9 INFERENCE IN FIRST-ORDER LOGIC

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

Chapter 7 defined the notion of **inference** and 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. This is significant, because more or less anything can be stated in first-order logic if you work hard enough at it.

Section 9.1 introduces inference rules for quantifiers and shows how to reduce first-order inference to propositional inference, albeit at 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; and resolution-based **theorem-proving** systems are described in Section 9.5. In general, one tries to use the most efficient method that can accommodate the facts and axioms that need to be expressed. Reasoning with fully general first-order sentences using resolution is usually less efficient than reasoning with definite clauses using forward or backward chaining.

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.

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

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 $\textsc{Subst}(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v \;\; \alpha}{\textsc{Subst}(\{v/g\}, \alpha)}$$

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

The corresponding **Existential Instantiation** rule for the existential quantifier is slightly more complicated. For any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \;\; \alpha}{\textsc{Subst}(\{v/k\}, \alpha)} \,.$$

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 the instantiation process is just giving a name to that object. Naturally, 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,

SKOLEM CONSTANT

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.

---

[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.

As well as being more complicated than Universal Instantiation, Existential Instantiation plays a slightly different role in inference. 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, once we have added the sentence $Kill(Murderer, Victim)$, we no longer need the sentence $\exists\, x\ Kill(x, Victim)$. 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

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 will 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

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

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

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

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 $Evil(John)$.

PROPOSITIONALIZATION

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 $King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$. Indeed, the inference of $Evil(John)$ from the sentences

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

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

A first-order inference rule

The inference that John is evil works like this: 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 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 $\{x/John\}$ 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 find a substitution both for the variables in the implication sentence and for the variables in the sentences to be matched. 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**: 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,

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

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 $King(John)$ $\quad$ p_1 is $King(x)$
$p_2{}'$ is $Greedy(y)$ $\quad$ p_2 is $Greedy(x)$
θ is $\{x/John, y/John\}$ $\quad$ 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,

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

This holds for the same reasons that the Universal Instantiation rule holds. 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

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

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

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

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 propositional to first-order logic. We will see in the rest of the 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 which are required to allow particular inferences to proceed. One potentially confusing point is that one sense Generalized Modus Ponens is less general than Modus Ponens (page 211): Modus Ponens allows any single α on the left-hand side of the implication, while Generalized Modus Ponens requires a special format for this sentence. It is generalized in the sense that it allows any number of P_i'.

Unification

UNIFICATION

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

UNIFIER

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 $Knows(John, x)$: whom does John know? Some answers to this query can be found 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.

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

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 z_{17} (a new variable name) without changing its meaning. Now the unification will work:

STANDARDIZING APART

$$\text{UNIFY}(Knows(John, x),\ Knows(z_{17}, Elizabeth)) = \{x/Elizabeth, z_{17}/John\} .$$

Exercise 9.7 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, $\text{UNIFY}(Knows(John, x), Knows(y, z))$ 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 of variables. 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 very 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. 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

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

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
     return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
  else if LIST?(x) and LIST?(y) then
     return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
          x, any expression
          θ, the substitution built up so far

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

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 function OP picks out the function symbol F and the function ARGS picks out the argument list (A, B).

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 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 the knowledge base in one long list; then, given a query q, call UNIFY(q, s) for every sentence s in 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, and 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

INDEXING

PREDICATE INDEXING

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[2] for efficient access.

Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol. In some applications, however, there are many clauses for a given predicate symbol. 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 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 $Employs(AIMA.org, y)$, 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 $Employs(AIMA.org, Richard)$, the queries are

$Employs(AIMA.org, Richard)$	Does AIMA.org employ Richard?
$Employs(x, Richard)$	Who employs Richard?
$Employs(AIMA.org, y)$	Whom does AIMA.org 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, 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 industrial and commercial databases, the problem has received substantial technology development.

[2] A hash table is a data structure for storing and retrieving information indexed by fixed *keys*. For practical purposes, a hash table can be considered to have constant storage and retrieval times, even when the table contains a very large number of items.

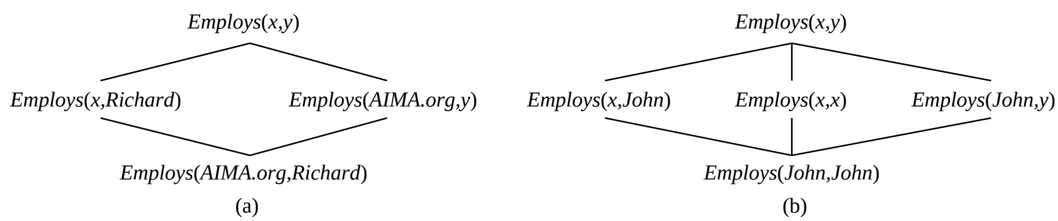


Figure 9.2 (a) The subsumption lattice whose lowest node is the sentence $Employs(AIMA.org, Richard)$. (b) The subsumption lattice for the sentence $Employs(John, John)$.

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 and how it can be implemented efficiently. 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 reasoning with forward chaining can be much more efficient than resolution theorem proving. Therefore it is often worthwhile to try to build a knowledge base using only definite clauses so that the cost of resolution can be avoided.

First-order definite clauses

First-order definite clauses closely resemble propositional definite clauses (page 217): 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.) Definite clauses are a suitable normal form for use with Generalized Modus Ponens.

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":

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

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

$$Owns(Nono, M_1) \quad (9.4)$$

$$Missile(M_1) \quad (9.5)$$

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

$$Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono) . \quad (9.6)$$

We will also need to know that missiles are weapons:

$$Missile(x) \Rightarrow Weapon(x) \quad (9.7)$$

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

$$Enemy(x, America) \Rightarrow Hostile(x) . \quad (9.8)$$

"West, who is American ...":

$$American(West) . \quad (9.9)$$

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

$$Enemy(Nono, America) . \quad (9.10)$$

DATALOG

This knowledge base contains no function symbols and is therefore an instance of the class of **Datalog** knowledge bases—that is, sets of first-order definite clauses with no function symbols. We will see that the absence of function symbols makes inference much easier.

A simple forward-chaining algorithm

RENAMING

The first forward chaining algorithm we will consider is a very simple one, as 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 will 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.

function FOL-FC-ASK(KB, α) **returns** a substitution or $false$
  inputs: KB, the knowledge base, a set of first-order definite clauses
    α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
    $new \leftarrow \{\}$
    for each sentence r **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-APART(r)
      for each θ 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(θ, q)
        if q' is not a renaming of some sentence already in KB or new **then do**
          add q' to new
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not $fail$ **then return** ϕ
    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.

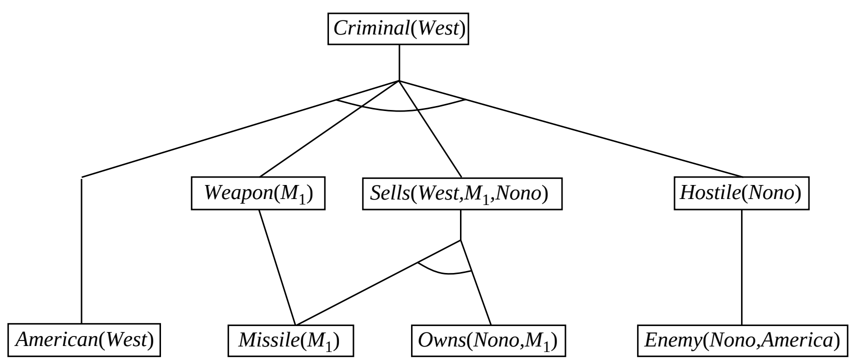


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

- 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 219); 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 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 219.) 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.

