CHAPTER

5

USING PREDICATE LOGIC

Nature cares nothing for logic, our human logic: she has her own, which we do not recognize and do not acknowledge until we are crushed under its wheel.

—Ivan Turgenev
(1818–1883), Russian novelist and playwright

In this chapter, we begin exploring one particular way of representing facts — the language of logic. Other representational formalisms are discussed in later chapters. The logical formalism is appealing because it immediately suggests a powerful way of deriving new knowledge from old — mathematical deduction. In this formalism, we can conclude that a new statement is true by proving that it follows from the statements that are already known. Thus the idea of a proof, as developed in mathematics as a rigorous way of demonstrating the truth of an already believed proposition, can be extended to include deduction as a way of deriving answers to questions and solutions to problems.

One of the early domains in which AI techniques were explored was mechanical theorem proving, by which was meant proving statements in various areas of mathematics. For example, the Logic Theorist [Newell *et al.*, 1963] proved theorems from the first chapter of Whitehead and Russell's *Principia Mathematica* [1950]. Another theorem prover [Gelernter *et al.*, 1963] proved theorems in geometry. Mathematical theorem proving is still an active area of AI research. (See, for example, Wos *et al.* [1984].) But, as we show in this chapter, the usefulness of some mathematical techniques extends well beyond the traditional scope of mathematics. It turns out that mathematics is no different from any other complex intellectual endeavor in requiring both reliable deductive mechanisms and a mass of heuristic knowledge to control what would otherwise be a completely intractable search problem.

At this point, readers who are unfamiliar with propositional and predicate logic may want to consult a good introductory logic text before reading the rest of this chapter. Readers who want a more complete and formal presentation of the material in this chapter should consult Chang and Lee [1973]. Throughout the chapter, we use the following standard logic symbols: "$\rightarrow$" (*material implication*), "$\neg$" (*not*), "$\vee$" (*or*), "$\wedge$" (*and*), "$\forall$" (*for all*), and "$\exists$" (*there exists*).

5.1 REPRESENTING SIMPLE FACTS IN LOGIC

Let's first explore the use of propositional logic as a way of representing the sort of world knowledge that an AI system might need. Propositional logic is appealing because it is simple to deal with and a decision procedure for it exists. We can easily represent real-world facts as logical *propositions* written as *well-formed formulas* (*wff's*) in propositional logic, as shown in Fig. 5.1. Using these propositions, we could, for example, conclude from the fact that it is raining the fact that it is not sunny. But very quickly we run up against the limitations of propositional logic. Suppose we want to represent the obvious fact stated by the classical sentence

It is raining.
RAINING

It is sunny.
SUNNY

It is windy.
WINDY

If it is raining, then it is not sunny.
$RAINING \rightarrow \neg SUNNY$

Fig. 5.1 *Some Simple Facts in Prepositional Logic*

Socrates is a man.

We could write:

SOCRATESMAN

But if we also wanted to represent

Plato is a man.

we would have to write something such as:

PLATOMAN

which would be a totally separate assertion, and we would not be able to draw any conclusions about similarities between Socrates and Plato. It would be much better to represent these facts as:

MAN(SOCRATES)
MAN(PLATO)

since now the structure of the representation reflects the structure of the knowledge itself. But to do that, we need to be able to use predicates applied to arguments. We are in even more difficulty if we try to represent the equally classic sentence

All men are mortal.

We could represent this as:

MORTALMAN

But that fails to capture the relationship between any individual being a man and that individual being a mortal. To do that, we really need variables and quantification unless we are willing to write separate statements about the mortality of every known man.

So we appear to be forced to move to first-order predicate logic (or just predicate logic, since we do not discuss higher order theories in this chapter) as a way of representing knowledge because it permits representations of things that cannot reasonably be represented in prepositional logic. In predicate logic, we can represent real-world facts as *statements* written as wff's.

But a major motivation for choosing to use logic at all is that if we use logical statements as a way of representing knowledge, then we have available a good way of reasoning with that knowledge. Determining the validity of a proposition in propositional logic is straightforward, although it may be computationally hard. So before we adopt predicate logic as a good medium for representing knowledge, we need to ask whether it also provides a good way of reasoning with the knowledge. At first glance, the answer is yes. It provides a way of deducing new statements from old ones. Unfortunately, however, unlike propositional logic, it does not possess a decision procedure, even an exponential one. There do exist procedures that will find a proof of a proposed theorem if indeed it is a theorem. But these procedures are not guaranteed to halt if the proposed statement is not a theorem. In other words, although first-order predicate logic is not decidable, it is semidecidable. A simple such procedure is to use the rules of inference to generate theorem's from the axioms in some orderly fashion, testing each to see if it is the one for which a proof is sought. This method is not particularly efficient, however, and we will want to try to find a better one.

Although negative results, such as the fact that there can exist no decision procedure for predicate logic, generally have little direct effect on a science such as AI, which seeks positive methods for doing things, this particular negative result is helpful since it tells us that in our search for an efficient proof procedure, we should he content if we find one that will prove theorems, even if it is not guaranteed to halt if given a nontheorem. And the fact that there cannot exist a decision procedure that halts on all possihle inputs does not mean that there cannot exist one that will halt on almost all the inputs it would see in the process of trying to solve real problems. So despite the theoretical undecidability of predicate logic, it can still serve as a useful way of representing and manipulating some of the kinds of knowledge that an AI system might need.

Let's now explore the use of predicate logic as a way of representing knowledge by looking at a specific example. Consider the following set of sentences:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

The facts described by these sentences can be represented as a set of wff's in predicate logic as follows:

1. Marcus was a man.

   man(*Marcus*)

   This representation captures the critical fact of Marcus being a man. It fails to capture some of the information in the English sentence, namely the notion of past tense. Whether this omission is acceptable or not depends on the use to which we intend to put the knowledge. For this simple example, it will be all right.

2. Marcus was a Pompeian.

   $Pompeian(Marcus)$

3. All Pompeians were Romans.

   $\forall x : Pompeian(x) \rightarrow Roman(x)$

4. Caesar was a ruler.

   $ruler(Caesar)$

   Here we ignore the fact that proper names are often not references to unique individuals, since many people share the same name. Sometimes deciding which of several people of the same name is being referred to in a particular statement may require a fair amount of knowledge and reasoning.
5. All Romans were either loyal to Caesar or hated him.

   $\forall x : Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

   In English, the word "or" sometimes means the logical *inclusive-or* and sometimes means the logical *exclusive-or* (XOR). Here we have used the inclusive interpretation. Some people will argue, however, that this English sentence is really stating an,exclusive-or. To express that, we would have to write:

   $\forall x : Romman(x) \rightarrow [(loyal\ to(x, Caesar) \vee hate(x, Caesar)) \wedge$
   $\neg(loyalto(x, Caesar) \wedge hate(x, Caesar))]$

6. Everyone is loyal to someone.

   $\forall x : \rightarrow y : loyalto(x,y)$

   A major problem that arises when trying to convert English sentences into logical statements is the scope of quantifiers. Does this sentence say, as we have assumed in writing the logical formula above, that for each person there exists someone to whom he or she is loyal, possibly a different someone for everyone? Or does it say that there exists someone to whom everyone is loyal (which would be written as $\exists y : \forall x : loyalto(x,y)$)? Often only one of the two interpretations seems likely, so people tend to favor it.
7. People only try to assassinate rulers they are not loyal to.

   $\forall x : \forall y : person(x) \wedge ruler(y) \wedge tryassassinate(x,y) \rightarrow \neg\ loyalto(x,y)$

   This sentence, too, is ambiguous. Does it mean that the only rulers that people try to assassinate are those to whom they are not loyal (the interpretation used here), or does it mean that the only thing people try to do is to assassinate rulers to whom they are not loyal?

   In representing this sentence the way we did, we have chosen to write "try to assassinate" as a single predicate. This gives a fairly simple representation with which we can reason about trying to assassinate. But using this representation, the connections between trying to assassinate and trying to do other things and between trying to assassinate and actually assassinating could not be made easily. If such connections were necessary, we would need to choose a different representation.
8. Marcus tried to assassinate Caesar.

   $tryassassinate\ (Marcus, Caesar)$

From this brief attempt to convert English sentences into logical statements, it should be clear how difficult the task is. For a good description of many issues involved in this process, see Reichenbach [1947].

Now suppose that we want to use these statements to answer the question

Was Marcus loyal to Caesar?

