14 AUTOMATED REASONING

For how is it possible, says that acute man, that when a concept is given me, I can go beyond it and connect with it another which is not contained in it, in such a manner as if the latter necessarily belonged to the former?

—IMMANUEL KANT, "Prolegomena to a Future Metaphysics"

Any rational decision may be viewed as a conclusion reached from certain premises. . . . The behavior of a rational person can be controlled, therefore, if the value and factual premises upon which he bases his decisions are specified for him.

—SIMON, *Decision-Making and Administrative Organization*, 1944

Reasoning is an art and not a science. . . .

—WOS ET AL., *Automated Reasoning*, 1984

14.0 Introduction to Weak Methods in Theorem Proving

Wos et al. (1984) describe an *automated reasoning* program as one that "employs an unambiguous and exacting notation for representing information, precise inference rules for drawing conclusions, and carefully delineated strategies to control those inference rules." They add that applying strategies to inference rules to deduce new information is an art: "A good choice for representation includes a notation that increases the chance for solving a problem and includes information that, though not necessary, is helpful. A good choice of inference rules is one that meshes well with the chosen representation. A good choice for strategies is one that controls the inference rules in a manner that sharply increases the effectiveness of the reasoning program."

Automated reasoning, as just described, uses weak problem-solving methods. It uses a uniform representation such as the first-order predicate calculus (Chapter 2), the Horn clause calculus (Section 14.3), or the clause form used for resolution (Section 14.2). Its inference rules are sound and, whenever possible, complete. It uses general strategies such as breadth-first, depth-first, or best-first search and, as we see in this chapter, heuristics such as *set of support* and *unit preference* to combat the combinatorics of exhaustive search. The design of search strategies, and especially heuristic search strategies, is very much an art; we cannot guarantee that they will find a useful solution to a problem using reasonable amounts of time and memory.

Weak method problem solving is an important tool in its own right as well as an essential basis for strong method problem solving. Production systems and rule-based expert system shells are both examples of weak method problem solvers. Even though the rules of the production system or rule-based expert system encode strong problem-solving heuristics, their application is usually supported by general (weak method) inference strategies.

Techniques for weak method problem solving have been the focus of AI research from its beginning. Often these techniques come under the heading of *theorem proving*, although we prefer the more generic title *automated reasoning*. We begin this chapter (Section 14.1) with an early example of automated reasoning, the *General Problem Solver*, and its use of *means–ends analysis* and *difference tables* to control search.

In Section 14.2 we present an important product of research in automated reasoning, the *resolution refutation system*. We discuss the representation language, the resolution inference rule, the search strategies, and the answer extraction processes used in resolution theorem proving. As an example of Horn clause reasoning, in Section 14.3 we describe the inference engine for Prolog, and show how that language contributes to a philosophy of declarative programming with an interpreter based on a resolution theorem prover. We conclude this chapter (Section 14.4) with some brief comments on *natural deduction*, equality handling, and more sophisticated inference rules.

14.1 The General Problem Solver and Difference Tables

The *General Problem Solver (GPS)* (Newell and Simon 1963*b*; Ernst and Newell 1969) came out of research by Allen Newell and Herbert Simon at Carnegie Mellon University, then Carnegie Institute of Technology. Its roots are in an earlier computer program called the *Logic Theorist (LT)* of Newell, Shaw, and Simon (Newell and Simon 1963*a*). The LT program proved many of the theorems in Russell and Whitehead's *Principia Mathematica* (Whitehead and Russell 1950).

As with all weak method problem solvers, the Logic Theorist employed a uniform representation medium and sound inference rules and adopted several strategies or heuristic methods to guide the solution process. The Logic Theorist used the propositional calculus (Section 2.1) as its representation medium. The inference rules were *substitution*, *replacement*, and *detachment*.

Substitution allows any expression to be substituted for every occurrence of a symbol in a proposition that is an axiom or theorem already known to be true. For instance, $(B \vee B) \rightarrow B$ may have the expression $\neg A$ substituted for B to produce $(\neg A \vee \neg A) \rightarrow \neg A$.

Replacement allows a connective to be replaced by its definition or an equivalent form. For example, the logical equivalence of $\neg A \vee B$ and $A \rightarrow B$ can lead to the replacement of $(\neg A \vee \neg A)$ with $(A \rightarrow \neg A)$.

Detachment is the inference rule we called modus ponens (Chapter 2).

The LT applies these inference rules in a breadth-first, goal-driven fashion to the theorem to be proved, attempting to find a series of operations that lead to axioms or theorems known to be true. The strategy of LT consists of four methods organized in an *executive routine*:

First, the substitution method is directly applied to the current goal, attempting to match it against all known axioms and theorems.

Second, if this fails to lead to a proof, all possible detachments and replacements are applied to the goal and each of these results is tested for success using substitution. If substitution fails to match any of these with the goal, then they are added to a *subproblem list*.

Third, the chaining method, employing the transitivity of implication, is used to find a new subproblem that, if solved, would provide a proof. Thus, if $a \rightarrow c$ is the problem and $b \rightarrow c$ is found, then $a \rightarrow b$ is set up as a new subproblem.

Fourth, if the first three methods fail on the original problem, go to the subproblem list and select the next untried subproblem.

The executive routine continues to apply these four methods until either the solution is found, no more problems remain on the subproblem list, or the memory and time allotted to finding the proof are exhausted. In this fashion, the logic theorist executes a goal-driven, breadth-first search of the problem space.

Part of the executive routine that enables the substitution, replacement, and detachment inference rules is the *matching process*. Suppose we wish to prove $p \rightarrow (q \rightarrow p)$. The matching process first identifies one of the axioms, $p \rightarrow (q \vee p)$, as more appropriate than the others–that is, more nearly matching in terms of a domain-defined difference–because the main connective, here $\rightarrow$, is the same in both expressions. Second, the matching process confirms that the expressions to the left of the main connective are identical. Finally, matching identifies the difference between expressions to the right of the main connective. This final difference, between $\rightarrow$ and $\vee$, suggests the obvious replacement for proving the theorem. The matching process helps control the (exhaustive) search that would be necessary for applying all substitutions, replacements, and detachments. In fact, the matching eliminated enough of the trial and error to make the LT into a successful problem solver.

A sample LT proof shows the power of the matching process. Theorem 2.02 of *Principia Mathematica* is $p \rightarrow (q \rightarrow p)$. Matching finds the axiom $p \rightarrow (q \vee p)$ as appropriate for replacement. Substitution of $\neg q$ for q proves the theorem. Matching,

controlling substitution, and replacement rules proved this theorem directly without any search through other axioms or theorems.

In another example, suppose we wish LT to prove:

$(p \rightarrow \neg p) \rightarrow \neg p.$

1.	$(A \vee A) \rightarrow A$	Matching identifies best axiom of five available.
2.	$(\neg A \vee \neg A) \rightarrow \neg A$	Substitution of $\neg A$ for A in order to apply
3.	$(A \rightarrow \neg A) \rightarrow \neg A$	replacement of $\rightarrow$ for $\vee$ and $\neg$, followed by
4.	$(p \rightarrow \neg p) \rightarrow \neg p$	substitution of p for A.

QED

The original LT proved this theorem in about 10 seconds using five axioms. The actual proof took two steps and required no search. Matching selected the appropriate axiom for the first step because its form was much like the conclusion it was trying to establish: (expression) $\rightarrow$ proposition. Then $\neg$ A was substituted for A. This allowed the replacement of the second and final step, which was itself motivated by the goal requiring a $\rightarrow$ rather than a $\vee$.

The Logic Theorist not only was the first example of an automated reasoning system but also demonstrated the importance of search strategies and heuristics in a reasoning program. In many instances LT found solutions in a few steps that exhaustive search might never find. Some theorems were not solvable by the LT, and Newell et al. pointed out improvements that might make their solution possible.

At about this time, researchers at Carnegie and others at Yale (Moore and Anderson 1954) began to examine think-aloud protocols of human subjects solving logic problems. Although their primary goal was to identify human processes that could solve this class of problem, researchers began to compare human problem solving with computer programs, such as the Logic Theorist. This was to become the first instance of what is now referred to as *information processing psychology*, where an explanation of the observed behavior of an organism is provided by a program of primitive information processes that generates that behavior (Newell et al. 1958). This research was also some of the first work that founded the modern discipline of *Cognitive Science* (see Section 16.2, Luger 1994).

Closer scrutiny of these first protocols showed many ways that LT's solutions differed from those of the human subjects. The human behavior showed strong evidence of a matching and difference reduction mechanism referred to as a *means–ends analysis*. In means–ends analysis the difference reduction methods (the *means*) were strongly linked to the specific differences to be reduced (the *ends*): the operators for difference reduction were indexed by the differences they could reduce.

In a very simple example, if the start statement was $p \rightarrow q$ and the goal was $\neg p \vee q$, the differences would include the $\rightarrow$ symbol in the start and $\vee$ in the goal, as well as the difference of p in the start and $\neg$ p in the goal. The difference table would contain the different ways that a $\rightarrow$ could be replaced by a $\vee$ and that $\neg$ could be removed. These transformations would be attempted one at a time until the differences were removed and the theorem was proven.

In most interesting problems the differences between start and goal could not be directly reduced. In this case an operator (from the table) was sought to partially reduce the difference. The entire procedure was applied recursively to these results until no differences existed. This might also require following different search paths, represented by different applications of reductions.

Figure 14.1a, from Newell and Simon (1963b), presents twelve transformation rules, the middle column, for solving logic problems. The right column gives directives as to when the transformations are to be used.

R 1.	$A \cdot B \rightarrow B \cdot A$ $A \vee B \rightarrow B \vee A$	Applies to main expression only.
R 2.	$A \supset B \rightarrow {\sim}B \supset {\sim}A$	Applies to main expression only.
R 3.	$A \cdot A \leftrightarrow A$ $A \vee A \leftrightarrow A$	A and B are two main expressions.
R 4.	$A \cdot (B \cdot C) \leftrightarrow (A \cdot B) \cdot C$ $A \vee (B \vee C) \leftrightarrow (A \vee B) \vee C$	A and $A \supset B$ are two main expressions.
R 5.	$A \vee B \leftrightarrow {\sim}({\sim}A \cdot {\sim}B)$	$A \supset B$ and $B \supset C$ are two main expressions.
R 6.	$A \supset B \leftrightarrow {\sim}A \vee B$	
R 7.	$A \cdot (B \vee C) \leftrightarrow (A \cdot B) \vee (A \cdot C)$ $A \vee (B \cdot C) \leftrightarrow (A \vee B) \cdot (A \vee C)$	
R 8.	$A \cdot B \rightarrow A$ $A \cdot B \rightarrow B$	Applies to main expression only.
R 9.	$A \rightarrow A \vee X$	Applies to main expression only.
R 10.	$\left.\begin{matrix} A \\ B \end{matrix}\right\} \rightarrow A \cdot B$	A and B are two main expressions.
R 11.	$\left.\begin{matrix} A \\ A \supset B \end{matrix}\right\} \rightarrow B$	A and $A \supset B$ are two main expressions.
R 12.	$\left.\begin{matrix} A \supset B \\ B \supset C \end{matrix}\right\} \rightarrow A \supset C$	$A \supset B$ and $B \supset C$ are two main expressions.

Figure 14.1a Transformation rules for logic problems, from Newell and Simon (1961).

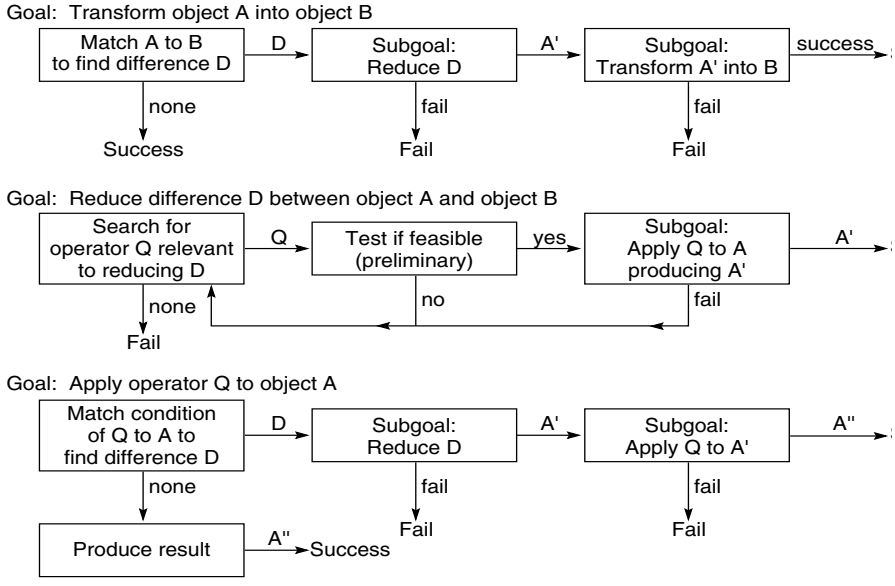
1.	$(R \supset \sim P) \cdot (\sim R \supset Q)$	$\sim (\sim Q \cdot P)$
2.	$(\sim R \vee \sim P) \cdot (R \vee Q)$	Rule *6* applied to left and right of 1.
3.	$(\sim R \vee \sim P) \cdot (\sim R \supset Q)$	Rule *6* applied to left of 1.
4.	$R \supset \sim P$	Rule *8* applied to 1.
5.	$\sim R \vee \sim P$	Rule *6* applied to 4.
6.	$\sim R \supset Q$	Rule *8* applied to 1.
7.	$R \vee Q$	Rule *6* applied to 6.
8.	$(\sim R \vee \sim P) \cdot (R \vee Q)$	Rule *10* applied to 5. and 7.
9.	$P \supset \sim R$	Rule *2* applied to 4.
10.	$\sim Q \supset R$	Rule *2* applied to 6.
11.	$P \supset Q$	Rule *12* applied to 6. and 9.
12.	$\sim P \vee Q$	Rule *6* applied to 11.
13.	$\sim (P \cdot \sim Q)$	Rule *5* applied to 12.
14.	$\sim (\sim Q \cdot P)$	Rule *1* applied to 13. QED.

