9 INFERENCE IN FIRST-ORDER LOGIC

In which we define effective procedures for answering questions posed in first-order logic.

Chapter 7 showed how sound and complete inference can be achieved for propositional logic. In this chapter, we extend those results to obtain algorithms that can answer any answerable question stated in first-order logic. Section 9.1 introduces inference rules for quantifiers and shows how to reduce first-order inference to propositional inference, albeit at potentially great expense. Section 9.2 describes the idea of **unification**, showing how it can be used to construct inference rules that work directly with first-order sentences. We then discuss three major families of first-order inference algorithms. **Forward chaining** and its applications to **deductive databases** and **production systems** are covered in Section 9.3; **backward chaining** and **logic programming** systems are developed in Section 9.4. Forward and backward chaining can be very efficient, but are applicable only to knowledge bases that can be expressed as sets of Horn clauses. General first-order sentences require resolution-based **theorem proving**, which is described in Section 9.5.

9.1 PROPOSITIONAL VS. FIRST-ORDER INFERENCE

This section and the next introduce the ideas underlying modern logical inference systems. We begin with some simple inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers. These rules lead naturally to the idea that *first-order* inference can be done by converting the knowledge base to *propositional* logic and using *propositional* inference, which we already know how to do. The next section points out an obvious shortcut, leading to inference methods that manipulate first-order sentences directly.

9.1.1 Inference rules for quantifiers

Let us begin with universal quantifiers. Suppose our knowledge base contains the standard folkloric axiom stating that all greedy kings are evil:

$$\forall x \;\; King(x) \land Greedy(x) \;\Rightarrow\; Evil(x) \;.$$

Then it seems quite permissible to infer any of the following sentences:

$$\begin{array}{l} King(John) \land Greedy(John) \Rightarrow Evil(John) \\ King(Richard) \land Greedy(Richard) \Rightarrow Evil(Richard) \\ King(Father(John)) \land Greedy(Father(John)) \Rightarrow Evil(Father(John)) . \\ \quad \vdots \end{array}$$

UNIVERSAL INSTANTIATION

GROUND TERM

The rule of **Universal Instantiation** (**UI** for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable.[1] To write out the inference rule formally, we use the notion of **substitutions** introduced in Section 8.3. Let $\text{SUBST}(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v \;\; \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

for any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

EXISTENTIAL INSTANTIATION

In the rule for **Existential Instantiation**, the variable is replaced by a single *new constant symbol*. The formal statement is as follows: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \;\; \alpha}{\text{SUBST}(\{v/k\}, \alpha)} .$$

For example, from the sentence

$$\exists x \;\; Crown(x) \land OnHead(x, John)$$

we can infer the sentence

$$Crown(C_1) \land OnHead(C_1, John)$$

as long as C_1 does not appear elsewhere in the knowledge base. Basically, the existential sentence says there is some object satisfying a condition, and applying the existential instantiation rule just gives a name to that object. Of course, that name must not already belong to another object. Mathematics provides a nice example: suppose we discover that there is a number that is a little bigger than 2.71828 and that satisfies the equation $d(x^y)/dy = x^y$ for x. We can give this number a name, such as e, but it would be a mistake to give it the name of an existing object, such as π. In logic, the new name is called a **Skolem constant**. Existential Instantiation is a special case of a more general process called **skolemization**, which we cover in Section 9.5.

SKOLEM CONSTANT

Whereas Universal Instantiation can be applied many times to produce many different consequences, Existential Instantiation can be applied once, and then the existentially quantified sentence can be discarded. For example, we no longer need $\exists x \; Kill(x, Victim)$ once we have added the sentence $Kill(Murderer, Victim)$. Strictly speaking, the new knowledge base is not logically equivalent to the old, but it can be shown to be **inferentially equivalent** in the sense that it is satisfiable exactly when the original knowledge base is satisfiable.

INFERENTIAL EQUIVALENCE

[1] Do not confuse these substitutions with the extended interpretations used to define the semantics of quantifiers. The substitution replaces a variable with a term (a piece of syntax) to produce a new sentence, whereas an interpretation maps a variable to an object in the domain.

9.1.2 Reduction to propositional inference

Once we have rules for inferring nonquantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. In this section we give the main ideas; the details are given in Section 9.5.

The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of *all possible* instantiations. For example, suppose our knowledge base contains just the sentences

$$\begin{aligned}&\forall x \;\; King(x) \land Greedy(x) \;\Rightarrow\; Evil(x)\\ &King(John)\\ &Greedy(John)\\ &Brother(Richard, John)\;.\end{aligned} \tag{9.1}$$

Then we apply UI to the first sentence using all possible ground-term substitutions from the vocabulary of the knowledge base—in this case, $\{x/John\}$ and $\{x/Richard\}$. We obtain

$$\begin{aligned}&King(John) \land Greedy(John) \;\Rightarrow\; Evil(John)\\ &King(Richard) \land Greedy(Richard) \;\Rightarrow\; Evil(Richard)\;,\end{aligned}$$

and we discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences—$King(John)$, $Greedy(John)$, and so on—as proposition symbols. Therefore, we can apply any of the complete propositional algorithms in Chapter 7 to obtain conclusions such as $Evil(John)$.

This technique of **propositionalization** can be made completely general, as we show in Section 9.5; that is, every first-order knowledge base and query can be propositionalized in such a way that entailment is preserved. Thus, we have a complete decision procedure for entailment ... or perhaps not. There is a problem: when the knowledge base includes a function symbol, the set of possible ground-term substitutions is infinite! For example, if the knowledge base mentions the $Father$ symbol, then infinitely many nested terms such as $Father(Father(Father(John)))$ can be constructed. Our propositional algorithms will have difficulty with an infinitely large set of sentences.

Fortunately, there is a famous theorem due to Jacques Herbrand (1930) to the effect that if a sentence is entailed by the original, first-order knowledge base, then there is a proof involving just a *finite* subset of the propositionalized knowledge base. Since any such subset has a maximum depth of nesting among its ground terms, we can find the subset by first generating all the instantiations with constant symbols ($Richard$ and $John$), then all terms of depth 1 ($Father(Richard)$ and $Father(John)$), then all terms of depth 2, and so on, until we are able to construct a propositional proof of the entailed sentence.

We have sketched an approach to first-order inference via propositionalization that is **complete**—that is, any entailed sentence can be proved. This is a major achievement, given that the space of possible models is infinite. On the other hand, we do not know until the proof is done that the sentence *is* entailed! What happens when the sentence is *not* entailed? Can we tell? Well, for first-order logic, it turns out that we cannot. Our proof procedure can go on and on, generating more and more deeply nested terms, but we will not know whether it is stuck in a hopeless loop or whether the proof is just about to pop out. This is very much like the halting problem for Turing machines. Alan Turing (1936) and Alonzo Church (1936) both proved, in rather different ways, the inevitability of this state of affairs. *The question of entailment for first-order logic is* ***semidecidable****—that is, algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every nonentailed sentence.*

9.2 Unification and Lifting

The preceding section described the understanding of first-order inference that existed up to the early 1960s. The sharp-eyed reader (and certainly the computational logicians of the early 1960s) will have noticed that the propositionalization approach is rather inefficient. For example, given the query $Evil(x)$ and the knowledge base in Equation (9.1), it seems perverse to generate sentences such as $King(Richard) \land Greedy(Richard) \Rightarrow Evil(Richard)$. Indeed, the inference of $Evil(John)$ from the sentences

$$\forall x \;\; King(x) \land Greedy(x) \;\Rightarrow\; Evil(x)$$
$$King(John)$$
$$Greedy(John)$$

seems completely obvious to a human being. We now show how to make it completely obvious to a computer.

9.2.1 A first-order inference rule

The inference that John is evil—that is, that $\{x/John\}$ solves the query $Evil(x)$—works like this: to use the rule that greedy kings are evil, find some x such that x is a king and x is greedy, and then infer that this x is evil. More generally, if there is some substitution θ that makes each of the conjuncts of the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying θ. In this case, the substitution $\theta = \{x/John\}$ achieves that aim.

We can actually make the inference step do even more work. Suppose that instead of knowing $Greedy(John)$, we know that *everyone* is greedy:

$$\forall y \;\; Greedy(y) \; . \tag{9.2}$$

Then we would still like to be able to conclude that $Evil(John)$, because we know that John is a king (given) and John is greedy (because everyone is greedy). What we need for this to work is to find a substitution both for the variables in the implication sentence and for the variables in the sentences that are in the knowledge base. In this case, applying the substitution $\{x/John, y/John\}$ to the implication premises $King(x)$ and $Greedy(x)$ and the knowledge-base sentences $King(John)$ and $Greedy(y)$ will make them identical. Thus, we can infer the conclusion of the implication.

GENERALIZED MODUS PONENS

This inference process can be captured as a single inference rule that we call **Generalized Modus Ponens**:[2] For atomic sentences p_i, $p_i{}'$, and q, where there is a substitution θ

such that $\text{SUBST}(\theta, p_i{}') = \text{SUBST}(\theta, p_i)$, for all i,

$$\frac{p_1{}',\ p_2{}',\ \ldots,\ p_n{}',\ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}\ .$$

There are $n+1$ premises to this rule: the n atomic sentences $p_i{}'$ and the one implication. The conclusion is the result of applying the substitution θ to the consequent q. For our example:

$p_1{}'$ is $King(John)$	p_1 is $King(x)$
$p_2{}'$ is $Greedy(y)$	p_2 is $Greedy(x)$
θ is $\{x/John, y/John\}$	q is $Evil(x)$
$\text{SUBST}(\theta, q)$ is $Evil(John)$.	

It is easy to show that Generalized Modus Ponens is a sound inference rule. First, we observe that, for any sentence p (whose variables are assumed to be universally quantified) and for any substitution θ,

$$p \models \text{SUBST}(\theta, p)$$

holds by Universal Instantiation. It holds in particular for a θ that satisfies the conditions of the Generalized Modus Ponens rule. Thus, from $p_1{}', \ldots, p_n{}'$ we can infer

$$\text{SUBST}(\theta, p_1{}') \wedge \ldots \wedge \text{SUBST}(\theta, p_n{}')$$

and from the implication $p_1 \wedge \ldots \wedge p_n \Rightarrow q$ we can infer

$$\text{SUBST}(\theta, p_1) \wedge \ldots \wedge \text{SUBST}(\theta, p_n) \ \Rightarrow\ \text{SUBST}(\theta, q)\ .$$

Now, θ in Generalized Modus Ponens is defined so that $\text{SUBST}(\theta, p_i{}') = \text{SUBST}(\theta, p_i)$, for all i; therefore the first of these two sentences matches the premise of the second exactly. Hence, $\text{SUBST}(\theta, q)$ follows by Modus Ponens.

LIFTING

Generalized Modus Ponens is a **lifted** version of Modus Ponens—it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic. We will see in the rest of this chapter that we can develop lifted versions of the forward chaining, backward chaining, and resolution algorithms introduced in Chapter 7. The key advantage of lifted inference rules over propositionalization is that they make only those substitutions that are required to allow particular inferences to proceed.