It seems that using 7 and 8, we should be able to prove that Marcus was not loyal to Caesar (again ignoring the distinction between past and present tense). Now let's try to produce a formal proof, reasoning backward from the desired goal:

$\neg loyalto(Marcus, Caesar)$

In order to prove the goal, we need to use the rules of inference to transform it into another goal (or possibly a set of goals) that can in turn be transformed, and so on, until there are no unsatisfied goals remaining. This process may require the search of an AND-OR graph (as described in Section 3.4) when there are alternative ways of satisfying individual goals. Here, for simplicity, we show only a single path. Figure 5.2 shows an attempt to produce a proof of the goal by reducing the set of necessary but as yet unattained goals to the empty set. The attempt fails, however, since there is no way to satisfy the goal *person* (*Murcus*) with the statements we have available.

The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. We need to add the representation of another fact to our system, namely:

$$
\begin{array}{l}
\neg\, loyalto(Marcus, Caesar) \\
\quad\quad \uparrow \quad\quad (7, \text{substitution}) \\
person(Marcus) \wedge \\
ruler(Caesar) \wedge \\
tryassassinate(Marcus, Caesar) \\
\quad\quad \uparrow \quad\quad (4) \\
person(Marcus) \\
tryassassinate(Marcus, Caesar) \\
\quad\quad \uparrow \quad\quad (8) \\
person(Marcus)
\end{array}
$$

Fig. 5.2 *An Attempt to Prove ¬loyalto(Marcus,Caesar)*

9. All men are people.
$\forall : man(x) \rightarrow person(x)$
Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.

From this simple example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones:

- Many English sentences are ambiguous (for example, 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
- There is often a choice of how to represent the knowledge (as discussed in connection with 1, and 7 above). Simple representations are desirable, but they may preclude certain kinds of reasoning. The expedient representation for a particular set of sentences depends on the use to which the knowledge contained in the sentences will be put.

- Even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand. In order to be able to use a set of statements effectively, it is usually necessary to have access to another set of statements that represent facts that people consider too obvious to mention. We discuss this issue further in Section 10.3.

An additional problem arises in situations where we do not know in advance which statements to deduce. In the example just presented, the object was to answer the question "Was Marcus loyal to Caesar?" How would a program decide whether it should try to prove

loyalto(Marcus, Caesar)
¬loyalto(Marcus, Caesar)

There are several things it could do. It could abandon the strategy we have outlined of reasoning backward from a proposed truth to the axioms and instead try to reason forward and see which answer it gets to. The problem with this approach is that, in general, the branching factor going forward from the axioms is so great that it would probably not get to either answer in any reasonable amount of time. A second thing it could do is use some sort of heuristic rules for deciding which answer is more likely and then try to prove that one first. If it fails to find a proof after some reasonable amount of effort, it can try the other answer. This notion of limited effort is important, since any proof procedure we use may not halt if given a nontheorem. Another thing it could do is simply try to prove both answers simultaneously and stop when one effort is successful. Even here, however, if there is not enough information available to answer the question with certainty, the program may never halt. Yet a fourth strategy is to try both to prove one answer and to disprove it, and to use information gained in one of the processes to guide the other.

5.2 REPRESENTING INSTANCE AND *ISA* RELATIONSHIPS

In Chapter 4, we discussed the specific attributes *instance* and *isa* and described the important role they play in a particularly useful form of reasoning, property inheritance. But if we look back at the way we just represented our knowledge about Marcus and Caesar, we do not appear to have used these attributes at all. We certainly have not used predicates with those names. Why not? The answer is that although we have not used the predicates *instance* and *isa* explicitly, we have captured the relationships they are used to express, namely class membership and class inclusion.

Figure 5.3 shows the first five sentences of the last section represented in logic in three different ways. The first part of the figure contains the representations we have already discussed. In these representations, class membership is represented with unary predicates (such as *Roman*), each of which corresponds to a class. Asserting that $P(x)$ is true is equivalent to asserting that x is an instance (or element) of P. The second part of the figure contains representations that use the *instance* predicate explicitly. The predicate *instance* is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs. But these representations do not use an explicit *isa* predicate. Instead, subclass relationships, such as that between Pompeians and Romans, are described as shown in sentence 3. The implication rule there states that if an object is an instance of the subclass *Pompeian* then it is an instance of the superclass *Roman*. Note that this rule is equivalent to the standard set-theoretic definition of the subclass-superclass relationship. The third part contains representations that use both the *instance* and *isa* predicates explicitly. The use of the *isa* predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided. This additional axiom describes how an *instance* relation and *an isa* relation can be combined to derive a new *instance* relation. This one additional axiom is general, though, and does not need to be provided separately for additional *isa* relations.

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\forall x : Pompeian(x) \rightarrow Roman(x)$
4. $ruler(Caesar)$
5. $\forall x : Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

1. $instance(Marcus, man)$
2. $instance(Marcus, Pompeian)$
3. $\forall x : instance(x, Pompeian) \rightarrow instance(x, Roman)$
4. $instance(Caesar, ruler)$
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

1. $instance(Marcus, man)$
2. $instance(Marcus, Pompeian)$
3. $isa(Pompeian, Roman)$
4. $instance(Caesar, ruler)$
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$
6. $\forall x : \forall y : \forall z : instance(x, y) \wedge isa(y, z) \rightarrow instance(x, z)$

Fig. 5.3 *Three Ways of Representing Class Membership*

These examples illustrate two points. The first is fairly specific. It is that, although class and superclass memberships are important facts that need to be represented, those memberships need not be represented with predicates labeled *instance* and *isa*. In fact, in a logical framework it is usually unwieldy to do that, and instead unary predicates corresponding to the classes are often used. The second point is more general. There are usually several different ways of representing a given fact within a particular representational framework, be it logic or anything else. The choice depends partly on which deductions need to be supported most efficiently and partly on taste. The only important thing is that within a particular knowledge base consistency of representation is critical. Since any particular inference rule is designed to work on one particular form of representation, it is necessary that all the knowledge to which that rule is intended to apply be in the form that the rule demands. Many errors in the reasoning performed by knowledge-based programs are the result of inconsistent representation decisions. The moral is simply to be careful.

There is one additional point that needs to be made here on the subject of the use of *isa* hierarchies in logic-based systems. The reason that these hierarchies are so important is not that they permit the inference of superclass membership. It is that by permitting the inference of superclass membership, they permit the inference of other properties associated with membership in that superclass. So, for example, in our sample knowledge base it is important to be able to conclude that Marcus is a Roman because we have some relevant knowledge about Romans, namely that they either hate Caesar or are loyal to him. But recall that in the baseball example of Chapter 4, we were able to associate knowledge with superclasses that could then be overridden by more specific knowledge associated either with individual instances or with subclasses. In other words, we recorded default values that could be accessed whenever necessary. For example, there was a height associated with adult males and a different height associated with baseball players. Our procedure for manipulating the *isa* hierarchy guaranteed that we always found the correct (i.e., most specific) value for any attribute. Unfortunately, reproducing this result in logic is difficult.

Suppose, for example, that, in addition to the facts we already have, we add the following.[1]

$Pompeian(Paulus)$
$\neg\, [loyalto(Paulus, Caesar) \vee hate(Paulus, Caesar)]$

[1] For convenience, we now return to our original notation using unary predicates to denote class relations.

In other words, suppose we want to make Paulus an exception to the general rule about Romans and their feelings toward Caesar. Unfortunately, we cannot simply add these facts to our existing knowledge base the way we could just add new nodes into a semantic net. The difficulty is that if the old assertions are left unchanged, then the addition of the new assertions makes the knowledge base inconsistent. In order to restore consistency, it is necessary to modify the original assertion to which an exception is being made. So our original sentence 5 must become:

$$\forall x : Roman(x) \wedge \neg eq(x, Paulus) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$$

In this framework, every exception to a general rule' must be stated twice, once in a particular statement and once in an exception list that forms part of the general rule. This makes the use of general rules in this framework less convenient and less efficient when there are exceptions than is the use of general rules in a semantic net.

A further problem arises when information is incomplete and it is not possible to prove that no exceptions apply in a particular instance. But we defer consideration of this problem until Chapter 7.