Figure 14.1b A proof of a theorem in propositional calculus, from Newell and Simon (1961).

Figure 14.1b presents a proof, from Newell and Simon (1963*b*), generated by a human subject. Before the proof, the transformation rules of Figure 14.1a are available to the subject, who, without experience in formal logic, is asked to change the expression $(R \supset \neg P) \bullet (\neg R \supset Q)$ to $\neg (\neg Q \bullet P)$. In the notation of Chapter 2 ~ is $\neg$, • is $\wedge$, and $\supset$ is $\rightarrow$. The $\rightarrow$ or $\leftrightarrow$ in Figure 14.1a indicates a legal replacement. The rightmost column of Figure 14.1b indicates the rule of Figure 14.1a that is applied at each step of the proof.

Newell and Simon (1963b) called the problem-solving strategies of the human subject *difference reduction* and the general process of using transformations appropriate for reducing specific problem differences *means–ends analysis.* The algorithm for applying means–ends analysis using difference reductions is the *General Problem Solver (GPS)*.

Figure 14.2 presents the control diagram and the table of connections for GPS. The goal is to transform expression A into expression B. The first step is to locate a difference D between A and B. The subgoal, reduce D, is identified in the second box of the first line; the third box indicates that difference reduction is recursive. The "reduction" is the second line, where an operator Q is identified for the difference D. Actually, a list of operators is identified from the table of connections. This list provides ordered alternatives for difference reduction should the chosen operator not be acceptable, for example, by not passing the *feasibility test*. In the third line of Figure 14.2 the operator is applied and D is reduced.



For the logic task of the text:

Feasibility test (preliminary)
- Is the mean connective the same (e.g., A•B → B fails against P∨Q)?
- Is the operator too big (e.g., (A∨B)•(A∨C)→A∨(B•C) fails against P•Q)?
- Is the operator too easy (e.g., A→A•A applies to anything)?
- Are the side conditions satisfied (e.g., R8 applies only to main expressions)?

Table of connections

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
Add terms			X				X		X	X	X	X
Delete terms			X				X	X			X	X
Change connective					X	X	X					
Change sign					X							
Change lower sign		X			X	X						
Change grouping				X			X					
Change position	X	X										

X means some variant of the rule is relevant. GPS will pick the appropriate variant.

Figure 14.2 Flow chart and difference reduction table for the General Problem Solver, from Newell and Simon (1963b).

The GPS model of problem solving requires two components. The first is the general procedure just described for comparing two state descriptions and reducing their differences. The second component of GPS is the *table of connections,* giving the links between problem differences and the specific transformations that reduce them, appropriate to an application area. Figure 14.2, gives differences and their reductions (the twelve transformations from Figure 14.1a) for propositional calculus expressions. Other tables of connections could be built for reducing differences in algebraic forms, or for tasks such as Towers of Hanoi, or for more complex games such as chess. Because of this modularity of the table of differences, i.e., the tables are changed out for different applications, the prob-

lem solver was called *general*. A number of the different application areas of GPS are described by Ernst and Newell (1969).

The actual structuring of the difference reductions of a particular problem domain helps organize the search for that domain. A heuristic or priority order for reduction of different difference classes is implicit in the order of the transformations within the difference reduction table. This priority order might put the more generally applicable transformations before the specialized ones or give whatever order some domain expert might deem most appropriate.

A number of research directions evolved from work in the General Problem Solver. One of these is the use of AI techniques to analyze human problem-solving behavior. In particular, the production system replaced the means–ends methods of GPS as the preferred form for modeling human information processing (Chapter 16). The production rules in modern rule-based expert systems replaced the specific entries in GPS's table of differences (Chapter 8).

In another interesting evolution of GPS, the difference table itself evolved in a further fashion, becoming the *operator table* for *planning* such as STRIPS and ABSTRIPS. Planning is important in robot problem solving. To accomplish a task, such as to go to the next room and bring back an object, the computer must develop a *plan*. This plan orchestrates the actions of the robot: put down anything it is now holding, go to the door of the present room, go through the door, find the required room, go through the door, go over to the object, and so on. Plan formation for STRIPS, the Stanford Research Institute Problem Solver (Fikes and Nilsson 1971, Fikes et al. 1972, Sacerdotti 1974) uses an operator table not unlike the GPS table of differences. Each operator (primitive act of the robot) in this table has an attached set of *preconditions* that are much like the feasibility tests of Figure 14.2. The operator table also contains *add* and *delete lists*, these are used to update the model of the "world" once the operator itself has been applied. We presented a STRIPS-like planner in Section 7.4. and then build it in Prolog in the auxiliary software materials.

To summarize, the first models of automated reasoning in AI are found in the Logic Theorist and General Problem Solver developed at Carnegie Institute. Already these programs offered the full prerequisites for weak method problem solving: a uniform representation medium, a set of sound inference rules, and a set of methods or strategies for applying these rules. The same components make up the *resolution proof procedures*, a modern and more powerful basis for automated reasoning.

14.2 Resolution Theorem Proving

14.2.1 Introduction

Resolution is a technique for proving theorems in the propositional or predicate calculus that has been a part of AI problem-solving research from the mid-1960s (Bledsoe 1977, Robinson 1965, Kowalski 1979*b*). Resolution is a sound inference rule that, when used to produce a *refutation* (Section 14.2.3), is also complete. In an important practical

application, resolution theorem proving, particularly the resolution refutation system, has made the current generation of Prolog interpreters possible (Section 14.3).

The resolution principle, introduced in an important paper by Robinson (1965), describes a way of finding contradictions in a database of clauses with minimum use of substitution. Resolution refutation proves a theorem by negating the statement to be proved and adding this negated goal to the set of axioms that are known (have been assumed) to be true. It then uses the resolution rule of inference to show that this leads to a contradiction. Once the theorem prover shows that the negated goal is inconsistent with the given set of axioms, it follows that the original goal must be consistent. This proves the theorem.

Resolution refutation proofs involve the following steps:

1. Put the premises or axioms into *clause form* (14.2.2).
2. Add the negation of what is to be proved, in clause form, to the set of axioms.
3. *Resolve* these clauses together, producing new clauses that logically follow from them (14.2.3).
4. Produce a contradiction by generating the empty clause.
5. The substitutions used to produce the empty clause are those under which the opposite of the negated goal (what was originally to be proven) is true (14.2.4).

Resolution is a sound inference rule in the sense of Chapter 2. However, it is not complete. Resolution is *refutation complete*; that is, the empty or null clause can always be generated whenever a contradiction in the set of clauses exists. More is said on this topic when we present strategies for refutation in Section 14.2.4.

Resolution refutation proofs require that the axioms and the negation of the goal be placed in a normal form called *clause form*. Clause form represents the logical database as a set of disjunctions of *literals*. A literal is an atomic expression or the negation of an atomic expression.

The most common form of resolution, called *binary resolution*, is applied to two clauses when one contains a literal and the other its negation. If these literals contain variables, the literals must be unified to make them equivalent. A new clause is then produced consisting of the disjuncts of all the predicates in the two clauses minus the literal and its negative instance, which are said to have been "resolved away." The resulting clause receives the unification substitution under which the predicate and its negation are found as "equivalent".

Before this is made more precise in the subsequent subsections, we take a simple example. Resolution produces a proof similar to one produced already with modus ponens. This is not intended to show that these inference rules are equivalent (resolution is actually more general than modus ponens) but to give the reader a feel for the process.

We wish to prove that "Fido will die" from the statements that "Fido is a dog" and "all dogs are animals" and "all animals will die." Changing these three premises to predicates and applying modus ponens gives:

1. All dogs are animals: $\forall$(X) (dog (X) $\rightarrow$ animal (X)).
2. Fido is a dog: dog (fido).
3. Modus ponens and {fido/X} gives: animal (fido).
4. All animals will die: $\forall$(Y) (animal (Y) $\rightarrow$ die (Y)).
5. Modus ponens and {fido/Y} gives: die (fido).

Equivalent reasoning by resolution converts these predicates to clause form:

	PREDICATE FORM	CLAUSE FORM
1.	$\forall$(X) (dog) (X) $\rightarrow$ animal (X))	$\neg$ dog (X) $\vee$ animal (X)
2.	dog (fido)	dog (fido)
3.	$\forall$(Y) (animal (Y) $\rightarrow$ die (Y))	$\neg$ animal (Y) $\vee$ die (Y)

Negate the conclusion that Fido will die:

4.	$\neg$ die (fido)	$\neg$ die (fido)

Resolve clauses having opposite literals, producing new clauses by resolution as in Figure 14.3. This process is often called *clashing*.

The symbol $\square$ in Figure 14.3 indicates that the empty clause is produced and the contradiction found. The $\square$ symbolizes the clashing of a predicate and its negation: the situation where two mutually contradictory statements are present in the clause space. These are clashed to produce the empty clause. The sequence of substitutions (unifications) used to make predicates equivalent also gives us the value of variables under which a goal is true. For example, had we asked whether something would die, our negated goal would have been $\neg$ ($\exists$ (Z) die (Z)), rather than $\neg$ die(fido). The substitution {fido/Z} in Figure 14.3 would determine that fido is an instance of an animal that will die. The issues implicit in this example are made clear in the remainder of Section 14.2.

14.2.2 Producing the Clause Form for Resolution Refutations

The resolution proof procedure requires all statements in the database describing a situation to be converted to a standard form called *clause* form. This is motivated by the fact that resolution is an operator on pairs of disjuncts to produce new disjuncts. The form the database takes is referred to as a *conjunction of disjuncts*. It is a *conjunction* because all the clauses that make up the database are assumed to be true at the same time. It is a *disjunction* in that each of the individual clauses is expressed with disjunction (or $\vee$) as the connective. Thus the entire database of Figure 14.3 may be represented in clause form as:

($\neg$ dog (X) $\vee$ animal (X)) $\wedge$ ($\neg$ animal (Y) $\vee$ die (Y)) $\wedge$ (dog (fido)).

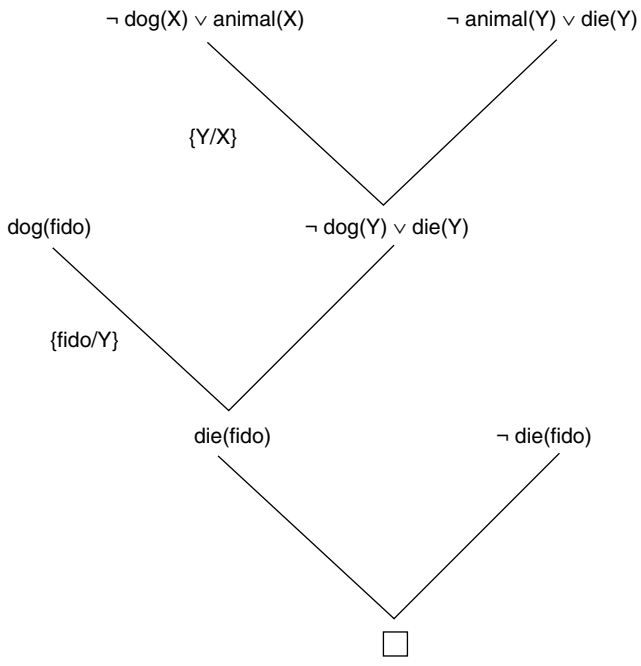


Figure 14.3 Resolution proof for the dead dog problem.

To this expression we add (by conjunction) the negation of what we wish to prove, in this case ¬ die(fido). Generally, the database is written as a set of disjunctions and the ∧ operators are omitted.