9.2.2 Unification

UNIFICATION

UNIFIER

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a **unifier** for them if one exists:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q)\ .$$

Let us look at some examples of how UNIFY should behave. Suppose we have a query $AskVars(Knows(John, x))$: whom does John know? Answers to this query can be found

[2] Generalized Modus Ponens is more general than Modus Ponens (page 249) in the sense that the known facts and the premise of the implication need match only up to a substitution, rather than exactly. On the other hand, Modus Ponens allows any sentence α as the premise, rather than just a conjunction of atomic sentences.

by finding all sentences in the knowledge base that unify with $Knows(John, x)$. Here are the results of unification with four different sentences that might be in the knowledge base:

$$\begin{aligned}
&\textsc{Unify}(Knows(John, x),\ Knows(John, Jane)) = \{x/Jane\}\\
&\textsc{Unify}(Knows(John, x),\ Knows(y, Bill)) = \{x/Bill, y/John\}\\
&\textsc{Unify}(Knows(John, x),\ Knows(y, Mother(y))) = \{y/John, x/Mother(John)\}\\
&\textsc{Unify}(Knows(John, x),\ Knows(x, Elizabeth)) = fail\ .
\end{aligned}$$

The last unification fails because x cannot take on the values $John$ and $Elizabeth$ at the same time. Now, remember that $Knows(x, Elizabeth)$ means "Everyone knows Elizabeth," so we *should* be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in $Knows(x, Elizabeth)$ to x_{17} (a new variable name) without changing its meaning. Now the unification will work:

STANDARDIZING APART

$$\textsc{Unify}(Knows(John, x),\ Knows(x_{17}, Elizabeth)) = \{x/Elizabeth, x_{17}/John\}\ .$$

Exercise 9.12 delves further into the need for standardizing apart.

There is one more complication: we said that UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier. For example, $\textsc{Unify}(Knows(John, x), Knows(y, z))$ could return $\{y/John, x/z\}$ or $\{y/John, x/John, z/John\}$. The first unifier gives $Knows(John, z)$ as the result of unification, whereas the second gives $Knows(John, John)$. The second result could be obtained from the first by an additional substitution $\{z/John\}$; we say that the first unifier is *more general* than the second, because it places fewer restrictions on the values of the variables. It turns out that, for every unifiable pair of expressions, there is a single **most general unifier** (or MGU) that is unique up to renaming and substitution of variables. (For example, $\{x/John\}$ and $\{y/John\}$ are considered equivalent, as are $\{x/John, y/John\}$ and $\{x/John, y/x\}$.) In this case it is $\{y/John, x/z\}$.

MOST GENERAL UNIFIER

An algorithm for computing most general unifiers is shown in Figure 9.1. The process is simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed. For example, $S(x)$ can't unify with $S(S(x))$. This so-called **occur check** makes the complexity of the entire algorithm quadratic in the size of the expressions being unified. Some systems, including all logic programming systems, simply omit the occur check and sometimes make unsound inferences as a result; other systems use more complex algorithms with linear-time complexity.

OCCUR CHECK

9.2.3 Storage and retrieval

Underlying the TELL and ASK functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions. STORE(s) stores a sentence s into the knowledge base and FETCH(q) returns all unifiers such that the query q unifies with some

```
function UNIFY($x, y, \theta$) returns a substitution to make $x$ and $y$ identical
  inputs: $x$, a variable, constant, list, or compound expression
          $y$, a variable, constant, list, or compound expression
          $\theta$, the substitution built up so far (optional, defaults to empty)

  if $\theta$ = failure then return failure
  else if $x = y$ then return $\theta$
  else if VARIABLE?($x$) then return UNIFY-VAR($x, y, \theta$)
  else if VARIABLE?($y$) then return UNIFY-VAR($y, x, \theta$)
  else if COMPOUND?($x$) and COMPOUND?($y$) then
     return UNIFY($x$.ARGS, $y$.ARGS, UNIFY($x$.OP, $y$.OP, $\theta$))
  else if LIST?($x$) and LIST?($y$) then
     return UNIFY($x$.REST, $y$.REST, UNIFY($x$.FIRST, $y$.FIRST, $\theta$))
  else return failure
```

```
function UNIFY-VAR($var, x, \theta$) returns a substitution

  if $\{var/val\} \in \theta$ then return UNIFY($val, x, \theta$)
  else if $\{x/val\} \in \theta$ then return UNIFY($var, val, \theta$)
  else if OCCUR-CHECK?($var, x$) then return failure
  else return add $\{var/x\}$ to $\theta$
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

sentence in the knowledge base. The problem we used to illustrate unification—finding all facts that unify with $Knows(John, x)$—is an instance of FETCHing.

The simplest way to implement STORE and FETCH is to keep all the facts in one long list and unify each query against every element of the list. Such a process is inefficient, but it works, and it's all you need to understand the rest of the chapter. The remainder of this section outlines ways to make retrieval more efficient; it can be skipped on first reading.

We can make FETCH more efficient by ensuring that unifications are attempted only with sentences that have *some* chance of unifying. For example, there is no point in trying to unify $Knows(John, x)$ with $Brother(Richard, John)$. We can avoid such unifications by **indexing** the facts in the knowledge base. A simple scheme called **predicate indexing** puts all the $Knows$ facts in one bucket and all the $Brother$ facts in another. The buckets can be stored in a hash table for efficient access.

INDEXING

PREDICATE INDEXING

Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol. Sometimes, however, a predicate has many clauses. For example, suppose that the tax authorities want to keep track of who employs whom, using a predicate $Employs(x, y)$. This would be a very large bucket with perhaps millions of employers

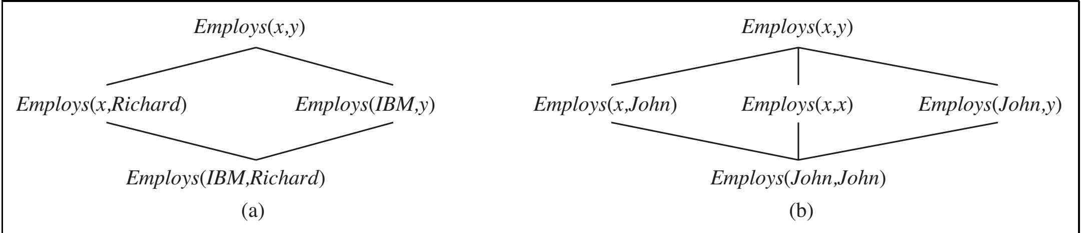


Figure 9.2 (a) The subsumption lattice whose lowest node is $Employs(IBM, Richard)$. (b) The subsumption lattice for the sentence $Employs(John, John)$.

and tens of millions of employees. Answering a query such as $Employs(x, Richard)$ with predicate indexing would require scanning the entire bucket.

For this particular query, it would help if facts were indexed both by predicate and by second argument, perhaps using a combined hash table key. Then we could simply construct the key from the query and retrieve exactly those facts that unify with the query. For other queries, such as $Employs(IBM, y)$, we would need to have indexed the facts by combining the predicate with the first argument. Therefore, facts can be stored under multiple index keys, rendering them instantly accessible to various queries that they might unify with.

Given a sentence to be stored, it is possible to construct indices for *all possible* queries that unify with it. For the fact $Employs(IBM, Richard)$, the queries are

$Employs(IBM, Richard)$	Does IBM employ Richard?
$Employs(x, Richard)$	Who employs Richard?
$Employs(IBM, y)$	Whom does IBM employ?
$Employs(x, y)$	Who employs whom?

SUBSUMPTION LATTICE

These queries form a **subsumption lattice**, as shown in Figure 9.2(a). The lattice has some interesting properties. For example, the child of any node in the lattice is obtained from its parent by a single substitution; and the "highest" common descendant of any two nodes is the result of applying their most general unifier. The portion of the lattice above any ground fact can be constructed systematically (Exercise 9.5). A sentence with repeated constants has a slightly different lattice, as shown in Figure 9.2(b). Function symbols and variables in the sentences to be stored introduce still more interesting lattice structures.

The scheme we have described works very well whenever the lattice contains a small number of nodes. For a predicate with n arguments, however, the lattice contains $O(2^n)$ nodes. If function symbols are allowed, the number of nodes is also exponential in the size of the terms in the sentence to be stored. This can lead to a huge number of indices. At some point, the benefits of indexing are outweighed by the costs of storing and maintaining all the indices. We can respond by adopting a fixed policy, such as maintaining indices only on keys composed of a predicate plus each argument, or by using an adaptive policy that creates indices to meet the demands of the kinds of queries being asked. For most AI systems, the number of facts to be stored is small enough that efficient indexing is considered a solved problem. For commercial databases, where facts number in the billions, the problem has been the subject of intensive study and technology development..

9.3 Forward Chaining

A forward-chaining algorithm for propositional definite clauses was given in Section 7.5. The idea is simple: start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made. Here, we explain how the algorithm is applied to first-order definite clauses. Definite clauses such as $Situation \Rightarrow Response$ are especially useful for systems that make inferences in response to newly arrived information. Many systems can be defined this way, and forward chaining can be implemented very efficiently.

9.3.1 First-order definite clauses

First-order definite clauses closely resemble propositional definite clauses (page 256): they are disjunctions of literals of which *exactly one is positive*. A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:

$$King(x) \land Greedy(x) \Rightarrow Evil(x) .$$
$$King(John) .$$
$$Greedy(y) .$$

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified. (Typically, we omit universal quantifiers when writing definite clauses.) Not every knowledge base can be converted into a set of definite clauses because of the single-positive-literal restriction, but many can. Consider the following problem:

> The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

We will prove that West is a criminal. First, we will represent these facts as first-order definite clauses. The next section shows how the forward-chaining algorithm solves the problem.

"... it is a crime for an American to sell weapons to hostile nations":

$$American(x) \land Weapon(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x) . \tag{9.3}$$

"Nono ... has some missiles." The sentence $\exists\, x\ Owns(Nono, x) \land Missile(x)$ is transformed into two definite clauses by Existential Instantiation, introducing a new constant M_1:

$$Owns(Nono, M_1) \tag{9.4}$$

$$Missile(M_1) \tag{9.5}$$

"All of its missiles were sold to it by Colonel West":

$$Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono) . \tag{9.6}$$

We will also need to know that missiles are weapons:

$$Missile(x) \Rightarrow Weapon(x) \tag{9.7}$$

and we must know that an enemy of America counts as "hostile":

$$Enemy(x, America) \Rightarrow Hostile(x) . \tag{9.8}$$

"West, who is American ...":

$$American(West) . \tag{9.9}$$

"The country Nono, an enemy of America ...":

$$Enemy(Nono, America) . \tag{9.10}$$

DATALOG

This knowledge base contains no function symbols and is therefore an instance of the class of **Datalog** knowledge bases. Datalog is a language that is restricted to first-order definite clauses with no function symbols. Datalog gets its name because it can represent the type of statements typically made in relational databases. We will see that the absence of function symbols makes inference much easier.

9.3.2 A simple forward-chaining algorithm

RENAMING

The first forward-chaining algorithm we consider is a simple one, shown in Figure 9.3. Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered (assuming that just one answer is required) or no new facts are added. Notice that a fact is not "new" if it is just a **renaming** of a known fact. One sentence is a renaming of another if they are identical except for the names of the variables. For example, $Likes(x, IceCream)$ and $Likes(y, IceCream)$ are renamings of each other because they differ only in the choice of x or y; their meanings are identical: everyone likes ice cream.

We use our crime problem to illustrate how FOL-FC-ASK works. The implication sentences are (9.3), (9.6), (9.7), and (9.8). Two iterations are required:

- On the first iteration, rule (9.3) has unsatisfied premises.
  Rule (9.6) is satisfied with $\{x/M_1\}$, and $Sells(West, M_1, Nono)$ is added.
  Rule (9.7) is satisfied with $\{x/M_1\}$, and $Weapon(M_1)$ is added.
  Rule (9.8) is satisfied with $\{x/Nono\}$, and $Hostile(Nono)$ is added.
- On the second iteration, rule (9.3) is satisfied with $\{x/West, y/M_1, z/Nono\}$, and $Criminal(West)$ is added.

Figure 9.4 shows the proof tree that is generated. Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB. Such a knowledge base is called a **fixed point** of the inference process. Fixed points reached by forward chaining with first-order definite clauses are similar to those for propositional forward chaining (page 258); the principal difference is that a first-order fixed point can include universally quantified atomic sentences.

FOL-FC-ASK is easy to analyze. First, it is **sound**, because every inference is just an application of Generalized Modus Ponens, which is sound. Second, it is **complete** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses. For Datalog knowledge bases, which contain no function symbols, the proof of completeness is fairly easy. We begin by counting the number of

