Chapter 3
Automated Reasoning in Full

This chapter provides you with a spirited introduction to the elements of automated reasoning, beginning with a brisk review of logic (which you may choose to bypass, depending on the type of traveler you are). The elements include a language for presenting the question or problem of interest to your automated reasoning assistant, various ways for drawing conclusions (inference rules), diverse means for controlling the reasoning (strategies), a procedure for simplifying and canonicalizing (demodulation), a procedure for coping with redundancy (subsumption), and a test for assignment completion (proof by contradiction). As for the applications of automated reasoning, Chapters 4 through 7 will serve you well.

In addition to focusing on the highlights and key features that can be found in an automated reasoning program, this chapter describes specific reasoning programs used at Argonne National Laboratory in the preceding two decades. For historical purposes, I include a brief discussion of the automated reasoning program AURA that was first used to obtain some of the results presented in this book. I also briefly describe the program ITP. However, since this book offers a copy of the program OTTER, the main focus is on that program.

In particular, I shall concentrate on some of the special features found in OTTER, features that are not always present in other reasoning programs. Finally, in Section 3.4.5, I include a detailed discussion of an inference rule that “builds in” equality. The rule, paramodulation, is heavily used, for example, for various applications covered in Chapter 6. This building in of equality illustrates yet another dimension of automated reasoning programs.

In addition to providing a review of the preceding material, I have a second motive in including this chapter. As remarked in Chapter 1, some who read this book are already familiar with the basic elements of logic and, therefore, Chapter 2 in part presents information that is already known. Other readers of the book may know a little about automated reasoning and, therefore, find that other parts of Chapter 2 cover familiar ground. Because of these considerations, this chapter proceeds at a rather brisk pace. If you find that you are one of those who know logic and who perhaps also know something about automated reasoning, or if you simply
prefer a faster pace, you can begin with this chapter rather than with Chapter 2. Since I shall be somewhat formal in this chapter, and since I include less explanatory material, you may wish to consult either of the two earlier chapters when and if some point is unclear. Chapter 2 especially contains more explanatory material and more detailed examples.

3.1 Logic

Automated reasoning is concerned with programs that aid in solving problems and in answering questions where reasoning is required. The reasoning referred to here is logical reasoning, not common-sense or probabilistic reasoning. The conclusions must follow inevitably from the facts or assumptions from which they are drawn. To see how an automated reasoning program works, you must first be familiar with certain elements of logic.

The logical operators are and, or, not, if-then (implies), and is-equivalent-to. These operators are used to produce "new" statements from given statements. By definition, not is applied to a single statement, and the remaining operators are applied to pairs of statements. (A single statement or an element of a pair of statements may itself be a compound statement.) The truth or falsity of the new statement depends, of course, on the truth or falsity of the statement or statements to which the operator is applied.

3.1.1 and

The and of two statements is true if and only if both statements are true. For example, the statement that "Arkansas begins with a vowel and ends with a consonant" is true because the two statements to which and is applied are each true. On the other hand, if Arkansas is replaced by Ohio, the statement is false because one of the two component statements is false—"Ohio ends with a consonant" is false.

3.1.2 or

The or of two statements is true if and only if at least one of the statements is true; hence it is true if both statements are true. For example, the statement that "(Oklahoma begins with a vowel and ends with a vowel) or (begins with a consonant and ends with a consonant)" is true. Although the second half of the statement is false, the statement is true because the first half is true. For another example, the statement that mothers are older than their children or fathers are older than their children is true. In this example, both substatements are true; thus their or is true. In everyday language, as opposed to logic, (usually) the or of two statements is true when exactly one of them is true.
3.1.3 not

The not of a statement is true if and only if the statement is false, and therefore false if and only if the statement is true. For example, the statement that “Oklahoma does not end with a consonant” is true. Its truth follows from the fact that the statement “Oklahoma ends with a consonant” is false. Similarly, the statement “mothers are not older than their children” is false, because they are older.

To determine the truth of compound statements involving not and some other operator, you simply apply the rules for each operator. For example, the statement “Oklahoma begins with a vowel and does not end with a consonant” is true. Can you now see what the rule is for the not of the not of a statement? The rule simply is that the nots cancel. A statement is true if and only if the not of the not of it is true, and a similar comment can be made for false.