We now present an algorithm, consisting of a sequence of transformations, for reducing any set of predicate calculus statements to clause form. It has been shown (Chang and Lee 1973) that these transformations may be used to reduce any set of predicate calculus expressions to a set of clauses that are inconsistent if and only if the original set of expressions is inconsistent. The clause form will not be strictly equivalent to the original set of predicate calculus expressions in that certain interpretations may be lost. This occurs because skolemization restricts the possible substitutions for existentially quantified variables (Chang and Lee 1973). It will, however, preserve unsatisfiability. That is, if there was a contradiction (a refutation) within the original set of predicate calculus expressions, a contradiction exists in the clause form. The transformations do not sacrifice completeness for refutation proofs.

We demonstrate this process of conjunctive normal form reduction through an example and give a brief description rationalizing each step. These are not intended to be proofs of the equivalence of these transformations across all predicate calculus expressions.

In the following expression, according to the conventions of Chapter 2, uppercase letters indicate variables (W, X, Y, and Z); lowercase letters in the middle of the alphabet indicate constants or bound variables (l, m, and n); and early alphabetic lowercase letters

indicate the predicate names (a, b, c, d, and e). To improve readability of the expressions, we use two types of brackets: ( ) and [ ], and remove redundant brackets. As an example, consider the following expression, where X, Y, and Z are variables and l a constant:

(i) $(\forall X)([a(X) \wedge b(X)] \rightarrow [c(X,l) \wedge (\exists Y)((\exists Z)[c(Y,Z)] \rightarrow d(X,Y))]) \vee (\forall X)(e(X))$

1. First we eliminate the $\rightarrow$ by using the equivalent form proved in Chapter 2: $a \rightarrow b \equiv \neg a \vee b$. This transformation reduces the expression in (i) above:

   (ii) $(\forall X)(\neg [a(X) \wedge b(X)] \vee [c(X,l) \wedge (\exists Y)((\exists Z)[\neg c(Y,Z)] \vee d(X,Y))] \vee (\forall X)(e(X))$

2. Next we reduce the scope of negation. This may be accomplished using a number of the transformations of Chapter 2. These include:

   $\neg (\neg a) \equiv a$
   $\neg (\exists X)\, a(X) \equiv (\forall X) \neg a(X)$
   $\neg (\forall X)\, b(X) \equiv (\exists X) \neg b(X)$
   $\neg (a \wedge b) \equiv \neg a \vee \neg b$
   $\neg (a \vee b) \equiv \neg a \wedge \neg b$

   Using the fourth equivalences (ii) becomes:

   (iii) $(\forall X)([\neg a(X) \vee \neg b(X)] \vee [c(X,l) \wedge (\exists Y)((\exists Z)[\neg c(Y,Z)] \vee d(X,Y))]) \vee (\forall X)(e(X))$

3. Next we standardize by renaming all variables so that variables bound by different quantifiers have unique names. As indicated in Chapter 2, because variable names are "dummies" or "place holders," the particular name chosen for a variable does not affect either the truth value or the generality of the clause. Transformations used at this step are of the form:

   $((\forall X)a(X) \vee (\forall X)b(X)) \equiv (\forall X)a(X) \vee (\forall Y)b(Y)$

   Because (iii) has two instances of the variable X, we rename:

   (iv) $(\forall X)([\neg a(X) \vee \neg b(X)] \vee [c(X,l) \wedge (\exists Y)((\exists Z)[\neg c(Y,Z)] \vee d(X,Y))]) \vee (\forall W)(e(W))$

4. Move all quantifiers to the left without changing their order. This is possible because step 3 has removed the possibility of any conflict between variable names. (iv) now becomes:

   (v) $(\forall X)(\exists Y)(\exists Z)(\forall W)([\neg a(X) \vee \neg b(X)] \vee [c(X,l) \wedge (\neg c(Y,Z) \vee d(X,Y))] \vee e(W))$

   After step 4 the clause is said to be in *prenex normal* form, because all the quantifiers are in front as a *prefix* and the expression or *matrix* follows after.

5. At this point all existential quantifiers are eliminated by a process called *skolemization*. Expression (v) has an existential quantifier for Y. When an expression contains an existentially quantified variable, for example, (∃Z)(foo(…, Z,…)), it may be concluded that there is an assignment to Z under which foo is true. Skolemization identifies such a value. Skolemization does not necessarily show *how* to produce such a value; it is only a method for giving a name to an assignment that *must* exist. If k represents that assignment, then we have foo(…,k,…). Thus:

   (∃X)(dog(X)) may be replaced by dog(fido)

   where the name fido is picked from the domain of definition of X to represent that individual X. fido is called a *skolem constant*. If the predicate has more than one argument and the existentially quantified variable is within the scope of universally quantified variables, the existential variable must be a function of those other variables. This is represented in the skolemization process:

   (∀X) (∃Y) (mother(X,Y))

   This expression indicates that every person has a mother. Every person is an X and the existing mother will be a function of the particular person X that is picked. Thus skolemization gives:

   (∀X) mother(X, m(X))

   which indicates that each X has a mother (the m of that X). In another example:

   (∀X)(∀Y)(∃Z)(∀W)(foo(X,Y,Z,W))

   is skolemized to:

   (∀X)(∀Y)(∀W)(foo(X,Y,f(X,Y),W)).

   The existentially quantified Y and Z are within the scope (to the right of) universally quantified X but not within the scope of W. Thus each will be replaced by a skolem function of X. Replacing Y with the skolem function f(X) and Z with g(X), (v) becomes:

   (vi) (∀X)(∀W)([¬ a(X) ∨ ¬ b(X)] ∨ [c(X,l) ∧ (¬ c(f(X),g(X)) ∨ d(X,f(X)))]) ∨ e(W))

   After skolemization, step 6 can take place, which simply drops the prefix.

6. Drop all universal quantification. By this point only universally quantified variables exist (step 5) with no variable conflicts (step 3). Thus all quantifiers can be dropped, and any proof procedure employed assumes all variables are universally quantified. Formula (vi) now becomes:

(vii) $[\neg a(X) \vee \neg b(X)] \vee [c(X,l) \wedge (\neg c(f(X),g(X)) \vee d(X,f(X)))] \vee e(W)$

7. Next we convert the expression to the conjunct of disjuncts form. This requires using the associative and distributive properties of $\wedge$ and $\vee$. Recall from Chapter 2 that

   $a \vee (b \vee c) = (a \vee b) \vee c$

   $a \wedge (b \wedge c) = (a \wedge b) \wedge c$

   which indicates that $\wedge$ or $\vee$ may be grouped in any desired fashion. The distributive property of Chapter 2 is also used, when necessary. Because

   $a \wedge (b \vee c)$

   is already in clause form, $\wedge$ is not distributed. However, $\vee$ must be distributed across $\wedge$ using:

   $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

   The final form of (vii) is:

   (viii) $[\neg a(X) \vee \neg b(X) \vee c(X,l) \vee e(W)] \wedge$
   $[\neg a(X) \vee \neg b(X) \vee \neg c(f(X),g(X)) \vee d(X,f(X)) \vee e(W)]$

8. Now call each conjunct a separate clause. In the example (viii) above there are two clauses:

   (ixa) $\neg a(X) \vee \neg b(X) \vee c(X,l) \vee e(W)$

   (ixb) $\neg a(X) \vee \neg b(X) \vee \neg c(f(X),g(X)) \vee d(X,f(X)) \vee e(W)$

9. The final step is to *standardize the variables apart* again. This requires giving the variable in each clause generated by step 8 different names. This procedure arises from the equivalence established in Chapter 2 that

   $(\forall X)(a(X) \wedge b(X)) \equiv (\forall X)\, a(X) \wedge (\forall Y)\, b(Y)$

   which follows from the nature of variable names as place holders. (ixa) and (ixb) now become, using new variable names U and V:

   (xa) $\neg a(X) \vee \neg b(X) \vee c(X,l) \vee e(W)$

   (xb) $\neg a(U) \vee \neg b(U) \vee \neg c(f(U),g(U)) \vee d(U,f(U)) \vee e(V)$

The importance of this final standardization becomes apparent only as we present the unification steps of resolution. We find the most general unification to make two predicates within two clauses equivalent, and then this substitution is made across all the variables of the same name within each clause. Thus, if some variables (needlessly) share

names with others, these may be renamed by the unification process with a subsequent (possible) loss of generality in the solution.

This nine-step process is used to change any set of predicate calculus expressions to clause form. The completeness property of resolution refutations is not lost. Next we demonstrate the resolution procedure for generating proofs from these clauses.

14.2.3 The Binary Resolution Proof Procedure

The *resolution refutation* proof procedure answers a query or deduces a new result by reducing a set of clauses to a contradiction, represented by the null clause ($\square$). The contradiction is produced by resolving pairs of clauses from the database. If a resolution does not produce a contradiction directly, then the clause produced by the resolution, the *resolvent*, is added to the database of clauses and the process continues.

Before we show how the resolution process works in the predicate calculus, we give an example from the propositional or variable-free calculus. Consider two *parent* clauses p1 and p2 from the propositional calculus:

$$\text{p1: } a_1 \vee a_2 \vee \cdots \vee a_n$$
$$\text{p2: } b_1 \vee b_2 \vee \cdots \vee b_m$$

having two literals a_i and b_j, where $1 < i \leq n$ and $1 \leq j \leq m$, such that $\neg a_i = b_j$. Binary resolution produces the clause:

$$a_1 \vee \cdots \vee a_{i-1} \vee a_{i+1} \vee \cdots \vee a_n \vee b_1 \vee \cdots \vee b_{j-1} \vee b_{j+1} \vee \cdots \vee b_m.$$

The notation above indicates that the resolvent is made up of the disjunction of all the literals of the two parent clauses except for the literals a_i and b_j.

A simple argument can give the intuition behind the resolution principle. Suppose

$$a \vee \neg b \text{ and } b \vee c$$

are both true statements. Observe that one of b and $\neg$ b must always be true and one always false (b $\vee \neg$ b is a tautology). Therefore, one of

$$a \vee c$$

must always be true. a $\vee$ c is the resolvent of the two parent clauses a $\vee \neg$ b and b $\vee$ c.

Consider now an example from the propositional calculus, where we want to prove a from the following axioms (of course, l $\leftarrow$ m $\equiv$ m $\rightarrow$ l for all propositions l and m):

$$a \leftarrow b \wedge c$$
$$b$$
$$c \leftarrow d \wedge e$$

$e \vee f$

$d \wedge \neg f$

We reduce the first axiom to clause form:

$a \leftarrow b \wedge c$

$a \vee \neg (b \wedge c)$ by $l \rightarrow m \equiv \neg l \vee m$

$a \vee \neg b \vee \neg c$ by de Morgan's law

The remaining axioms are reduced, and we have the following clauses:

$a \vee \neg b \vee \neg c$

b

$c \vee \neg d \vee \neg e$

$e \vee f$

d

$\neg f$

The resolution proof is found in Figure 14.4. First, the goal to be proved, a, is negated and added to the clause set. The derivation of $\square$ indicates that the database of clauses is inconsistent.

To use binary resolution in the predicate calculus, where each literal may contain variables, there must be a process under which two literals with different variable names, or one with a constant value, can be seen as equivalent. Unification was defined in Section 2.3.2 as the process for determining consistent and most general substitutions for making two predicates equivalent.

The algorithm for resolution on the predicate calculus is very much like that on the propositional calculus except that:

1. A literal and its negation in parent clauses produce a resolvent only if they unify under some substitution σ. σ is then applied to the resolvent before adding it to the clause set. We require that σ be the most general unifer of the parent clauses.

2. The unification substitutions used to find the contradiction offer variable bindings under which the original query is true. We explain this process, called *answer extraction*, in Section 14.2.5.

Occasionally, two or more literals in one clause have a unifying substitution. When this occurs there may not exist a refutation for a set of clauses containing that clause, even though the set may be contradictory. For instance, consider the clauses:

$p(X) \vee p(f(Y))$

$\neg p(W) \vee \neg p(f(Z))$

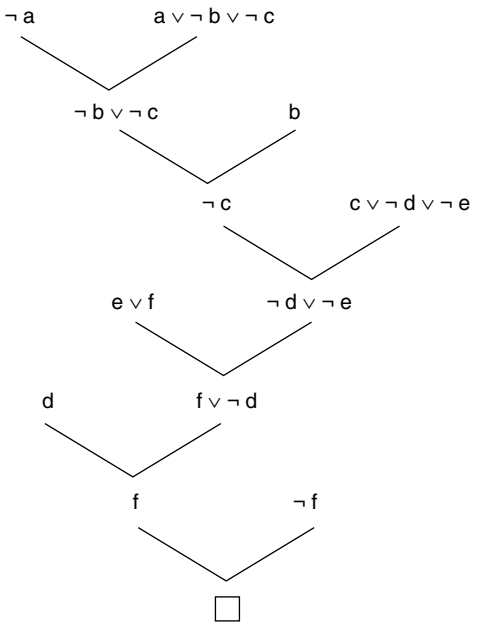


Figure 14.4 One resolution proof for an example from the propositional calculus.