```
function FOL-FC-ASK($KB$, $\alpha$) returns a substitution or $false$
  inputs: $KB$, the knowledge base, a set of first-order definite clauses
          $\alpha$, 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
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES($rule$)
      for each $\theta$ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
                for some $p'_1, \ldots, p'_n$ in $KB$
        $q' \leftarrow$ SUBST($\theta, q$)
        if $q'$ does not unify with some sentence already in $KB$ or $new$ then
          add $q'$ to $new$
          $\phi \leftarrow$ UNIFY($q', \alpha$)
          if $\phi$ is not $fail$ then return $\phi$
    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 STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

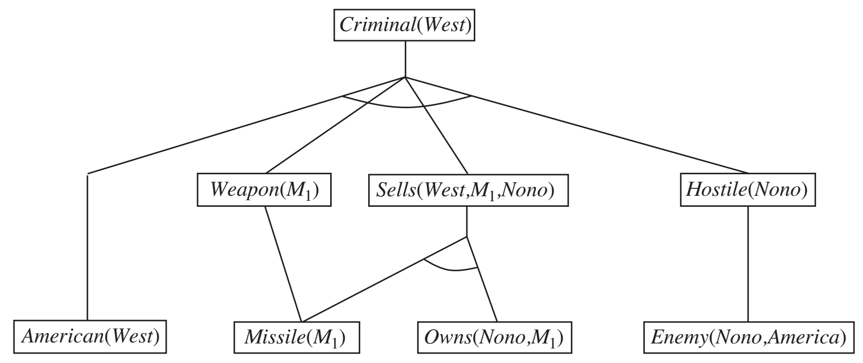


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

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 pn^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 chaining. (See page 258.) 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, 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 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

$$NatNum(0)$$
$$\forall n \;\; NatNum(n) \;\Rightarrow\; NatNum(S(n)) \,,$$

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

9.3.3 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.

PATTERN MATCHING

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

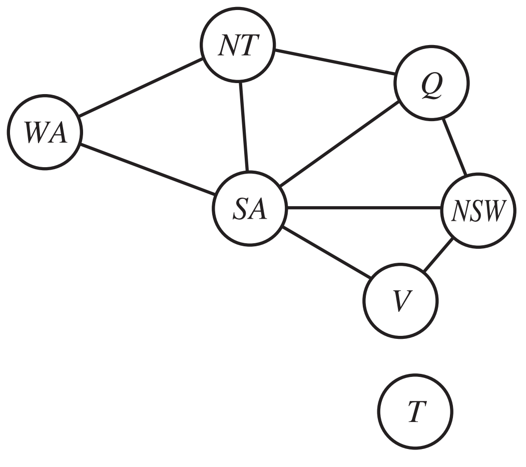
$$Missile(x) \Rightarrow Weapon(x) \,.$$

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

$$Missile(x) \wedge Owns(Nono, x) \;\Rightarrow\; Sells(West, x, Nono) \,.$$

Again, we can find all the objects owned by Nono 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 Nono and very few missiles, however, it would be better to find all the missiles first and then check whether they are owned by Nono. This is the **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 NP-hard, but good heuristics are available. For example, the **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 Nono.

CONJUNCT ORDERING



$$
\begin{aligned}
&\mathit{Diff}(wa, nt) \land \mathit{Diff}(wa, sa) \land \\
&\quad \mathit{Diff}(nt, q) \land \mathit{Diff}(nt, sa) \land \\
&\quad \mathit{Diff}(q, nsw) \land \mathit{Diff}(q, sa) \land \\
&\quad \mathit{Diff}(nsw, v) \land \mathit{Diff}(nsw, sa) \land \\
&\quad \mathit{Diff}(v, sa) \Rightarrow \mathit{Colorable}()
\end{aligned}
$$

$$
\begin{array}{ll}
\mathit{Diff}(\mathit{Red}, \mathit{Blue}) & \mathit{Diff}(\mathit{Red}, \mathit{Green}) \\
\mathit{Diff}(\mathit{Green}, \mathit{Red}) & \mathit{Diff}(\mathit{Green}, \mathit{Blue}) \\
\mathit{Diff}(\mathit{Blue}, \mathit{Red}) & \mathit{Diff}(\mathit{Blue}, \mathit{Green})
\end{array}
$$

(a) (b)

Figure 9.5 (a) Constraint graph for coloring the map of Australia. (b) The map-coloring CSP expressed as a single definite clause. Each map region is represented as a variable whose value can be one of the constants Red, Green or Blue.

The connection between pattern matching and constraint satisfaction is actually very close. We can view each conjunct as a constraint on the variables that it contains—for example, $\mathit{Missile}(x)$ is a unary constraint on x. Extending this idea, *we can express every finite-domain CSP as a single definite clause together with some associated ground facts.* Consider the map-coloring problem from Figure 6.1, shown again in Figure 9.5(a). An equivalent formulation as a single definite clause is given in Figure 9.5(b). Clearly, the conclusion $\mathit{Colorable}()$ can be inferred only if the CSP has a solution. Because CSPs in general include 3-SAT problems as special cases, we can conclude that *matching a definite clause against a set of facts is NP-hard.*

It might seem rather depressing that forward chaining has an NP-hard matching problem in its inner loop. There are three ways to cheer ourselves up:

DATA COMPLEXITY

- We can remind ourselves that most rules in real-world knowledge bases are small and simple (like the rules in our crime example) rather than large and complex (like the CSP formulation in Figure 9.5). It is common in the database world to assume that both the sizes of rules and the arities of predicates are bounded by a constant and to worry only about **data complexity**—that is, the complexity of inference as a function of the number of ground facts in the knowledge base. It is easy to show that the data complexity of forward chaining is polynomial.
- We can consider subclasses of rules for which matching is efficient. Essentially every Datalog clause can be viewed as defining a CSP, so matching will be tractable just when the corresponding CSP is tractable. Chapter 6 describes several tractable families of CSPs. For example, if the constraint graph (the graph whose nodes are variables and whose links are constraints) forms a tree, then the CSP can be solved in linear time. Exactly the same result holds for rule matching. For instance, if we remove South

Australia from the map in Figure 9.5, the resulting clause is

$$Diff(wa, nt) \land Diff(nt, q) \land Diff(q, nsw) \land Diff(nsw, v) \Rightarrow Colorable()$$

which corresponds to the reduced CSP shown in Figure 6.12 on page 224. Algorithms for solving tree-structured CSPs can be applied directly to the problem of rule matching.

- We can try to to eliminate redundant rule-matching attempts in the forward-chaining algorithm, as described next.

Incremental forward chaining

When we showed how forward chaining works on the crime example, we cheated; in particular, we omitted some of the rule matching done by the algorithm shown in Figure 9.3. For example, on the second iteration, the rule

$$Missile(x) \Rightarrow Weapon(x)$$

matches against $Missile(M_1)$ (again), and of course the conclusion $Weapon(M_1)$ is already known so nothing happens. Such redundant rule matching can be avoided if we make the following observation: *Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration $t-1$.* This is true because any inference that does not require a new fact from iteration $t-1$ could have been done at iteration $t-1$ already.

This observation leads naturally to an incremental forward-chaining algorithm where, at iteration t, we check a rule only if its premise includes a conjunct p_i that unifies with a fact p'_i newly inferred at iteration $t-1$. The rule-matching step then fixes p_i to match with p'_i, but allows the other conjuncts of the rule to match with facts from any previous iteration. This algorithm generates exactly the same facts at each iteration as the algorithm in Figure 9.3, but is much more efficient.

With suitable indexing, it is easy to identify all the rules that can be triggered by any given fact, and indeed many real systems operate in an "update" mode wherein forward chaining occurs in response to each new fact that is TELLed to the system. Inferences cascade through the set of rules until the fixed point is reached, and then the process begins again for the next new fact.

Typically, only a small fraction of the rules in the knowledge base are actually triggered by the addition of a given fact. This means that a great deal of redundant work is done in repeatedly constructing partial matches that have some unsatisfied premises. Our crime example is rather too small to show this effectively, but notice that a partial match is constructed on the first iteration between the rule

$$American(x) \land Weapon(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x)$$

and the fact $American(West)$. This partial match is then discarded and rebuilt on the second iteration (when the rule succeeds). It would be better to retain and gradually complete the partial matches as new facts arrive, rather than discarding them.

RETE

The **rete** algorithm[3] was the first to address this problem. The algorithm preprocesses the set of rules in the knowledge base to construct a sort of dataflow network in which each

[3] Rete is Latin for net. The English pronunciation rhymes with treaty.

node is a literal from a rule premise. Variable bindings flow through the network and are filtered out when they fail to match a literal. If two literals in a rule share a variable—for example, $Sells(x, y, z) \land Hostile(z)$ in the crime example—then the bindings from each literal are filtered through an equality node. A variable binding reaching a node for an n-ary literal such as $Sells(x, y, z)$ might have to wait for bindings for the other variables to be established before the process can continue. At any given point, the state of a rete network captures all the partial matches of the rules, avoiding a great deal of recomputation.

PRODUCTION SYSTEM

Rete networks, and various improvements thereon, have been a key component of so-called **production systems**, which were among the earliest forward-chaining systems in widespread use.[4] The XCON system (originally called R1; McDermott, 1982) was built with a production-system architecture. XCON contained several thousand rules for designing configurations of computer components for customers of the Digital Equipment Corporation. It was one of the first clear commercial successes in the emerging field of expert systems. Many other similar systems have been built with the same underlying technology, which has been implemented in the general-purpose language OPS-5.

COGNITIVE ARCHITECTURES

Production systems are also popular in **cognitive architectures**—that is, models of human reasoning—such as ACT (Anderson, 1983) and SOAR (Laird *et al.*, 1987). In such systems, the "working memory" of the system models human short-term memory, and the productions are part of long-term memory. On each cycle of operation, productions are matched against the working memory of facts. A production whose conditions are satisfied can add or delete facts in working memory. In contrast to the typical situation in databases, production systems often have many rules and relatively few facts. With suitably optimized matching technology, some modern systems can operate in real time with tens of millions of rules.

Irrelevant facts

The final source of inefficiency in forward chaining appears to be intrinsic to the approach and also arises in the propositional context. Forward chaining makes all allowable inferences based on the known facts, *even if they are irrelevant to the goal at hand.* In our crime example, there were no rules capable of drawing irrelevant conclusions, so the lack of directedness was not a problem. In other cases (e.g., if many rules describe the eating habits of Americans and the prices of missiles), FOL-FC-ASK will generate many irrelevant conclusions.