3.1.4 if-then and implies

An if-then (implies) statement is false if and only if both the if component is true and the then component is false. In all other cases, if-then statements are true. Notice that this usage—the logical usage—is different from everyday language. In everyday language, if-then statements are often assumed true. In fact, typically the if is assumed true with the further assumption that this forces the then to be true. From a logical viewpoint, what is the truth of the following statement: “If mothers are younger than their children, then all children are 20 years old”? This statement is true from a logical viewpoint since the if component is false. A related example is the statement that “if a state name (in the United States) begins with a B, then it ends with a vowel”. To prove this statement false, you would have to give a state whose name begins with a B and does not end with a vowel. Since no state exists with a name beginning with a B, the if-then statement is true. Such statements are called vacuously true, and do occasionally come into play.

3.1.5 is-equivalent-to

For statements P and Q, P is-equivalent-to Q is false if and only if one of P and Q is true while the other is false. The statement “P is-equivalent-to Q” can be rewritten as “(if P then Q) and (if Q then P)”. The operator is-equivalent-to comes into play in statements of the form “P if and only if Q”. Although is-equivalent-to is barely discussed in Chapter 2, it can be understood in terms of the discussion found there for if-then and and.

3.1.6 Relationships and Laws in Logic

The operators of if-then and is-equivalent-to can be replaced by using or, not, and and. The most common replacement or translation that you make when using
an automated reasoning program is that of translating if-then to or and not. The rule is, “(if P then Q)” translates to “((not P) or Q)”. Thus, the statement “Kim is male or female” is a correct translation of the statement “if Kim is not male then Kim is female”; you can see this by using the rule that nots cancel. But it is also a correct translation of the statement “if Kim is not female then Kim is male”.

Next are the distributive laws, stated in the manner in which they are applied for producing information in the form required by the language used with OTTER. First, “P and Q) or (P and R)” translates to “P and (Q or R)”. Second, “P or (Q and R)” translates to “(P or Q) and (P or R)”. Finally, there are DeMorgan’s laws. First, “not (P and Q)” translates to “((not P) or (not Q))”. Second, “not (P or Q)” translates to “(not P) and (not Q)”. The distributive laws and DeMorgan’s laws are used for translating complex statements into a form in which every pair of statements is joined with an and, while the only operators that occur within a statement are not and or. For example, “if (P and Q) then R” translates to “(not P) or (not Q) or R”.

Exercises

1. Which of the following statements is true?
   a. (2 + 4 > 6) and (0 + 1 < 2)
   b. (2 + 4 < 9) or (3 − 1 > 0)
   c. not ((1 − 1 = 0) and (3 − 2 = 2))
   d. if (1 − 1 = 0) then (2 − 2 = 0)
   e. if (1 − 2 = 0) then (2 − 2 = 0)
   f. if (1 − 2 = 0) then (2 − 1 = 0)
   g. (1 − 1 = 0) is-equivalent-to (2 − 2 = 0)
   h. (1 − 2 = 0) is-equivalent-to (2 − 2 = 0)
   i. (1 − 2 = 0) is-equivalent-to (2 − 1 = 0)

2. Which of the following translations can be made?
   a. not (not (1 + 2 = 3)) to (1 + 2 = 3)
   b. not (not (1 + 2 = 3)) to (2 + 1 = 3)
   c. not (not (not (1 + 2 = 3))) to not (1 + 2 = 3)
   d. not (not (not (1 + 2 = 3))) to (1 + 2 = 3)
   e. if (8 < 9) then (8 < 9 + 1) to not (8 < 9) or (8 < 9 + 1)
   f. if (8 < 9) then (8 < 9 + 1) to (8 < 9 + 1) or (not (8 < 9))
   g. not (if (2 + 1 = 3) then (1 + 2 = 3)) to (2 + 1 = 3) and (not (1 + 2 = 3))
   h. not (if (2 + 1 = 3) then (1 + 2 = 3)) to (1 + 2 = 3) and (not (2 + 1 = 3))
   i. not (if (2 + 1 = 3) then (4 / 2 = 2)) to (2 + 1 = 3) and (not (2 / 4 = 2))
3.2 A Language Understood by an Automated Reasoning Program

