by the knowledge base. There is no way to assert that a sentence is false in Prolog. This makes Prolog less expressive than first-order logic, but it is part of what makes Prolog more efficient and more concise. Consider the following Prolog assertions about some course offerings:

\[
\text{Course}(CS, 101), \text{ Course}(CS, 102), \text{ Course}(CS, 106), \text{ Course}(EE, 101).
\]

(9.11)

Under the unique names assumption, CS and EE are different (as are 101, 102, and 106), so this means that there are four distinct courses. Under the closed world assumption there are no other courses, so there are exactly four courses. But if these were assertions in FOL rather than in Prolog, then all we could say is that there are somewhere between one and infinity courses. That’s because the assertions (in FOL) do not deny the possibility that other unmentioned courses are also offered, nor do they say that the courses mentioned are different from each other. If we wanted to translate Equation (9.11) into FOL, we would get this:

\[
\text{Course}(d, n) \equiv (d = CS \land n = 101) \lor (d = CS \land n = 102) \\
\lor (d = CS \land n = 106) \lor (d = EE \land n = 101).
\]

(9.12)

This is called the completion of Equation (9.11). It expresses in FOL the idea that there are at most four courses. To express in FOL the idea that there are at least four courses, we need to write the completion of the equality predicate:

\[
x = y \iff (x = CS \land y = CS) \lor (x = EE \land y = EE) \lor (x = 101 \land y = 101) \\
\lor (x = 102 \land y = 102) \lor (x = 106 \land y = 106).
\]

The completion is useful for understanding database semantics, but for practical purposes, if your problem can be described with database semantics, it is more efficient to reason with Prolog or some other database semantics system, rather than translating into FOL and reasoning with a full FOL theorem prover.

### 9.4.6 Constraint logic programming

In our discussion of forward chaining (Section 9.3), we showed how constraint satisfaction problems (CSPs) can be encoded as definite clauses. Standard Prolog solves such problems in exactly the same way as the backtracking algorithm given in Figure 6.5.

Because backtracking enumerates the domains of the variables, it works only for finite-domain CSPs. In Prolog terms, there must be a finite number of solutions for any goal with unbound variables. (For example, the goal \(\text{diff}(Q, SA)\), which says that Queensland and South Australia must be different colors, has six solutions if three colors are allowed.) Infinite-domain CSPs—for example, with integer or real-valued variables—require quite different algorithms, such as bounds propagation or linear programming.

Consider the following example. We define \(\text{triangle}(X, Y, Z)\) as a predicate that holds if the three arguments are numbers that satisfy the triangle inequality:

\[
\text{triangle}(X, Y, Z) \iff X > 0, Y > 0, Z > 0, X + Y >= Z, Y + Z >= X, X + Z >= Y.
\]

If we ask Prolog the query \(\text{triangle}(3, 4, Z)\), it succeeds. On the other hand, if we ask \(\text{triangle}(3, 4, Z)\), no solution will be found, because the subgoal \(Z = 0\) cannot be handled by Prolog; we can’t compare an unbound value to 0.
Constraint logic programming (CLP) allows variables to be constrained rather than bound. A CLP solution is the most specific set of constraints on the query variables that can be derived from the knowledge base. For example, the solution to the triangle \((3, 4, Z)\) query is the constraint \(7 \geq Z\). Standard logic programs are just a special case of CLP in which the solution constraints must be equality constraints—that is, bindings.

CLP systems incorporate various constraint-solving algorithms for the constraints allowed in the language. For example, a system that allows linear inequalities on real-valued variables might include a linear programming algorithm for solving those constraints. CLP systems also adopt a much more flexible approach to solving standard logic programming queries. For example, instead of depth-first, left-to-right backtracking, they might use any of the more efficient algorithms discussed in Chapter 6, including heuristic conjunct ordering, backjumping, cutset conditioning, and so on. CLP systems therefore combine elements of constraint satisfaction algorithms, logic programming, and deductive databases.

Several systems that allow the programmer more control over the search order for inference have been defined. The MRS language (Genesereth and Smith, 1981; Russell, 1985) allows the programmer to write metarules to determine which conjuncts are tried first. The user could write a rule saying that the goal with the fewest variables should be tried first or could write domain-specific rules for particular predicates.

9.5 Resolution

The last of our three families of logical systems is based on resolution. We saw on page 250 that propositional resolution using refutation is a complete inference procedure for propositional logic. In this section, we describe how to extend resolution to first-order logic.

9.5.1 Conjunctive normal form for first-order logic

As in the propositional case, first-order resolution requires that sentences be in conjunctive normal form (CNF)—that is, a conjunction of clauses, where each clause is a disjunction of literals. Literals can contain variables, which are assumed to be universally quantified. For example, the sentence

\[
\forall x \ \text{American}(x) \land \text{Weapon}(y) \land \text{Sells}(x, y, z) \land \text{Hostile}(z) \land \text{Criminal}(x)
\]

becomes, in CNF,

\[
\neg\text{American}(x) \lor \text{Weapon}(y) \lor \neg\text{Sells}(x, y, z) \lor \neg\text{Hostile}(z) \lor \neg\text{Criminal}(x).
\]

Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. In particular, the CNF sentence will be unsatisfiable just when the original sentence is unsatisfiable, so we have a basis for doing proofs by contradiction on the CNF sentences.

\[\neg\text{A clause can also be represented as an implication with a conjunction of atoms in the premise and a disjunction of atoms in the conclusion (Exercise 7.13). This is called implicative normal form or Kowalski form (especially when written with a right-to-left implication symbol [Kowalski, 1979]) and is often much easier to read.}\]