something more like English. The first says that either $\epsilon \leq 0$ or $g(\epsilon) > 0$. This simply guarantees that $\delta = g(\epsilon)$ is greater than zero whenever ϵ is. The second clause is more interesting. It states that either $|f(x) - L| < \epsilon$ or one of the three conditions

$$\epsilon > 0, \quad |x - a| < \delta, \quad x \neq a$$

fails to hold. Perhaps the process of Skolemization and finding clausal form has helped you understand the notion of limits better—or maybe it made limits more confusing! ■

Unification

The unification problem is as follows. Suppose we have the Skolemized clausal form of a formula. Let $p(t_1, \ldots, t_n)$ and $\neg p(t'_1, \ldots, t'_n)$ be predicates from two different clauses, where t_i and t'_i are terms. Unification is the process of substituting terms for some (or all) variables so that, after the substitution, t_i and t'_i are the same term for $1 \leq i \leq n$. This is *unification*, and the substitution is called *a unifier* of $p(t_1, \ldots, t_n)$ and $p(t'_1, \ldots, t'_n)$.

I said "a unifier" because many unifications are possible. For example, a unifier of $p(X)$ and $p(Y)$ is obtained by substituting $f(a, Z)$ for both X and Y. Another unifier is obtained by replacing X and Y with c. Still another is obtained by substituting X for Y. The last unification is "more general" because the first two can be obtained by a further substitution for the variable X in the last substitution—either $f(a, Z)$ or c, respectively. The goal of unification is to obtain a "most general" unifier; that is, a unifier from which all others can be obtained by further substitutions. *A priori*, it's not clear that a *most* general unifier exists.

There are various ways of describing an algorithm for finding the most general unifier. The most easily understood involves depth-first traversal of ordered trees associated with predicates. The trees are called *formation trees*. The description of the tree construction parallels Definition 3.4 (p. 106) and part of Definition 3.5. Here's the definition and the tree construction in parallel.

The terms in $\mathcal{L}$ and the associated formation trees are defined recursively as follows.:

D1. Every variable as well as every constant is a term.

T1. A vertex labeled with any variable or constant is a formation tree.

D2. If $t_1, \ldots, t_n$ are terms and f is a function that takes n arguments, then $f(t_1, \ldots, t_n)$ is a term.

T2. If $t_1, \ldots, t_n$ are terms and f is a function that takes n arguments, then the ordered tree whose root is labeled f and whose ith child is the formation tree for t_i is a formation tree.

Finally, we have the atomic formulas:

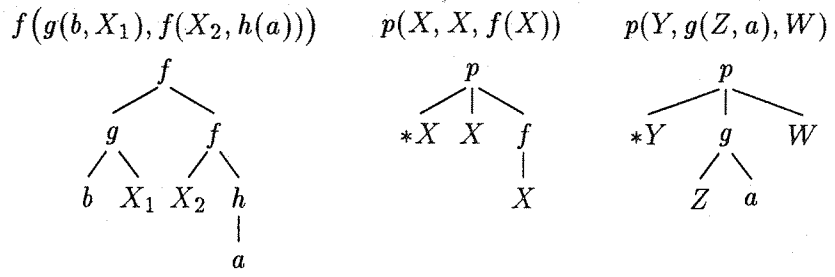

Figure 4.1 Some formation trees. Above each tree is the term or atomic formula to which it corresponds. The asterisks are explained in Example 4.7.

D3. If $t_1, \ldots, t_n$ are terms and p is a predicate that takes n arguments, then $p(t_1, \ldots, t_n)$ is an atomic formula.

T3. If $t_1, \ldots, t_n$ are terms and p is a predicate that takes n arguments, then the ordered tree whose root is labeled p and whose ith child is the formation tree for t_i is a formation tree.

Figure 4.1 shows some formation trees.

Notice that, since a variable takes no arguments, it can occur only as the leaf of a formation tree. Here's the algorithm for finding a most general unifier.

Algorithm 4.4 Most General Unifier

Let T_0 and T_1 be two formation trees. The algorithm terminates with both trees displaying the result of the most general unifier, or it reports failure. Start at the root of each tree and carry out a depth-first search and substitution as described below until both trees have been traversed.

1. Compare: Let L_i be the label at the present vertex in T_i. If $L_0 = L_1$, go to Step 3.
2. Substitute: Let U_i be the formation tree rooted at the current vertex in T_i. Since the roots L_0 and L_1 are not equal, there are three possibilities:
   (a) Neither L_0 nor L_1 is a variable: In this case, stop with failure.
   (b) One of L_i is a variable V and V appears in U_{1-i}. In this case, stop with failure.
   (c) One of L_i is a variable V and V does not appear in U_{1-i}. In this case, replace all leaves labeled V with a copy of U_{1-i}. (This substitutes U_{1-i} for V.)
3. Advance: If there are no more vertices in the depth-first search, terminate with success. Otherwise, move to the next depth-first vertex in both T_0 and T_1 and go to Step 1.

The subscript $1-i$ is used to refer to the tree other than T_i. This works since $(i = 0) \rightarrow (1 - i = 1)$ and $(i = 1) \rightarrow (1 - i = 0)$.