The reader should note that with simple resolution these clauses can be reduced only to equivalent or tautological forms but not to a contradiction, that is, no substitution can make them inconsistent.

This situation may be handled by *factoring* such clauses. If a subset of the literals in a clause has a most general unifier (Section 2.3.2), then the clause is replaced by a new clause, called a *factor* of that clause. The factor is the original clause with the most general unifier substitution applied and then redundant literals removed. For example, the two literals of the clause $p(X) \vee p(f(Y))$ will unify under the substitution $\{f(Y)/X\}$. We make the substitution in both literals to obtain the clause $p(f(Y)) \vee p(f(Y))$ and then replace this clause with its factor: $p(f(Y))$. Any resolution refutation system that includes factoring is refutation complete. Standardizing variables apart, Section 14.2.2 step 3, can be interpreted as a trivial application of factoring. Factoring may also be handled as part of the inference process in *hyperresolution* described in Section 14.4.2.

We now present an example of a resolution refutation for the predicate calculus. Consider the following story of the "happy student":

> Anyone passing his history exams and winning the lottery is happy. But anyone who studies or is lucky can pass all his exams. John did not study but he is lucky. Anyone who is lucky wins the lottery. Is John happy?

First change the sentences to predicate form:

Anyone passing his history exams and winning the lottery is happy.

$\forall$ X (pass (X,history) $\wedge$ win (X,lottery) $\rightarrow$ happy (X))

Anyone who studies or is lucky can pass all his exams.

$\forall$ X $\forall$ Y (study (X) $\vee$ lucky (X) $\rightarrow$ pass (X,Y))

John did not study but he is lucky.

$\neg$ study (john) $\wedge$ lucky (john)

Anyone who is lucky wins the lottery.

$\forall$ X (lucky (X) $\rightarrow$ win (X,lottery))

These four predicate statements are now changed to clause form (Section 14.2.2):

1. $\neg$ pass (X, history) $\vee$ $\neg$ win (X, lottery) $\vee$ happy (X)
2. $\neg$ study (Y) $\vee$ pass (Y, Z)
3. $\neg$ lucky (W) $\vee$ pass (W, V)
4. $\neg$ study (john)
5. lucky (john)
6. $\neg$ lucky (U) $\vee$ win (U, lottery)

Into these clauses is entered, in clause form, the negation of the conclusion:

7. $\neg$ happy (john)

The resolution refutation graph of Figure 14.5 shows a derivation of the contradiction and, consequently, proves that John is happy.

As a final example in this subsection we present the "exciting life" problem; suppose:

> All people who are not poor and are smart are happy. Those people who read are smart. John can read and is not poor. Happy people have exciting lives. Can anyone be found with an exciting life?

We assume $\forall$X (smart (X) $\equiv$ $\neg$ stupid (X)) and $\forall$Y (wealthy (Y) $\equiv$ $\neg$ poor (Y)), and get:

$\forall$X ($\neg$ poor (X) $\wedge$ smart (X) $\rightarrow$ happy (X))
$\forall$Y (read (Y) $\rightarrow$ smart (Y))
read (john) $\wedge$ $\neg$ poor (john)
$\forall$Z (happy (Z) $\rightarrow$ exciting (Z))

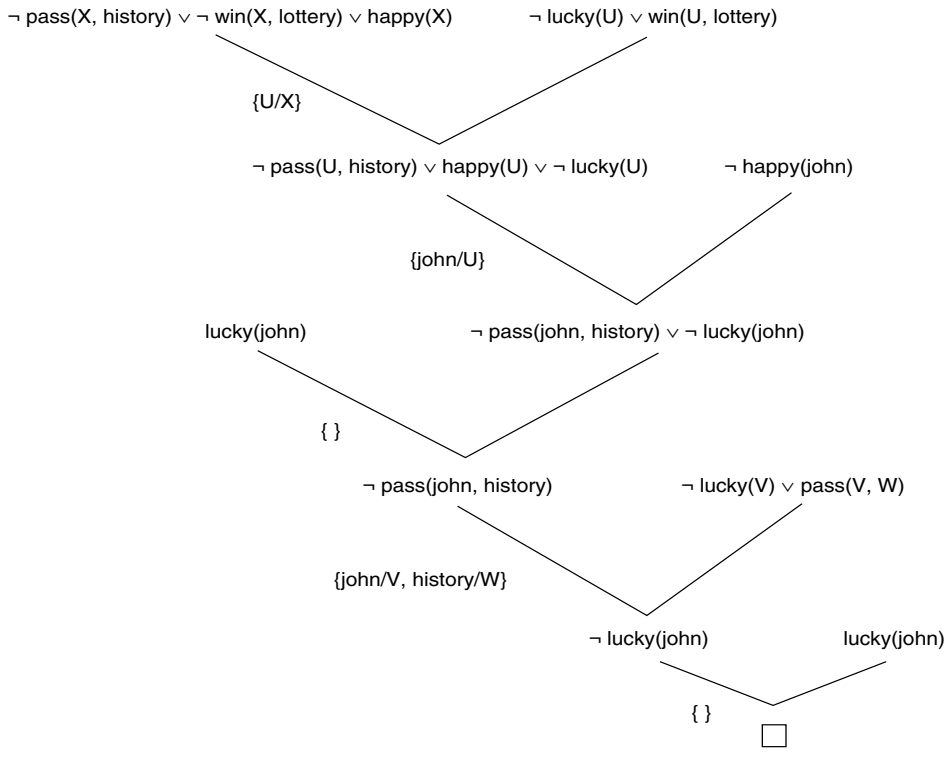


Figure 14.5 One resolution refutation for the happy student problem.

The negation of the conclusion is:

¬ ∃ W (exciting (W))

These predicate calculus expressions for the "exciting life" problem are transformed into the following clauses:

poor (X) ∨ ¬ smart (X) ∨ happy (X)
¬ read (Y) ∨ smart (Y)
read (john)
¬ poor (john)
¬ happy (Z) ∨ exciting (Z)
¬ exciting (W)

The resolution refutation for this example is found in Figure 14.6.

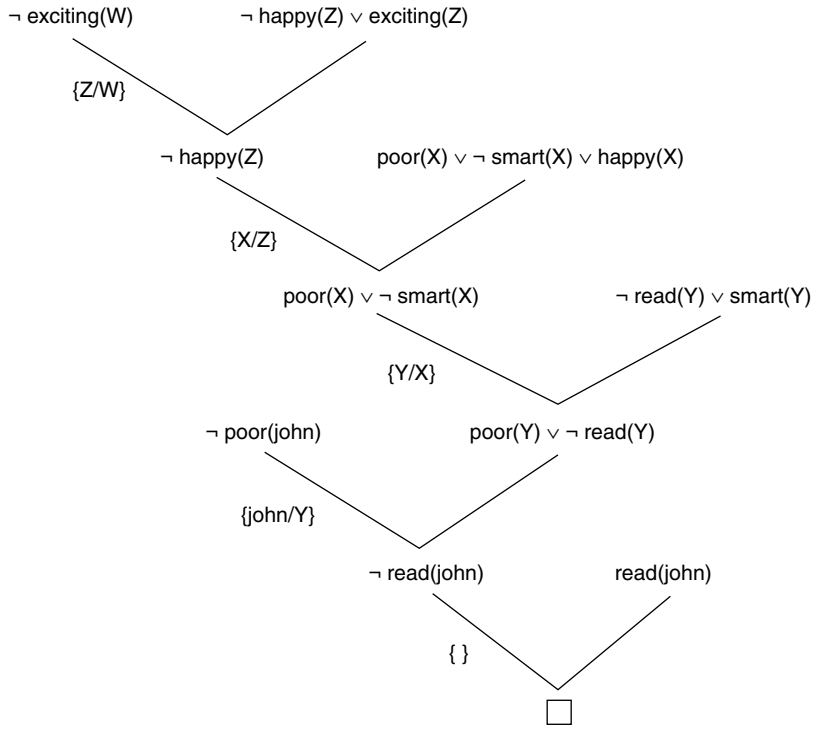


Figure 14.6 Resolution proof for the exciting life problem.

14.2.4 Strategies and Simplification Techniques for Resolution

A different proof tree within the search space for the problem of Figure 14.6 appears in Figure 14.7. There are some similarities in these proofs; for example, they both took five resolution steps. Also, the associative application of the unification substitutions found that john was the instance of the person with the "exciting life" in both proofs.

However, even these two similarities need not have occurred. When the resolution proof system was defined (Section 14.2.3) no order of clause combinations was implied. This is a critical issue: when there are N clauses in the clause space, there are N^2 ways of combining them or checking to see whether they can be combined at just the first level! The resulting set of clauses from this comparison is also large; if even 20% of them produce new clauses, the next round of possible resolutions will contain even more combinations than the first round. In a large problem this exponential growth will quickly get out of hand.

For this reason search heuristics are very important in resolution proof procedures, as they are in all weak method problem solving. As with the heuristics we considered in Chapter 4, there is no science that can determine the best strategy for any particular problem. Nonetheless, some general strategies can address the exponential combinatorics.

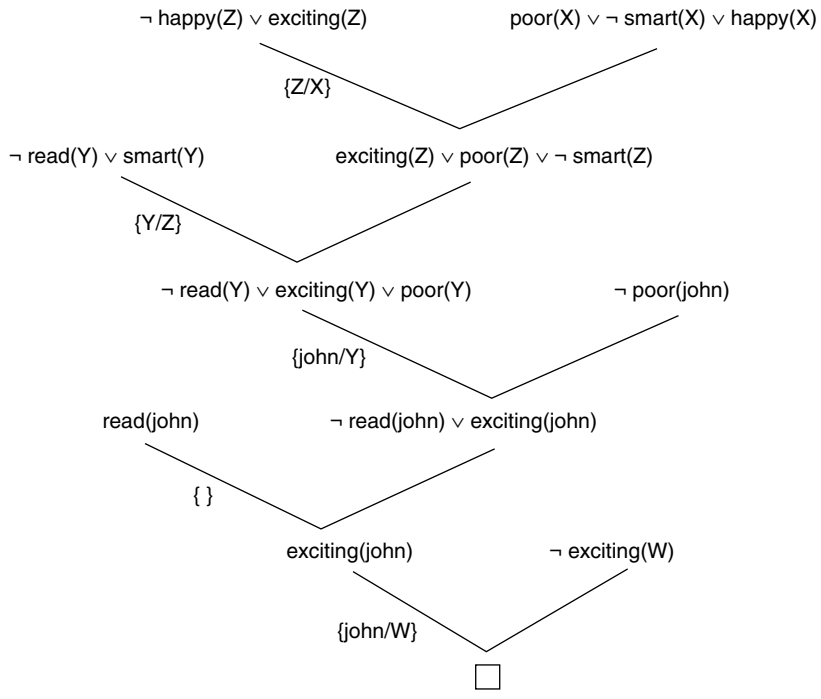


Figure 14.7 Another resolution refutation for the example of Figure 14.6.

Before we describe our strategies, we need to make several clarifications. First, based on the definition of unsatisfiability of an expression in Chapter 2, a *set of clauses is unsatisfiable* if no interpretation exists that establishes the set as satisfiable. Second, an inference rule is *refutation complete* if, given an unsatisfiable set of clauses, the unsatisfiability can be established by use of this inference rule alone. Resolution with factoring has this property (Chang and Lee 1973). Finally, a *strategy is complete* if by its use with a refutation-complete inference rule we can guarantee finding a refutation whenever a set of clauses is unsatisfiable. *Breadth-first* is an example of a complete strategy.

The *Breadth-First* Strategy The complexity analysis of exhaustive clause comparison just described was based on breadth-first search. Each clause in the clause space is compared for resolution with every clause in the clause space on the first round. The clauses at the second level of the search space are generated by resolving the clauses produced at the first level with all the original clauses. We generate the clauses at the nth level by resolving all clauses at level $n - 1$ against the elements of the original clause set and all clauses previously produced.

This strategy can quickly get out of hand for large problems. It does have an interesting property, however. Like any breadth-first search, it guarantees finding the

shortest solution path, because it generates every search state for each level before going any deeper. It also is a complete strategy in that, if it is continued long enough, it is guaranteed to find a refutation if one exists. Thus, when the problem is small, as are the ones we have presented as examples, the breadth-first strategy can be a good one. Figure 14.8 applies the breadth-first strategy to the "exciting life" problem.

The *Set of Support* Strategy An excellent strategy for large clause spaces is called the set of support (Wos and Robinson 1968). For a set of input clauses, S, we can specify a subset, T of S, called the set of support. The strategy requires that one of the resolvents in each resolution have an ancestor in the set of support. It can be proved that if S is unsatisfiable and S – T is satisfiable, then the set of support strategy is refutation complete (Wos et al. 1984).

