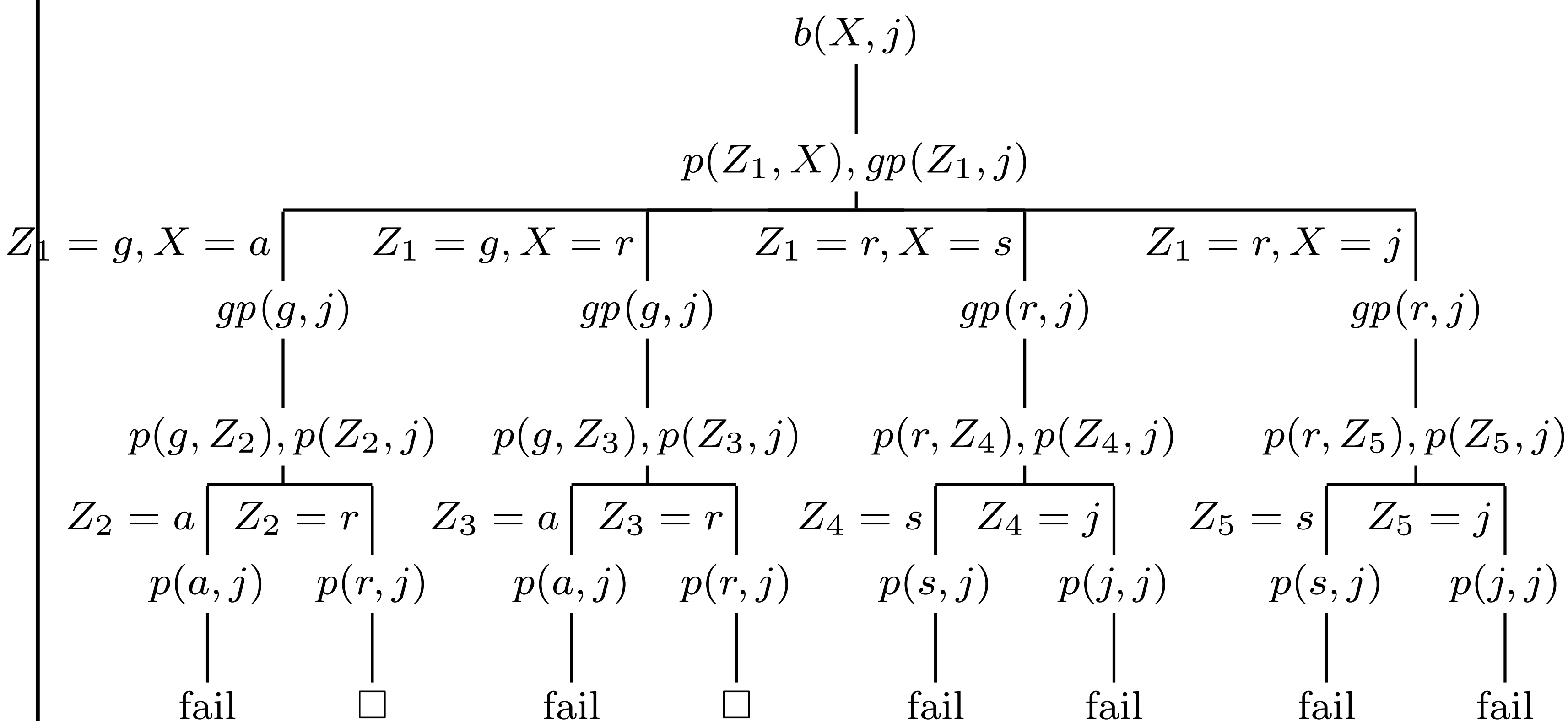
```
parent(g,a).
parent(g,r).
parent(r,s).
parent(r,j).

grandparent(X,Y):- parent(X,Z), parent(Z,Y).

before(X,Y):- parent(Z,X), grandparent(Z,Y).
```

The predicate `before` expresses the relation that one person is in the generation before another.