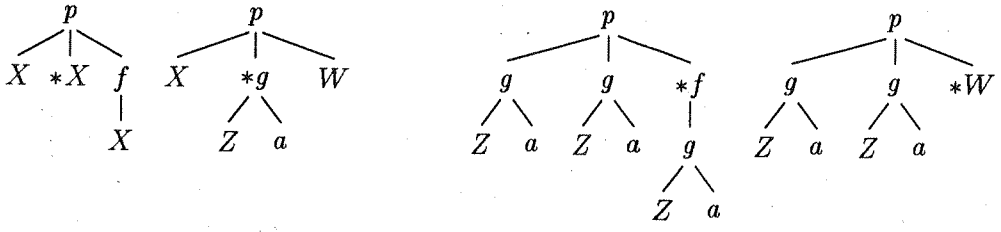

Figure 4.2 Application of the unification algorithm. See Example 4.7 for details.

Example 4.7 Unification

Let's see how the algorithm works on the rightmost trees in Figure 4.1. The asterisks indicate where the first disagreement occurs in the depth-first search. In this case, L_0 and L_1 are both variables so we can replace X with Y or Y with X. Choosing the former leads to the two trees on the left side of Figure 4.2.

Again, the vertices where disagreement occurs are marked by an asterisk. Now $L_0 = X$ and $L_1 = g$, so we must substitute U_1 for X everywhere. The results are shown on the right side of Figure 4.2. Finally, the last substitution that the algorithm requires is replacing W with $f(g(Z, a))$. The final result is $p(g(Z, a), g(Z, x), f(g(Z, a)))$.

Now let's make a slight change in the original T_1. Replace the variable W with X. In this case, the algorithm reaches its third disagreement at f in T_0 and g in T_1 (g instead of X because an earlier substitution for X). As a result, it reports failure because no substitution for a variable will change the function symbols.

Let's make a different change. Replace W with Z in T_1. The third disagreement is at the same place as before, but now there is a Z in T_1 and U_0 contains Z because U_0 corresponds to $f(g(Z, A))$. The algorithm ends in failure. Why can't we simply substitute $f(g(Z, A))$ for Z everywhere? If we were to do so, we would have to make the substitution in $f(g(Z, a))$ as well, and this would lead to an infinite repetition: $f(g(Z, a))$ becomes $f(g(f(g(Z, a)), a))$, which becomes $f(g(f(g(f(g(Z, a)), a)), a))$, and so on. It's a bit easier to see what's happening by looking at X and $h(X)$. Repeated substitution leads to $h(h(h(\ldots)))$, where the ellipsis indicates an infinite sequence of nested h's. ■

Theorem 4.5 Most General Unifier

Algorithm 4.4 always terminates. If it terminates in failure, no unification is possible. If it produces a unification, it is a most general one; that is, every other unification is obtainable from that produced by the algorithm by substituting for the variables it contains.

Since the algorithm starts with two trees, which are finite, it may seem that termination is obvious. This is not the case. Conceivably, the process of substitution could result in larger and larger trees such that the depth-first traversal is never finished. In fact, if we forget to include (b) in the algorithm, the algorithm can go on forever as described at the end of the previous example. This is what most (perhaps all) implementations of Prolog actually do.

Proof: How is termination proved? Very simply. At the start, the number of variables in the two formation trees is finite since the entire trees are finite. Because of (b) in the algorithm, each substitution eliminates all occurrences of a variable. Hence the number of substitutions done by the algorithm is finite. Consequently, the trees are changed only a finite number of times and so do not grow without bound.

Any disagreement that is found by the algorithm must be eliminated if a unification is to be found. The only substitutions that are allowed are those replacing a variable with a term. This explains why there must be a failure when (a) occurs in the algorithm. Case (b) must lead to failure because it leads to an infinite chain of substitutions as discussed near the end of the previous example.

This leaves (c). The substitution made there is the least possible to create unification. Mightn't a more restrictive substitution somehow allow more freedom elsewhere in the trees? Perhaps you intuitively see that this is not the case—and perhaps you don't, in which case some experimentation will probably convince you. To give a proof up to present-day standards of mathematical rigor, we'd have to spend time looking carefully at properties of substitutions. We won't do that, so the proof is not quite complete. ■

Aside. The entire discussion of unification could have been formulated with several atomic formulas in the predicate p rather than just two. You should find it relatively easy to adapt the algorithm.

Resolution

To prepare for resolution, we first put formulas in clausal form, ignoring the location of quantifiers. Then we use Skolemization to eliminate existential quantifiers and place universal quantifiers on the leftmost side of the formula. At this point, we're ready to obtain a contradiction by means of resolution. The process is the same as that for propositional calculus, except that we must make use of unification. We select two clauses, one containing an atomic formula $p(t_1, \ldots, t_n)$ and another containing the negation of the predicate, say $\neg p(t'_1, \ldots, t'_n)$. A most general unifier is found. The literals in the two clauses, with the exception of $p(t_1, \ldots, t_n)$ and $\neg p(t'_1, \ldots, t'_n)$, are placed in a new clause and the substitutions of the most general unifier are applied.