DEDUCTIVE DATABASES

MAGIC SET

One way to avoid drawing irrelevant conclusions is to use backward chaining, as described in Section 9.4. Another solution is to restrict forward chaining to a selected subset of rules, as in PL-FC-ENTAILS? (page 258). A third approach has emerged in the field of **deductive databases**, which are large-scale databases, like relational databases, but which use forward chaining as the standard inference tool rather than SQL queries. The idea is to rewrite the rule set, using information from the goal, so that only relevant variable bindings—those belonging to a so-called **magic set**—are considered during forward inference. For example, if the goal is $Criminal(West)$, the rule that concludes $Criminal(x)$ will be rewritten to include an extra conjunct that constrains the value of x:

$$Magic(x) \land American(x) \land Weapon(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x) .$$

[4] The word **production** in **production systems** denotes a condition–action rule.

The fact $Magic(West)$ is also added to the KB. In this way, even if the knowledge base contains facts about millions of Americans, only Colonel West will be considered during the forward inference process. The complete process for defining magic sets and rewriting the knowledge base is too complex to go into here, but the basic idea is to perform a sort of "generic" backward inference from the goal in order to work out which variable bindings need to be constrained. The magic sets approach can therefore be thought of as a kind of hybrid between forward inference and backward preprocessing.

9.4 BACKWARD CHAINING

The second major family of logical inference algorithms uses the **backward chaining** approach introduced in Section 7.5 for definite clauses. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof. We describe the basic algorithm, and then we describe how it is used in **logic programming**, which is the most widely used form of automated reasoning. We also see that backward chaining has some disadvantages compared with forward chaining, and we look at ways to overcome them. Finally, we look at the close connection between logic programming and constraint satisfaction problems.

9.4.1 A backward-chaining algorithm

Figure 9.6 shows a backward-chaining algorithm for definite clauses. FOL-BC-ASK(KB, $goal$) will be proved if the knowledge base contains a clause of the form $lhs \Rightarrow goal$, where lhs (left-hand side) is a list of conjuncts. An atomic fact like $American(West)$ is considered as a clause whose lhs is the empty list. Now a query that contains variables might be proved in multiple ways. For example, the query $Person(x)$ could be proved with the substitution $\{x/John\}$ as well as with $\{x/Richard\}$. So we implement FOL-BC-ASK as a **generator**—a function that returns multiple times, each time giving one possible result.

GENERATOR

Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the lhs of a clause must be proved. FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause to be brand-new variables, and then, if the rhs of the clause does indeed unify with the goal, proving every conjunct in the lhs, using FOL-BC-AND. That function in turn works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as we go. Figure 9.7 is the proof tree for deriving $Criminal(West)$ from sentences (9.3) through (9.10).

Backward chaining, as we have written it, is clearly a depth-first search algorithm. This means that its space requirements are linear in the size of the proof (neglecting, for now, the space required to accumulate the solutions). It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness. We will discuss these problems and some potential solutions, but first we show how backward chaining is used in logic programming systems.

```
function FOL-BC-ASK($KB$, $query$) returns a generator of substitutions
   return FOL-BC-OR($KB$, $query$, { })

generator FOL-BC-OR($KB$, $goal$, $\theta$) yields a substitution
   for each rule ($lhs \Rightarrow rhs$) in FETCH-RULES-FOR-GOAL($KB$, $goal$) do
      ($lhs$, $rhs$) ← STANDARDIZE-VARIABLES(($lhs$, $rhs$))
      for each $\theta'$ in FOL-BC-AND($KB$, $lhs$, UNIFY($rhs$, $goal$, $\theta$)) do
         yield $\theta'$

generator FOL-BC-AND($KB$, $goals$, $\theta$) yields a substitution
   if $\theta = failure$ then return
   else if LENGTH($goals$) = 0 then yield $\theta$
   else do
      $first$,$rest$ ← FIRST($goals$), REST($goals$)
      for each $\theta'$ in FOL-BC-OR($KB$, SUBST($\theta$, $first$), $\theta$) do
         for each $\theta''$ in FOL-BC-AND($KB$, $rest$, $\theta'$) do
            yield $\theta''$
```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

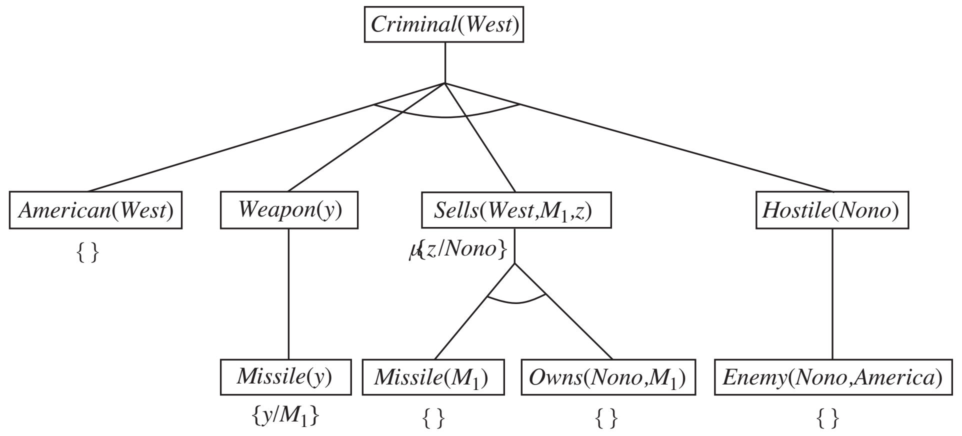


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally $Hostile(z)$, z is already bound to $Nono$.

9.4.2 Logic programming

Logic programming is a technology that comes fairly close to embodying the declarative ideal described in Chapter 7: that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge. The ideal is summed up in Robert Kowalski's equation,

$$Algorithm = Logic + Control\ .$$

PROLOG

Prolog is the most widely used logic programming language. It is used primarily as a rapid-prototyping language and for symbol-manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language (Pereira and Warren, 1980). Many expert systems have been written in Prolog for legal, medical, financial, and other domains.

Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order logic. Prolog uses uppercase letters for variables and lowercase for constants—the opposite of our convention for logic. Commas separate conjuncts in a clause, and the clause is written "backwards" from what we are used to; instead of $A \wedge B \Rightarrow C$ in Prolog we have `C :- A, B`. Here is a typical example:

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

The notation `[E|L]` denotes a list whose first element is `E` and whose rest is `L`. Here is a Prolog program for `append(X,Y,Z)`, which succeeds if list `Z` is the result of appending lists `X` and `Y`:

```
append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

In English, we can read these clauses as (1) appending an empty list with a list `Y` produces the same list `Y` and (2) `[A|Z]` is the result of appending `[A|X]` onto `Y`, provided that `Z` is the result of appending `X` onto `Y`. In most high-level languages we can write a similar recursive function that describes how to append two lists. The Prolog definition is actually much more powerful, however, because it describes a *relation* that holds among three arguments, rather than a *function* computed from two arguments. For example, we can ask the query `append(X,Y,[1,2])`: what two lists can be appended to give `[1,2]`? We get back the solutions

```
X=[]    Y=[1,2];
X=[1]   Y=[2];
X=[1,2] Y=[]
```

The execution of Prolog programs is done through depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base. Some aspects of Prolog fall outside standard logical inference:

- Prolog uses the database semantics of Section 8.2.8 rather than first-order semantics, and this is apparent in its treatment of equality and negation (see Section 9.4.5).
- There is a set of built-in functions for arithmetic. Literals using these function symbols are "proved" by executing code rather than doing further inference. For example, the

goal "`X is 4+3`" succeeds with `X` bound to 7. On the other hand, the goal "`5 is X+Y`" fails, because the built-in functions do not do arbitrary equation solving.[5]

- There are built-in predicates that have side effects when executed. These include input–output predicates and the `assert/retract` predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce confusing results—for example, if facts are asserted in a branch of the proof tree that eventually fails.
- The **occur check** is omitted from Prolog's unification algorithm. This means that some unsound inferences can be made; these are almost never a problem in practice.
- Prolog uses depth-first backward-chaining search with no checks for infinite recursion. This makes it very fast when given the right set of axioms, but incomplete when given the wrong ones.

Prolog's design represents a compromise between declarativeness and execution efficiency—inasmuch as efficiency was understood at the time Prolog was designed.

9.4.3 Efficient implementation of logic programs

The execution of a Prolog program can happen in two modes: interpreted and compiled. Interpretation essentially amounts to running the FOL-BC-ASK algorithm from Figure 9.6, with the program as the knowledge base. We say "essentially" because Prolog interpreters contain a variety of improvements designed to maximize speed. Here we consider only two.

First, our implementation had to explicitly manage the iteration over possible results generated by each of the subfunctions. Prolog interpreters have a global data structure, a stack of **choice points**, to keep track of the multiple possibilities that we considered in FOL-BC-OR. This global stack is more efficient, and it makes debugging easier, because the debugger can move up and down the stack.

CHOICE POINT

Second, our simple implementation of FOL-BC-ASK spends a good deal of time generating substitutions. Instead of explicitly constructing substitutions, Prolog has logic variables that remember their current binding. At any point in time, every variable in the program either is unbound or is bound to some value. Together, these variables and values implicitly define the substitution for the current branch of the proof. Extending the path can only add new variable bindings, because an attempt to add a different binding for an already bound variable results in a failure of unification. When a path in the search fails, Prolog will back up to a previous choice point, and then it might have to unbind some variables. This is done by keeping track of all the variables that have been bound in a stack called the **trail**. As each new variable is bound by UNIFY-VAR, the variable is pushed onto the trail. When a goal fails and it is time to back up to a previous choice point, each of the variables is unbound as it is removed from the trail.

TRAIL

Even the most efficient Prolog interpreters require several thousand machine instructions per inference step because of the cost of index lookup, unification, and building the recursive call stack. In effect, the interpreter always behaves as if it has never seen the program before; for example, it has to *find* clauses that match the goal. A compiled Prolog

[5] Note that if the Peano axioms are provided, such goals can be solved by inference within a Prolog program.