5.3 COMPUTABLE FUNCTIONS AND PREDICATES

In the example we explored in the last section, all the simple facts were expressed as combinations of individual predicates, such as:

$$tryassassinate(Marcus, Caesar)$$

This is fine if the number of facts is not very large or if the facts themselves are sufficiently unstructured that there is little alternative. But suppose we want to express simple facts, such as the following greater-than and less-than relationships:

$$\begin{array}{ll} gt(1,0) & lt(0,1) \\ gt(2,1) & lt(1,2) \\ gt(3,2) & lt(2,3) \\ \vdots & \vdots \end{array}$$

Clearly we do not want to have to write out the representation of each of these facts individually. For one thing, there are infinitely many of them. But even if we only consider the finite number of them that can be represented, say, using a single machine word per number, it would be extremely inefficient to store explicitly a large set of statements when we could, instead, so easily compute each one as we need it. Thus it becomes useful to augment our representation by these *computable predicates*. Whatever proof procedure we use, when it comes upon one of these predicates, instead of searching for it explicitly in the database or attempting to deduce it by further reasoning, we can simply invoke a procedure, which we will specify in addition to our regular rules, that will evaluate it and return true or false.

It is often also useful to have computable functions as well as computable predicates. Thus we might want to be able to evaluate the truth of

$$gt(2 + 3, 1)$$

To do so requires that we first compute the value of the plus function given the arguments 2 and 3, and then send the arguments 5 and 1 to *gt*.

The next example shows how these ideas of computable functions and predicates can be useful. It also makes use of the notion of equality and allows equal objects to be substituted for each other whenever it appears helpful to do so during a proof.

Consider the following set of facts, again involving Marcus:

1. **Marcus was a man.**

   man(Marcus)

   Again we ignore the issue of tense.
2. Marcus was a Pompeian.

   Pompeian(Marcus)

3. Marcus was born in 40 A.D.

   born(Marcus, 40)

   For simplicity, we will not represent A.D. explicitly, just as we normally omit it in everyday discussions. If we ever need to represent dates B.C., then we will have to decide on a way to do that, such as by using negative numbers. Notice that the representation of a sentence does not have to look like the sentence itself as long as there is a way to convert back and forth between them. This allows us to choose a representation, such as positive and negative numbers, that is easy for a program to work with.
4. All men are mortal.

   $\forall x: man(x) \rightarrow mortal(x)$

5. All Pompeians died when the volcano erupted in 79 A.D.

   $erupted(volcano, 79) \wedge \forall x : [Pompeian(x) \rightarrow died(x, 79)]$

   This sentence clearly asserts the two facts represented above. It may also assert another that we have not shown, namely that the eruption of the volcano caused the death of the Pompeians. People often assume causality between concurrent events if such causality seems plausible.

   Another problem that arises in interpreting this sentence is that of determining the referent of the phrase "the volcano." There is more than one volcano in the world. Clearly the one referred to here is Vesuvius, which is near Pompeii and erupted in 79 A.D. In general, resolving references such as these can require both a lot of reasoning and a lot of additional knowledge.
6. No mortal lives longer than 150 years.

   $\forall x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$

   There are several ways that the content of this sentence could be expressed. For example, we could introduce a function *age* and assert that its value is never greater than 150. The representation shown above is simpler, though, and it will suffice for this example.
7. It is now 1991.

   $now = 1991$

   Here we will exploit the idea of equal quantities that can be substituted for each other.

Now suppose we want to answer the question "Is Marcus alive?" A quick glance through the statements we have suggests that there may be two ways of deducing an answer. Either we can show that Marcus is dead because he was killed by the volcano or we can show that he must be dead because he would otherwise be more than 150 years old, which we know is not possible. As soon as we attempt to follow either of those paths rigorously, however, we discover, just as we did in the last example, that we need some additional knowledge. For example, our statements talk about dying, but they say nothing that relates to being alive, which is what the question is asking. So we add the following facts:

8. Alive means not dead.

   $\forall x : \forall t : [alive(x, t) \rightarrow \neg dead(x,t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$

   This is not strictly correct, since $\neg dead$ implies alive only for animate objects. (Chairs can be neither dead nor alive.) Again, we will ignore, this for now. This is an example of the fact that rarely do two expressions have truly identical meanings in all circumstances.

9. If someone dies, then he is dead at all later times.

   $\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$

   This representation says that one is dead in all years after the one in which one died. It ignores the question of whether one is dead in the year in which one died.

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\forall x : man(x) \rightarrow mortal(x)$
5. $\forall$: $Pompeian(x) \rightarrow died(x, 79)$
6. $erupted(yolcano, 79)$
7. $\forall_x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$
8. $now = 1991$
9. $\forall x : \forall t$: $[\text{alive}(x, t) \rightarrow \neg dead(x, t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$
10. $\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$

Fig. 5.4 *A Set of Facts about Marcus*

To answer that requires breaking time up into smaller units than years. If we do that, we can then add rules that say such things as "One is dead at *time* (*year 1, month 1*) if one died during (*year 1, month 1*) and *month 2* precedes *month 1*]." We can extend this to days, hours, etc., as necessary. But we do not want to reduce all time statements to that level of detail, which is unnecessary and often not available.

A summary of all the facts we have now represented is given in Fig. 5.4. (The numbering is changed slightly because sentence 5 has been split into two parts.) Now let's attempt to answer the question "Is Marcus alive?" by proving:

$\neg alive(Marcus, now)$

Two such proofs are shown in Fig. 5.5 and 5.6. The term *nil* at the end of each proof indicates that the list of conditions remaining to be proved is empty and so the proof has succeeded. Notice in those proofs that whenever a statement of the form:

$a \wedge b \rightarrow c$

was used, *a* and *b* were set up as independent subgoals. In one sense they are, but in another sense they are not if they share the same bound variables, since, in that case, consistent substitutions must be made in each of them. For example, in Fig. 5.6 look at the step justified by statement 3. We can satisfy the goal

$born(Marcus, t_1)$

using statement 3 by binding A to 40, but then we must also bind A to 40 in

$gt(now - t_1, 150)$

since the two t_1's were the same variable in statement 4, from which the two goals came. A good computational proof procedure has to include both a way of determining that a match exists and a way of guaranteeing uniform substitutions throughout a proof. Mechanisms for doing both those things are discussed below.

$$
\begin{array}{cl}
\neg alive(Marcus, now) & \\
\uparrow & (9, \text{substitution}) \\
dead(Marcus, now) & \\
\uparrow & (10, \text{substitution}) \\
died(Marcus, t_1) \wedge gt(now, t_1) & \\
\uparrow & (5, \text{substitution}) \\
Pompeian(Marcus) \wedge gt(now, 79) & \\
\uparrow & (2) \\
gt(now, 79) & \\
\uparrow & (8, \text{substitute equals}) \\
gt(1991, 79) & \\
\uparrow & (\text{compute gt}) \\
nil &
\end{array}
$$

Fig. 5.5 *One Way of Proving That Marcus Is Dead*

From looking at the proofs we have just shown, two things should be clear:

- Even very simple conclusions can require many steps to prove.
- A variety of processes, such as matching, substitution, and application of *modus ponens* are involved in the production of a proof. This is true even for the simple statements we are using. It would be worse if we had implications with more than a single term on the right or with complicated expressions involving *amis* and *ors* on the left.

The first of these observations suggests that if we want to be able to do nontnvial reasoning, we are going to need some statements that allow us to take bigger steps along the way. These should represent the facts that people gradually acquire as they become experts. How to get computers to acquire them is a hard problem for which no very good answer is known.

The second observation suggests that actually building a program to do what people do in producing proofs such as these may not be easy. In the next section, we introduce a proof procedure called *resolution* that reduces some of the complexity because it operates on statements that have first been converted to a single canonical form.

5.4 RESOLUTION

As we suggest above, it would be useful from a computational point of view ff we had a proof procedure that carried out in a single operation the variety of processes involved in reasoning with statements in predicate logic. Resolution is such a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form, which is described below.

$$
\begin{array}{ll}
\neg alive(Marcus, now) & \\
\quad\uparrow & (9, \text{substitution}) \\
dead(Marcus, now) & \\
\quad\uparrow & (7, \text{substitution}) \\
mortal(Marcus) \wedge & \\
born(Marcus, t_1) \wedge & \\
gt(now - t_1, 150) & \\
\quad\uparrow & (4, \text{substitution}) \\
man(Marcus) \wedge & \\
born(Marcus, t_1) \wedge & \\
gt(now - t_1, 150) & \\
\quad\uparrow & (1) \\
born(Marcus, t_1) \wedge & \\
gt(now - t_1, 150) & \\
\quad\uparrow & (3) \\
gt(now - 40, 150) & \\
\quad\uparrow & (8) \\
gt(1991 - 40, 150) & \\
\quad\uparrow & (\text{compute minus}) \\
gt(1951, 150) & \\
\quad\uparrow & (\text{compute gt}) \\
nil &
\end{array}
$$

Fig. 5.6 *Another Way of Proving That Marcus is Dead*

Resolution produces proofs by *refutation*. In other words, to prove a statement (i.e., show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e., that it is unsatisfiable). This approach contrasts with the technique that we have been using to generate proofs by chaining backward from the theorem to be proved to the axioms. Further discussion of how resolution operates will be much more straightforward after we have discussed the standard form in which statements will be represented, so we defer it until then.