If the original set of clauses is consistent, then any set of support that includes the negation of the original query meets these requirements. This strategy is based on the insight that the negation of what we want to prove true is going to be responsible for causing the clause space to be contradictory. The set of support forces resolutions between clauses of which at least one is either the negated goal clause or a clause produced by resolutions on the negated goal.

Figure 14.6 is an example of the set of support strategy applied to the exciting life problem. Because a set of support refutation exists whenever any refutation exists, the set of support can be made the basis of a complete strategy. One way to do this is to perform a breadth-first search for all possible set of support refutations. This, of course, will be much more efficient than breadth-first search of all clauses. One need only be sure that all resolvents of the negated goal clause are examined, along with all their descendants.

The *Unit Preference* Strategy Observe that in the resolution examples seen so far, the derivation of the contradiction is indicated by the clause with no literals. Thus, every time

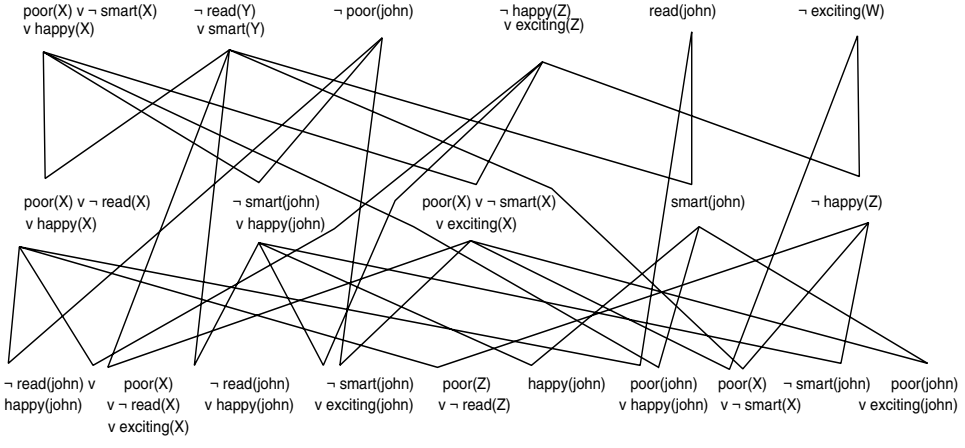


Figure 14.8 Complete state space for the exciting life problem generated by breadth-first search (to two levels).

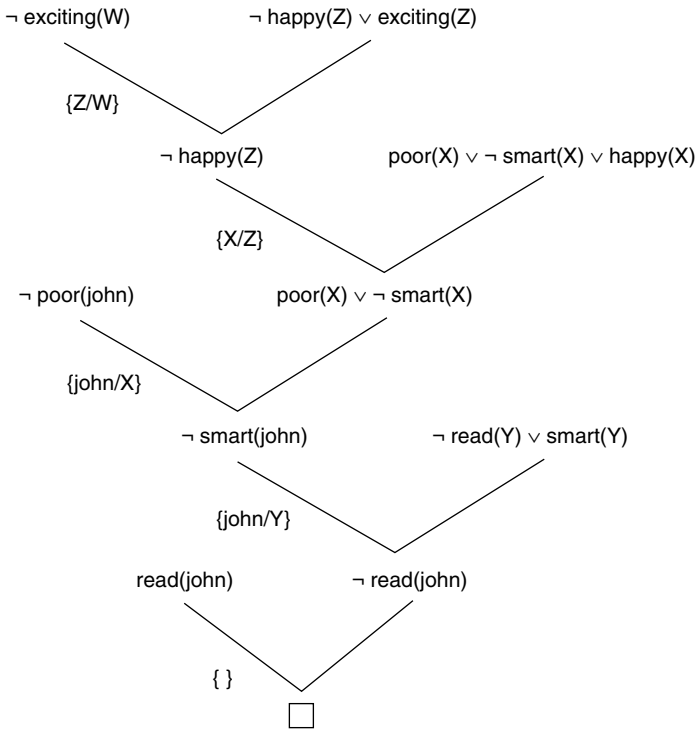


Figure 14.9 Using the unit preference strategy on the exciting life problem.

we produce a resultant clause that has fewer literals than the clauses that are resolved to create it, we are closer to producing the clause of no literals. In particular, resolving with a clause of one literal, called a *unit* clause, will guarantee that the resolvent is smaller than the largest parent clause. The unit preference strategy uses units for resolving whenever they are available. Figure 14.9 uses the unit preference strategy on the exciting life problem. The unit preference strategy along with the set of support can produce a more efficient complete strategy.

Unit resolution is a related strategy that requires that one of the resolvents always be a unit clause. This is a stronger requirement than the unit preference strategy. We can show that unit resolution is not complete using the same example that shows the incompleteness of linear input form.

The *Linear Input Form* Strategy The linear input form strategy is a direct use of the negated goal and the original axioms: take the negated goal and resolve it with one of the axioms to get a new clause. This result is then resolved with one of the axioms to get another new clause, which is again resolved with one of the axioms. This process continues until the empty clause is produced.

At each stage we resolve the clause most recently obtained with an axiom derived from the original problem statement. We never use a previously derived clause, nor do we resolve two of the axioms together. The linear input form is not a complete strategy, as can be seen by applying it to the following set of four clauses (which are obviously unsatisfiable). Regardless of which clause is taken as the negation of the goal, the linear input strategy cannot produce a contradiction:

$\neg a \vee \neg b$

$a \vee \neg b$

$\neg a \vee b$

$a \vee b$

Other Strategies and Simplification Techniques We have not attempted to present an exhaustive set of strategies or even the most sophisticated techniques for proving theorems using resolution inference. These are available in the literature, such as Wos et al. (1984) and Wos (1988). Our goal is rather to introduce the basic tools for this research area and to describe how these tools may be used in problem solving. As noted earlier, the resolution proof procedure is but another weak method problem-solving technique.

In this sense, resolution may serve as an inference engine for the predicate calculus, but an engine that requires much analysis and careful application of strategies before success. In a problem large enough to be interesting, randomly clashing expressions together with resolution is as hopeless as striking random terminal keys and hoping a quality paper will result.

The examples used in this chapter are trivially small and have all the clauses necessary (and only those necessary) for their solution. This is seldom true of interesting problems. We have given several simple strategies for combating these combinatorial complexities, and we will conclude this subsection by describing a few more important considerations in designing a resolution-based problem solver. Later we show (in Section 14.3) how a resolution refutation system, with an interesting combination of search strategies, provides a "semantics" for *logic programming*, especially for the design of Prolog interpreters.

A combination of strategies can be quite effective in controlling search—for instance, the use of set of support plus unit preference. Search heuristics may also be built into the design of rules (by creating a left-to-right ordering of literals for resolving). This order can be most effective for pruning the search space. This implicit use of strategy is important in Prolog programming (Section 14.3).

The generality of conclusions can be a criterion for designing a solution strategy. On one side it might be important to keep intermediate solutions as general as possible, as this allows them to be used more freely in resolution. Thus the introduction of any resolution with clauses that require specialization by binding variables, such as {john/X}, should be put off as long as possible. If, on the other side, a solution requires specific variable bindings, such as in the analysis of whether John has a staph infection, the {john/Person} and {staph/Infection} substitutions may restrict the search space and increase the probability and speed of finding a solution.

An important issue in selecting a strategy is the notion of completeness. It might be very important in some applications to know that a solution will be found (if one exists). This can be guaranteed by using only complete strategies.

We can also increase efficiency by speeding up the matching process. We can eliminate needless (and costly) unifications between clauses that cannot possibly produce new resolvents by indexing each clause with the literals it contains and whether they are positive or negative. This allows us directly to find potential resolvents for any clause. Also, we should eliminate certain clauses as soon as they are produced. First, any tautological clause need never be considered; these can never be falsified and so are of no use in a solution attempt.

Another type of clause that gives no new information is one that can be *subsumed*, that is, when a new clause has a more general instance already in the clause space. For example, if p(john) is deduced for a space that already contains $\forall$ X(p(X)), then p(john) may be dropped with no loss; in fact, there is a saving because there are fewer clauses in the clause space. Similarly, p(X) subsumes the clause p(X) $\vee$ q(X). Less general information does not add anything to more general information when both are in the clause space.

Finally, *procedural attachment* evaluates or otherwise processes without further search any clause that can yield new information. It does arithmetic, makes comparisons between atoms or clauses, or "runs" any other deterministic procedure that can add concrete information to the problem solving or in any manner constrain the solution process. For example, we may use a procedure to compute a binding for a variable when enough information is present to do so. This variable binding then restricts possible resolutions and prunes the search space.

Next we show how answers may be extracted from the resolution refutation process.

14.2.5 Answer Extraction from Resolution Refutations

The instances under which an hypothesis is true includes the set of substitutions with which the refutation is found. Therefore, retaining information on the unification substitutions made in the resolution refutation gives information for a correct answer. In this subsection we give three examples of this and introduce a bookkeeping method for extracting answers from a resolution refutation.

The answer recording method is simple: retain the original conclusion that was to be proved and, into that conclusion, introduce each unification that is made in the resolution process. Thus the original conclusion is the "bookkeeper" of all unifications that are made as part of the refutation. In the computational search for resolution refutations, this might require extra pointers, such as when more than one possible choice exists in the search for a refutation. A control mechanism such as backtracking may be necessary to produce alternative solution paths. But still, with a bit of care, this added information may be retained.

Let us see some examples of this process. In Figure 14.6, where a proof was found for the existence of a person with an exciting life, the unifications of Figure 14.10 were made. If we retain the original goal and apply all the substitutions of the refutation to this clause, we find the answer of which person it is who has an exciting life.

Figure 14.10 shows how a resolution refutation not only can show that "no one leads an exciting life" is false but also, in the process of that demonstration, can produce a happy person, John. This is a general result, where the unifications that produce a refutation are the same ones that produce the instances under which the original query is true.

A second example is the simple story:

Fido the dog goes wherever John, his master, goes. John is at the library. Where is Fido?

First we represent this story in predicate calculus expressions and then reduce these expressions to clause form. The predicates:

at (john,X) $\rightarrow$ at (fido,X)

at (john,library)

The clauses:

$\neg$ at (john,Y) $\vee$ at (fido,Y)

at (john,library)

The conclusion negated:

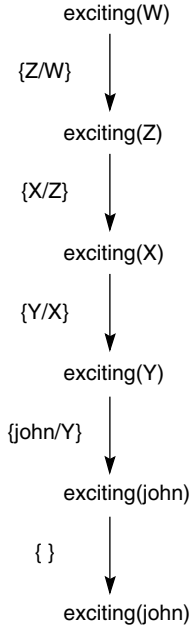


Figure 14.10 Unification substitutions of Figure 14.6 applied to the original query.

¬ at (fido,Z), Fido is nowhere!

Figure 14.11 gives the answer extraction process. The literal keeping track of unifications is the original question (where is Fido?):

at (fido,Z)

Once again, the unifications under which the contradiction is found tell how the original query is true: Fido is at the library.

The final example shows how the skolemization process can give the instance under which the answer may be extracted. Consider the following situation:

> Everyone has a parent. The parent of a parent is a grandparent. Given the person John, prove that John has a grandparent.

The following sentences represent the facts and relationships in the situation above. First, Everyone has a parent:

$$(\forall X)(\exists Y)\ p(X,Y)$$

A parent of a parent is a grandparent.

$$(\forall X)(\forall Y)(\forall Z)\ p(X,Y) \wedge p(Y,Z) \rightarrow gp(X,Z)$$

The goal is to find a W such that gp(john,W) or $\exists$ (W)(gp(john,W)). The negation of the goal is $\neg \exists$ (W)(gp(john,W)) or:

¬ gp(john,W)

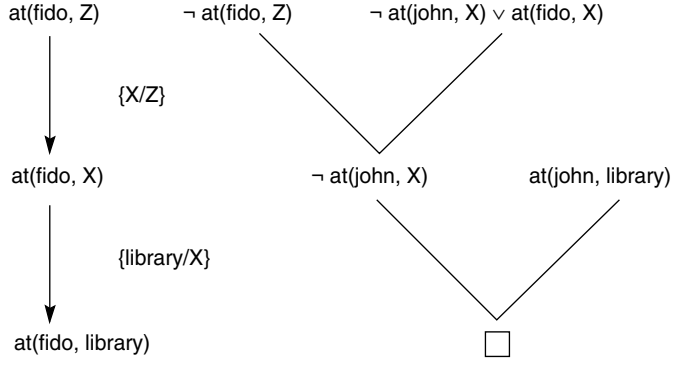


Figure 14.11 Answer extraction process on the finding fido problem.