SLD-trees

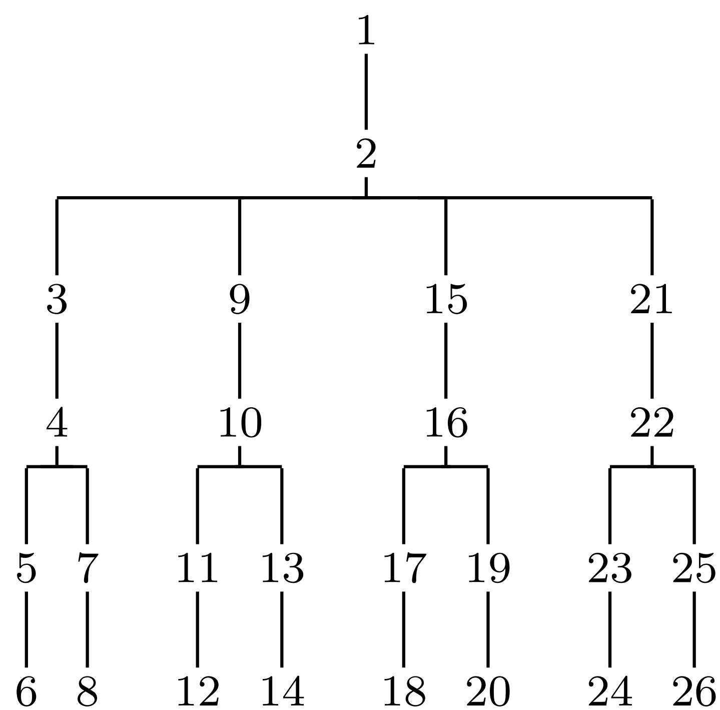
One can picture the search for a proof as a tree of possibilities.




$b(X, j)$
$p(Z_1, X), gp(Z_1, j)$
$Z_1 = g, X = a$
$Z_1 = g, X = r$
$Z_1 = r, X = s$
$Z_1 = r, X = j$
$gp(g, j)$
$gp(g, j)$
$gp(r, j)$
$gp(r, j)$
$p(g, Z_2), p(Z_2, j)$
$p(g, Z_3), p(Z_3, j)$
$p(r, Z_4), p(Z_4, j)$
$p(r, Z_5), p(Z_5, j)$
$Z_2 = a$
$Z_2 = r$
$Z_3 = a$
$Z_3 = r$
$Z_4 = s$
$Z_4 = j$
$Z_5 = s$
$Z_5 = j$
$p(a, j)$
$p(r, j)$
$p(a, j)$
$p(r, j)$
$p(s, j)$
$p(j, j)$
$p(s, j)$
$p(j, j)$
fail
$\square$
fail
$\square$
fail
fail
fail
fail

Depth-first search

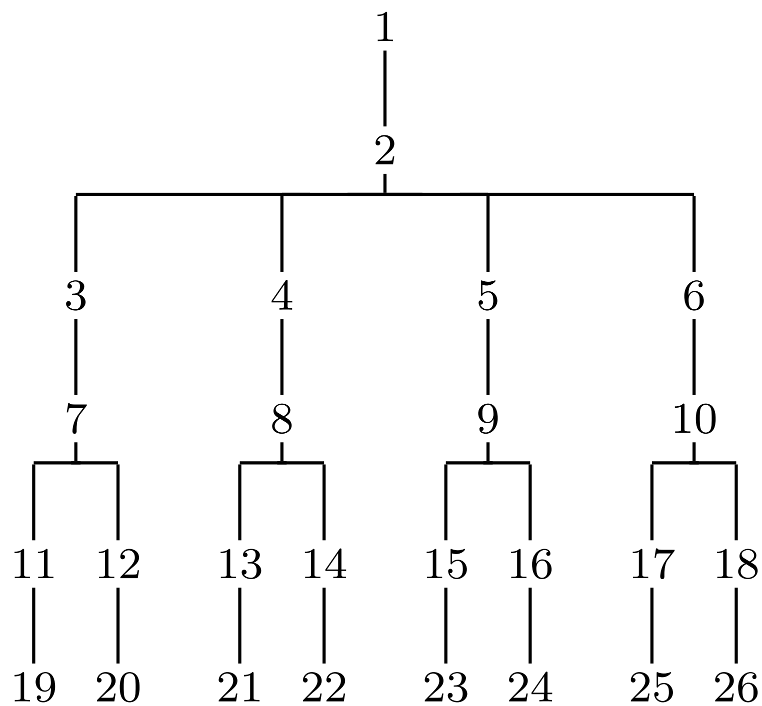
Fully explore each subtree to the left before moving on to the next subtree to the right. The nodes are traversed in the order below.



This is easy to implement without using too much memory.

Breadth-first search

Explore each successive level of the tree before looking at the next level down.



This requires far more memory than a depth-first search but it is nevertheless sometimes appropriate.