```
procedure APPEND($ax, y, az, continuation$)

    $trail \leftarrow$ GLOBAL-TRAIL-POINTER()
    if $ax$ = [ ] and UNIFY($y, az$) then CALL($continuation$)
    RESET-TRAIL($trail$)
    $a, x, z \leftarrow$ NEW-VARIABLE(), NEW-VARIABLE(), NEW-VARIABLE()
    if UNIFY($ax, [a \mid x]$) and UNIFY($az, [a \mid z]$) then APPEND($x, y, z, continuation$)
```

Figure 9.8 Pseudocode representing the result of compiling the Append predicate. The function NEW-VARIABLE returns a new variable, distinct from all other variables used so far. The procedure CALL(*continuation*) continues execution with the specified continuation.

program, on the other hand, is an inference procedure for a specific set of clauses, so it *knows* what clauses match the goal. Prolog basically generates a miniature theorem prover for each different predicate, thereby eliminating much of the overhead of interpretation. It is also possible to **open-code** the unification routine for each different call, thereby avoiding explicit analysis of term structure. (For details of open-coded unification, see Warren *et al.* (1977).)

OPEN-CODE

The instruction sets of today's computers give a poor match with Prolog's semantics, so most Prolog compilers compile into an intermediate language rather than directly into machine language. The most popular intermediate language is the Warren Abstract Machine, or WAM, named after David H. D. Warren, one of the implementers of the first Prolog compiler. The WAM is an abstract instruction set that is suitable for Prolog and can be either interpreted or translated into machine language. Other compilers translate Prolog into a high-level language such as Lisp or C and then use that language's compiler to translate to machine language. For example, the definition of the Append predicate can be compiled into the code shown in Figure 9.8. Several points are worth mentioning:

- Rather than having to search the knowledge base for Append clauses, the clauses become a procedure and the inferences are carried out simply by calling the procedure.
- As described earlier, the current variable bindings are kept on a trail. The first step of the procedure saves the current state of the trail, so that it can be restored by RESET-TRAIL if the first clause fails. This will undo any bindings generated by the first call to UNIFY.
- The trickiest part is the use of **continuations** to implement choice points. You can think of a continuation as packaging up a procedure and a list of arguments that together define what should be done next whenever the current goal succeeds. It would not do just to return from a procedure like APPEND when the goal succeeds, because it could succeed in several ways, and each of them has to be explored. The continuation argument solves this problem because it can be called each time the goal succeeds. In the APPEND code, if the first argument is empty and the second argument unifies with the third, then the APPEND predicate has succeeded. We then CALL the continuation, with the appropriate bindings on the trail, to do whatever should be done next. For example, if the call to APPEND were at the top level, the continuation would print the bindings of the variables.

CONTINUATION

Before Warren's work on the compilation of inference in Prolog, logic programming was too slow for general use. Compilers by Warren and others allowed Prolog code to achieve speeds that are competitive with C on a variety of standard benchmarks (Van Roy, 1990). Of course, the fact that one can write a planner or natural language parser in a few dozen lines of Prolog makes it somewhat more desirable than C for prototyping most small-scale AI research projects.

Parallelization can also provide substantial speedup. There are two principal sources of parallelism. The first, called **OR-parallelism**, comes from the possibility of a goal unifying with many different clauses in the knowledge base. Each gives rise to an independent branch in the search space that can lead to a potential solution, and all such branches can be solved in parallel. The second, called **AND-parallelism**, comes from the possibility of solving each conjunct in the body of an implication in parallel. AND-parallelism is more difficult to achieve, because solutions for the whole conjunction require consistent bindings for all the variables. Each conjunctive branch must communicate with the other branches to ensure a global solution.

OR-PARALLELISM

AND-PARALLELISM

9.4.4 Redundant inference and infinite loops

We now turn to the Achilles heel of Prolog: the mismatch between depth-first search and search trees that include repeated states and infinite paths. Consider the following logic program that decides if a path exists between two points on a directed graph:

```
path(X,Z) :- link(X,Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

A simple three-node graph, described by the facts `link(a,b)` and `link(b,c)`, is shown in Figure 9.9(a). With this program, the query `path(a,c)` generates the proof tree shown in Figure 9.10(a). On the other hand, if we put the two clauses in the order

```
path(X,Z) :- path(X,Y), link(Y,Z).
path(X,Z) :- link(X,Z).
```

then Prolog follows the infinite path shown in Figure 9.10(b). Prolog is therefore **incomplete** as a theorem prover for definite clauses—even for Datalog programs, as this example shows—because, for some knowledge bases, it fails to prove sentences that are entailed. Notice that forward chaining does not suffer from this problem: once `path(a,b)`, `path(b,c)`, and `path(a,c)` are inferred, forward chaining halts.

Depth-first backward chaining also has problems with redundant computations. For example, when finding a path from A_1 to J_4 in Figure 9.9(b), Prolog performs 877 inferences, most of which involve finding all possible paths to nodes from which the goal is unreachable. This is similar to the repeated-state problem discussed in Chapter 3. The total amount of inference can be exponential in the number of ground facts that are generated. If we apply forward chaining instead, at most n^2 `path(X,Y)` facts can be generated linking n nodes. For the problem in Figure 9.9(b), only 62 inferences are needed.

DYNAMIC PROGRAMMING

Forward chaining on graph search problems is an example of **dynamic programming**, in which the solutions to subproblems are constructed incrementally from those of smaller

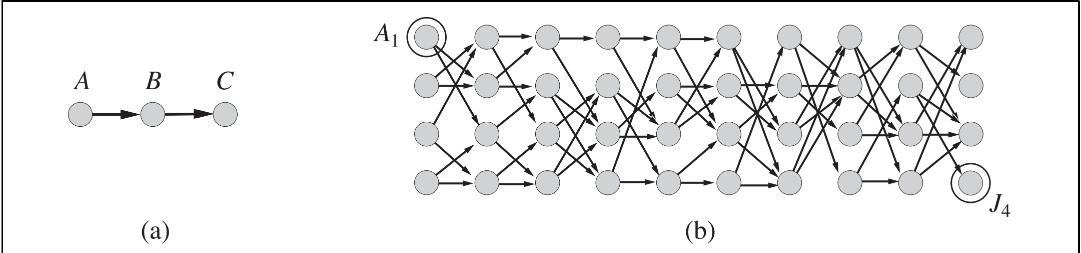


Figure 9.9 (a) Finding a path from A to C can lead Prolog into an infinite loop. (b) A graph in which each node is connected to two random successors in the next layer. Finding a path from A_1 to J_4 requires 877 inferences.

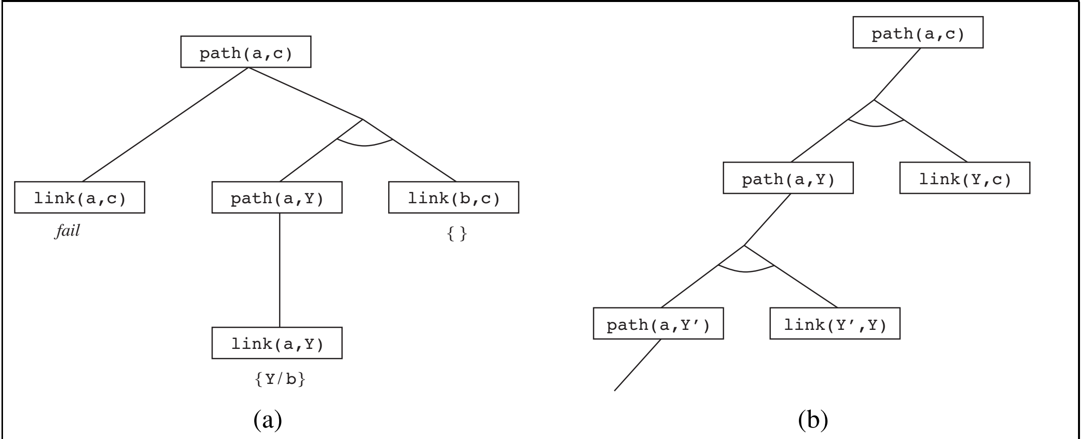


Figure 9.10 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.

subproblems and are cached to avoid recomputation. We can obtain a similar effect in a backward chaining system using **memoization**—that is, caching solutions to subgoals as they are found and then reusing those solutions when the subgoal recurs, rather than repeating the previous computation. This is the approach taken by **tabled logic programming** systems, which use efficient storage and retrieval mechanisms to perform memoization. Tabled logic programming combines the goal-directedness of backward chaining with the dynamic-programming efficiency of forward chaining. It is also complete for Datalog knowledge bases, which means that the programmer need worry less about infinite loops. (It is still possible to get an infinite loop with predicates like `father(X,Y)` that refer to a potentially unbounded number of objects.)

TABLED LOGIC PROGRAMMING

9.4.5 Database semantics of Prolog

Prolog uses database semantics, as discussed in Section 8.2.8. The unique names assumption says that every Prolog constant and every ground term refers to a distinct object, and the closed world assumption says that the only sentences that are true are those that are entailed 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:

$$Course(CS, 101),\ Course(CS, 102),\ Course(CS, 106),\ Course(EE, 101). \quad (9.11)$$

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:

$$\begin{aligned} Course(d, n) \quad \Leftrightarrow \quad & (d = CS \land n = 101) \lor (d = CS \land n = 102) \\ & \lor (d = CS \land n = 106) \lor (d = EE \land n = 101)\,. \end{aligned} \qquad (9.12)$$

COMPLETION

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:

$$\begin{aligned} x = y \quad \Leftrightarrow \quad & (x = CS \land y = CS) \lor (x = EE \land y = EE) \lor (x = 101 \land y = 101) \\ & \lor (x = 102 \land y = 102) \lor (x = 106 \land y = 106)\,. \end{aligned}$$

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 `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 `triangle(X,Y,Z)` as a predicate that holds if the three arguments are numbers that satisfy the triangle inequality:

```
triangle(X,Y,Z) :-
    X>0, Y>0, Z>0, X+Y>=Z, Y+Z>=X, X+Z>=Y.
```

If we ask Prolog the query `triangle(3,4,5)`, it succeeds. On the other hand, if we ask `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

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 >= Z >= 1`. 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.

METARULE

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.[6] Literals can contain variables, which are assumed to be universally quantified. For example, the sentence

$$\forall x \;\; \mathit{American}(x) \wedge \mathit{Weapon}(y) \wedge \mathit{Sells}(x,y,z) \wedge \mathit{Hostile}(z) \;\Rightarrow\; \mathit{Criminal}(x)$$

becomes, in CNF,

$$\neg \mathit{American}(x) \vee \neg\, \mathit{Weapon}(y) \vee \neg \mathit{Sells}(x,y,z) \vee \neg \mathit{Hostile}(z) \vee \mathit{Criminal}(x)\;.$$

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.

[6] 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.

The procedure for conversion to CNF is similar to the propositional case, which we saw on page 253. The principal difference arises from the need to eliminate existential quantifiers. We illustrate the procedure by translating the sentence "Everyone who loves all animals is loved by someone," or

$$\forall x\ [\forall y\ \mathit{Animal}(y) \Rightarrow \mathit{Loves}(x,y)] \Rightarrow [\exists y\ \mathit{Loves}(y,x)]\ .$$

The steps are as follows:

- **Eliminate implications**:

$$\forall x\ [\neg\forall y\ \neg\mathit{Animal}(y) \lor \mathit{Loves}(x,y)] \lor [\exists y\ \mathit{Loves}(y,x)]\ .$$

- **Move $\neg$ inwards**: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$\neg\forall x\ p$	becomes	$\exists x\ \neg p$
$\neg\exists x\ p$	becomes	$\forall x\ \neg p$.

  Our sentence goes through the following transformations:

$$\forall x\ [\exists y\ \neg(\neg\mathit{Animal}(y) \lor \mathit{Loves}(x,y))] \lor [\exists y\ \mathit{Loves}(y,x)]\ .$$
$$\forall x\ [\exists y\ \neg\neg\mathit{Animal}(y) \land \neg\mathit{Loves}(x,y)] \lor [\exists y\ \mathit{Loves}(y,x)]\ .$$
$$\forall x\ [\exists y\ \mathit{Animal}(y) \land \neg\mathit{Loves}(x,y)] \lor [\exists y\ \mathit{Loves}(y,x)]\ .$$

  Notice how a universal quantifier ($\forall y$) in the premise of the implication has become an existential quantifier. The sentence now reads "Either there is some animal that x doesn't love, or (if this is not the case) someone loves x." Clearly, the meaning of the original sentence has been preserved.