Efficient forward chaining

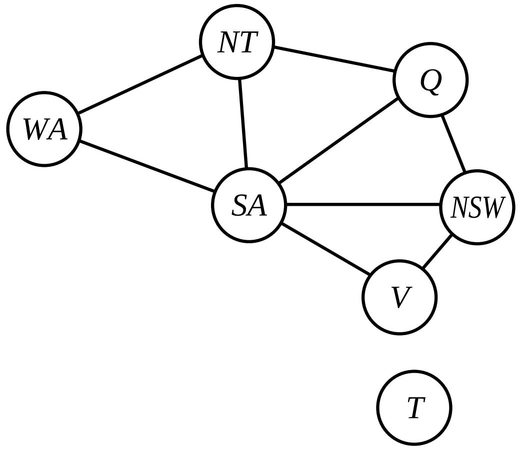
The forward chaining algorithm in Figure 9.3 is designed for ease of understanding rather than for efficiency of operation. There are three possible sources of complexity. First, the "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable 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 will address each of these sources 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

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



(a)

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

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

(b)

Figure 9.5 (a) Constraint graph for coloring the map of Australia (from Figure5.1). (b) The map-coloring CSP expressed as a single definite clause. Note that the domains of the variables are defined implicitly by the constants given in the ground facts for Diff.

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

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

Again, we can find all the objects owned by Nono in constant time per object; then, for each object, we could check if whether 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 **most constrained variable** heuristic used for CSPs in Chapter 5 would suggest ordering the conjuncts to look for missiles first if there are fewer missiles than objects that are owned by Nono.

CONJUNCT ORDERING

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 5.1, 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 3SAT 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:

- 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 database. It is easy to show that the data complexity of forward chaining is polynomial.

DATA COMPLEXITY

- 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 5 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

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

  which corresponds to the reduced CSP shown in Figure 5.11. Algorithms for solving tree-structured CSPs can be applied directly to the problem of rule matching.

- We can work hard to eliminate redundant rule matching attempts in the forward chaining algorithm, which is the subject of the next section.

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 constructing partial matches repeatedly 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 seriously. The algorithm preprocesses the set of rules in the knowledge base to construct a sort of dataflow network in which each 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 SYSTEMS

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 using 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 using 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 over a million 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. (See Section 7.5.) 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,

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

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

so the lack of directedness was not a problem. In other cases (e.g., if we have several rules describing the eating habits of Americans and the prices of missiles), FOL-FC-ASK will generate many irrelevant conclusions.

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; this approach was discussed in the propositional context. A third approach has emerged in the deductive database community, where forward chaining is the standard tool. The idea is to rewrite the rule set, using information from the goal, so that only relevant

MAGIC SET

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:

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

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. 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 will also see that backward chaining has some disadvantages compared with forward chaining, and we look at ways to overcome them. Finally, we will look at the close connection between logic programming and constraint satisfaction problems.

A backward chaining algorithm

Figure 9.6 shows a simple backward-chaining algorithm, FOL-BC-ASK. It is called with a list of goals containing a single element, the original query, and returns the set of all substitutions satisfying the query. The list of goals can be thought of as a "stack" waiting to be worked on; if *all* of them can be satisfied, then the current branch of the proof succeeds. The algorithm takes the first goal in the list and finds every clause in the knowledge base whose positive literal, or **head**, unifies with the goal. Each such clause creates a new recursive call in which the premise, or **body**, of the clause is added to the goal stack. Remember that facts are clauses with a head but no body, so when a goal unifies with a known fact, no new subgoals are added to the stack and the goal is solved. Figure 9.7 is the proof tree for deriving $Criminal(West)$ from sentences (9.3) through (9.10).

```
function FOL-BC-ASK(KB, goals, θ) returns a set of substitutions
  inputs: KB, a knowledge base
          goals, a list of conjuncts forming a query (θ already applied)
          θ, the current substitution, initially the empty substitution { }
  local variables: answers, a set of substitutions, initially empty

  if goals is empty then return {θ}
  q′ ← SUBST(θ, FIRST(goals))
  for each sentence r in KB where STANDARDIZE-APART(r) = (p1 ∧ ... ∧ pn ⇒ q)
         and θ′ ← UNIFY(q, q′) succeeds
     new_goals ← [p1, ..., pn|REST(goals)]
     answers ← FOL-BC-ASK(KB, new_goals, COMPOSE(θ′, θ)) ∪ answers
  return answers
```

Figure 9.6 A simple backward-chaining algorithm.

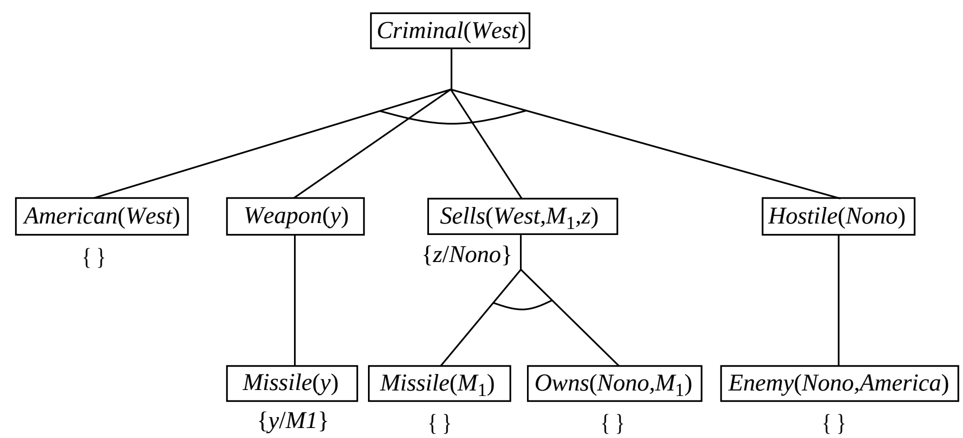


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. 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 $Nono$.

COMPOSITION

The algorithm uses **composition** of substitutions. COMPOSE(θ_1, θ_2) is the substitution whose effect is identical to the effect of applying each substitution in turn. That is,

$$\text{SUBST}(\text{COMPOSE}(\theta_1, \theta_2), p) = \text{SUBST}(\theta_2, \text{SUBST}(\theta_1, p)) .$$

In the algorithm, the current variable bindings, which are stored in θ, are composed with the bindings resulting from unifying the goal with the clause head, giving a new set of current bindings for the recursive call.

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 will see how backward chaining is used in logic programming systems.

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 by far the most widely used logic programming language. Its users number in the hundreds of thousands. 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. Clauses are written with the head preceding the body; "`:-`" is used for left-implication, commas separate literals in the body, and a period marks the end of a sentence:

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

Prolog includes "syntactic sugar" for list notation and arithmetic. As an example, 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`. This definition of `append` appears fairly similar to the corresponding definition in Lisp, but is actually much more powerful. For example, we can ask the query `append(A,B,[1,2])`: what two lists can be appended to give `[1,2]`? We get back the solutions

```
A=[]    B=[1,2]
A=[1]   B=[2]
A=[1,2] B=[]
```

The execution of Prolog programs is done via 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:

- 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 some confusing effects—for example, if facts are asserted in a branch of the proof tree that eventually fails.
- Prolog allows a form of negation called **negation as failure**. A negated goal `not P` is considered proved if the system fails to prove `P`. Thus, the sentence

  ```
  alive(X) :- not dead(X).
  ```

  can be read as "Everyone is alive if not provably dead."
- Prolog has an equality operator, `=`, but it lacks the full power of logical equality. An equality goal succeeds if the two terms are *unifiable* and fails otherwise. So `X+Y=2+3` succeeds with `X` bound to `2` and `Y` bound to `3`, but `morningstar=eveningstar` fails. (In classical logic, the latter equality might or might not be true.) No facts or rules about equality can be asserted.
- The **occur check** is omitted from Prolog's unification algorithm. This means that some unsound inferences can be made; these are seldom a problem except when using Prolog for mathematical theorem proving.

The decisions made in the design of Prolog represent a compromise between declarativeness and execution efficiency—inasmuch as efficiency was understood at the time Prolog was designed. We will return to this subject after looking at how Prolog is implemented.

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, instead of constructing the list of all possible answers for each subgoal before continuing to the next, Prolog interpreters generate one answer and a "promise" to generate the rest when the current answer has been fully explored. This promise is called a **choice point**. When the depth-first search completes its exploration of the possible solutions arising from the current answer and backs up to the choice point, the choice point is expanded to yield a new answer for the subgoal and a new choice point. This approach saves both time and space. It also provides a very simple interface for debugging because at all times there is only a single solution path under consideration.