5.4.1 Conversion to Clause Form

Suppose we know that all Romans who know Marcus either hate Caesar or think that anyone who hates anyone is crazy. We could represent that in the following wff:

$$
\begin{aligned}
&\forall x : [Roman(x) \wedge know(x, Marcus)] \rightarrow \\
&\quad [hate(x, Caesar) \vee (\forall y : \exists z : hate(y, z) \rightarrow thinkcrazy(x, y))]
\end{aligned}
$$

To use this formula in a proof requires a complex matching process. Then, having matched one piece of it, such as *thinkcrazy*(x,y), it is necessary to do the right thing with the rest of the formula including the pieces in which the matched part is embedded and those in which it is not. If the formula were in a simpler form, this proc;ess would be much easier. The formula would be easier to work with if

- It were flatter, i.e., there was less embedding of components.
- The quantifiers were separated from the rest of the formula so that they did not need to be considered.

Conjunctive normal form [Davis and Putnam, 1960] has both of these properties. For example, the formula given above for the feelings of Romans who know Marcus would be represented in conjunctive normal form as

$$
\begin{aligned}
&\neg Roman(x) \wedge \neg know(x, Marcus) \vee \\
&\quad hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, z)
\end{aligned}
$$

Since there exists an algorithm for converting any wff into conjunctive normal form, we lose no generality if we employ a proof procedure (such as resolution) that operates only on wff's in this form. In fact, for resolution to work, we need to go one step further. We need to reduce a set of wff's to a set of *clauses*, where a clause is defined to be a wff in conjunctive normal form but with no instances of the connector A. We can do this by first converting each wff into conjunctive normal form and then breaking apart each such expression into clauses, one for each conjunct. All the conjuncts will be considered to be conjoined together as the proof procedure operates. To convert a wff into clause form, perform the following sequence of steps.

Algorithm: Convert to Clause Form

1. Eliminate $\rightarrow$, using the fact that $a \rightarrow b$ is equivalent to $\neg a \vee b$. Performing this transformation on the wff given above yields

   $$\forall x : \neg[Roman(x) \wedge know(x, Marcus)] \vee [hate(x, Caesar) \vee (\forall y : \neg(\exists z : hate(y, z)) \vee thinkcrazy(x,y))]$$

2. Reduce the scope of each $\neg$ to a single term, using the fact that $\neg(\neg p) = p$, deMorgan's laws [which say that $\neg(a \wedge b) = \neg a \vee \neg b$ and $\neg(a \vee b) = \neg a \wedge \neg b$], and the standard correspondences between quantifiers [$\neg\forall x : P(x) = \exists x : \neg P(x)$ and $\neg\exists x : P(x) = \forall x : \neg P(x)$]. Performing this transformation on the wff from step 1 yields

   $$\forall x : [\neg Roman(x) \vee \neg know(x, Marcus)] \vee [hate(x, Caesar) \vee (\forall y : \forall z : \neg hate(y, z) \vee thinkcrazy(x, y))]$$

3. Standardize variables so that each quantifier binds a unique variable. Since variables are just dummy names, this process cannot affect the truth value of the wff. For example, the formula

   $$\forall x : P(x) \vee \forall x : Q(x)$$

   would be converted to

   $$\forall x : P(x) \vee \forall y : Q(y)$$

   This step is in preparation for the next.

4. Move all quantifiers to the left of the formula without changing their relative order. This is possible since there is no conflict among variable names. Performing this operation on the formula of step 2, we get

   $$\forall x : \forall y : \forall z : [\neg Roman(x) \vee \neg know(x, Marcus)] \vee [hate(x, Caesar) \vee (\neg hate(y, z) \vee thinkcrazy(x,y))]$$

   At this point, the formula is in what is known as *prenex normal form*. It consists of a *prefix* of quantifiers followed by a *matrix*, which is quantifier-free.

5. Eliminate existential quantifiers. A formula that contains an existentially quantified variahle asserts that there is a value that can be substituted for the variable that makes the formula true. We can eliminate the quantifier by substituting for the variable a reference to a function that produces the desired value. Since we do not necessarily know how to produce the value, we must create a new function name for every such replacement. We make no assertions about these functions except that they must exist. So, for example, the formula

   $$\exists y : President(y)$$

   can be transformed into the formula

   $$President(S1)$$

where SI is a function with no arguments that somehow produces a value that satisfies President.
If existential quantifiers occur within the scope of universal quantifiers, then the value that satisfies the predicate may depend on the values of the universally quantified variables. For example, in the formula

$\forall x : \exists y : father\text{-}of(y,x)$

the value of y that satisfies *father-of* depends on the particular value of x. Thus we must generate functions with the same number of arguments as the number of universal quantifiers in whose scope the expression occurs. So this example would be transformed into

$\forall x : father\text{-}of(S2(x),x))$

These generated functions are called *Skolem functions.* Sometimes ones with no arguments are called *Skolem constants*.

6. Drop the prefix. At this point, all remaining variables are universally quantified, so the prefix can just be dropped and any proof procedure we use can simply assume that any variable it sees is universally quantified. Now the formula produced in step 4 appears as

$$[\neg Roman(x) \vee \neg know(x, Marcus)] \vee$$
$$[hate(x, Caesar) \vee (\neg hate(y, z) \vee thinkcrazy(x, y))]$$

7. Convert the matrix into a conjunction of disjuncts. In the case of our example, since there are no *and's,* it is only necessary to exploit the associative property of *or* [i.e., $(a \wedge b) \vee c = (a \vee c) \wedge (b \wedge c)$] and simply remove the parentheses, giving

$$\neg Roman(x) \vee \neg know(x, Marcus) \vee$$
$$hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, y)$$

However, it is also frequently necessary to exploit the distributive property [i.e., $(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$]. For example, the formula

$$(winter \wedge wearingboots) \vee (summer \wedge wearingsandals)$$

becomes, after one application of the rule

$$[winter \vee (summer \wedge wearingsandals)]$$
$$\wedge [wearingboots \vee (summer \wedge wearingsandals)]$$

and then, after a second application, required since there are still conjuncts joined byOR's,

$$(winter \vee summer) \wedge$$
$$(winter \vee wearingsandals) \wedge$$
$$(wearingboots \vee summer) \wedge$$
$$(wearingboots \vee wearingsandals)$$

8. Create a separate clause corresponding to each conjunct. In order for a wff to be true, all the clauses that are generated from it must be true. If we are going to be working with several wff's, all the clauses generated by each of them can now be combined to represent the same set of facts as were represented by the original wff's.
9. Standardize apart the variables in the set of clauses generated in step 8. By this we mean rename the variables so that no two clauses make reference to the same variable. In making this transformation, we rely on the fact that

$(\forall x : P(x) \wedge Q(x)) = \forall x : P(x) \wedge \forall x : Q(x)$

Thus since each clause is a separate conjunct and since all the variables are universally quantified, there need be no relationship between the variables of two clauses, even if they were generated from the same wff.

Performing this final step of standardization is important because during the resolution procedure it is sometimes necessary to instantiate a universally quantified variable (i.e., substitute for it a particular value). But, in general, we want to keep clauses in their most general form as long as possible. So when a variable is instantiated, we want to know the minimum number of substitutions that must be made to preserve the truth value of the system.

After applying this entire procedure to a set of wff's, we will have a set of clauses, each of which is a disjunction of *literals*. These clauses can now be exploited by the resolution procedure to generate proofs.

5.4.2 The Basis of Resolution

The resolution procedure is a simple iterative process: at each step, two clauses, called the *parent clauses*, are compared (*resolved*), yielding a new clause that has been inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose that there are two clauses in the system:

$winter \vee summer$
$\neg winter \vee cold$

Recall that this means that both clauses must be true (i.e., the clauses, although they look independent, are really conjoined).

Now we observe that precisely one of *winter* and $\neg$*winter* will be true at any point. If *winter* is true, then *cold* must be true to guarantee the truth of the second clause. If $\neg$*winter* is true, then *summer* must be true to guarantee the truth of the first clause. Thus we see that from these two clauses we can deduce

$summer \vee cold$

This is the deduction that the resolution procedure will make. Resolution operates by taking two clauses that each contain the same literal, in this example, *winter*. The literal must occur in positive form in one clause and in negative form in the other. The *resolvent* is obtained by combining all of the literals of the two parent clauses except the ones that cancel.