Throughout this book, the clause language is used to represent a problem to an automated reasoning program. A clause is a statement that is the or of its literals. A clause may be empty, the or of no literals. A literal is a well-formed formula that consists of an n-ary predicate symbol with n greater than or equal to 0, possibly preceded by the not symbol “~”, followed by its n arguments. The language does not allow not of not, and simply cancels the two nots. A predicate symbol is a symbol that denotes a relation. The arguments of a predicate can be variables, constants, or functions with their arguments. The arguments of a function can be variables, constants, or functions with their arguments. If a function has no arguments, it is a constant. All variables that occur in a clause are implicitly treated as universally quantified, and therefore to be read as “for all” or as “for each”. An implicit and occurs between clauses. For most practical problems, this language of clauses is sufficiently powerful for attempting to solve them. (Actually, if you rely on set theory, the clause language is sufficient to address virtually any question or problem; see the CD-ROM for appropriate clauses for set theory.)

A literal prefixed with a not symbol is called a negative literal. A literal that is not so prefixed is called a positive literal. All literals are classed as negative or positive literals. A clause containing no negative literals is called a positive clause. A clause containing no positive literals is called a negative clause. The empty clause, containing no literals, is both positive and negative. A clause containing at least one positive and at least one negative literal is called a mixed clause. A clause containing exactly one literal is called a unit clause. If the literal in a unit clause is positive, then the clause is called a positive unit clause. A negative unit clause contains one negative literal. A clause containing more than one literal is called a nonunit clause.

For examples of the terminology, consider the following problem. What is a correct translation of “The job of nurse is held by a male”? More precisely, let the statement to be translated read, “For all x, if x holds the job of nurse, then x is male”.

A correct translation is

\(-\text{HASAJOB}(x,\text{nurse}) \mid \text{MALE}(x)\).

where “ | ” is the symbol for or. This statement is a nonunit clause: the clause contains two literals, one negative and one positive. The predicates that occur in this clause are HASAJOB and MALE. Variables are universally quantified; in particular, FEMALE(x) is to be read as “for all x, FEMALE(x)”. The predicate HASAJOB has two arguments, while MALE has but one.

The clause

\text{EQUALP}(\text{Pete},\text{husband(Thelma)}).
is a positive unit clause, one of whose arguments is the function husband that
in turn has the argument Thelma, and the other argument is the 0-ary function
(constant) Pete. This clause exhibits one of the few “built-in” concepts available
to an automated reasoning program, namely, equality. While almost all concepts
must be carefully specified for such a program, since reasoning programs must be
given all of the properties that sufficiently pin down a concept, for equality such is
not always necessary. Some reasoning programs—OTTER being a prime example—
can be instructed to interpret a predicate beginning with the letters EQUAL (or
EQ) to be treated as you would treat equality.

To state that Thelma is female, and Steve is male, and everyone is male or
female, you write the following clauses.

\[
\begin{align*}
\text{FEMALE(Thelma).} \\
\text{MALE(Steve).} \\
\text{MALE(x) \mid FEMALE(x).}
\end{align*}
\]

The and is implicit, while the or is represented with “ \mid ”. If you were being very
careful, you would replace the last clause with

\[
\text{-PERSON(x) \mid MALE(x) \mid FEMALE(x).}
\]
to state that the variable \( x \) in this clause ranges over people.

By convention, duplicate literals in a clause are always removed automatically,
for \( P \text{ or } P \) is logically equivalent to \( P \). In addition, the order of the literals within
a clause as well as the order of the clauses themselves has no effect on the meaning
or information conveyed to a reasoning program. The order of either can, however,
affect the efficiency with which a reasoning program attacks a problem.

### 3.2.1 Variables

Variables in the language used throughout the book are universally quantified and
are therefore to be interpreted as “for all”. A name is a sequence of characters all
of which are letters or numerals. For OTTER, a variable is a name beginning with
one of the letters between lower-case \( u \) and lower-case \( z \), inclusive. Conversely, (for
OTTER) a name beginning with a letter between lower-case \( u \) and lower-case \( z \),
inclusive, is a variable.

Rather than employing existentially quantified variables—for example, there exists an \( x \)—the clause language requires their replacement by functions. The statement
that each married woman has a husband can be rigorously phrased as “for each married woman, there exists a husband of that woman who is married to her”.