CHOICE POINT

Second, our simple implementation of FOL-BC-ASK spends a good deal of time generating and composing substitutions. Prolog implements substitutions using logic variables

[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($trail$)
$a \leftarrow$ NEW-VARIABLE(); $x \leftarrow$ NEW-VARIABLE(); $z \leftarrow$ NEW-VARIABLE()
if UNIFY($ax, [a — x]$) and UNIFY($az, [a — z]$) **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 so far used. The procedure CALL($continuation$) continues execution with the specified continuation.

that can 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 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 implementors 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. There are several points 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.

CONTINUATIONS

- 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, 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.

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.

OR-PARALLELISM

AND-PARALLELISM

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.

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).
```

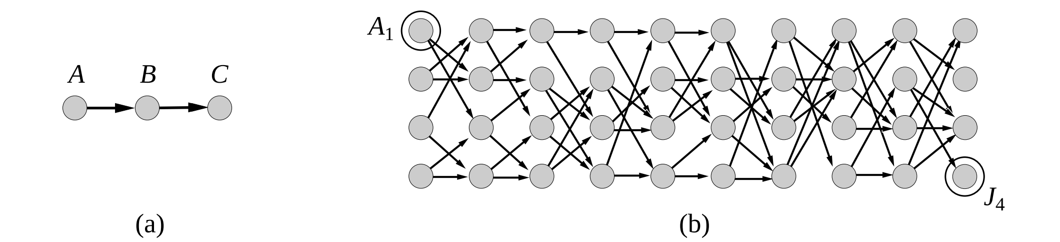


Figure 9.9 (a) Finding a path from A to C can lead Prolog into an infinite loop. (b) A graph in which each node is connected to two random successors in the next layer. Finding a path from A_1 to J_4 requires 877 inferences.

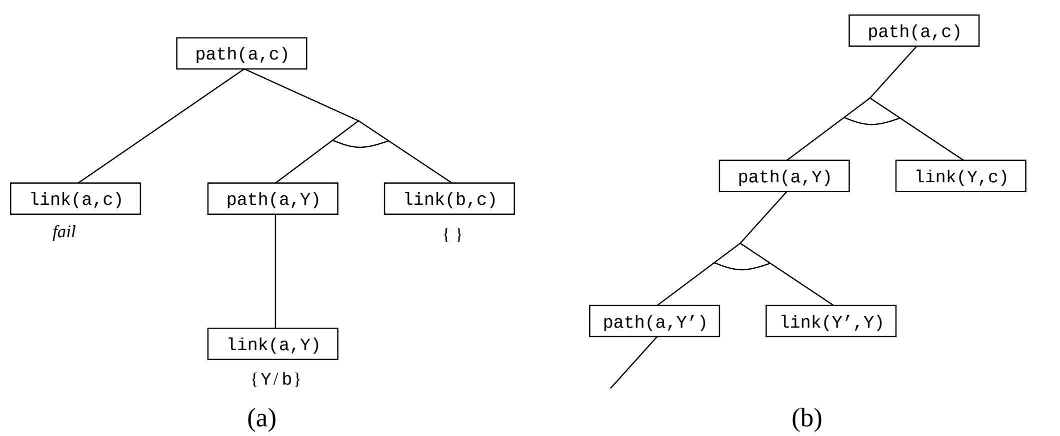


Figure 9.10 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.

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 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 programs, which means that the programmer need worry less about infinite loops.

TABLED LOGIC PROGRAMMING

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 5.3.

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.

The following clause succeeds if three numbers 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)`, this works fine. 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. The difficulty is that variables in Prolog must be in one of two states: unbound or bound to a particular term.

CONSTRAINT LOGIC PROGRAMMING

Binding a variable to a particular term can be viewed as an extreme form of constraint, namely an equality constraint. **Constraint logic programming** (CLP) allows variables to be *constrained* rather than *bound*. A solution to a constraint logic program 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 5, including heuristic conjunct ordering, backjumping, cutset conditioning, and so on. CLP systems therefore combine elements of constraint satisfaction algorithms, logic programming, and deductive databases.

CLP systems can also take advantage of the variety of CSP search optimizations described in Chapter 5, such as variable and value ordering, forward checking, and intelligent backtracking. Several systems have been defined that allow the programmer more control over the search order for inference. For example, 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.

METARULES

9.5 RESOLUTION

The last of our three families of logical systems is based on **resolution**. We saw in Chapter 7 that propositional resolution is a refutation complete inference procedure for propositional logic. In this section, we will see how to extend resolution to first-order logic.

The question of the existence of complete proof procedures is of direct concern to mathematicians. If a complete proof procedure can be found for mathematical statements, two things follow: first, all conjectures can be established mechanically; second, all of mathematics can be established as the logical consequence of a set of fundamental axioms. The question of completeness has therefore generated some of the most important mathematical work of the 20th century. In 1930, the German mathematician Kurt Gödel proved the first **completeness theorem** for first-order logic, showing that any entailed sentence has a finite proof. (No really *practical* proof procedure was found until J. A. Robinson published the resolution algorithm in 1965.) In 1931, Gödel proved an even more famous **incompleteness theorem**. The theorem states that a logical system that includes the principle of induction—without which very little of discrete mathematics can be constructed—is necessarily incomplete. Hence, there are sentences that are entailed, but have no finite proof within the system. The needle may be in the metaphorical haystack, but no procedure can guarantee that it will be found.

COMPLETENESS THEOREM

INCOMPLETENESS THEOREM

Despite Gödel's theorem, resolution-based theorem provers have been applied widely to derive mathematical theorems, including several for which no proof was known previously. Theorem provers have also been used to verify hardware designs and to generate logically correct programs, among other applications.

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

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

becomes, in CNF,

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

[6] A clause can also be represented as an implication with a conjunction of atoms on the left and a disjunction of atoms on the right, as shown in Exercise 7.12. This form, sometimes called **Kowalski form** when written with a right-to-left implication symbol (Kowalski, 1979b), is often much easier to read.

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.

The procedure for conversion to CNF is very similar to the propositional case, which we saw on page 215. The principal difference arises from the need to eliminate existential quantifiers. We will illustrate the procedure by translating the sentence "Everyone who loves all animals is loved by someone," or

$$\forall\, x \;\; [\forall\, y \;\; Animal(y) \;\Rightarrow\; Loves(x,y)] \;\Rightarrow\; [\exists\, y \;\; Loves(y,x)] \;.$$

The steps are as follows:

◇ **Eliminate implications**:

$$\forall\, x \;\; [\neg\forall\, y \;\; \neg Animal(y) \vee Loves(x,y)] \vee [\exists\, y \;\; Loves(y,x)] \;.$$

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

$$\begin{array}{lll} \neg\forall\, x \;\; p & \text{becomes} & \exists\, x \;\; \neg p \\ \neg\exists\, x \;\; p & \text{becomes} & \forall\, x \;\; \neg p \;. \end{array}$$

Our sentence goes through the following transformations:

$$\begin{array}{l} \forall\, x \;\; [\exists\, y \;\; \neg(\neg Animal(y) \vee Loves(x,y))] \vee [\exists\, y \;\; Loves(y,x)] \;. \\ \forall\, x \;\; [\exists\, y \;\; \neg\neg Animal(y) \wedge \neg Loves(x,y)] \vee [\exists\, y \;\; Loves(y,x)] \;. \\ \forall\, x \;\; [\exists\, y \;\; Animal(y) \wedge \neg Loves(x,y)] \vee [\exists\, y \;\; Loves(y,x)] \;. \end{array}$$

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 $(\forall\, x \quad P(x)) \vee (\exists\, x \quad Q(x))$ 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

$$\forall\, x \;\; [\exists\, y \;\; Animal(y) \wedge \neg Loves(x,y)] \vee [\exists\, z \;\; Loves(z,x)] \;.$$

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. If we apply this rule to our sample sentence, however, we obtain

$$\forall\, x \;\; [Animal(A) \wedge \neg Loves(x,A)] \vee Loves(B,x)$$

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:

$$\forall\, x \;\; [Animal(F(x)) \wedge \neg Loves(x,F(x))] \vee Loves(G(x),x) \;.$$

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:

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

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

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

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 $G(x)$ refers to someone who might love x.) Fortunately, humans seldom need look at CNF sentences—the translation process is easily automated.

The resolution inference rule

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule given on page 214. 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

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

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

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

BINARY RESOLUTION

The rule we have just given is 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.

Example proofs

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

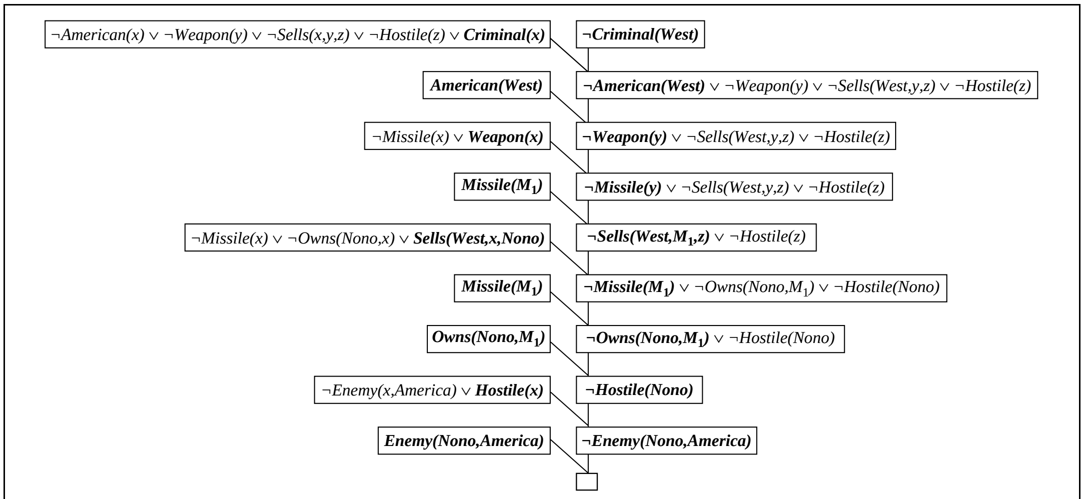


Figure 9.11 A resolution proof that West is a criminal.