If the clause that is produced is the empty clause, then a contradiction has been found. For example, the two clauses

$winter$
$\neg winter$

will produce the empty clause. If a contradiction exists, then eventually it will be found. Of course, if no contradiction exists, it is possible that the procedure will never terminate, although as we will see, there are often ways of detecting that no contradiction exists.

So far, we have discussed only resolution in prepositional logic. In predicate logic, the situation is more complicated since we must consider all possible ways of substituting values for the variables. The theoretical basis of the resolution procedure in predicate logic is Herbrand's theorem [Chang and Lee, 1973], which tells us the following:

- To show that a set of clauses S is unsatisfiable, it is necessary to consider only interpretations over a particular set, called the *Herbrand universe* of S.
- A set of clauses S is unsatisfiable if and only if a finite subset of ground instances (in which all bound variables have had a value substituted for them) of S is unsatisfiable.

The second part of the theorem is important if there is to exist any computational procedure for proving unsatisfiability, since in a finite amount of time no procedure will be able to examine an infinite set. The first part suggests that one way to go about finding a contradiction is to try systematically the possible substitutions and see if each produces a contradiction. But that is highly inefficient. The resolution principle, first introduced by Robinson [1965], provides a way of finding contradictions by trying a minimum number of substitutions. The idea is-to keep clauses in their general form as long as possible and only introduce specific substitutions when they are required. For more details on different kinds of resolution, see Stickel [1988].

5.4.3 Resolution in Propositional Logic

In order to make it clear how resolution works, we first present the resolution procedure for propositional logic. We then expand it to include predicate logic.

In propositional logic, the procedure for producing a proof by resolution of proposition P with respect to a set of axioms F is the following.

Algorithm: Propositional Resolution

1. Convert all the propositions of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resulting clause, called the *resolvent*, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and $\neg L$ such that one of the parent clauses contains L and the other contains $\neg L$, then select one such pair and eliminate both L and $\neg L$ from the resolvent.
   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Let's look at a simple example. Suppose we are given the axioms shown in the first column of Fig. 5.7 and we want to prove R. First we convert the axioms to clause form, as shown in the second column of the figure.

Given Axioms	Converted to Clause Form	
P	P	(1)
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$	(2)
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$	(3)
	$\neg T \vee Q$	(4)
T	T	(5)

Fig. 5.7 *A Few Facts in Propositional Logic*

Then we negate R, producing $\neg R$, which is already in clause form. Then we begin selecting pairs of clauses to resolve together. Although any pair of clauses can be resolved, only those pairs that contain complementary literals will produce a resolvent that is likely to lead to the goal of producing the empty clause (shown as a box). We might, for example, generate the sequence of resolvents shown in Fig. 5.8. We begin by resolving with the clause $\neg R$ since that is one of the clauses that must be involved in the contradiction we are trying to find.

One way of viewing the resolution process is that it takes a set of clauses that are all assumed to be true and, based on information provided by the others, it generates new clauses that represent restrictions on the way each of those original clauses can be made true. A contradiction occurs when a clause becomes so restricted that there is no way it can be true. This is indicated by the generation of the empty clause. To see how this works, let's look again at the example. In order for proposition 2 to be true., one of three things must be true: $\neg P$, $\neg Q$, or R. But we are assuming that $\neg R$ is true. Given that, the only way for proposition 2 to be true is for one of two things to be true: $\neg P$ or $\neg Q$. That is what the first resolvent clause says. But proposition 1 says that P is true, which means that $\neg P$ cannot be true, which leaves only one way for proposition 2 to be true, namely for $\neg Q$ to be true (as shown in the second resolvent clause). Proposition 4 can be true if either $\neg T$ or Q is true. But since we now know that $\neg Q$ must be true, the only way for proposition 4 to be true is for $\neg T$ to be true (the third resolvent). But proposition 5 says that T is true. Thus there is no way for all of these clauses to be true in a single interpretation. This is indicated by the empty clause (the last resolvent).

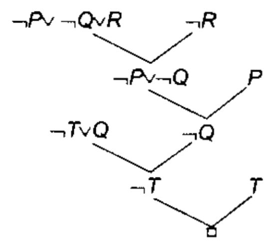


Fig. 5.8 *Resolution in Propositional Logic*

5.4.4 The Unification Algorithm

In propositional logic, it is easy to determine that two literals cannot both be true at the same time. Simply look for L and $\neg L$ In predicate logic, this matching process is more complicated since the arguments of the predicates must be considered. For example, *man*(*John*) and $\neg$*man*(*John*) is a contradiction, while *man*(*John*) artd $\neg$*man*(*Spot*) is not Thus, in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. There is a straightforward recursive procedure, called the *unification algorithm,* that does just this.

The basic idea of unification is very simple. To attempt to unify two literals, we first check if their initial predicate symbols are the same. If so, we can proceed. Otherwise, there is no way they can be unified, regardless of their arguments. For example, the two literals

tryassassinate(*Marcus*, *Caesar*)
hate(*Marcus*, *Caesar*)

cannot be unified. If the predicate symbols match, then we must check the arguments, one pair at a time. If the first matches, we can continue with the second, and so on. To test each argument pair, we can simply call the unification procedure recursively. The matching rules are simple. Different constants or predicates cannot match; identical ones can. A variable can match another variable, any constant, or a predicate expression, with the restriction that the predicate expression must not contain any instances of the variable being matched.

The only complication in this procedure is that we must find a single, consistent substitution for the entire literal, not separate ones for each piece of it. To do this, we must take each substitution that we find and apply it to the remainder of the literals before we continue trying to unify them. For example, suppose we want to unify the expressions

$P(x,x)$
$P(y,z)$

The two instances of P match fine. Next we compare x and y, and decide that if we substitute y for x, they could match. We will write that substitution as

y/x

(We could, of course, have decided instead to substitute x for y, since they are both just dummy variable names. The algorithm will simply pick one of these two substitutions.) But now, if we simply continue and match x and z. we produce tne substitution z/x. But we cannot substitute bothy and z for x, so we have not produced a consistent substitution.

What we need to do after finding the first substitutiony/x is to make that substitution throughout the literals, giving

$P(y, y)$
$P(y, z)$

Now we can attempt to unify arguments v and z, which succeeds with the substitution z/y. The entire unification process has now succeeded with a substitution that is the composition of the two substitutions we found. We write the composition as

$(z/y)(y/x)$

following standard notation for function composition. In general, the substitution $(a_1/a_2, a_3/a_4, \ldots)(b_1/b_2, b_3/b_4, \ldots)\ldots$ means to apply all the substitutions of the right- most list, then take the result and apply all the ones of the next list, and so forth, until all substitutions have been applied.

The object of the unification procedure is to discover at least one substitution that causes two literals to match. Usually, if there is one such substitution there are many. ' For example, the literals

hate(x, y)
hate$(Marcus, z)$

could be unified with any of the following substitutions:

$(Marcus/x, z/y)$
$(Marcus/x, y/z)$
$(Marcus/x, Caesar/y, Caesar/z)$
$(Marcus/x, Polonius/y, Polonius/z)$

The first two of these are equivalent except for lexical variation. But the second two, although they produce a match, also produce a substitution that is more restrictive than absolutely necessary for the match. Because the final substitution produced by the unification process will be used by the resolution procedure, it is useful to generate the most general unifier possible. The algorithm shown below will do that.

Having explained the operation of the unification algorithm, we can now state it concisely. We describe a procedure Unify(Ll, L2), which returns as its value a list representing the composition of the substitutions that were performed during the match. The empty list, NIL, indicates that a match was fpund without any substitutions. The list consisting of the single value FAIL indicates that the unification procedure failed.

Algorithm: Unify(Ll, L2)

1. If $L1$ or $L2$ are both variables or constants, then:
   (a) If $L1$ and $L2$ are identical, then return NIL.
   (b) Else if $L1$ is a variable, then if $L1$ occurs in $L2$ then return {FAIL}, else return $(L2/L1)$.
   (c) Else if $L2$ is a variable then if $L2$ occurs in $L1$ then return {FAIL}, else return $(L1/L2)$.
   (d) Else return {FAIL}.