In clause form, this statement becomes

\[
\text{-MARRIEDWOMAN(x) \mid MARRIEDTO(x,husband(x)).}
\]

with the function “husband” replacing the existentially quantified variable that
could have been employed.
The rule is that existentially quantified variables are replaced by functions, where the function has as its arguments all universally quantified variables that appear before the existentially quantified variable. In the preceding example—for all $x$ there exists a $y$ such that if $x$ is a married woman, then $x$ is married to $y$—the function husband must, therefore, have as its argument the variable $x$. Existentially quantified variables that depend on no universally quantified variables translate into constants. In the alphabet, there exists a letter denoted by the variable $y$ such that, for all letters denoted by the variable $x$, $y$ occurs earlier in the alphabet than $x$. In this example, the variable $y$ is an existentially quantified variable depending on no universally quantified variables. Thus, $y$ can be replaced with a constant, say $a$.

\[
\text{EARLIER}(a, x).
\]

Thus, there is a crucial difference between “for all $x$ there exists a $y$” and “there exists an $x$ for all $y$”. In the first case, the corresponding clause (or clauses) contains a function of one variable, while in the latter a constant is employed.

The final point to note about variables is that their meaning is relevant only to the clause in which they occur. Thus, if the variable $x$ occurs in two different clauses, no implied connection between the clauses exists because of sharing a variable. In other words, the “for all” that is to be read for the variables in a clause governs only that one clause.

Exercises

3. Convert the following statements to clauses. (Note that, in the second exercise that follows, if the variables are interchanged in the SISTER literal, $z$ could be an aunt, but $z$ could also be an uncle.)
   a. if $\text{MOTHER(Mary, Sam)}$ and $\text{SISTER(Linda, Mary)}$ then $\text{AUNT(Linda, Sam)}$
   b. if $\text{MOTHER(x, y)}$ and $\text{SISTER(z, x)}$ then $\text{AUNT(z, y)}$
   c. if $\text{MOTHER(x, y)}$ and $(\text{not SISTER(x, z)})$ then $(\text{not AUNT(z, y)})$
   d. if $\text{MOTHER(x, y)}$ or $\text{FATHER(x, y)}$ then $\text{PARENT(x, y)}$
   e. $\text{not (POSITIVE(x) and NEGATIVE(x))}$

4. How would the following statements appear in clause form?
   a. For any $x$, there exists a $y$ such that $\text{GREATERTHAN(y, x)}$.
   b. There exists an $x$ such that for all $y$ $\text{GREATERTHAN(y, x)}$.
   c. For any $x$ and $y$, there exists a $z$ such that $(z = x + y)$.
   d. For any $x$, there exists a $y$ such that for all $z$ $(\text{if } (x < z) \text{ then } (y < z))$.
   e. There exists an $x$ such that for all $y$, there exists a $z$ such that $(y + z > x)$ or $(y + z = x)$.
   f. There exists an $x$ such that for all $y$, there exists a $z$ such that $(y + z > x)$ and $(y - z < x)$. 
3.3 Submitting a Problem to a Reasoning Program

An automated reasoning program begins its attack on a problem with the information you supply. You pick the problem, you tell it what you wish it to know, and you give it instructions about how to proceed—instructions discussed later concerning, among other items, how to reason and how to control the reasoning; see, respectively, Sections 3.4 and 3.9.

3.3.1 Assumptions and Axioms

To use an automated reasoning program, you must supply it with various clauses. One set of clauses corresponds to the general information and description of the problem domain; with OTTER, such information is often placed in a list called usable. The information that these clauses correspond to includes the assumptions or axioms of the problem domain. Also included in this set of clauses are those that correspond to useful, general information.

For example, if the puzzle is about jobs people might hold, the clause

\[ \text{MALE}(x) \lor \text{FEMALE}(x) \]

corresponds to an axiom about people—all of them are male or female. The clause

\[ \text{FEMALE}(\text{Roberta}) \]

gives useful information about Roberta. Since an automated reasoning program knows nothing about a problem unless you tell it the information, the set of clauses that corresponds to the assumptions and axioms must contain enough to pin down the various concepts relevant to the problem.

3.3.2 Special Facts and the Special Hypothesis

Another set of clauses that you supply to an automated reasoning program is called the special hypothesis. The special hypothesis of a problem is that which narrows the study to a particular domain. Two examples, one from everyday situations and one from mathematics, nicely illustrate the concept.