Figure 7.12, so we will not repeat it here. Instead, we will 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 chose 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 really 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. $\forall x \;\; [\forall y \;\; Animal(y) \;\Rightarrow\; Loves(x,y)] \;\Rightarrow\; [\exists y \;\; Loves(y,x)]$

B. $\forall x \;\; [\exists y \;\; Animal(y) \wedge Kills(x,y)] \;\Rightarrow\; [\forall z \;\; \neg Loves(z,x)]$

C. $\forall x \;\; Animal(x) \;\Rightarrow\; Loves(Jack, x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\forall x \;\; Cat(x) \Rightarrow Animal(x)$

¬G. $\neg Kills(Curiosity, Tuna)$

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

A1. $Animal(F(x)) \vee Loves(G(x), x)$

A2. $\neg Loves(x, F(x)) \vee Loves(G(x), x)$

B. $\neg Animal(y) \vee \neg Kills(x,y) \vee \neg Loves(z,x)$

C. $\neg Animal(x) \vee Loves(Jack, x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\neg Cat(x) \vee Animal(x)$

¬G. $\neg Kills(Curiosity, Tuna)$

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.

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

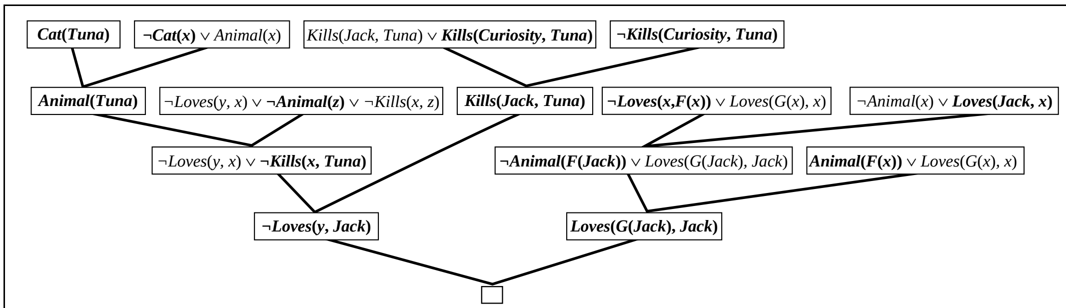


Figure 9.12 A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $Loves(G(Jack), Jack)$.

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 $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$ 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 $\neg Kills(w, Tuna) \vee Answer(w)$. 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 $Answer(Curiosity) \vee Answer(Jack)$, which does not constitute an answer.

ANSWER LITERAL

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 will 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, using the negated-goal method that we described earlier in the Chapter.

We will 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 the original proof due to Robinson, with some simplifications from Genesereth and Nilsson (1987). The basic structure of the proof is shown in Figure 9.13; it proceeds 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.

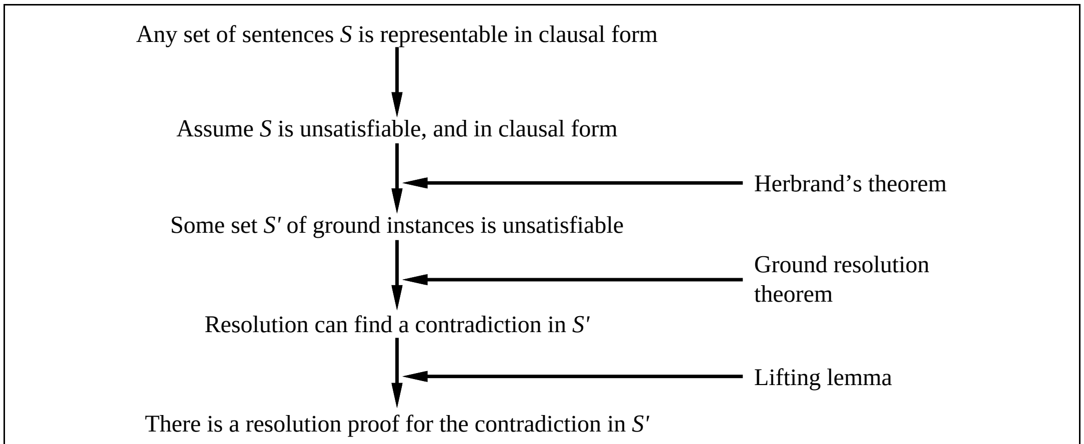


Figure 9.13 Structure of a completeness proof for resolution.

To carry out the first step, we will 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 constructible 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

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

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 217) 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, 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:

> $\forall\, i \quad i$ 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's basic lemma implies the following fact:

> 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.

Dealing with equality

None of the inference methods described so far in this chapter handle equality. There are three distinct approaches that 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:

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

Given these sentences, a standard inference procedure such as resolution can perform tasks requiring equality reasoning, such as solving mathematical equations.

Another way to deal with equality is with an additional inference rule. The simplest rule, **demodulation**, takes a unit clause $x = y$ and substitutes y for any term that unifies with x in some other clause. More formally, we have

DEMODULATION

◊ **Demodulation**: For any terms x, y, and z, where $\text{UNIFY}(x, z) = \theta$ and $m_n[z]$ is a literal containing z:

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

Demodulation is typically used for simplifying expressions using collections of assertions such as $x + 0 = x$, $x^1 = x$, and so on. 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 $\text{UNIFY}(x, z) = \theta$,

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

Unlike demodulation, paramodulation yields a complete inference procedure for first-order logic with equality.

EQUATIONAL UNIFICATION

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 some amount of 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 constraint logic programming systems described in Section 9.4.

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

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 $\neg P \vee \neg Q \vee R$) always yields a clause (in this case, $\neg Q \vee R$) 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 preference by itself does not, however, reduce the branching factor in medium-sized problems enough to make them solvable by resolution. It is, nonetheless, a useful heuristic that can be combined with other strategies.

UNIT RESOLUTION

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 knowledge bases. Unit resolution proofs on Horn knowledge bases resemble forward chaining.

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. The set-of-support strategy does just that. It starts by identifying a subset of the sentences called the **set of support**. Every resolution combines a sentence from the set of support with another sentence and adds the resolvent 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 will be complete. A common approach is to 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 proof trees that are often easy for humans to understand, because they are goal-directed.

Input resolution

INPUT RESOLUTION

LINEAR RESOLUTION

In the **input resolution** 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 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.

Subsumption

SUBSUMPTION

The **subsumption** method eliminates all sentences that are subsumed by (i.e., 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.

Theorem provers

Theorem provers (also known as automated reasoners) differ from logic programming languages in two ways. First, most logic programming languages handle only Horn clauses, whereas theorem provers accept full first-order logic. Second, Prolog programs intertwine logic and control. The programmer's choice `A :- B, C` instead of `A :- C, B` affects the execution of the program. In most theorem provers, the syntactic form chosen for sentences does not affect the results. Theorem provers still need control information to operate efficiently, but that information is usually kept distinct from the knowledge base, rather than being part of the knowledge representation itself. Most of the research in theorem provers involves finding control strategies that are generally useful, as well as increasing the speed.

Design of a theorem prover

In this section, we describe the theorem prover OTTER (Organized Techniques for Theorem-proving and Effective Research) (McCune, 1992), with particular attention to its control strategy. In preparing a problem for OTTER, the user must divide the knowledge into four parts:

- A set of clauses known as the **set of support** (or sos), which defines the important facts about the problem. Every resolution step resolves a member of the set of support against another axiom, so the search is focused on the set of support.
- A set of **usable axioms** that are outside the set of support. These provide background knowledge about the problem area. The boundary between what is part of the problem (and thus in sos) and what is background (and thus in the usable axioms) is up to the user's judgment.
- A set of equations known as **rewrites** or **demodulators**. Although demodulators are equations, they are always applied in the left to right direction. Thus, they define a canonical form into which all terms will be simplified. For example, the demodulator $x + 0 = x$ says that every term of the form $x + 0$ should be replaced by the term x.
- A set of parameters and clauses that defines the control strategy. In particular, the user specifies a heuristic function to control the search and a filtering function to eliminate some subgoals as uninteresting.

OTTER works by continually resolving an element of the set of support against one of the usable axioms. Unlike Prolog, it 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. At each step, OTTER moves the "lightest" clause in the set of support to the usable list and adds to the usable list some immediate consequences of resolving the lightest clause with elements of the usable list. OTTER halts when it has found a refutation or when there are no more clauses in the set of support. The algorithm is shown in more detail in Figure 9.14.