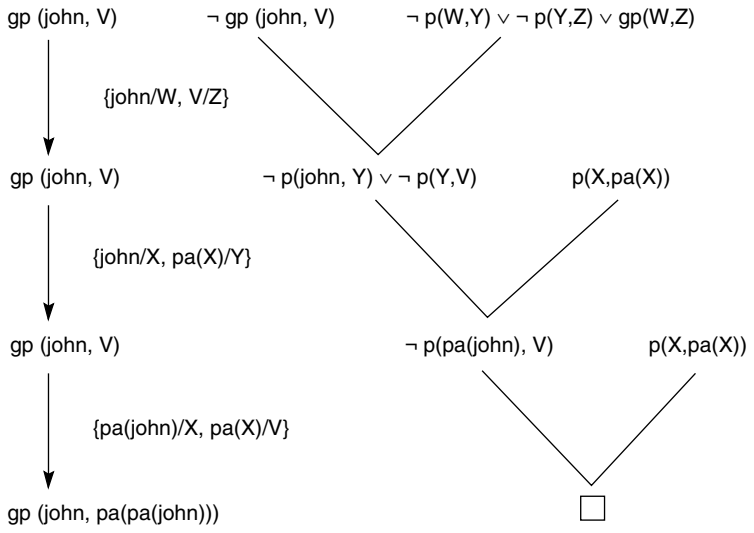


Figure 14.12 Skolemization as part of the answer extraction process.

In the process of putting this set of predicates into clause form for the resolution refutation, the existential quantifier in the first predicate (everyone has a parent) requires a skolem function. This skolem function would be the obvious function: take the given X and find the parent of X. Let's call this the pa(X) for "find a parental ancestor for X ." For John this would be either his father or his mother. The clause form for the predicates of this problem is:

p(X,pa(X))

¬ p(W,Y) ∨ ¬ p(Y,Z) ∨ gp(W,Z)

¬ gp(john,V)

The resolution refutation and answer extraction process for this problem are presented in Figure 14.12. Note that the unification substitutions in the answer are

gp(john,pa(pa(john)))

The answer to the question of whether John has a grandparent is to "find the parental ancestor of John's parental ancestor." The skolemized function allows us to compute this result.

The general process for answer extraction just described may be used in all resolution refutations, whether they be with general unifications as in Figures 14.10 and 14.11 or from evaluating the skolem function as in Figure 14.12. The process yields an answer. The method is really quite simple: the situation (unifications) under which the contradiction is found is an instance under which the opposite of the negated conclusion (the original

query of the problem) is true. Although this subsection has not given a mathematical demonstrated of why this is true in every instance, we have shown several examples of how the process works; further discussion can be found in the literature (Nilsson 1980, Wos et al. 1984).

14.3 Prolog and Automated Reasoning

14.3.1 Introduction

Only by understanding the implementation of a computer language can we properly guide its use, control its side effects, and have confidence in its results. In this section we describe the semantics of Prolog, and relate it to the issues in automated reasoning presented in the previous section.

A serious criticism of the resolution proof procedure, Section 14.2, is that it requires a totally homogeneous database to represent the problem. When predicate calculus descriptors are reduced or transformed to clause form, important heuristic problem-solving information is left out. The omitted information is not the truth or fallacy of any part of the problem but rather the control hints or procedural descriptions on how to *use* the information. For example, a negated goal clause in a resolution format might be of the form:

$$a \vee \neg b \vee c \vee \neg d$$

where a, b, c, and d are literals. The resolution inference mechanism applies a search strategy to deduce the empty clause. All literals are open to the strategy and the one used depends on the particular strategy selected. The strategies used to guide resolution theorem proving are weak heuristics; they do not incorporate deep knowledge of a specific problem domain.

For example, the negated goal clause in the resolution example above might be a transformation of the predicate calculus statement:

$$a \leftarrow b \wedge \neg c \wedge d$$

This can be understood as "to see whether a is true go out and see whether b is true and c is false and d is true." The rule was intended as a *procedure* for solving a and implements heuristic information specific to this use. Indeed, the subgoal b might offer the easiest way to falsify the entire predicate, so the order "try b then see whether c is false then test d" could save much problem-solving time and cost. The implicit heuristic says "test the easiest way to falsify the problem first, then if this is passed go ahead and generate the remaining (perhaps much more difficult) part of the solution." Human experts design procedures and relationships that not only are true but also contain information critical for *using* this truth. In most interesting problem-solving situations we cannot afford to ignore these heuristics (Kowalski 1979*b*).

In the next section we introduce Horn clauses and use their procedural interpretation as an explicit strategy that preserves this heuristic information.

14.3.2 Logic Programming and Prolog

To understand the mathematical foundations of Prolog, we first define *logic programming*. Once we have done this, we will add an explicit search strategy to logic programming to approximate the search strategy, sometimes referred to as the *procedural semantics*, of Prolog. To get full Prolog, we also discuss the use of *not* and the *closed world assumption*.

Consider a database of clauses prepared for resolution refutation, as in Section 14.2. If we restrict this set to clauses that have at most one positive literal (zero or more negative literals), we have a clause space with some interesting properties. First, problems describable with this set of clauses preserve unsatisfiability for resolution refutations, or are refutation complete, Section 14.2. Second, an important benefit of restricting our representation to this subclass of all clauses is a very efficient search strategy for refutations: a linear input form, unit preference based, left-to-right and depth-first goal reduction. With well-founded recursion (recursive calls that eventually terminate) and occurs checking, this strategy guarantees finding refutations if the clause space is unsatisfiable (van Emden and Kowalski 1976). A Horn clause contains at most one positive literal, which means it is of the form

$$a \vee \neg b_1 \vee \neg b_2 \vee \cdots \vee \neg b_n$$

where a and all the b_is are positive literals. To emphasize the key role of the one positive literal in resolutions, we generally write Horn clauses as implications with the positive literal as the conclusion:

$$a \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_n$$

Before we discuss further the search strategy, we formally define this subset of clauses, called *Horn clauses*. These, together with a *nondeterministic* goal reduction strategy, are said to constitute a *logic program*.

DEFINITION

LOGIC PROGRAM

A *logic program* is a set of universally quantified expressions in first-order predicate calculus of the form:

$$a \leftarrow b_1 \wedge b_2 \wedge b_3 \wedge \cdots \wedge b_n$$

The a and b_i are all positive literals, sometimes referred to as atomic goals. The a is the clause *head*, the conjunction of b_i, the *body*.

These expressions are the *Horn clauses* of the first-order predicate calculus. They come in three forms: first, when the original clause has no positive literals; second, when it has no negative literals; and third, when it has one positive and one or more negative literals. These cases are 1, 2, and 3, respectively:

1. $\leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_n$
   called a *headless* clause or *goals* to be tried: b_1 and b_2 and . . . and b_n.
2. $a_1 \leftarrow$
   $a_2 \leftarrow$
   .
   .
   $a_n \leftarrow$
   called the *facts*.
3. $a \leftarrow b_1 \wedge \cdots \wedge b_n$
   called a *rule* relation.

Horn clause calculus allows only the forms just presented; there may be only one literal to the left of $\leftarrow$ and this literal must be positive. All literals to the right of $\leftarrow$ are also positive.

The reduction of clauses that have at most one positive literal into Horn form requires three steps. First, select the positive literal in the clause, if there is a positive literal, and move this literal to the very left (using the commutative property of $\vee$). This single positive literal becomes the *head* of the Horn clause, as just defined. Second, change the entire clause to Horn form by the rule:

$$a \vee \neg b_1 \vee \neg b_2 \vee \cdots \vee \neg b_n \equiv a \leftarrow \neg (\neg b_1 \vee \neg b_2 \vee \cdots \vee \neg b_n)$$

Finally, use de Morgan's law to change this specification to:

$$a \leftarrow b_1 \wedge b_2 \cdots \wedge b_n$$

where the commutative property of $\wedge$ can be used to order the b_i subgoals.

It should be noted that it may not be possible to transform clauses from an arbitrary clause space to Horn form. Some clauses, such as $p \vee q$, have no Horn form. To create a Horn clause, there can be at most one positive literal in the original clause. If this criterion is not met it may be necessary to rethink the original predicate calculus specification for the problem. The payoff for Horn form representation is an efficient refutation strategy, as we see shortly.

The computation algorithm for logic programs proceeds by nondeterministic goal reduction. At each step of the computation where there is a goal of the form:

$$\leftarrow a_1 \wedge a_2 \wedge \cdots \wedge a_n$$

the interpreter *arbitrarily* chooses some a_i for $1 \leq i \leq n$. It then *nondeterministically* chooses a clause:

$$a^1 \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_n$$

such that the a^1 unifies with a_i with substitution ς and uses this clause to reduce the goal. The new goal then becomes:

$$\leftarrow (a_1 \wedge \cdots \wedge a_{i-1} \wedge b_1 \wedge b_2 \wedge \cdots \wedge b_n \wedge a_{i+1} \wedge \cdots \wedge a_n)\varsigma$$

This process of nondeterministic goal reduction continues until the computation terminates with the goal set empty.

If we eliminate the nondeterminism by imposing an order on the reduction of subgoals, we do not change the result of the computation. All results that can be found nondeterministically can be found through an exhaustive ordered search. However, by reducing the amount of nondeterminism, we can define strategies that prune unnecessary branches from the space. Thus, a major concern of practical logic programming languages is to provide the programmer with facilities to control and, when possible, reduce the amount of nondeterminism. These facilities allow the programmer to influence both the order in which the goals are reduced and the set of clauses that are used to reduce each goal. (As in any graph search, precautions must be taken to prevent infinite cycles in the proof.)

The abstract specification of a logic program has a clean semantics, that of the resolution refutation system. van Emden and Kowalski (1976) show that the smallest interpretation on which a logic program is true is *the* interpretation of the program. The price paid by practical programming languages, such as Prolog, is that executing programs by these may compute only a subset of their associated interpretations (Shapiro 1987).

Sequential Prolog is an approximation to an interpreter for the logic programming model, designed for efficient execution on von Neumann computers. This is the interpreter that we have used so far in this text. Sequential Prolog uses both the order of goals in a clause and the order of clauses in the program to control the search for a proof. When a number of goals are available, Prolog always pursues them left to right. In the search for a unifiable clause on a goal, the possible clauses are checked in the order they are presented by the programmer. When each selection is made, a backtracking pointer is placed with the recorded unification that allows other clauses to be used (again, in the programmer's order) should the original selection of a unifiable clause fail. If this attempt fails across all possible clauses in the clause space, then the computation fails. With cut, an attempt to use efficiently the depth-first backtracking search, Section 14.1.5, the interpreter may not, in fact, visit all clause combinations (interpretations) in the search space.

More formally, given a goal:

$$\leftarrow a_1 \wedge a_2 \wedge a_3 \cdots \wedge a_n$$

and a program P, the Prolog interpreter sequentially searches for the first clause in P whose head unifies with a_1. This clause is then used to reduce the goals. If:

$$a^1 \leftarrow b_1 \wedge b_2 \wedge \ldots \wedge b_n$$

is the reducing clause with ξ the unification, the goal clause then becomes:

$$\leftarrow (b_1 \wedge b_2 \wedge \cdots \wedge b_n \wedge a_2 \wedge a_3 \wedge \cdots \wedge a_n)\xi$$

The Prolog interpreter then continues by trying to reduce the leftmost goal, b_1 in this example, using the first clause in the program P that unifies with b_1. Suppose it is:

$$b^1 \leftarrow c_1 \wedge c_2 \wedge \cdots \wedge c_p$$

under unification ϕ. The goal then becomes:

$$\leftarrow (c_1 \wedge c_2 \wedge \cdots \wedge c_p \wedge b_2 \wedge \cdots \wedge b_n \wedge a_2 \wedge a_3 \wedge \cdots \wedge a_n)\xi\ \phi$$

Note that the goal list is treated as a stack enforcing depth-first search. If the Prolog interpreter ever fails to find a unification that solves a goal it then backtracks to its most recent unification choice point, restores all bindings made since that choice point, and chooses the next clause that will unify (by the order in P). In this way, Prolog implements its left-to-right, depth-first search of the clause space.

If the goal is reduced to the null clause ($\square$) then the composition of unifications that made the reductions:

$$\leftarrow (\square)\xi\ \phi \cdots \omega$$

(here $\xi\ \phi \cdots \omega$), provides an interpretation under which the original goal clause was true.

Besides backtracking on the order of clauses in a program, sequential Prolog allows the cut or "!". As described in Section 14.1.5, a cut may be placed in a clause as a goal itself. The interpreter, when encountering the cut, is committed to the current execution path and in particular to that subset of unifications made since the choice of the clause containing the cut. It also commits the interpreter to the choice of that clause itself as the only method for reducing the goal. Should failure be encountered within the clause after the cut, the entire clause fails.