2. If the initial predicate symbols in $L1$ and $L2$ are not identical, then return {FAIL).
3. If LI and $L2$ have a different number of arguments, then return {FAIL}.
4. Set *SUBST* to NIL. (At the end of this procedure, *SUBST* will contain all the substitutions used to unify $L1$ and $L2$.)
5. For $i \leftarrow 1$ to number of arguments in $L1$:
   (a) Call Unify with the /th argument of $L1$ and the ith argument of $L2$, putting result in S.
   (b) If S contains FAIL then return {FAIL}.
   (c) If S is not equal to NIL then:
      (i) Apply S to the remainder of both $L1$ and $L2$.
      (ii) *SUBST* : = APPEND(S, *SUBST*).
6. Return *SUBST*.

The only part of this algorithm that we have not yet discussed is the check in steps 1(b) and 1(c) to make sure that an expression involving a given variable is not unified with that variable. Suppose we were attempting to unify the expressions

$$f(x,x)$$
$$f(g(x),g(x))$$

If we accepted $g(x)$ as a substitution for x, then we would have to substitute it for x in the remainder of the expressions. But this leads to infinite recursion since it will never be possible to eliminate x.

Unification has deep mathematical roots and is a useful operation in many AI programs, for example, theorem provers and natural language parsers. As a result, efficient data structures and algorithms for unification have been developed. For an introduction to these techniques and applications, see Knight [1989].

5.4.5 Resolution in Predicate Logic

We now have an easy way of determining that two literals are contradictory—they are if one of them can be unified with the negation of the other. So, for example, $man(x)$ and $\neg man(Spot)$ are contradictory, since $man(x)$ and $man(Spot)$ can be unified. This corresponds to the intuition that says that $man(x)$ cannot be true for all x if there is known to be some x, say Spot, for which $man(x)$ is false. Thus in order to use resolution for expressions in the predicate logic, we use the unification algorithm to locate pairs of literals that cancel out.

We also need to use the unifier produced by the unification algorithm to generate the resolvent clause. For example, suppose we want to resolve two clauses:

1. $man(Marcus)$
2. $\neg man(x_1) \vee mortal(x_1)$

The literal *man{Marcus)* can be unified with the literal *man{x\)* with the substitution $Marcus/x_1$, telling us that for $x_1 = Marcus$, $\neg man(Marcus)$ is false. But we cannot simply cancel out the two *man* literals as we did in propositional logic and generate the resolvent $mortal(x_1)$. Clause 2 says that for a given x_1, either $\neg man(x_1)$ or $mortal(x_1)$. So for it to be true, we can now conclude only that $mortal(Marcus)$ must be true. It is not necessary that $mortal(x_1)$ be true for all x_1, since for some values of x_1, $\neg man(x_1)$ might be true, making $mortal(x_1)$ irrelevant to the truth of the complete clause. So the resolvent generated by clauses 1 and 2 must be $mortal(Marcus)$, which we get by applying the result of the unification process to.the resolvent. The resolution process can then proceed from there to discover whether $mortal(Marcus)$ leads to a contradiction with other available clauses.

This example illustrates the importance of standardizing variables apart during the process of converting expressions to clause form. Given that that standardization has bee'n done, it is easy to determine how the unifier must be used to perform substitutions to create the resolvent. If two instances of the same variable occur, then they must be given identical substitutions.

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements F and a statement to be proved P:

Algorithm: Resolution

1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obttfHied in 1.
3. Repeat until either a contradiction is found, no progress can be made, or a prede- termined amount of effort has been expended.
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals $T1$ and $\neg T2$ such that one of the parent clauses contains $T2$ and the other contains $T1$ and if $T1$ and $T2$ are unifiable, then neither $T1$ nor $T2$ should appear in the resolvent. We call $T1$ and $T2$ *Complementary literals*. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.
   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

If the choice of clauses to resolve together at each step is made in certain systematic ways, then the resolution procedure will find a contradiction if one exists. However, it may take a very long time. There exist strategies for making the choice that can speed up the process considerably:

- Only resolve pairs of clauses that contain complementary literals, since only such resolutions produce new clauses that are harder to satisfy than their parents. To facilitate this, index clauses by the predicates they contain, combined with an indication of whether the predicate is negated. Then, given a particular clause, possible resolvents that contain a complementary occurrence of one of its predi- cates can be located directly.
- Eliminate certain clauses as soon as they are generated so that they cannot partic- ipate in later resolutions. Two kinds of clauses should be eliminated: tautologies (which can never be unsatisfied) and clauses that are subsumed by other clauses (i.e., they are easier ;o satisfy. For example, $P \vee Q$ is subsumed by P.)
- Whenever possible, resolve either with one of the clauses that is part of the statement we are trying to refute or with a clause generated by a resolution with such a clause. This is called the *set-of-support strategy* and corresponds to the intuition that the contradiction we are looking for must involve the statement we are trying to prove. Any other contradiction would say that the previously believed statements were inconsistent.
- Whenever possible, resolve with clauses that have a single literal. Such resolutions generate new clauses with fewer literals than the larger of their parent clauses and thus are probably closer to the goal of a resolvent with zero terms. This method is called the *unit-preference strategy.*

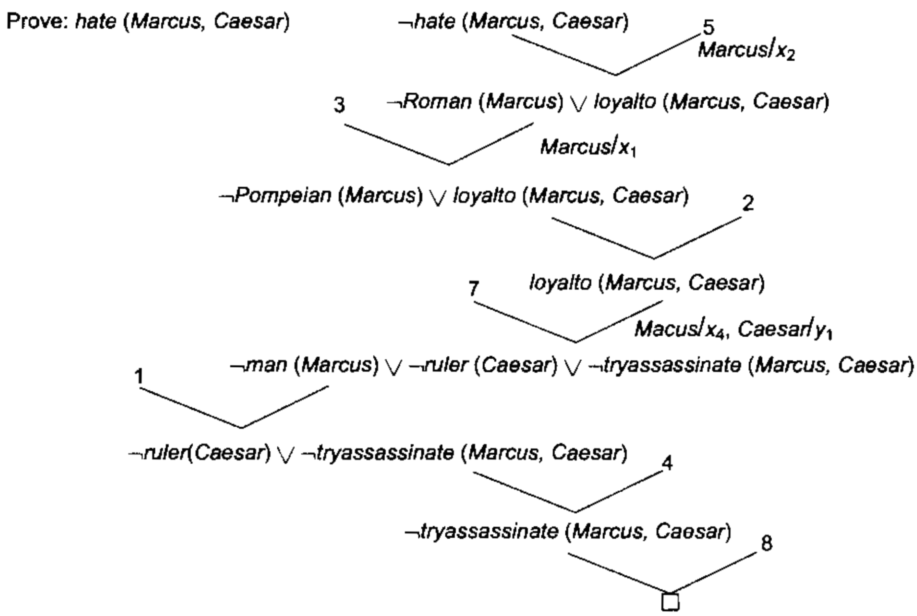
Let's now return to our discussion of Marcus and show how resolution can be used to prove new things about him. Let's first consider the set of statements introduced in Section 5.1. To use them in resolution proofs, we must convert them to clause form as described in Section 5.4.1. Figure 5.9(a) shows the results of that conversion. Figure 5.9(b) shows a resolution proof of the statement

$$hate(Marcus, Caesar)$$

Axioms in clause form:

1. *man(Marcus)*
2. *Pompeian(Marcus)*
3. $\neg Pompeian(x_1) \vee Roman(x_1)$
4. *ruler(Caesar)*
5. $\neg Roman(x_2) \vee loyalto(x_2, Caesar) \vee hate(x_2, Caesar)$
6. $loyalto(x_3, f1(x_3))$
7. $\neg man(x_4) \vee \neg ruler(y_1) \vee \neg tryassassinate(x_4, y_1) \vee loyalto(x_4, y_1)$
8. *tryassassinate(Marcus, Caesar)*

(a)



(b)

Fig. 5.9 *A Resolution Proof*

Of course, many more resolvents could have been generated than we have shown, but we used the heuristics described above to guide the search. Notice that what we have done here essentially is to reason backward from the statement we want to show is a contradiction through a set of intermediate conclusions to the final conclusion of inconsistency.

Suppose our actual goal in proving the assertion

hate(Marcus, Caesar)

was to answer the question "Did Marcus hate Caesar?" In that case, we might just as easily have attempted to prove the statement

$\neg hate(Marcus, Caesar)$

To do so, we would have added

hate(Marcus, Caesar)

to the set of available clauses and begun the resolution process. But immediately we notice that there are no clauses that contain a literal involving $\neg$*hate*. Since the resolution process can only generate new clauses that are composed of combinations of literals from already existing clauses, we know that no such clause can be generated and thus we conclude that *hate(Marcus,Caesar)* will not produce a contradiction with the known statements. This is an example of the kind of situation in which the resolution procedure can detect that no contradiction exists. Sometimes this situation is detected not at the beginning of a proof, but part way through, as shown in the example in Figure 5.10(*a*), based on the axioms given in Fig. 5.9.