```
procedure OTTER(sos, usable)
   inputs: sos, a set of support—clauses defining the problem (a global variable)
           usable, background knowledge potentially relevant to the problem

   repeat
      clause ← the lightest member of sos
      move clause from sos to usable
      PROCESS(INFER(clause, usable), sos)
   until sos = [ ] or a refutation has been found
--------------------------------------------------------------------------------
function INFER(clause, usable) returns clauses

   resolve clause with each member of usable
   return the resulting clauses after applying FILTER
--------------------------------------------------------------------------------
procedure PROCESS(clauses, sos)

   for each clause in clauses do
      clause ← SIMPLIFY(clause)
      merge identical literals
      discard clause if it is a tautology
      sos ← [clause — sos]
      if clause has no literals then a refutation has been found
      if clause has one literal then look for unit refutation
```

Figure 9.14 Sketch of the OTTER theorem prover. Heuristic control is applied in the selection of the "lightest" clause and in the FILTER function that eliminates uninteresting clauses from consideration.

Extending Prolog

An alternative way to build a theorem prover is to start with a Prolog compiler and extend it to get a sound and complete reasoner for full first-order logic. This was the approach taken in the Prolog Technology Theorem Prover, or PTTP (Stickel, 1988). PTTP includes five significant changes to Prolog to restore completeness and expressiveness:

- The occurs check is put back into the unification routine to make it sound.
- The depth-first search is replaced by an iterative deepening search. This makes the search strategy complete and takes only a constant factor more time.
- Negated literals (such as $\neg P(x)$) are allowed. In the implementation, there are two separate routines, one trying to prove P and one trying to prove $\neg P$.

- A clause with n atoms is stored as n different rules. For example, $A \Leftarrow B \wedge C$ would also be stored as $\neg B \Leftarrow C \wedge \neg A$ and as $\neg C \Leftarrow B \wedge \neg A$. This technique, known as **locking**, means that the current goal need be unified with only the head of each clause, yet it still allows for proper handling of negation.

LOCKING

- Inference is made complete (even for non-Horn clauses) by the addition of the linear input resolution rule: If the current goal unifies with the negation of one of the goals on the stack, then that goal can be considered solved. This is a way of reasoning by contradiction. Suppose the original goal is P and this is reduced by a series of inferences to the goal $\neg P$. This establishes that $\neg P \Rightarrow P$, which is logically equivalent to P.

Despite these changes, PTTP retains the features that make Prolog fast. Unifications are still done by modifying variables directly, with unbinding done by unwinding the trail during backtracking. The search strategy is still based on input resolution, meaning that every resolution is against one of the clauses given in the original statement of the problem (rather than a derived clause). This makes it feasible to compile all the clauses in the original statement of the problem.

The main drawback of PTTP is that the user has to relinquish all control over the search for solutions. Each inference rule is used by the system both in its original form and in the contrapositive form. This can lead to unintuitive searches. For example, consider the rule

$$(f(x,y) = f(a,b)) \Leftarrow (x = a) \wedge (y = b)\ .$$

As a Prolog rule, this is a reasonable way to prove that two f terms are equal. But PTTP would also generate the contrapositive:

$$(x \neq a) \Leftarrow (f(x,y) \neq f(a,b)) \wedge (y = b)\ .$$

It seems that this is a wasteful way to prove that any two terms x and a are different.

Theorem provers as assistants

So far, we have thought of a reasoning system as an independent agent that has to make decisions and act on its own. Another use of theorem provers is as an assistant, providing advice to, say, a mathematician. In this mode the mathematician acts as a supervisor, mapping out the strategy for determining what to do next and asking the theorem prover to fill in the details. This alleviates the problem of semi-decidability to some extent, because the supervisor can cancel a query and try another approach if the query is taking too much time. A theorem prover can also act as a **proof-checker**, where the proof is given by a human as a series of fairly large steps; the individual inferences required to show that each step is sound are filled in by the system.

PROOF-CHECKER

SOCRATIC REASONER

A **Socratic reasoner** is a theorem prover whose ASK function is incomplete, but which can always arrive at a solution if asked the right series of questions. Thus, Socratic reasoners make good assistants, provided that there is a supervisor to make the right series of calls to ASK. ONTIC (McAllester, 1989) is a Socratic reasoning system for mathematics.

Practical uses of theorem provers

Theorem provers have come up with novel mathematical results. The SAM (Semi-Automated Mathematics) program was the first, proving a lemma in lattice theory (Guard *et al.*, 1969). The AURA program has also answered open questions in several areas of mathematics (Wos and Winker, 1983). The Boyer–Moore theorem prover (Boyer and Moore, 1979) has been used and extended over many years and was used by Natarajan Shankar to give the first fully rigorous formal proof of Gödel's Incompleteness Theorem (Shankar, 1986). The OTTER program is one of the strongest theorem provers; it has been used to solve several open questions in combinatorial logic. The most famous of these concerns **Robbins algebra**. In 1933, Herbert Robbins proposed a simple set of axioms that appeared to define Boolean algebra, but no proof of this could be found (despite serious work by several mathematicians including Alfred Tarski himself). On October 10, 1996, after eight days of computation, EQP (a version of OTTER) found a proof (McCune, 1997).

ROBBINS ALGEBRA

Theorem provers can be applied to the problems involved in the **verification** and **synthesis** of both hardware and software, because both domains can be given correct axiomatizations. 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 software, the axioms state the properties of each syntactic element of the programming language. (Reasoning about programs is quite similar to reasoning about actions in the situation calculus.) An algorithm is verified by showing that its outputs meet the specifications for all inputs. The RSA public key encryption algorithm and the Boyer–Moore string-matching algorithm have been verified this way (Boyer and Moore, 1984). In the case of hardware, the axioms describe the interactions between signals and circuit elements. (See Chapter 8 for an example.) The design of a 16-bit adder has been verified by AURA (Wojcik, 1983). Logical reasoners designed specially for verification have been able to verify entire CPUs, including their timing properties (Srivas and Bickford, 1990).

VERIFICATION

SYNTHESIS

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 Simon (1963). The idea is to prove a theorem to the effect that "there exists a program p satisfying a certain specification." If the proof is constrained to be constructive, the program can be extracted. 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 is also an active area of research. In the area of hardware synthesis, the AURA theorem prover has been applied to design circuits that are more compact than any previous design (Wojciechowski and Wojcik, 1983). For many circuit designs, propositional logic is sufficient because the set of interesting propositions is fixed by the set of circuit elements. The application of propositional inference in hardware synthesis is now a standard technique having many large-scale deployments (see, e.g., Nowick *et al.* (1993)).

DEDUCTIVE SYNTHESIS

These same techniques are now starting to be applied to software verification as well, 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).

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 for instantiating quantifiers in order to propositionalize the inference problem. Typically, this approach is very slow.
- The use of **unification** to identify appropriate substitutions for variables eliminates the instantiation step in first-order proofs, making the process much more efficient.
- 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** programs 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 programs and runs in polynomial time.
- Backward chaining is used in **logic programming systems** such as **Prolog**, 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**.