In the first example, you are asked to plan a party and are given various bits of general information concerning food and beverage and the like, and you are also told (the narrowing information) that the party’s purpose is to celebrate a birthday. The narrowing information is the special hypothesis, reducing the domain (from among parties in general) to that of a birthday party. In the second example, you are asked to prove that commutativity is present for all groups in which the square of \( x \) (for all elements \( x \)) is the identity \( e \). The special hypothesis for this theorem is the property that the square of (every) \( x \) is \( e \).
3.3.3 Denial of the Goal or Theorem

The final set of clauses that ordinarily is given to an automated reasoning program corresponds to assuming that the goal cannot be reached or, equivalently, that the conclusion is false. The clause or clauses are called the denial of the goal or the theorem. Recall, as discussed in Chapter 2, that the usual procedure in using an automated reasoning program to prove some statement is to assume the statement is false. If, in fact, the statement does follow from the remaining facts and properties of the problem, then assuming it is false eventually leads to a contradiction. The inclusion of such a statement or such statements provides a very convenient means for an automated reasoning program to test for completion of the task. When I said “ordinarily is given”, I had in mind that some uses of a reasoning program are simply for finding information, and so no particular result is being sought. Thus, when simply seeking additional information, nothing exists on which to key for seeking a contradiction.

However, the typical use of a reasoning program is that of proving some particular fact or of answering some specific question. For example, if you were trying to prove that Roberta is the teacher, you would typically assume that she is not the teacher. You would include the clause

\[-\text{HASAJOB}(\text{Roberta}, \text{teacher}).\]

as the denial of the goal or theorem. As a trickier example, if you were trying to match four given people to eight specific jobs with each of the four holding two, the denial would consist, in effect, of assuming that the task of finding the two jobs for each of the four people could not be completed.

Exercise

5. Consider the following puzzle.
   a. The only animals in this house are cats.
   b. Every animal that loves to gaze at the moon is suitable for a pet.
   c. When I detest an animal, I avoid it.
   d. No animals are carnivorous, unless they prowl at night.
   e. No cat fails to kill mice.
   f. No animals ever take to me, except that that are in this house.
   g. Giraffes are not suitable for pets.
   h. None but carnivora kill mice.
   i. I detest animals that do not take to me.
   j. Animals that prowl at night always love to gaze at the moon.

The object is to prove that “I always avoid a giraffe”.
First, write the assumptions of the problem in clause form. To do this, use
predicates that specify classes of animals. For example, the first statement, in clause form, is

$$\neg \text{INHOUSE}(x) \lor \text{CAT}(x).$$

Then write clauses that correspond to the denial of the statement to be proved.

6. What clauses would you write for problem 5 if a single predicate $\text{ISA}(x,y)$ were used, where $\text{ISA}(x,y)$ is taken to mean that “$x$ is a member of the group $y$”?

3.4 Inference Rules

An inference rule is an algorithm that, when successfully applied to some given set of hypotheses or premisses, yields a conclusion that follows inevitably and logically from the premisses. (The words “premiss” and “premisses” are technical terms in logic, and thus the unusual spelling.) Inference rules—types of reasoning—are the means for yielding possibly new information from given information. The conclusion obtained by applying some inference rule may, of course, not be new, because that information may already be present.

3.4.1 Unification

Two literals can be unified if there exists a replacement for the variables in both such that, after the replacement, the resulting literals are identical except possibly for the sign. The replacement, of course, substitutes the same term for all occurrences of a given variable. The procedure of unification is the means for unifying two literals. Unification is also used to unify two terms, as occurs in demodulation (Section 3.7) and also occurs in the inference rule paramodulation (discussed in Section 3.4.5). The unification procedure always obtains the most general replacement that can be found, if one exists, for the two expressions to make them identical. When unification is attempted for two given literals, the fact that either is negative or positive is ignored. However, for most of the inference rules, a pair of literals is considered only when they are opposite in sign—one negative and one positive.