But suppose our knowledge base contained the two additional statements

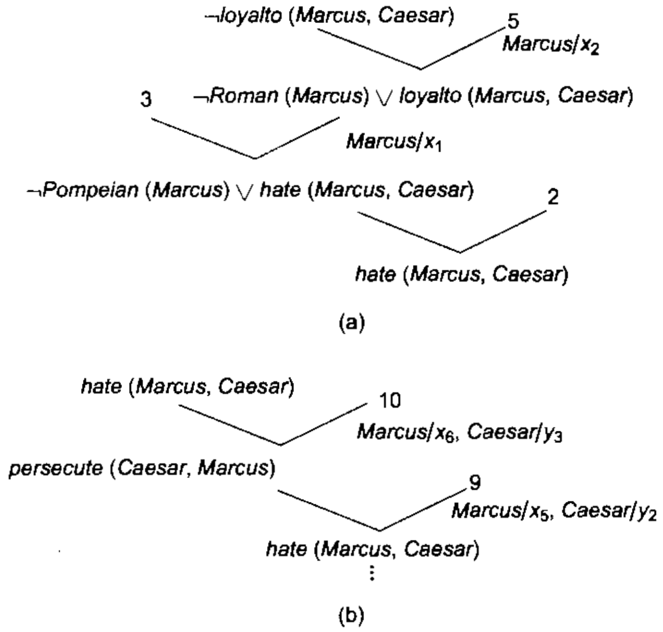


Fig. 5.10 *An Unsuccessful Attempt at Resolution*

9. *persecute*$(x, y) \rightarrow$ *hate*(y, x)
10. *hate*$(x, y) \rightarrow$ *persecute*(y, x)

Converting to clause form, we get

9. $\neg$*persecute*$(x_5, y_2) \vee$ *hate*(y_2, x_5)
10. $\neg$*hate*$(x_6, y_3) \vee$ *persecute*(y_3, x_6)

These statements enable the proof of Fig. 5.10(*a*) to continue as shown in Fig. 5.10(*b*). Now to detect that there is no contradiction we must discover that the only resolvents that can be generated have been generated before. In other words, although we can generate resolvents, we can generate no new ones.

Given:

1. $\neg father(x, y) \vee \neg woman(x)$
   (i.e., $father(x, y) \rightarrow \neg woman(x)$)
2. $\neg mother(x, y) \vee woman(x)$
   (i.e., $mother(x, y) \rightarrow woman(x)$)
3. $mother(Chris, Mary)$
4. $father(Chris, Bill)$

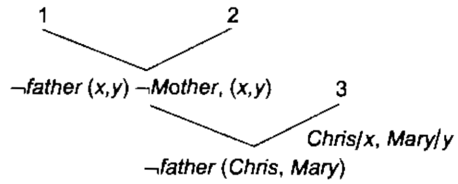


Fig. 5.11 *The Need to Standardize Variables*

Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $horn(Marcus, 40)$
4. $\neg man(x_1) \vee mortal(x_1)$
5. $\neg Pompeian(x_2) \vee died(x_2, 79)$
6. $erupted(volcano, 79)$
7. $\neg morta(x_3) \vee \neg born(x_3, t_1) \vee \neg gt(t_2 - t_1, 150) \vee dead(x_3, t_2)$
8. $now = 2008$

9a. $\neg alive(x_4, t_3) \vee \neg dead(x_4, t_3)$

9b. $dead(x_5, t_4) \vee alive(x_5, t_4)$

10. $\neg died(x_6\ t_5) \vee \neg gt(t_6, t_5) \vee dead(x_6, t_6)$

Prove: $\neg alive(Marcus, now)$

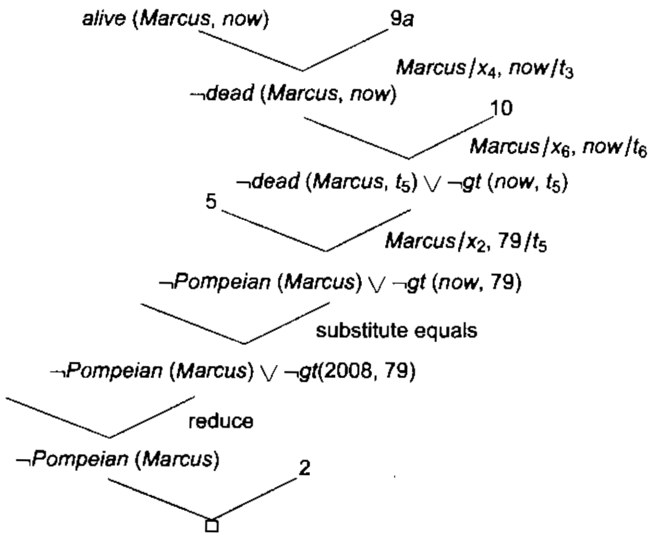


Fig. 5.12 *Using Resolution with Equality and Reduce*

Recall that the final step of the process of converting a set of formulas to clause form was to standardize apart the variables that appear in the final clauses. Now that we have discussed the resolution procedure, we can see clearly why this step is so important. Figure 5.11 shows an example of the difficulty that may arise if standardization is not done. Because the variable y occurs in both clause 1 and clause 2, the substitution at the second resolution step produces a clause that is too restricted and so does not lead to the contradiction that is present in the database. If, instead, the clause

$\neg father(Chris, y)$

had been produced, the contradiction with clause 4 would have emerged. This would have happened if clause 2 had been rewritten as

$\neg mother(a, b) \vee woman(a)$

In its pure form, resolution requires all the knowledge it uses to be represented in the form of clauses. But as we pointed out in Section 5.3, it is often more efficient to represent certain kinds of information in the form of computable functions, computable predicates, and equality relationships. It is not hard to augment resolution to handle this sort of knowledge. Figure 5.12 shows, a resolution proof of the statement

$\neg alive(Marcus, now)$

based on the statements given in Section 5.3. We have added two ways of generating new clauses, in addition to the resolution rule:

- Substitution of one value for another to which it is equal.
- Reduction of computable predicates. If the predicate evaluates to FALSE, it can simply be dropped, since adding V FALSE to a disjunction cannot change its truth value. If the predicate evaluates to TRUE, then the generated clause is a tautology and cannot lead to a contradiction.

5.4.6 The Need to Try Several Substitutions

Resolution provides a very good way of finding a refutation proof without actually trying all the substitutions that Herbrand's theorem suggests might be necessary. But it does not always eliminate the necessity of trying more than one substitution. For example, suppose we know, in addition to the statements in Section 5.1, that

$hate(Marcus, Paulus)$
$hate(Marcus, Julian)$

Now if we want to prove that Marcus hates some ruler, we would be likely to try each substitution shown in Figure 5.13(a) and (b) before finding the contradiction shown in (c). Sometimes there is no way short of very good luck to avoid trying several substitutions.

5.4.7 Question Answering

Very early in the history of AI it was realized that theorem-proving techniques could be applied to the problem of answering questions. As we have already suggested, this seems natural since both deriving theorems from axioms and deriving new facts (answers) from old facts employ the process of deduction. We have already shown how resolution can be used to answer yes-no questions, such as "Is Marcus alive?" In this section, we show how resolution can be used to answer fill-in-the-blank questions, such as "When did Marcus die?" or

"Who tried to assassinate a ruler?" Answering these questions involves finding a known statement that matches the terms given in the question and then responding with another piece of that same statement that fills the slot demanded by the question. For example, to answer the question "When did Marcus die?" we need a statement of the form

died(Marcus, ??)

with ?? actually filled in by some particular year. So, since we can prove the statement

died(Marcus, 79)

we can respond with the answer 79.