- 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. 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

SYLLOGISM

Logical inference was studied extensively in Greek mathematics. The type of inference most carefully studied by Aristotle was the **syllogism**, which is a kind of inference rule. Aristotle's syllogisms did include elements of first-order logic, such as quantification, but were restricted to unary predicates. Syllogisms were categorized by "figures" and "moods," depending on the order of the terms (which we would call predicates) in the sentences, the degree of generality (which we would today interpret through quantifiers) applied to each term, and whether each term is negated. The most fundamental syllogism is that of the first mood of the first figure:

All S are M.
All M are P.
Therefore, all S are P.

Aristotle tried to prove the validity of other syllogisms by “reducing” them to those of the first figure. He was much less precise in describing what this “reduction” should involve than he was in characterizing the syllogistic figures and moods themselves.

Gottlob Frege, who developed full first-order logic in 1879, based his system of inference on a large collection of logically valid schemas plus a single inference rule, Modus Ponens. Frege took advantage of the fact that the effect of an inference rule of the form “From P, infer Q” can be simulated by applying Modus Ponens to P along with a logically valid schema $P \Rightarrow Q$. This “axiomatic” style of exposition, using Modus Ponens plus a number of logically valid schemas, was employed by a number of logicians after Frege; most notably, it was used in *Principia Mathematica* (Whitehead and Russell, 1910).

Inference rules, as distinct from axiom schemas, were the focus of the **natural deduction** approach, introduced by Gerhard Gentzen (1934) and by Stanisław Jaśkowski (1934). Natural deduction is called “natural” because it does not require conversion to (unreadable) normal form and because its inference rules are intended to appear natural to humans. Prawitz (1965) offers a book-length treatment of natural deduction. Gallier (1986) uses Gentzen’s approach to expound the theoretical underpinnings of automated deduction.

The invention of clausal form was a crucial step in the development of a deep mathematical analysis of first-order logic. 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 and Skolem functions were introduced, appropriately enough, by Thoralf Skolem (1920). The general procedure for skolemization is given by Skolem (1928), along with the important notion of the Herbrand universe.

Herbrand’s theorem, named after the French logician Jacques Herbrand (1930), has played a vital role in the development of automated reasoning methods, both before and after Robinson’s introduction of resolution. This is reflected in our reference to the “Herbrand universe” rather than the “Skolem universe,” even though Skolem really invented the concept. Herbrand can also be regarded as the inventor of unification. Gödel (1930) built on the ideas of Skolem and Herbrand to show that first-order logic has a complete proof procedure. Alan Turing (1936) and Alonzo Church (1936) simultaneously showed, using very different proofs, that validity in first-order logic was not decidable. The excellent text by Enderton (1972) explains all of these results in a rigorous yet moderately understandable fashion.

Although McCarthy (1958) had suggested the use of first-order logic for representation and reasoning in AI, the first such systems were developed by logicians interested in mathematical theorem proving. It was Abraham Robinson who proposed the use of propositionalization and Herbrand’s theorem, and Gilmore (1960) who wrote the first program based on this approach. Davis and Putnam (1960) used clausal form and produced a program that attempted to find refutations by substituting members of the Herbrand universe for variables to produce ground clauses and then looking for propositional inconsistencies among the ground clauses. Prawitz (1960) developed the key idea of letting the quest for propositional inconsistency drive the search process, and generating terms from the Herbrand universe only when it was necessary to do so in order to establish propositional inconsistency. After further development by other researchers, this idea led J. A. Robinson (no relation) to develop the resolution method (Robinson, 1965). The so-called inverse method developed at about the same time by the Soviet researcher S. Maslov (1964, 1967), based on somewhat different principles, offers similar computational advantages over propositionalization. Wolfgang Bibel's (1981) **connection method** can be viewed as an extension of this approach.

After the development of resolution, work on first-order inference proceeded in several different directions. In AI, resolution was adopted for question-answering systems by Cordell Green and Bertram Raphael (1968). A somewhat less formal approach was taken by Carl Hewitt (1969). His PLANNER language, although never fully implemented, was a precursor to logic programming and included directives for forward and backward chaining and for negation as failure. A subset known as MICRO-PLANNER (Sussman and Winograd, 1970) was implemented and used in the SHRDLU natural language understanding system (Winograd, 1972). Early AI implementations put a good deal of effort into data structures that would allow efficient retrieval of facts; this work is covered in AI programming texts (Charniak *et al.*, 1987; Norvig, 1992; Forbus and de Kleer, 1993).

By the early 1970s, **forward chaining** was well established in AI as an easily understandable alternative to resolution. It was used in a wide variety of systems, ranging from Nevins's geometry theorem prover (Nevins, 1975) to the R1 expert system for VAX configuration (McDermott, 1982). AI applications typically involved large numbers of rules, so it was important to develop efficient rule-matching technology, particularly for incremental updates. The technology for **production systems** was developed to support such applications. The production system language OPS-5 (Forgy, 1981; Brownston *et al.*, 1985) was used for R1 and for the SOAR cognitive architecture (Laird *et al.*, 1987). OPS-5 incorporated the rete match process (Forgy, 1982). SOAR, which generates new rules to cache the results of previous computations, can create very large rule sets—over 1,000,000 rules in the case of the TACAIR-SOAR system for controlling simulated fighter aircraft (Jones *et al.*, 1998). CLIPS (Wygant, 1989) was a C-based production system language developed at NASA that allowed better integration with other software, hardware, and sensor systems and was used for spacecraft automation and several military applications.

The area of research known as **deductive databases** has also contributed a great deal to our understanding of forward inference. It began with a workshop in Toulouse in 1977, organized by Jack Minker, that brought together experts in logical inference and database systems (Gallaire and Minker, 1978). A recent historical survey (Ramakrishnan and Ullman, 1995) says, "Deductive [database] systems are an attempt to adapt Prolog, which has a 'small data' view of the world, to a 'large data' world." Thus, it aims to meld relational database technology, which is designed for retrieving large *sets* of facts, with Prolog-based inference technology, which typically retrieves one fact at a time. Texts on deductive databases include Ullman (1989) and Ceri *et al.* (1990).

Influential work by Chandra and Harel (1980) and Ullman (1985) led to the adoption of Datalog as a standard language for deductive databases. "Bottom-up" inference, or forward chaining, also became the standard—partly because it avoids the problems with nontermination and redundant computation that occur with backward chaining and partly because it has a more natural implementation in terms of the basic relational database operations. The development of the **magic sets** technique for rule rewriting by Bancilhon *et al.* (1986) allowed forward chaining to borrow the advantage of goal-directedness from backward chaining. Equalizing the arms race, tabled logic programming methods (see page 313) borrow the advantage of dynamic programming from forward chaining.

Much of our understanding of the complexity of logical inference has come from the deductive database community. Chandra and Merlin (1977) first showed that matching a single nonrecursive rule (a **conjunctive query** in database terminology) can be NP-hard. Kuper and Vardi (1993) proposed **data complexity**—that is, complexity as a function of database size, viewing rule size as constant—as a suitable measure for query answering. Gottlob *et al.* (1999b) discuss the connection between conjunctive queries and constraint satisfaction, showing how hypertree decomposition can optimize the matching process.

