function \textsc{FOL-FC-Ask}(KB, q) returns a substitution or false
inputs: KB, the knowledge base, a set of first-order definite clauses
a, the query, an atomic sentence

local variables: new, the new sentences inferred on each iteration

repeat until new is empty
new \leftarrow \{ \}
for each rule in KB do
    \( \varphi \rightarrow A \ldots A p_m \)
    for each \( p_i \) such that
    \( \text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p'_1 A \ldots A p'_n) \)
    for some \( p'_1, \ldots, p'_n \) in KB
    \( q' \leftarrow \text{SUBST}(\theta, q) \)
    if \( q' \) does not unify with some sentence already in KB or new then
        add \( q' \) to new
    \( \phi \leftarrow \text{UNIFY}(q', \alpha) \)
    if \( \phi \) is not fail then return \( q' \)
add new to KB
return false

Figure 9.3 A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function \textsc{Standardize-Variables} replaces all variables in its arguments with new ones that have not been used before.

The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level. Possible facts that can be added, which determines the maximum number of iterations. Let \( k \) be the maximum arity (number of arguments) of any predicate, \( p \) be the number of predicates, and \( n \) be the number of constant symbols. Clearly, there can be no more than \( p^m n^k \) distinct ground facts, so after this many iterations the algorithm must have reached a fixed point. Then we can make an argument very similar to the proof of completeness for propositional forward
The details of how to make the transition from propositional to first-order completeness are given for the resolution algorithm in Section 9.5.

For general definite clauses with function symbols, \texttt{FOL-FC-Ask} can generate infinitely many new facts, so we need to be more careful. For the case in which an answer to the query sentence \( q \) is entailed, we must appeal to \textit{Herbrand's theorem} to establish that the algorithm will find a proof. (See Section 9.5 for the resolution case.) If the query has no answer, the algorithm could fail to terminate in some cases. For example, if the knowledge base includes the Peano axioms

\[
\text{NatNum}(0) \\
\text{do } \text{NatNum}(n) \rightarrow \text{NatNum}(S(n))
\]

then forward chaining adds \( \text{NatNum}(S(0)) \), \( \text{NatNum}(S(S(0))) \), \( \text{NatNum}(S(S(S(0)))) \), and so on. This problem is unavoidable in general. As with general first-order logic, entailment with definite clauses is semidecidable.

93.3 Efficient forward chaining

The forward-chaining algorithm in Figure 9.3 is designed for ease of understanding rather than for efficiency of operation. There are three possible sources of inefficiency. First, the "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the knowledge base. This is often called pattern matching and can be very expensive. Second, the algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the knowledge base on each iteration. Finally, the algorithm might generate many facts that are irrelevant to the goal. We address each of these issues in turn.

Matching rules against known facts

The problem of matching the premise of a rule against the facts in the knowledge base might seem simple enough. For example, suppose we want to apply the rule

\[
\text{Missile}(x) \rightarrow \text{Weapon}(x).
\]

Then we need to find all the facts that unify with \( \text{Missile}(x) \); in a suitably indexed knowledge base, this can be done in constant time per fact. Now consider a rule such as

\[
\text{Missile}(x) \rightarrow \text{Owns}(N, \text{None}), \text{Sells}(\text{West}, \text{at}, \text{None}).
\]

Again, we can find all the objects owned by Nano in constant time per object; then, for each object, we could check whether it is a missile. If the knowledge base contains many objects owned by Nano and very few missiles, however, it would be better to find all the missiles first and then check whether they are owned by Nano. This is the \textbf{conjunct ordering problem}: find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized. It turns out that finding the optimal ordering is \textbf{NP-hard}, but good heuristics are available. For example, the \textbf{minimum-remaining-values (MRV)} heuristic used for CSPs in Chapter 6 would suggest ordering the conjuncts to look for missiles first if fewer missiles than objects are owned by Nano.