It turns out that the resolution procedure provides an easy way of locating just the statement we need and finding a proof for it. Let's continue with the example question

Prove:	$\exists x : hate(Marcus,x) \wedge ruler(x)$
(negate):	$\neg\exists x : hate(Marcus,x) \wedge ruler(x)$
(clausify):	$\neg hate(Marcus,x) \vee \neg ruler(x)$

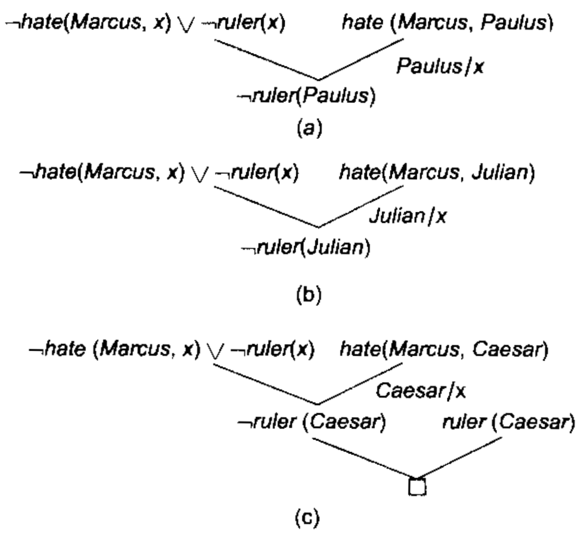


Fig. 5.13 *Trying Several Substitutions*

'When did Marcus die?" In order to be able to answer this question, it must first be true that Marcus died. Thus it must be the case that

$\exists t : died(Marcus,t)$

A reasonable first step then might be to try to prove this. To do so using resolution, we attempt to show that

$\neg\exists t: died(Marcus, t)$

produces a contradiction. What does it mean for that statement to produce a contradiction? Either it conflicts with a statement of the form

$\forall t: died(Marcus, t)$

where t is a variable, in which case we can either answer the question by reporting that there are many times at which Marcus died, or we can simply pick one such time and respond with it. The other possibility is that we produce a contradiction with one or more specific statements of the form

died(*Marcus, date*)

for some specific value of *date*. Whatever value of date we use in producing that contradiction is the answer we want. The value that proves that there is a value (and thus the inconsistency of the statement that there is no such value) is exactly the value we want.

Figure 5.14(a) shows how the resolution process finds the statement for which we are looking. The answer to the question can then be derived from the chain of unifications that lead back to the starting clause. We can eliminate the necessity for this final step by adding an additional expression to the one we are going to use to try to find a contradiction. This new expression will simply be the one we are trying to prove true (i.e., it will be the negation of the expression that is actually used in the resolution). We can tag it with a special marker so that it will not interfere with the resolution process. (In the figure, it is underlined.) It will just get carried along, but each time unification is done, the variables in this dummy expression will be bound just as are the ones in the clauses that are actively being used. Instead of terminating on reaching the nil clause, the resolution procedure will terminate when all that is left is the dummy expression. The bindings of its variables at that point provide the answer to the question. Figure 5.14(fr) shows how this process produces an answer to our question.

Unfortunately, given a particular representation of the facts in a system, there will usually be some questions that cannot be answered using this mechanism. For example, suppose that we want to answer the question "What happened in 79 A.D.?" \ising the statements in Section 5.3. In order to answer the question, we need to prove that something happened in 79. We need to prove

$\exists x : event(x, 79)$

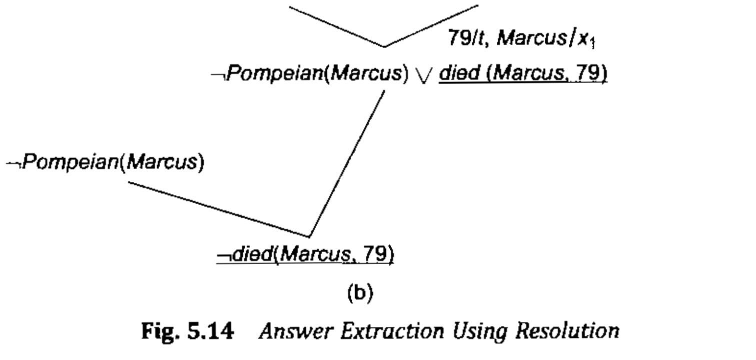


Fig. 5.14 *Answer Extraction Using Resolution*

and to discover a value for x. But we do not have any statements of the form $event(x, y)$.

We can, however, answer the question if we change our representation. Instead of saying

$$erupted(volcano, 79)$$

we can say

$$event(erupted(volcano), 79)$$

Then the simple proof shown in Fig. 5.15 enables us to answer the question.

This new representation has the drawback that it is more complex than the old one. And it still does not make it possible to answer all conceivable questions. In general, it is necessary to decide on the kinds of questions that will be asked and to design a representation appropriate for those questions.

$\neg event(x,79) \lor \underline{event(x, 79)}$ $\quad$ $event\ (erupted\ (volcano),79)$

$erupted\ (volcano)/x$

$\underline{event\ (erupted\ (volcano),79)}$

Fig. 5.15 *Using the New Representation*

Of course, yes-no and fill-in-the-blank questions are not the only kinds one could ask. For example, we might ask how to do something. So we have not yet completely solved the problem of question answering. In later chapters, we discuss some other methods for answering a variety of questions. Some of them exploit resolution; others do not.

5.5 NATURAL DEDUCTION

In the last section, we introduced resolution as an easily implementable proof procedure that relies for its simplicity on a uniform representation of the statements it uses. Unfortunately, uniformity has its price—everything looks the same. Since everything looks the same, there is no easy way to select those statements that are the most likely to be useful in solving a particular problem. In converting everything to clause form, we often lose valuable heuristic information that is contained in the original representation of the facts. For example, suppose we believe that all judges who are not crooked are well-educated, which can be represented as

$$\forall x : judge(x) \land \neg crooked(x) \rightarrow educated(x)$$

In this form, the statement suggests a way of deducing that someone is educated. But when the same statement is converted to clause form,

$$\neg judge(x) \lor crooked(x) \lor educated(x)$$

it appears also to be a way of deducing that someone is not a judge by showing that he is not crooked and not educated. Of course, in a logical sense, it is. But it is almost certainly not the best way, or even a very good way, to go about showing that someone is not a judge. The heuristic information contained in the original statement has been lost in the transformation.

Another problem with the use of resolution as the basis of a theorem-proving system is that people do not think in resolution. Thus it is very difficult for a person to interact with a resolution theorem prover, either to give it advice or to be given advice by it. Since proving very hard things is something that computers still do poorly, it is important from a practical standpoint that such interaction be possible. To facilitate it, we are foiced to look for a way of doing machine theorem proving that corresponds more closely to the processes used in human theorem proving. We are thus led to what we call, mostly by definition, *natural deduction.*

Natural deduction is not a precise term. Rather it describes a melange of techniques, used in combination to solve problems that are not tractable by any one method alone. One common technique is to arrange knowledge, not by predicates, as we have been doing, but rather by the objects involved in the predicates. Some techniques for doing this are described in Chapter 9. Another technique is to use a set of rewrite rules that not only describe logical implications but also suggest the way that those implications can be exploited in proofs.

For a good survey of the variety of techniques that can be exploited in a natural deduction system, see Bledsoe [1977]. Although the emphasis in that paper is on proving mathematical theorems, many of the ideas in it can be applied to a variety of domains in which it is necessary to deduce new statements from known ones. For another discussion of theorem proving using natural mechanisms, see Boyer and Moore [1988], which describes a system for reasoning about programs. It places particular emphasis on the use of mathematical induction as a proof technique.

SUMMARY

In this chapter we showed how predicate logic can be used as the basis of a technique for knowledge representation. We also discussed a problem-solving technique, resolution, that can be applied when knowledge is represented in this way. The resolution procedure is not guaranteed to halt if given a nontheorem to prove. But is it guaranteed io halt and find a contradiction if one exists? This is called the *completeness* question. In the form in which we have presented the algorithm, the answer to this question is no. Some small changes, usually not implemented in theorem-proving systems, must be made to guarantee completeness. But, from a computational point of view, completeness is not the only important question. Instead, we must ask whether a proof can be found in the limited amount of time that is available. There are two ways to approach achieving this computational goal. The first is to search for good heuristics that can inform a theorem-proving program. Current theorem-proving research attempts to do this. The other approach is to change not the program but the data given to the program. In this approach, we recognize that a knowledge base that is just a list of logical assertions possesses no structure. Suppose an information-bearing structure could be imposed on such a knowledge base. Then that additional information could be used to guide the program that uses the knowledge. Such a program majuiot look a lot like a theorem prover, although it will still be a knowledge-based problem solver. We discuss this idea further in Chapter 9.

A second difficulty with the use of theorem proving in AI systems is that there are some kinds of information that are not easily represented in predicate logic. Consider the following examples:

- "It is very hot today." How can relative degrees of heat be represented?
- "Blond-haired people often have blue eyes." How can the amount of certainty be represented?
- "If there is no evidence to the contrary, assume that any adult you meet knows how to read." How can we represent that one fact should be inferred from the absence of another?
- "It's better to have more pieces on the board than the opponent has." How can we represent this kind of heuristic information?
- "I know Bill thinks the Giants will win, but I think they are going to lose." How can several different