- **Standardize variables**: For sentences like $(\exists x\, P(x)) \lor (\exists x\, Q(x))$ which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

$$\forall x\ [\exists y\ \mathit{Animal}(y) \land \neg\mathit{Loves}(x,y)] \lor [\exists z\ \mathit{Loves}(z,x)]\ .$$

SKOLEMIZATION

- **Skolemize**: **Skolemization** is the process of removing existential quantifiers by elimination. In the simple case, it is just like the Existential Instantiation rule of Section 9.1: translate $\exists x\ P(x)$ into $P(A)$, where A is a new constant. However, we can't apply Existential Instantiation to our sentence above because it doesn't match the pattern $\exists v\ \alpha$; only parts of the sentence match the pattern. If we blindly apply the rule to the two matching parts we get

$$\forall x\ [\mathit{Animal}(A) \land \neg\mathit{Loves}(x,A)] \lor \mathit{Loves}(B,x)\ ,$$

  which has the wrong meaning entirely: it says that everyone either fails to love a particular animal A or is loved by some particular entity B. In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person. Thus, we want the Skolem entities to depend on x and z:

$$\forall x\ [\mathit{Animal}(F(x)) \land \neg\mathit{Loves}(x,F(x))] \lor \mathit{Loves}(G(z),x)\ .$$

SKOLEM FUNCTION

  Here F and G are **Skolem functions**. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears. As with Existential Instantiation, the Skolemized sentence is satisfiable exactly when the original sentence is satisfiable.

- **Drop universal quantifiers**: At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers:

  $$[\mathit{Animal}(F(x)) \land \neg \mathit{Loves}(x, F(x))] \lor \mathit{Loves}(G(z), x) \ .$$

- **Distribute $\lor$ over $\land$**:

  $$[\mathit{Animal}(F(x)) \lor \mathit{Loves}(G(z), x)] \land [\neg \mathit{Loves}(x, F(x)) \lor \mathit{Loves}(G(z), x)] \ .$$

  This step may also require flattening out nested conjunctions and disjunctions.

The sentence is now in CNF and consists of two clauses. It is quite unreadable. (It may help to explain that the Skolem function $F(x)$ refers to the animal potentially unloved by x, whereas $G(z)$ refers to someone who might love x.) Fortunately, humans seldom need look at CNF sentences—the translation process is easily automated.

9.5.2 The resolution inference rule

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule given on page 253. Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain complementary literals. Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one *unifies with* the negation of the other. Thus, we have

$$\frac{\ell_1 \lor \cdots \lor \ell_k, \qquad m_1 \lor \cdots \lor m_n}{\textsc{Subst}(\theta, \ell_1 \lor \cdots \lor \ell_{i-1} \lor \ell_{i+1} \lor \cdots \lor \ell_k \lor m_1 \lor \cdots \lor m_{j-1} \lor m_{j+1} \lor \cdots \lor m_n)}$$

where $\textsc{Unify}(\ell_i, \neg m_j) = \theta$. For example, we can resolve the two clauses

$$[\mathit{Animal}(F(x)) \lor \mathit{Loves}(G(x), x)] \quad \text{and} \quad [\neg \mathit{Loves}(u, v) \lor \neg \mathit{Kills}(u, v)]$$

by eliminating the complementary literals $\mathit{Loves}(G(x), x)$ and $\neg \mathit{Loves}(u, v)$, with unifier $\theta = \{u/G(x), v/x\}$, to produce the **resolvent** clause

$$[\mathit{Animal}(F(x)) \lor \neg \mathit{Kills}(G(x), x)] \ .$$

BINARY RESOLUTION

This rule is called the **binary resolution** rule because it resolves exactly two literals. The binary resolution rule by itself does not yield a complete inference procedure. The full resolution rule resolves subsets of literals in each clause that are unifiable. An alternative approach is to extend **factoring**—the removal of redundant literals—to the first-order case. Propositional factoring reduces two literals to one if they are *identical*; first-order factoring reduces two literals to one if they are *unifiable*. The unifier must be applied to the entire clause. The combination of binary resolution and factoring is complete.

9.5.3 Example proofs

Resolution proves that $KB \models \alpha$ by proving $KB \land \neg\alpha$ unsatisfiable, that is, by deriving the empty clause. The algorithmic approach is identical to the propositional case, described in

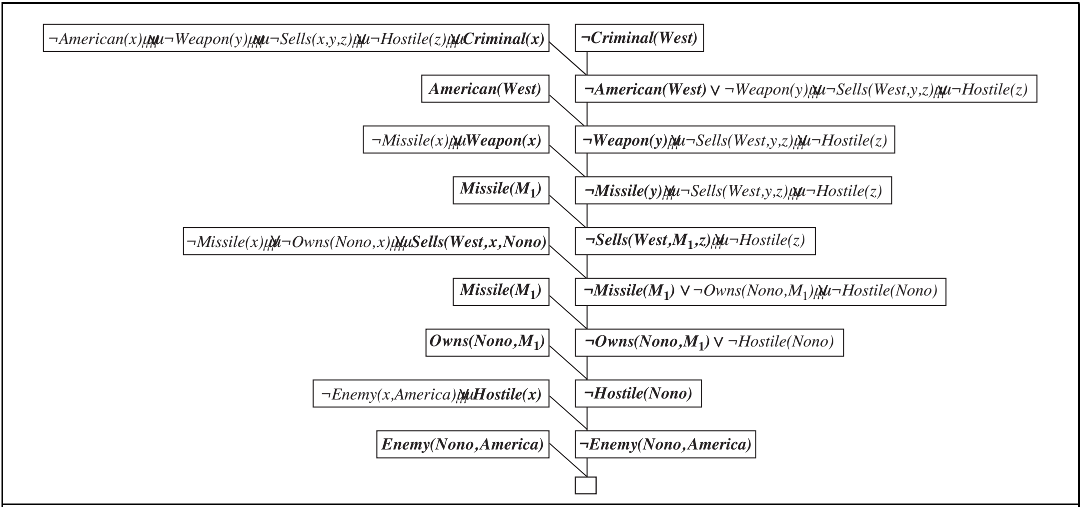

¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x,y,z) ∨ ¬Hostile(z) ∨ **Criminal(x)**
¬Criminal(West)
American(West)
¬American(West) ∨ ¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ **Weapon(x)**
¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ ¬Owns(Nono,x) ∨ **Sells(West,x,Nono)**
¬Sells(West,M1,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(M1) ∨ ¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
Owns(Nono,M1)
¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
¬Enemy(x,America) ∨ **Hostile(x)**
¬Hostile(Nono)
Enemy(Nono,America)
¬Enemy(Nono,America)

Figure 9.11 A resolution proof that West is a criminal. At each step, the literals that unify are in bold.

Figure 7.12, so we need not repeat it here. Instead, we give two example proofs. The first is the crime example from Section 9.3. The sentences in CNF are

$$\neg American(x) \lor \neg Weapon(y) \lor \neg Sells(x,y,z) \lor \neg Hostile(z) \lor Criminal(x)$$
$$\neg Missile(x) \lor \neg Owns(Nono,x) \lor Sells(West,x,Nono)$$
$$\neg Enemy(x,America) \lor Hostile(x)$$
$$\neg Missile(x) \lor Weapon(x)$$
$$Owns(Nono,M_1) \qquad Missile(M_1)$$
$$American(West) \qquad Enemy(Nono,America)\ .$$

We also include the negated goal $\neg Criminal(West)$. The resolution proof is shown in Figure 9.11. Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. This is characteristic of resolution on Horn clause knowledge bases. In fact, the clauses along the main spine correspond *exactly* to the consecutive values of the *goals* variable in the backward-chaining algorithm of Figure 9.6. This is because we always choose to resolve with a clause whose positive literal unified with the leftmost literal of the "current" clause on the spine; this is exactly what happens in backward chaining. Thus, backward chaining is just a special case of resolution with a particular control strategy to decide which resolution to perform next.

Our second example makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English, the problem is as follows:

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

A. $\forall x\ [\forall y\ \mathit{Animal}(y) \Rightarrow \mathit{Loves}(x,y)] \Rightarrow [\exists y\ \mathit{Loves}(y,x)]$

B. $\forall x\ [\exists z\ \mathit{Animal}(z) \land \mathit{Kills}(x,z)] \Rightarrow [\forall y\ \neg \mathit{Loves}(y,x)]$

C. $\forall x\ \mathit{Animal}(x) \Rightarrow \mathit{Loves}(\mathit{Jack}, x)$

D. $\mathit{Kills}(\mathit{Jack}, \mathit{Tuna}) \lor \mathit{Kills}(\mathit{Curiosity}, \mathit{Tuna})$

E. $\mathit{Cat}(\mathit{Tuna})$

F. $\forall x\ \mathit{Cat}(x) \Rightarrow \mathit{Animal}(x)$

¬G. $\neg \mathit{Kills}(\mathit{Curiosity}, \mathit{Tuna})$

Now we apply the conversion procedure to convert each sentence to CNF:

A1. $\mathit{Animal}(F(x)) \lor \mathit{Loves}(G(x), x)$

A2. $\neg \mathit{Loves}(x, F(x)) \lor \mathit{Loves}(G(x), x)$

B. $\neg \mathit{Loves}(y, x) \lor \neg \mathit{Animal}(z) \lor \neg \mathit{Kills}(x, z)$

C. $\neg \mathit{Animal}(x) \lor \mathit{Loves}(\mathit{Jack}, x)$

D. $\mathit{Kills}(\mathit{Jack}, \mathit{Tuna}) \lor \mathit{Kills}(\mathit{Curiosity}, \mathit{Tuna})$

E. $\mathit{Cat}(\mathit{Tuna})$

F. $\neg \mathit{Cat}(x) \lor \mathit{Animal}(x)$

¬G. $\neg \mathit{Kills}(\mathit{Curiosity}, \mathit{Tuna})$

The resolution proof that Curiosity killed the cat is given in Figure 9.12. In English, the proof could be paraphrased as follows:

> Suppose Curiosity did not kill Tuna. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore, Curiosity killed the cat.

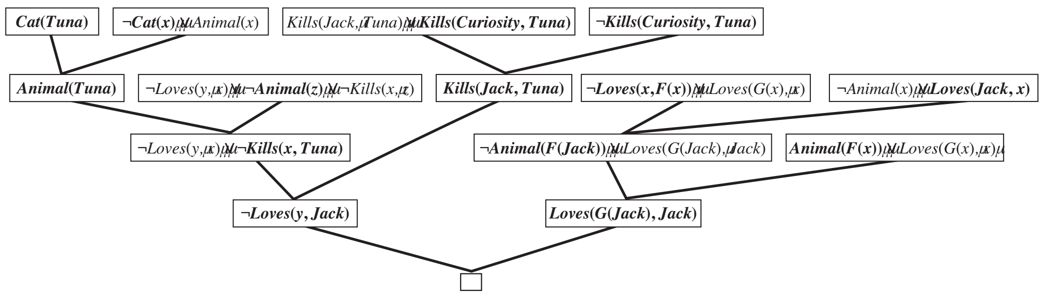

Cat(Tuna)
¬Cat(x) ∨ Animal(x)
Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)
¬Kills(Curiosity, Tuna)
Animal(Tuna)
¬Loves(y, x) ∨ ¬Animal(z) ∨ ¬Kills(x, z)
Kills(Jack, Tuna)
¬Loves(x, F(x)) ∨ Loves(G(x), x)
¬Animal(x) ∨ Loves(Jack, x)
¬Loves(y, x) ∨ ¬Kills(x, Tuna)
¬Animal(F(Jack)) ∨ Loves(G(Jack), Jack)
Animal(F(x)) ∨ Loves(G(x), x)
¬Loves(y, Jack)
Loves(G(Jack), Jack)

Figure 9.12 A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $\mathit{Loves}(G(\mathit{Jack}), \mathit{Jack})$. Notice also in the upper right, the unification of $\mathit{Loves}(x, F(x))$ and $\mathit{Loves}(\mathit{Jack}, x)$ can only succeed after the variables have been standardized apart.