CONJUNCTIVE QUERY

DATA COMPLEXITY

As mentioned earlier, **backward chaining** for logical inference appeared in Hewitt's PLANNER language (1969). Logic programming *per se* evolved independently of this effort. A restricted form of linear resolution called **SL-resolution** was developed by Kowalski and Kuehner (1971), building on Loveland's **model elimination** technique (1968); when applied to definite clauses, it becomes **SLD-resolution**, which lends itself to the interpretation of definite clauses as programs (Kowalski, 1974, 1979a, 1979b). Meanwhile, in 1972, the French researcher Alain Colmerauer had developed and implemented **Prolog** for the purpose of parsing natural language—Prolog's clauses were intended initially as context-free grammar rules (Roussel, 1975; Colmerauer *et al.*, 1973). Much of the theoretical background for logic programming was developed by Kowalski, working with Colmerauer. The semantic definition using least fixed points is due to Van Emden and Kowalski (1976). Kowalski (1988) and Cohen (1988) provide good historical overviews of the origins of Prolog. *Foundations of Logic Programming* (Lloyd, 1987) is a theoretical analysis of the underpinnings of Prolog and other logic programming languages.

SL-RESOLUTION

SLD-RESOLUTION

Efficient Prolog compilers are generally based on the Warren Abstract Machine (WAM) model of computation developed by David H. D. Warren (1983). Van Roy (1990) showed that the application of additional compiler techniques, such as type inference, made Prolog programs competitive with C programs in terms of speed. The Japanese Fifth Generation project, a 10-year research effort beginning in 1982, was based completely on Prolog as the means to develop intelligent systems.

Methods for avoiding unnecessary looping in recursive logic programs were developed independently by Smith *et al.* (1986) and Tamaki and Sato (1986). The latter paper also included memoization for logic programs, a method developed extensively as **tabled logic programming** by David S. Warren. Swift and Warren (1994) show how to extend the WAM to handle tabling, enabling Datalog programs to execute an order of magnitude faster than forward-chaining deductive database systems.

Early theoretical work on constraint logic programming was done by Jaffar and Lassez (1987). Jaffar *et al.* (1992a) developed the CLP(R) system for handling real-valued constraints. Jaffar *et al.* (1992b) generalized the WAM to produce the CLAM (Constraint Logic Abstract Machine) for specifying implementations of CLP. Ait-Kaci and Podelski (1993) describe a sophisticated language called LIFE, which combines CLP with functional programming and with inheritance reasoning. Kohn (1991) describes an ambitious project to use constraint logic programming as the foundation for a real-time control architecture, with applications to fully automatic pilots.

There are a number of textbooks on logic programming and Prolog. *Logic for Problem Solving* (Kowalski, 1979b) is an early text on logic programming in general. Prolog texts include Clocksin and Mellish (1994), Shoham (1994), and Bratko (2001). Marriott and Stuckey (1998) provide excellent coverage of CLP. Until its demise in 2000, the *Journal of Logic Programming* was the journal of record; it has now been replaced by *Theory and Practice of Logic Programming*. Logic programming conferences include the International Conference on Logic Programming (ICLP) and the International Logic Programming Symposium (ILPS).

Research into **mathematical theorem proving** began even before the first complete first-order systems were developed. Herbert Gelernter's Geometry Theorem Prover (Gelernter, 1959) used heuristic search methods combined with diagrams for pruning false subgoals and was able to prove some quite intricate results in Euclidean geometry. Since that time, however, there has not been very much interaction between theorem proving and AI.

Early work concentrated on completeness. Following Robinson's seminal paper, the demodulation and paramodulation rules for equality reasoning were introduced by Wos *et al.* (1967) and Wos and Robinson (1968), respectively. These rules were also developed independently in the context of term rewriting systems (Knuth and Bendix, 1970). The incorporation of equality reasoning into the unification algorithm is due to Gordon Plotkin (1972); it was also a feature of QLISP (Sacerdoti *et al.*, 1976). Jouannaud and Kirchner (1991) survey equational unification from a term rewriting perspective. Efficient algorithms for standard unification were developed by Martelli and Montanari (1976) and Paterson and Wegman (1978).

In addition to equality reasoning, theorem provers have incorporated a variety of special-purpose decision procedures. Nelson and Oppen (1979) proposed an influential scheme for integrating such procedures into a general reasoning system; other methods include Stickel's (1985) "theory resolution" and Manna and Waldinger's (1986) "special relations."

A number of control strategies have been proposed for resolution, beginning with the unit preference strategy (Wos *et al.*, 1964). The set of support strategy was proposed by Wos *et al.* (1965), to provide a degree of goal-directedness in resolution. Linear resolution first appeared in Loveland (1970). Genesereth and Nilsson (1987, Chapter 5) provide a short but thorough analysis of a wide variety of control strategies.

Guard *et al.* (1969) describe the early SAM theorem prover, which helped to solve an open problem in lattice theory. Wos and Winker (1983) give an overview of the contributions of the AURA theorem prover toward solving open problems in various areas of mathematics and logic. McCune (1992) follows up on this, recounting the accomplishments of AURA's successor, OTTER, in solving open problems. Weidenbach (2001) describes SPASS, one of the strongest current theorem provers. *A Computational Logic* (Boyer and Moore, 1979) is the basic reference on the Boyer-Moore theorem prover. Stickel (1988) covers the Prolog Technology Theorem Prover (PTTP), which combines the advantages of Prolog compilation with the completeness of model elimination (Loveland, 1968). SETHEO (Letz *et al.*, 1992) is another widely used theorem prover based on this approach; it can perform several million inferences per second on 2000-model workstations. LEANTAP (Beckert and Posegga, 1995) is an efficient theorem prover implemented in only 25 lines of Prolog.

Early work in automated program synthesis was done by Simon (1963), Green (1969a), and Manna and Waldinger (1971). The transformational system of Burstall and Darlington (1977) used equational reasoning for recursive program synthesis. KIDS (Smith, 1990, 1996) is one of the strongest modern systems; it operates as an expert assistant. Manna and Waldinger (1992) give a tutorial introduction to the current state of the art, with emphasis on their own deductive approach. *Automating Software Design* (Lowry and McCartney, 1991) collects a number of papers in the area. The use of logic in hardware design is surveyed by Kern and Greenstreet (1999); Clarke *et al.* (1999) cover model checking for hardware verification.

Computability and Logic (Boolos and Jeffrey, 1989) is a good reference on completeness and undecidability. Many early papers in mathematical logic are to be found in *From Frege to Gödel: A Source Book in Mathematical Logic* (van Heijenoort, 1967). The journal of record for the field of pure mathematical logic (as opposed to automated deduction) is *The Journal of Symbolic Logic*. Textbooks geared toward automated deduction include the classic *Symbolic Logic and Mechanical Theorem Proving* (Chang and Lee, 1973), as well as more recent works by Wos *et al.* (1992), Bibel (1993), and Kaufmann *et al.* (2000). The anthology *Automation of Reasoning* (Siekmann and Wrightson, 1983) includes many important early papers on automated deduction. Other historical surveys have been written by Loveland (1984) and Bundy (1999). The principal journal for the field of theorem proving is the *Journal of Automated Reasoning*; the main conference is the annual Conference on Automated Deduction (CADE). Research in theorem proving is also strongly related to the use of logic in analyzing programs and programming languages, for which the principal conference is Logic in Computer Science.

EXERCISES

9.1 Prove from first principles that Universal Instantiation is sound and that Existential Instantiation produces an inferentially equivalent knowledge base.

EXISTENTIAL INTRODUCTION