Procedurally, the cut makes it unnecessary to retain backtrack pointers for the reducing clause and all its components *before* the cut. Thus, cut can mean that only some of the possible interpretations of the model are ever computed.

We summarize our discussion of sequential Prolog by comparing it to the resolution refutation model of Section 14.2.

1. The resolution clause space is a superset of Horn clause expressions in logic programming. Each clause must have at most one positive literal to be in Horn form.
2. The following structures represent the problem in Horn form:
   a. The goals,

      $$\leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_n$$

      are a list of clause statements that make up the goals to be tested by resolution refutation. Each a_i is in turn negated, unified with, and reduced until the empty clause is found (if this is possible).

b. The facts,

$$a_1 \leftarrow$$
$$a_2 \leftarrow$$
$$\vdots$$
$$a_n \leftarrow$$

are each separate clauses for resolution. Finally,

c. The Horn clause rules or axioms,

$$a \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_n$$

allow us to reduce matching subgoals.

3. With a unit preference, *linear input form* strategy (always preferring fact clauses and using the negated goal and its descendant resolvents; see Section 14.2.4) and applying a left-to-right, depth-first (with backtracking) order for selecting clauses for resolutions, the resolution theorem prover is acting as a Prolog interpreter. Because this strategy is refutation complete, its use guarantees that the solution will be found (provided that part of the set of interpretations is not pruned away by using cut).

4. Finally, the composition of unifications in the proof provides the answer (interpretation) for which the goal is true. This is exactly equivalent to the answer extraction process of Section 14.2.5. Recording the composition of unifications in the goal literal produces each answer interpretation.

An important issue with Prolog is that we wish to use negative literals on the right hand side of Horn clause rules. This requires interpreters to enforce the *closed world assumption*. In predicate calculus, the proof of $\neg\, p(X)$ is exactly the proof that $p(X)$ is logically false. That is, $p(X)$ is false under every interpretation that makes the axiom set true. The Prolog interpreter, based on the unification algorithm of Chapter 2, offers a more restricted result than the general resolution refutation of Section 14.2. Rather than trying all interpretations, it examines only those interpretations that are made explicit in the database. We now axiomatize these constraints to see exactly the restrictions implicit in current Prolog environemnts.

For every predicate p, and every variable X belonging to p, suppose $a_1, a_2, \ldots, a_n$ make up the domain of X. The Prolog interpreter, using unification, enforces:

1. The *unique name* axiom. For all atoms of the domain $a_i \neq a_j$ unless they are identical. This implies that atoms with distinct names are distinct.

2. The *closed world* axiom.

$$p(X) \rightarrow p(a_1) \vee p(a_2) \vee \cdots \vee p(a_n).$$

This means the only possible instances of a relation are those implied by the clauses present in the problem specification.

3. The *domain closure* axiom.

   $$(X = a_1) \vee (X = a_2) \vee \cdots \vee (X = a_n).$$

   This guarantees that the atoms occurring in the problem specification constitute all and the only atoms.

These three axioms are implicit in the action of the Prolog interpreter. They may be seen as added to the set of Horn clauses making up a problem description and thus as constraining the set of possible interpretations to a Prolog query.

Intuitively, this means that Prolog assumes as false all goals that it cannot prove to be true. This can introduce anomalies: if a goal's truth value is actually unknown to the current database, Prolog will assume it to be false.

Other limitations are implicit in Prolog, as they seem to be in all computing languages. The most important of these, besides the problem of negation as failure, represent violations of the semantic model for logic programming. In particular, there is the lack of an occurs check (see Section 2.3; this allows a clause to unify with a subset of itself) and the use of cut. The current generation of Prolog interpreters should be looked at pragmatically. Some problems arise because "no efficient way is currently known" to get around the issue (this is the situation with the occurs check); others arise from attempts to optimize use of the depth-first with backtrack search (explicitl controlling search with the cut). Many of the anomalies of Prolog are a result of trying to implement the nondeterministic semantics of pure logic programming on a sequential computer. This includes the problems introduced by the cut.

In the final section of Chapter 14 we introduce alternative inferencing schemes for automated reasoning.

14.4 Further Issues in Automated Reasoning

We described weak method problem solvers as using (a) a *uniform representation medium* for (b) *sound inference rules* that focus on syntactic features of the representation and are guided by (c) *methods or strategies* for combating the combinatorics of exhaustive search. We conclude this chapter with further comments on each of these aspects of the weak method solution process.

14.4.1 Uniform Representations for Weak Method Solutions

The resolution proof procedure requires us to place all our axioms in clause form. This uniform representation then allows us to resolve clauses and simplifies the design of problem solving heuristics. One major disadvantage of this approach is that much valuable heuristic information can be lost in this uniform encoding.

The if . . . then format of a rule often conveys more information for use of modus ponens or production system search than one of its syntactic variants. It also offers us an efficient way to use the rule. For instance, suppose we want to represent the abductive

inference, Section 8.0, If the engine does not turn over and the lights do not come on then the battery may be dead. This rule suggests how to check the battery.

The disjunctive form of the same rule obscures this heuristic information about how the rule should be applied. If we express this rule in predicate calculus $\neg$ turns_over $\wedge$ $\neg$ lights $\rightarrow$ battery, the clause form of this rule is this: turns_over $\vee$ lights $\vee$ battery. This clause can have a number of equivalent forms, and each of these represents a different implication.

($\neg$ turns_over $\wedge$ $\neg$ lights) $\rightarrow$ battery
($\neg$ turns_over $\rightarrow$ (battery $\vee$ lights))
($\neg$ battery $\wedge$ $\neg$ lights) $\rightarrow$ turns_over
($\neg$ battery $\rightarrow$ (turns_over $\vee$ lights))

and so on.

To retain heuristic information in the automated reasoning process several researchers, including Nilsson (1980) and Bundy (1988), advocate reasoning methods that encode heuristics by forming rules according to the way in which the human expert might design the rule relationships. We have proposed this approach already in our and/or graph reasoning in Section 3.3 and the Prolog form of automated reasoning of Section 14.3. Rule-based expert systems also allow the programmer to control search through the structure of rules. We develop the idea further with the next two examples, one data-driven and the second goal-driven. Both of these retain the form of implications and use this information to guide search through an and/or graph.

Consider, for use in data-driven reasoning, the following facts, rules (axioms), and goal:

Fact:

($a \vee (b \wedge c)$)

Rules (or axioms):

($a \rightarrow (d \wedge e)$)

($b \rightarrow f$)

($c \rightarrow (g \vee h)$)

Goal:

$\leftarrow e \vee f$

The proof of $e \vee f$ is found in the and/or graph of Figure 14.13. Note the use of and connectors on $\vee$ relations and the or connectors on $\wedge$ relations in the data-driven search space. If we are given that either a or $b \wedge c$ is true, then we must reason with both disjuncts to guarantee that our argument is truth preserving; hence these two paths are conjoined. When b and c are true, on the other hand, we can continue to explore either of these conjuncts. Rule matching takes any intermediate state, such as c, and replaces it with

the conclusion of a rule, such as $(g \vee h)$, whose premise matches that state. The discovery of both states e and f in Figure 14.13 indicates that the goal $(e \vee f)$ is established.

In a similar fashion we can use matching of rules on and/or graphs for goal-driven reasoning. When a goal description includes a $\vee$, as in the example of Figure 14.14, then either alternative can be explored independently to establish the goal. If the goal is a conjunction, then, of course, both conjuncts must be established.

Goal:

$(a \vee (b \wedge c))$

Rules (or axioms):

$(f \wedge d) \rightarrow a$

$(e \rightarrow (b \wedge c))$

$(g \rightarrow d)$

Fact:

$f \wedge g$

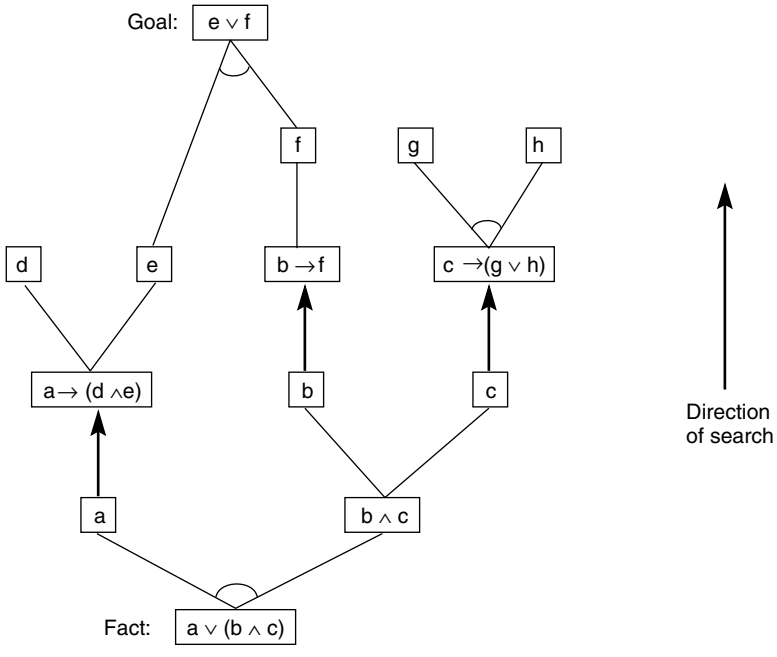


Figure 14.13 Data-driven reasoning with an and/or graph in the propositional calculus.

Although these examples are taken from the propositional calculus, a similar search is generated using predicate calculus facts and rules. Unification makes literals compatible for applying inference rules across different branches of the search space. Of course, unifications must be consistent (that is, unifiable) across different and branches of the search space.

This subsection has suggested solution methods to help preserve heuristic information within the representational medium for weak method problem solving. This is essentially the way the inference engines of expert systems allow the programmer to specify control and heuristic information in a rule. Expert systems rely on the rule form, such as the ordering of rules or the ordering of the premises within a rule, for control of search rather than depending totally on general weak problem-solving methods. What is lost in this approach is the ability to apply uniform proof procedures, such as resolution, across the full set of rules. As can be noted in the examples of Figures 14.13 and 14.14, modus ponens may still be used, however. Production system control using either depth-first, breadth-first, or best-first search offers one weak method reasoning architecture for implementing rule systems (see examples in Chapters 4, 6, and 8).

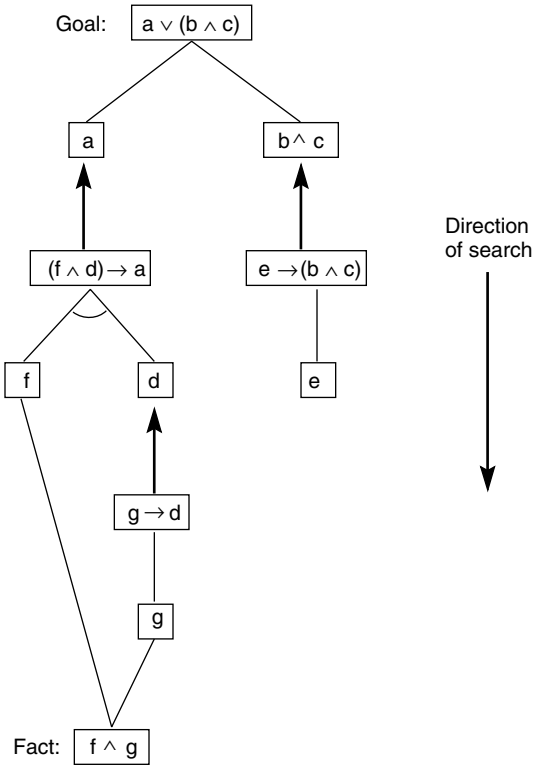


Figure 14.14 Goal-driven reasoning with an and/or graph in the propositional calculus.

14.4.2 Alternative Inference Rules

Resolution is the most general sound inference rule we have presented so far. Several more sophisticated inference rules have been created in an attempt to make resolution more efficient. We briefly consider two of these: *hyperresolution* and *paramodulation*.

Resolution, as we have presented it, is actually a special variant called *binary resolution*: exactly two parent clauses are clashed. A successful application of hyperresolution replaces a sequence of binary resolutions to produce one clause. Hyperresolution clashes, in a single step, a clause with some negative literals, referred to as the *nucleus*, and a number of clauses with all positive literals, called the *satellites*. These satellites must have one positive literal that will match with a negative literal of the nucleus. There must also be one satellite for each negative literal of the nucleus. Thus the result of an application of hyperresolution is a clause with all positive literals.

An advantage of hyperresolution is that a clause of all positive literals is produced from each hyperresolution inference, and the clause space itself is kept smaller because no intermediate results are produced. Unifications across all clauses in the inference step must be consistent.