The proof answers the question “Did Curiosity kill the cat?” but often we want to pose more general questions, such as “Who killed the cat?” Resolution can do this, but it takes a little more work to obtain the answer. The goal is $\exists\, w\ Kills(w, Tuna)$, which, when negated, becomes $\neg Kills(w, Tuna)$ in CNF. Repeating the proof in Figure 9.12 with the new negated goal, we obtain a similar proof tree, but with the substitution $\{w/Curiosity\}$ in one of the steps. So, in this case, finding out who killed the cat is just a matter of keeping track of the bindings for the query variables in the proof.

NONCONSTRUCTIVE PROOF

Unfortunately, resolution can produce **nonconstructive proofs** for existential goals. For example, $\neg Kills(w, Tuna)$ resolves with $Kills(Jack, Tuna) \lor Kills(Curiosity, Tuna)$ to give $Kills(Jack, Tuna)$, which resolves again with $\neg Kills(w, Tuna)$ to yield the empty clause. Notice that w has two different bindings in this proof; resolution is telling us that, yes, someone killed Tuna—either Jack or Curiosity. This is no great surprise! One solution is to restrict the allowed resolution steps so that the query variables can be bound only once in a given proof; then we need to be able to backtrack over the possible bindings. Another solution is to add a special **answer literal** to the negated goal, which becomes $\neg Kills(w, Tuna) \lor Answer(w)$. Now, the resolution process generates an answer whenever a clause is generated containing just a *single* answer literal. For the proof in Figure 9.12, this is $Answer(Curiosity)$. The nonconstructive proof would generate the clause $Answer(Curiosity) \lor Answer(Jack)$, which does not constitute an answer.

ANSWER LITERAL

9.5.4 Completeness of resolution

This section gives a completeness proof of resolution. It can be safely skipped by those who are willing to take it on faith.

REFUTATION COMPLETENESS

We show that resolution is **refutation-complete**, which means that *if* a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by the set of sentences. Hence, it can be used to find all answers to a given question, $Q(x)$, by proving that $KB \land \neg Q(x)$ is unsatisfiable.

We take it as given that any sentence in first-order logic (without equality) can be rewritten as a set of clauses in CNF. This can be proved by induction on the form of the sentence, using atomic sentences as the base case (Davis and Putnam, 1960). Our goal therefore is to prove the following: *if S is an unsatisfiable set of clauses, then the application of a finite number of resolution steps to S will yield a contradiction.*

Our proof sketch follows Robinson’s original proof with some simplifications from Genesereth and Nilsson (1987). The basic structure of the proof (Figure 9.13) is as follows:

1. First, we observe that if S is unsatisfiable, then there exists a particular set of *ground instances* of the clauses of S such that this set is also unsatisfiable (Herbrand’s theorem).
2. We then appeal to the **ground resolution theorem** given in Chapter 7, which states that propositional resolution is complete for ground sentences.
3. We then use a **lifting lemma** to show that, for any propositional resolution proof using the set of ground sentences, there is a corresponding first-order resolution proof using the first-order sentences from which the ground sentences were obtained.

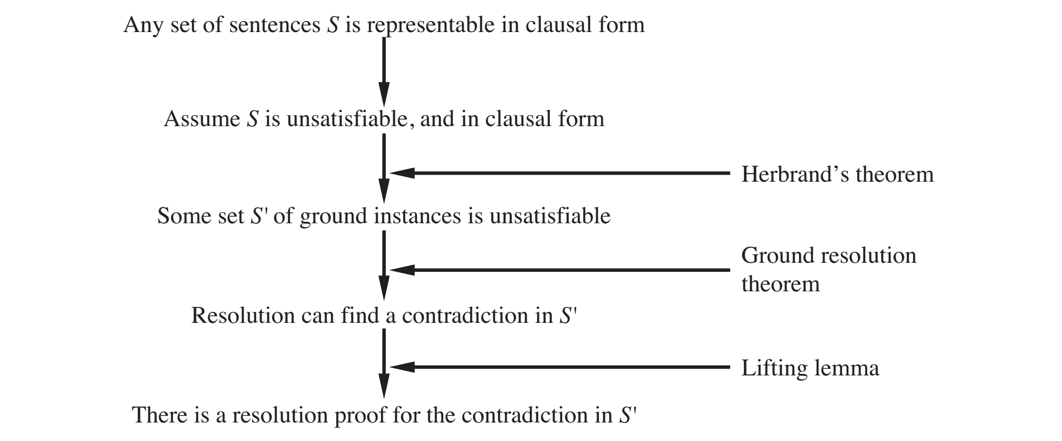

Any set of sentences S is representable in clausal form
Assume S is unsatisfiable, and in clausal form
Herbrand's theorem
Some set S' of ground instances is unsatisfiable
Ground resolution theorem
Resolution can find a contradiction in S'
Lifting lemma
There is a resolution proof for the contradiction in S'

Figure 9.13 Structure of a completeness proof for resolution.

To carry out the first step, we need three new concepts:

HERBRAND UNIVERSE

- **Herbrand universe**: If S is a set of clauses, then H_S, the Herbrand universe of S, is the set of all ground terms constructable from the following:
  a. The function symbols in S, if any.
  b. The constant symbols in S, if any; if none, then the constant symbol A.

  For example, if S contains just the clause $\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$, then H_S is the following infinite set of ground terms:

  $$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), \ldots\} .$$

SATURATION

- **Saturation**: If S is a set of clauses and P is a set of ground terms, then $P(S)$, the saturation of S with respect to P, is the set of all ground clauses obtained by applying all possible consistent substitutions of ground terms in P with variables in S.

HERBRAND BASE

- **Herbrand base**: The saturation of a set S of clauses with respect to its Herbrand universe is called the Herbrand base of S, written as $H_S(S)$. For example, if S contains solely the clause just given, then $H_S(S)$ is the infinite set of clauses

  $$\begin{array}{l}\{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B),\\ \ \neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B),\\ \ \neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B),\\ \ \neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \ldots\ \}\end{array}$$

HERBRAND'S THEOREM

These definitions allow us to state a form of **Herbrand's theorem** (Herbrand, 1930):

> If a set S of clauses is unsatisfiable, then there exists a finite subset of $H_S(S)$ that is also unsatisfiable.

Let S' be this finite subset of ground sentences. Now, we can appeal to the ground resolution theorem (page 255) to show that the **resolution closure** $RC(S')$ contains the empty clause. That is, running propositional resolution to completion on S' will derive a contradiction.

Now that we have established that there is always a resolution proof involving some finite subset of the Herbrand base of S, the next step is to show that there is a resolution

GÖDEL'S INCOMPLETENESS THEOREM

By slightly extending the language of first-order logic to allow for the **mathematical induction schema** in arithmetic, Kurt Gödel was able to show, in his **incompleteness theorem**, that there are true arithmetic sentences that cannot be proved.

The proof of the incompleteness theorem is somewhat beyond the scope of this book, occupying, as it does, at least 30 pages, but we can give a hint here. We begin with the logical theory of numbers. In this theory, there is a single constant, 0, and a single function, S (the successor function). In the intended model, $S(0)$ denotes 1, $S(S(0))$ denotes 2, and so on; the language therefore has names for all the natural numbers. The vocabulary also includes the function symbols $+$, $\times$, and $Expt$ (exponentiation) and the usual set of logical connectives and quantifiers. The first step is to notice that the set of sentences that we can write in this language can be enumerated. (Imagine defining an alphabetical order on the symbols and then arranging, in alphabetical order, each of the sets of sentences of length 1, 2, and so on.) We can then number each sentence α with a unique natural number $\#\alpha$ (the **Gödel number**). This is crucial: number theory contains a name for each of its own sentences. Similarly, we can number each possible proof P with a Gödel number $G(P)$, because a proof is simply a finite sequence of sentences.

Now suppose we have a recursively enumerable set A of sentences that are true statements about the natural numbers. Recalling that A can be named by a given set of integers, we can imagine writing in our language a sentence $\alpha(j, A)$ of the following sort:

> $\forall i \;\; i$ is not the Gödel number of a proof of the sentence whose Gödel number is j, where the proof uses only premises in A.

Then let σ be the sentence $\alpha(\#\sigma, A)$, that is, a sentence that states its own unprovability from A. (That this sentence always exists is true but not entirely obvious.)

Now we make the following ingenious argument: Suppose that σ *is* provable from A; then σ is false (because σ says it cannot be proved). But then we have a false sentence that is provable from A, so A cannot consist of only true sentences—a violation of our premise. Therefore, σ is *not* provable from A. But this is exactly what σ itself claims; hence σ is a true sentence.

So, we have shown (barring $29\frac{1}{2}$ pages) that for any set of true sentences of number theory, and in particular any set of basic axioms, there are other true sentences that *cannot* be proved from those axioms. This establishes, among other things, that we can never prove all the theorems of mathematics *within any given system of axioms*. Clearly, this was an important discovery for mathematics. Its significance for AI has been widely debated, beginning with speculations by Gödel himself. We take up the debate in Chapter 26.

proof using the clauses of S itself, which are not necessarily ground clauses. We start by considering a single application of the resolution rule. Robinson stated this lemma:

> Let C_1 and C_2 be two clauses with no shared variables, and let C'_1 and C'_2 be ground instances of C_1 and C_2. If C' is a resolvent of C'_1 and C'_2, then there exists a clause C such that (1) C is a resolvent of C_1 and C_2 and (2) C' is a ground instance of C.

LIFTING LEMMA

This is called a **lifting lemma**, because it lifts a proof step from ground clauses up to general first-order clauses. In order to prove his basic lifting lemma, Robinson had to invent unification and derive all of the properties of most general unifiers. Rather than repeat the proof here, we simply illustrate the lemma:

$$\begin{aligned}
C_1 &= \neg P(x, F(x, A)) \lor \neg Q(x, A) \lor R(x, B) \\
C_2 &= \neg N(G(y), z) \lor P(H(y), z) \\
C'_1 &= \neg P(H(B), F(H(B), A)) \lor \neg Q(H(B), A) \lor R(H(B), B) \\
C'_2 &= \neg N(G(B), F(H(B), A)) \lor P(H(B), F(H(B), A)) \\
C' &= \neg N(G(B), F(H(B), A)) \lor \neg Q(H(B), A) \lor R(H(B), B) \\
C &= \neg N(G(y), F(H(y), A)) \lor \neg Q(H(y), A) \lor R(H(y), B) \ .
\end{aligned}$$

We see that indeed C' is a ground instance of C. In general, for C'_1 and C'_2 to have any resolvents, they must be constructed by first applying to C_1 and C_2 the most general unifier of a pair of complementary literals in C_1 and C_2. From the lifting lemma, it is easy to derive a similar statement about any sequence of applications of the resolution rule:

> For any clause C' in the resolution closure of S' there is a clause C in the resolution closure of S such that C' is a ground instance of C and the derivation of C is the same length as the derivation of C'.

From this fact, it follows that if the empty clause appears in the resolution closure of S', it must also appear in the resolution closure of S. This is because the empty clause cannot be a ground instance of any other clause. To recap: we have shown that if S is unsatisfiable, then there is a finite derivation of the empty clause using the resolution rule.