9.2 From $Likes(Jerry, IceCream)$ it seems reasonable to infer $\exists\, x \;\; Likes(x, IceCream)$. Write down a general inference rule, **Existential Introduction**, that sanctions this inference. State carefully the conditions that must be satisfied by the variables and terms involved.

9.3 Suppose a knowledge base contains just one sentence, $\exists\, x \; AsHighAs(x, Everest)$. Which of the following are legitimate results of applying Existential Instantiation?

a. $AsHighAs(Everest, Everest)$.

b. $AsHighAs(Kilimanjaro, Everest)$.

c. $AsHighAs(Kilimanjaro, Everest) \land AsHighAs(BenNevis, Everest)$
(after two applications).

9.4 For each pair of atomic sentences, give the most general unifier if it exists:

a. $P(A, B, B)$, $P(x, y, z)$.
b. $Q(y, G(A, B))$, $Q(G(x, x), y)$.
c. $Older(Father(y), y)$, $Older(Father(x), John)$.
d. $Knows(Father(y), y)$, $Knows(x, x)$.

9.5 Consider the subsumption lattices shown in Figure 9.2.

a. Construct the lattice for the sentence $Employs(Mother(John), Father(Richard))$.
b. Construct the lattice for the sentence $Employs(IBM, y)$ ("Everyone works for IBM"). Remember to include every kind of query that unifies with the sentence.
c. Assume that STORE indexes each sentence under every node in its subsumption lattice. Explain how FETCH should work when some of these sentences contain variables; use as examples the sentences in (a) and (b) and the query $Employs(x, Father(x))$.

9.6 Suppose we put into a logical database a segment of the U.S. census data listing the age, city of residence, date of birth, and mother of every person, using social security numbers as identifying constants for each person. Thus, George's age is given by $Age($443-65-1282$, 56)$. Which of the indexing schemes S1–S5 following enable an efficient solution for which of the queries Q1–Q4 (assuming normal backward chaining)?

- ♢ **S1**: an index for each atom in each position.
- ♢ **S2**: an index for each first argument.
- ♢ **S3**: an index for each predicate atom.
- ♢ **S4**: an index for each *combination* of predicate and first argument.
- ♢ **S5**: an index for each *combination* of predicate and second argument and an index for each first argument (nonstandard).

- ♢ **Q1**: $Age($443-44-4321$, x)$
- ♢ **Q2**: $ResidesIn(x, Houston)$
- ♢ **Q3**: $Mother(x, y)$
- ♢ **Q4**: $Age(x, 34) \wedge ResidesIn(x, TinyTownUSA)$

9.7 One might suppose that we can avoid the problem of variable conflict in unification during backward chaining by standardizing apart all of the sentences in the knowledge base once and for all. Show that, for some sentences, this approach cannot work. (*Hint*: Consider a sentence, one part of which unifies with another.)

9.8 Explain how to write any given 3-SAT problem of arbitrary size using a single first-order definite clause and no more than 30 ground facts.

9.9 Write down logical representations for the following sentences, suitable for use with Generalized Modus Ponens:

a. Horses, cows, and pigs are mammals.

b. An offspring of a horse is a horse.

c. Bluebeard is a horse.

d. Bluebeard is Charlie's parent.

e. Offspring and parent are inverse relations.

f. Every mammal has a parent.

9.10 In this question we will use the sentences you wrote in Exercise 9.9 to answer a question using a backward-chaining algorithm.

a. Draw the proof tree generated by an exhaustive backward-chaining algorithm for the query $\exists\, h \;\; Horse(h)$, where clauses are matched in the order given.

b. What do you notice about this domain?

c. How many solutions for h actually follow from your sentences?

d. Can you think of a way to find all of them? (*Hint*: You might want to consult Smith *et al.* (1986).)

9.11 A popular children's riddle is "Brothers and sisters have I none, but that man's father is my father's son." Use the rules of the family domain (Chapter 8) to show who that man is. You may apply any of the inference methods described in this chapter. Why do you think that this riddle is difficult?

9.12 Trace the execution of the backward chaining algorithm in Figure 9.6 when it is applied to solve the crime problem. Show the sequence of values taken on by the $goals$ variable, and arrange them into a tree.

9.13 The following Prolog code defines a predicate `P`:

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

a. Show proof trees and solutions for the queries `P(A,[1,2,3])` and `P(2,[1,A,3])`.

b. What standard list operation does `P` represent?

9.14 In this exercise, we will look at sorting in Prolog.

a. Write Prolog clauses that define the predicate `sorted(L)`, which is true if and only if list `L` is sorted in ascending order.

b. Write a Prolog definition for the predicate `perm(L,M)`, which is true if and only if `L` is a permutation of `M`.

c. Define `sort(L,M)` (`M` is a sorted version of `L`) using `perm` and `sorted`.

d. Run `sort` on longer and longer lists until you lose patience. What is the time complexity of your program?

e. Write a faster sorting algorithm, such as insertion sort or quicksort, in Prolog.

9.15 In this exercise, we will look at the recursive application of rewrite rules, using logic programming. A rewrite rule (or **demodulator** in OTTER terminology) is an equation with a specified direction. For example, the rewrite rule $x+0 \rightarrow x$ suggests replacing any expression that matches $x + 0$ with the expression x. The application of rewrite rules is a central part of equational reasoning systems. We will use the predicate `rewrite(X,Y)` to represent rewrite rules. For example, the earlier rewrite rule is written as `rewrite(X+0,X)`. Some terms are *primitive* and cannot be further simplified; thus, we will write `primitive(0)` to say that 0 is a primitive term.

- **a**. Write a definition of a predicate `simplify(X,Y)`, that is true when `Y` is a simplified version of `X`—that is, when no further rewrite rules are applicable to any subexpression of `Y`.
- **b**. Write a collection of rules for the simplification of expressions involving arithmetic operators, and apply your simplification algorithm to some sample expressions.
- **c**. Write a collection of rewrite rules for symbolic differentiation, and use them along with your simplification rules to differentiate and simplify expressions involving arithmetic expressions, including exponentiation.

9.16 In this exercise, we will consider the implementation of search algorithms in Prolog. Suppose that `successor(X,Y)` is true when state `Y` is a successor of state `X`; and that `goal(X)` is true when `X` is a goal state. Write a definition for `solve(X,P)`, which means that `P` is a path (list of states) beginning with `X`, ending in a goal state, and consisting of a sequence of legal steps as defined by `successor`. You will find that depth-first search is the easiest way to do this. How easy would it be to add heuristic search control?

9.17 How can resolution be used to show that a sentence is valid? Unsatisfiable?

9.18 From "Horses are animals," it follows that "The head of a horse is the head of an animal." Demonstrate that this inference is valid by carrying out the following steps:

- **a**. Translate the premise and the conclusion into the language of first-order logic. Use three predicates: $HeadOf(h, x)$ (meaning "h is the head of x"), $Horse(x)$, and $Animal(x)$.
- **b**. Negate the conclusion, and convert the premise and the negated conclusion into conjunctive normal form.
- **c**. Use resolution to show that the conclusion follows from the premise.

9.19 Here are two sentences in the language of first-order logic:

(A): $\forall x \;\; \exists y \;\; (x \geq y)$

(B): $\exists y \;\; \forall x \;\; (x \geq y)$

- **a**. Assume that the variables range over all the natural numbers $0, 1, 2, \ldots, \infty$ and that the "$\geq$" predicate means "is greater than or equal to." Under this interpretation, translate (A) and (B) into English.
- **b**. Is (A) true under this interpretation?

c. Is (B) true under this interpretation?

d. Does (A) logically entail (B)?

e. Does (B) logically entail (A)?

f. Using resolution, try to prove that (A) follows from (B). Do this even if you think that (B) does not logically entail (A); continue until the proof breaks down and you cannot proceed (if it does break down). Show the unifying substitution for each resolution step. If the proof fails, explain exactly where, how, and why it breaks down.

g. Now try to prove that (B) follows from (A).

9.20 Resolution can produce nonconstructive proofs for queries with variables, so we had to introduce special mechanisms to extract definite answers. Explain why this issue does not arise with knowledge bases containing only definite clauses.

9.21 We said in this chapter that resolution cannot be used to generate all logical consequences of a set of sentences. Can any algorithm do this?