As an example of hyperresolution, consider the following clause set:

$\neg$ married(X,Y) $\vee$ $\neg$ mother(X,Z) $\vee$ father(Y,Z)

married(kate,george) $\vee$ likes(george,kate)

mother(kate,sarah)

We draw a conclusion in one step using hyperresolution:

father(george,sarah) $\vee$ likes(george,kate)

The first clause in the example is the nucleus; the second two are satellites. The satellites are all positive, and there is one for each negative literal in the nucleus. Note how the nucleus is just the clause form for the implication:

married(X,Y) $\wedge$ mother(X,Z) $\rightarrow$ father(Y,Z)

The conclusion of this rule is part of the final result. Note that there are no intermediate results, such as:

$\neg$ mother(kate,Z) $\vee$ father(george,Z) $\vee$ likes(george,kate)

which we would find in any binary resolution proof applied to the same clause space.

Hyperresolution is sound and complete when used by itself. When combined with other strategies, such as set of support, completeness may be compromised (Wos et al. 1984). It does require special search strategies to organize the satellite and nucleus clauses, although in most environments where hyperresolution is used, the clauses are often indexed by the name and positive or negative property of each literal. This makes it efficient to prepare the nucleus and satellite clauses for the hyperresolution inference.

An important and difficult issue in the design of theorem-proving mechanisms is the control of equality. Especially complex are application areas, such as mathematics, where most facts and relationships have multiple representations, such as can be obtained by applying the associative and commutative properties to expressions. To convince yourself of this with a very simple example, consider the multiple ways the arithmetic expression 3 + (4 + 5) can be represented, including 3 + ((4 + 0) + 5). This is a complex issue in that expressions need to be substituted for, unified with, and checked for equality with other expressions within automated mathematical problem solving.

Demodulation is the process of rephrasing or rewriting expressions so they automatically take on a chosen canonical form. The unit clauses used to produce this canonical form are called *demodulators*. Demodulators specify the equality of different expressions, allowing us to replace an expression with its canonical form. With proper use of demodulators all newly produced information is reduced to a specified form before it is placed in the clause space. For example, we might have a demodulator:

```
equal(father(father(X)),grandfather(X))
```

and the new clause:

```
age(father(father(sarah)),86).
```

Before adding this new clause to the clause space, we apply the demodulator and add instead:

```
age(grandfather(sarah),86).
```

The equality problem here is one of naming. Do we wish to classify a person as father(father(X)) or grandfather(X)? Similarly, we can pick out canonical names for all family relations: a brother(father(Y)) is uncle(Y), etc. Once we pick the canonical names to store information under, we then design demodulators such as the equal clause to reduce all new information to this determined form. Note that demodulators are always unit clauses.

Paramodulation is a generalization of equality substitution at the term level. For example, given the expression:

```
older(mother(Y),Y)
```

and the equality relationship:

```
equal(mother(sarah),kate)
```

we can conclude with paramodulation:

```
older(kate,sarah)
```

Note the term-level matching and replacement of {sarah/Y} and mother(sarah) for kate. A vital difference between demodulation and paramodulation is that the latter allows a nontrivial replacement of variables in both the arguments of the equality predicate and the predicate into which the substitution is made. Demodulation does replacement based on the demodulator. Multiple demodulators may be used to get an expression into its final form; paramodulation is usually used only once in any situation.

We have given several simple examples of these powerful inference mechanisms. They should be seen as more general techniques for use in a resolution clause space. Like all the other inference rules we have seen, these are tightly linked to the chosen representation and must be controlled by appropriate strategies.

14.4.3 Question Answering Supported by Resolution Refutation

In Section 14.2.5 we demonstrated a natural fit between the process of resolution refutation and the generation of answers for the original query (the theorem to be proved). Stuart Shapiro and his colleagues (Burhans and Shapiro 2005) demonstrate how resolution refutations also offer a useful paradigm for question answering.

Burhans and Shipiro note that the refutation process partitions the clauses generated during resolution refutations into three classes, each appropriate for answering different types of questions about the theorem under consideration. In their research clauses that are relevant are all identified as possible answers, where relevance is determined according to a question (the theorem to be proved) and a knowledge base (the clause set used in the proof), where any clause descended from the clause form of the negated goal is deemed relevant.

The refutation proof is partitioned into three answer classes: *specific*, *generic*, and *hypothetical*. These classes are distinguished by the way the literals making up the clause share variables. The result is a context independent, i.e., it can be used with any set-of-support based refutation, pragmatics for question answering.

Specific answers, sometimes referred to as *extensional* answers (Green 1969a, b), are associated with the set of ground terms corresponding to the constants substituted for variable values in the process of producing a refutation. For example, the query "is there anyone at home?" would be replaced by the negated goal, "there is no one at home" and would lead to a refutation when it is shown that "Amy is at home". This set of individuals that can be produced through the refutation process provide the specific answers.

Generic answers, also called *intensional* answers (Green 1969a,b), correspond to all non-ground instances of answering clauses that can be associated with the refutation. For example, for the query "is there anyone at home?", any variable-based clauses, such as "all the children are at home", provided these clauses are contained in or are inferable from the clause space, make up the set of generic answers.

Finally, hypothetical answers, sometimes called *conditional* or *qualified*, take the form of a rule whose antecedent has some ground variables and whose consequent matches the negated answer query. For example, "is there anyone at home?" might match with the clause "if Allen is at home then Amy is at home" with the subsequent search to determine whether Allen is at home. For more details see Burhans and Shiparo (2005).

14.4.4 Search Strategies and Their Use

Sometimes the domain of application puts special demands on the inference rules and heuristics for guiding their use. We have already seen the use of demodulators for assistance in equality substitution. Bledsoe, in his *natural deduction system*, identifies two important strategies for preparing theorems for resolution proof. He calls these strategies s*plit* and *reduce* (Bledsoe 1971).

Blades designed his strategies for use in mathematics and, in particular, for application to *set theory*. The effect of these strategies is to break a theorem into parts to make it easier to prove by conventional methods such as resolution. Split takes various mathematical forms and splits them to appropriate pieces. The proof of $A \wedge B$ is equivalent to the proof of A and the proof of B. Similarly, the proof of $A \leftrightarrow B$ is the proof of $A \rightarrow B$ and the proof of $A \leftarrow B$.

The heuristic reduce also attempts to break down large proofs to their components. For example, the proof of $s \in A \cap B$ may be decomposed into the proofs of $s \in A$ and $s \in B$. Again, to prove a property true of $\neg (A \cup B)$ prove the property for $\neg A$ and for $\neg B$. By breaking up larger proofs into smaller pieces, Bledsoe hopes to contain the search space. His heuristics also include a limited use of equality substitution.

As mentioned throughout this book, the appropriate use of heuristics is very much an art that takes into account the application area as well as the representation and inference rules used. We close this chapter by citing some general proverbs, all of which are sometimes false but which can, with careful use, be very effective. These proverbs sum up thoughts taken from researchers in the area (Bledsoe 1971, Nilsson 1980, Wos et al. 1984, Dallier 1986, Wos 1988) as well as our own reflections on weak method problem solvers. We state them without further comment.

Use, whenever possible, clauses with fewer literals.

Break the task into subtasks before employing general inferencing.

Use equality predicates whenever this is appropriate.

Use demodulators to create canonical forms.

Use paramodulation when inferencing with equality predicates.

Use strategies that preserve “completeness”.

Use set of support strategies, for these contain the potential contradiction.

Use units within resolution, as these shorten the resulting clause.

Perform subsumption checks with new clauses.

Use an ordering mechanism on clauses and literals within the clauses that reflect your intuitions and problem-solving expertise.

14.5 Epilogue and References

Weak method solvers require choosing a representational medium, inference mechanisms, and search strategies. These three choices are intricately interwoven and cannot be made in isolation from each other. The application domain also affects the choice of representation, inference rules, and strategies. The "proverbs" at the end of the previous section should be considered in making these choices.

Resolution is the process of constraining possible interpretations until the clause space with the inclusion of the negated goal is inconsistent. We do not address the soundness of resolution or the completeness of resolution refutations. These proofs are based on Herbrand's theorem (Chang and Lee 1973) and the notion of possible interpretations of the clause set. The interested reader is encouraged to go to the references for these proofs.

A number of other references are appropriate: *Automated Theorem Proving: A Logical Basis* offers a formal approach (Loveland 1978). A number of classic early papers in the field are collected in a series *The Automation of Reasoning: Collected Papers, 1957 to 1970* (Siekmann and Wrightson 1983*a*, *b*). Nilsson (1980), Weyhrauch (1980), Genesereth and Nilsson (1987), Kowalski (1979*b*), Lloyd (1984) Wos et al. (1984), and Wos (1988) offer summaries of important concepts in automated reasoning. Robinson (1965) and Bledsoe (1977) have made fundamental contributions to the field. An important research contribution is made by Boyer and Moore (1979). Early theorem-proving work by Newell and Simon and their colleagues is reported in *Computers and Thought* (Feigenbaum and Feldman 1963) and *Human Problem Solving* (Newell and Simon 1972). Question answering based on resolution refutations is described by Burhans and Shapiro (2005).

CADE, the Conference on Automated DEduction is a major forum for the presentation of results in automated reasoning. Model checking, verification systems, and scalable knowledge representation, are current research issues (McAllester 1999, Ganzinger et al. 1999, Veroff and Spinks 2006). Research by Wos and his colleagues at Argonne National Laboratory is important. Veroff (1997) has published a set of essays in honor of Wos. Bundy's work (1983, 1988) in automated reasoning at the University of Edinburgh is also important, as is extending the Boyer-Moore theorem prover at the University of Texas, Austin, see *Computer-Aided Reasoning: ACL2 Case Studies*, Kaufmann et al. (2000).

14.6 Exercises

1. Take the logic-based financial advisor of Section 2.4, put the predicates describing the problem into clause form, and use resolution refutations to answer queries such as whether a particular investor should make an investment(combination).

2. Use resolution to prove Wirth's statement in Exercise 12, Chapter 2.

3. Use resolution to answer the query in Example 3.3.4.

4. In Chapter 5 we presented a simplified form of the knight's tour. Take the path3 rule, put it in clause form, and use resolution to answer queries such as path3(3,6). Next, use the recursive path call, in clause form, to answer queries.

5. How might you use resolution to implement a "production system" search?

6. How would you do data-driven reasoning with resolution? Use this to address the search space of Exercise 1. What problems might arise in a large problem space?

7. Use resolution for queries in the farmer, wolf, goat, and cabbage problem of Section 15.3.

8. Use resolution to solve the following puzzle from Wos et al. (1984). Four people: Roberta, Thelma, Steve, and Pete hold eight different jobs, each person with exactly two jobs. The jobs are, chef, guard, nurse, telephonist, police officer, teacher, actor, and boxer. The nurse is a male. The husband of the chef is the telephonist. Roberta is not a boxer. Pete has no education over ninth grade. Roberta, the chef, and the police officer went golfing together. Who holds which jobs? Show how the addition of a sex bias can change the problem.

9. Work out two examples for hyperresolution where the nucleus has at least four literals.

10. Write a demodulator for sum that would cause clauses of the form equal(ans, sum(5, sum(6, minus(6)))) to be reduced to equal(ans, sum(5, 0)). Write a further demodulator to reduce this last result to equal(ans, 5).

11. Pick a "canonical set" of six family relations. Write demodulators to reduce alternative forms of relations to the set. For example, your "mother's brother" is "uncle."

12. Take the happy student problem of Figure 14.5 and apply three of the refutation strategies of Section 14.2.4 to its solution.

13. Put the following predicate calculus expression in clause form:

$$\forall\ (X)(p(X) \rightarrow \{\forall\ (Y)[p(Y) \rightarrow p(f(X,Y))] \wedge \neg\ \forall\ (Y)[q(X,Y) \rightarrow p(Y)]\})$$

14. Create the and/or graph for the following data-driven predicate calculus deduction.

    Fact: $\neg\ d(f) \vee [b(f) \wedge c(f)]$.

    Rules: $\neg\ d(X) \rightarrow \neg\ a(X)$ and $b(Y) \rightarrow e(Y)$ and $g(W) \leftarrow c(W)$.

    Prove: $\neg\ a(Z) \vee e(Z)$.

15. Prove the linear input form strategy is not refutation complete.

16. Create the and/or graph for the following problem. Why is it impossible to conclude the goal:

    $r(Z) \vee s(Z)$?

    Fact: $p(X) \vee q(X)$.

    Rules: $p(a) \rightarrow r(a)$ and $q(b) \rightarrow s(b)$.

17. Use factoring and resolution to produce a refutation for the following clauses: $p(X) \vee p(f(Y))$ and $\neg\ p(W) \vee \neg\ p(f(Z))$. Try to produce a refutation without factoring.

18. Derive a resolution proof of the theorem of Figure 14.1.

19. An alternative semantic model for logic programming is that of *Flat Concurrent Prolog*. Compare Prolog as seen in Section 14.3 with Flat Concurrent Prolog (Shapiro 1987).