The lifting of theorem proving from ground clauses to first-order clauses provides a vast increase in power. This increase comes from the fact that the first-order proof need instantiate variables only as far as necessary for the proof, whereas the ground-clause methods were required to examine a huge number of arbitrary instantiations.

9.5.5 Equality

None of the inference methods described so far in this chapter handle an assertion of the form $x = y$. Three distinct approaches can be taken. The first approach is to axiomatize equality—to write down sentences about the equality relation in the knowledge base. We need to say that equality is reflexive, symmetric, and transitive, and we also have to say that we can substitute equals for equals in any predicate or function. So we need three basic axioms, and then one for each predicate and function:

$$\begin{array}{l}
\forall x \;\; x = x \\
\forall x, y \;\; x = y \;\Rightarrow\; y = x \\
\forall x, y, z \;\; x = y \land y = z \;\Rightarrow\; x = z \\
\\
\forall x, y \;\; x = y \;\Rightarrow\; (P_1(x) \;\Leftrightarrow\; P_1(y)) \\
\forall x, y \;\; x = y \;\Rightarrow\; (P_2(x) \;\Leftrightarrow\; P_2(y)) \\
\quad\quad\vdots \\
\forall w, x, y, z \;\; w = y \land x = z \;\Rightarrow\; (F_1(w, x) = F_1(y, z)) \\
\forall w, x, y, z \;\; w = y \land x = z \;\Rightarrow\; (F_2(w, x) = F_2(y, z)) \\
\quad\quad\vdots
\end{array}$$

Given these sentences, a standard inference procedure such as resolution can perform tasks requiring equality reasoning, such as solving mathematical equations. However, these axioms will generate a lot of conclusions, most of them not helpful to a proof. So there has been a search for more efficient ways of handling equality. One alternative is to add inference rules rather than axioms. The simplest rule, **demodulation**, takes a unit clause $x = y$ and some clause α that contains the term x, and yields a new clause formed by substituting y for x within α. It works if the term within α unifies with x; it need not be exactly equal to x. Note that demodulation is directional; given $x = y$, the x always gets replaced with y, never vice versa. That means that demodulation can be used for simplifying expressions using demodulators such as $x + 0 = x$ or $x^1 = x$. As another example, given

$$\begin{array}{l}
\mathit{Father}(\mathit{Father}(x)) = \mathit{PaternalGrandfather}(x) \\
\mathit{Birthdate}(\mathit{Father}(\mathit{Father}(\mathit{Bella})), 1926)
\end{array}$$

we can conclude by demodulation

$$\mathit{Birthdate}(\mathit{PaternalGrandfather}(\mathit{Bella}), 1926) \; .$$

More formally, we have

DEMODULATION

- **Demodulation**: For any terms x, y, and z, where z appears somewhere in literal m_i and where $\text{UNIFY}(x, z) = \theta$,

$$\frac{x = y, \qquad\qquad m_1 \lor \cdots \lor m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), m_1 \lor \cdots \lor m_n)} \; .$$

  where SUBST is the usual substitution of a binding list, and $\text{SUB}(x, y, m)$ means to replace x with y everywhere that x occurs within m.

The rule can also be extended to handle non-unit clauses in which an equality literal appears:

PARAMODULATION

- **Paramodulation**: For any terms x, y, and z, where z appears somewhere in literal m_i, and where $\text{UNIFY}(x, z) = \theta$,

$$\frac{\ell_1 \lor \cdots \lor \ell_k \lor x = y, \qquad\qquad m_1 \lor \cdots \lor m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), \text{SUBST}(\theta, \ell_1 \lor \cdots \lor \ell_k \lor m_1 \lor \cdots \lor m_n)} \; .$$

For example, from

$$P(F(x, B), x) \lor Q(x) \qquad \text{and} \qquad F(A, y) = y \lor R(y)$$

we have $\theta = \text{UNIFY}(F(A, y), F(x, B)) = \{x/A, y/B\}$, and we can conclude by paramodulation the sentence

$$P(B, A) \lor Q(A) \lor R(B) .$$

Paramodulation yields a complete inference procedure for first-order logic with equality.

A third approach handles equality reasoning entirely within an extended unification algorithm. That is, terms are unifiable if they are *provably* equal under some substitution, where "provably" allows for equality reasoning. For example, the terms $1 + 2$ and $2 + 1$ normally are not unifiable, but a unification algorithm that knows that $x + y = y + x$ could unify them with the empty substitution. **Equational unification** of this kind can be done with efficient algorithms designed for the particular axioms used (commutativity, associativity, and so on) rather than through explicit inference with those axioms. Theorem provers using this technique are closely related to the CLP systems described in Section 9.4.

EQUATIONAL UNIFICATION

9.5.6 Resolution strategies

We know that repeated applications of the resolution inference rule will eventually find a proof if one exists. In this subsection, we examine strategies that help find proofs *efficiently*.

UNIT PREFERENCE

Unit preference: This strategy prefers to do resolutions where one of the sentences is a single literal (also known as a **unit clause**). The idea behind the strategy is that we are trying to produce an empty clause, so it might be a good idea to prefer inferences that produce shorter clauses. Resolving a unit sentence (such as P) with any other sentence (such as $\neg P \lor \neg Q \lor R$) always yields a clause (in this case, $\neg Q \lor R$) that is shorter than the other clause. When the unit preference strategy was first tried for propositional inference in 1964, it led to a dramatic speedup, making it feasible to prove theorems that could not be handled without the preference. **Unit resolution** is a restricted form of resolution in which every resolution step must involve a unit clause. Unit resolution is incomplete in general, but complete for Horn clauses. Unit resolution proofs on Horn clauses resemble forward chaining.

The OTTER theorem prover (Organized Techniques for Theorem-proving and Effective Research, McCune, 1992), uses a form of best-first search. Its heuristic function measures the "weight" of each clause, where lighter clauses are preferred. The exact choice of heuristic is up to the user, but generally, the weight of a clause should be correlated with its size or difficulty. Unit clauses are treated as light; the search can thus be seen as a generalization of the unit preference strategy.

SET OF SUPPORT

Set of support: Preferences that try certain resolutions first are helpful, but in general it is more effective to try to eliminate some potential resolutions altogether. For example, we can insist that every resolution step involve at least one element of a special set of clauses—the *set of support*. The resolvent is then added into the set of support. If the set of support is small relative to the whole knowledge base, the search space will be reduced dramatically.

We have to be careful with this approach because a bad choice for the set of support will make the algorithm incomplete. However, if we choose the set of support S so that the remainder of the sentences are jointly satisfiable, then set-of-support resolution is complete. For example, one can use the negated query as the set of support, on the assumption that the original knowledge base is consistent. (After all, if it is not consistent, then the fact that the query follows from it is vacuous.) The set-of-support strategy has the additional advantage of generating goal-directed proof trees that are often easy for humans to understand.

INPUT RESOLUTION

Input resolution: In this strategy, every resolution combines one of the input sentences (from the KB or the query) with some other sentence. The proof in Figure 9.11 on page 348 uses only input resolutions and has the characteristic shape of a single "spine" with single sentences combining onto the spine. Clearly, the space of proof trees of this shape is smaller than the space of all proof graphs. In Horn knowledge bases, Modus Ponens is a kind of input resolution strategy, because it combines an implication from the original KB with some other sentences. Thus, it is no surprise that input resolution is complete for knowledge bases that are in Horn form, but incomplete in the general case. The **linear resolution** strategy is a slight generalization that allows P and Q to be resolved together either if P is in the original KB or if P is an ancestor of Q in the proof tree. Linear resolution is complete.

LINEAR RESOLUTION

SUBSUMPTION

Subsumption: The subsumption method eliminates all sentences that are subsumed by (that is, more specific than) an existing sentence in the KB. For example, if $P(x)$ is in the KB, then there is no sense in adding $P(A)$ and even less sense in adding $P(A) \vee Q(B)$. Subsumption helps keep the KB small and thus helps keep the search space small.

Practical uses of resolution theorem provers

SYNTHESIS

VERIFICATION

Theorem provers can be applied to the problems involved in the **synthesis** and **verification** of both hardware and software. Thus, theorem-proving research is carried out in the fields of hardware design, programming languages, and software engineering—not just in AI.

In the case of hardware, the axioms describe the interactions between signals and circuit elements. (See Section 8.4.2 on page 309 for an example.) Logical reasoners designed specially for verification have been able to verify entire CPUs, including their timing properties (Srivas and Bickford, 1990). The AURA theorem prover has been applied to design circuits that are more compact than any previous design (Wojciechowski and Wojcik, 1983).

In the case of software, reasoning about programs is quite similar to reasoning about actions, as in Chapter 7: axioms describe the preconditions and effects of each statement. The formal synthesis of algorithms was one of the first uses of theorem provers, as outlined by Cordell Green (1969a), who built on earlier ideas by Herbert Simon (1963). The idea is to constructively prove a theorem to the effect that "there exists a program p satisfying a certain specification." Although fully automated **deductive synthesis**, as it is called, has not yet become feasible for general-purpose programming, hand-guided deductive synthesis has been successful in designing several novel and sophisticated algorithms. Synthesis of special-purpose programs, such as scientific computing code, is also an active area of research.

DEDUCTIVE SYNTHESIS

Similar techniques are now being applied to software verification by systems such as the SPIN model checker (Holzmann, 1997). For example, the Remote Agent spacecraft control program was verified before and after flight (Havelund *et al.*, 2000). The RSA public key encryption algorithm and the Boyer–Moore string-matching algorithm have been verified this way (Boyer and Moore, 1984).

9.6 SUMMARY

We have presented an analysis of logical inference in first-order logic and a number of algorithms for doing it.

- A first approach uses inference rules (**universal instantiation** and **existential instantiation**) to **propositionalize** the inference problem. Typically, this approach is slow, unless the domain is small.
- The use of **unification** to identify appropriate substitutions for variables eliminates the instantiation step in first-order proofs, making the process more efficient in many cases.
- A lifted version of **Modus Ponens** uses unification to provide a natural and powerful inference rule, **generalized Modus Ponens**. The **forward-chaining** and **backward-chaining** algorithms apply this rule to sets of definite clauses.
- Generalized Modus Ponens is complete for definite clauses, although the entailment problem is **semidecidable**. For **Datalog** knowledge bases consisting of function-free definite clauses, entailment is decidable.
- Forward chaining is used in **deductive databases**, where it can be combined with relational database operations. It is also used in **production systems**, which perform efficient updates with very large rule sets. Forward chaining is complete for Datalog and runs in polynomial time.
- Backward chaining is used in **logic programming systems**, which employ sophisticated compiler technology to provide very fast inference. Backward chaining suffers from redundant inferences and infinite loops; these can be alleviated by **memoization**.
- Prolog, unlike first-order logic, uses a closed world with the unique names assumption and negation as failure. These make Prolog a more practical programming language, but bring it further from pure logic.
- The generalized **resolution** inference rule provides a complete proof system for first-order logic, using knowledge bases in conjunctive normal form.
- Several strategies exist for reducing the search space of a resolution system without compromising completeness. One of the most important issues is dealing with equality; we showed how **demodulation** and **paramodulation** can be used.
- Efficient resolution-based theorem provers have been used to prove interesting mathematical theorems and to verify and synthesize software and hardware.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Gottlob Frege, who developed full first-order logic in 1879, based his system of inference on a collection of valid schemas plus a single inference rule, Modus Ponens. Whitehead and Russell (1910) expounded the so-called *rules of passage* (the actual term is from Herbrand (1930)) that are used to move quantifiers to the front of formulas. Skolem constants