For example, the clauses

$$\text{FEMALE}(\text{Roberta}).$$

$$\neg \text{MALE}(x) \lor \neg \text{FEMALE}(x).$$

contain literals that can be unified, yielding

$$\neg \text{MALE}(\text{Roberta}).$$

stating that Roberta is not male. The unification procedure finds that a replacement of the variable $x$ by Roberta is required. The two literals that are unified are $\text{FEMALE}(\text{Roberta})$ and $\neg \text{FEMALE}(x)$. 
Inference Rules

A somewhat subtle point in unification is exhibited by the following example.

\[ \text{GE}(\text{sum}(x, \text{sum}(y, \text{minus}(x))), y). \]
\[ -\text{GE}(\text{sum}(x, x), y) \mid \text{ACCEPTABLE}(\text{pair}(x, y)). \]

The first clause says that \( x + (y - x) \) is greater than or equal to \( y \). The second clause says that if \( x + x \) is greater than or equal to \( y \), then the pair \( x, y \) is acceptable. Do you see why the two literals in the predicate GE cannot be unified? Even better, do you understand the basic rule that describes such situations that cannot be successfully unified?

The two literals cannot be unified because no (uniform) replacement of variables exists that yields two new and identical (except for sign) literals. (The fact that the same variable \( x \) appears in both clauses is not relevant; recall that a variable has meaning only for the clause in which it appears. Thus, to attempt to unify the two literals from the two clauses, you must in effect rename the variables before making the attempt so that no variables are shared by the two clauses.) As for the principle explaining such failures, note that unification can be attempted by a left-to-right symbol matching. Before applying the algorithm to literals from different clauses, the variables must be renamed so that no variable is shared. In this example, the first step of this matching process dictates replacing the variable that first occurs in each literal by a common variable, say \( z \). The next matching step attempts to then match the new variable \( z \) from the second clause with the term \( \text{sum}(y, \text{minus}(z)) \) from the first clause. Thus, failure occurs, since unification does not succeed if at some point in the matching process a variable is matched with a term that already contains that variable. Of course, successful unification requires that the predicates be the same, when applied to two literals.

3.4.2 Binary Resolution

The object of an application of binary resolution is to produce a new clause from two existing clauses, each of which happens to contain an appropriate literal. Formally, binary resolution is that inference rule that takes two clauses, selects a literal in each of the same predicate but of opposite sign, and yields a clause providing that the two selected literals unify. The result of a successful application of binary resolution is obtained by applying the replacement that unification finds to the two clauses, deleting from each (only) the descendant of the selected literal, and taking the or of the remaining literals of the two clauses. Of course in practice, when you apply binary resolution to a pair of clauses, you must rename the variables in the two clauses so that no variable is shared.

Binary resolution is a generalization of the inference rule that yields \((Q \text{ or } R)\) from \((P \text{ or } Q)\) and \((\text{not } P \text{ or } R)\). Equivalently, binary resolution is a generalization of the inference rule that yields \((Q \text{ or } R)\) from \((\text{if } P \text{ then } Q)\) and \((\text{if } \text{not } P \text{ then } R)\). The generalization allows \( Q \) or \( R \) (or both, see Section 3.5) to be empty. The
generalization also allows for unification to be required to produce the required P and -P. From a more classical viewpoint, binary resolution is a generalization of both modus ponens and syllogism.

For example, binary resolution applied to the two clauses

\[-\text{HASAJOB}(x,\text{nurse}) \mid \text{MALE}(x).\]
\[-\text{MALE}(x) \mid -\text{FEMALE}(x).\]

yields

\[-\text{HASAJOB}(x,\text{nurse}) \mid -\text{FEMALE}(x).\]

as the result. Binary resolution places no restrictions on either of the two clauses to which it is being applied, and no restriction on the resulting clause.

### 3.4.3 UR-Resolution

The object of an application of UR-resolution is to produce a unit clause from a set of clauses one of which is a nonunit clause while the remaining are unit clauses. Formally, UR-resolution is that inference rule that applies to a set of clauses one of which must be a nonunit clause, the remaining must be unit clauses, and the result of successful application must be a unit clause. Furthermore, the nonunit clause must contain exactly one more literal than the number of (not necessarily distinct) unit clauses in the set to which UR-resolution is being applied. In addition, except for one literal, the literals of the nonunit clause must be paired with the unit clauses such that each literal has the predicate of its paired unit clause, the two members of a pair are opposite in sign, and the members of a pair unify. For UR-resolution to succeed, a simultaneous replacement of all variables in all clauses in the set under consideration must exist that, when applied to each pair, makes the pair identical except for sign. The result of a successful application is obtained by applying the most general replacement that exists and that satisfies the given criterion, and canceling all but the one literal of the nonunit clause after the replacement has been made in it.

A successful application of UR-resolution can be viewed as a sequence of binary resolutions in which each is required to involve exactly one unit clause. However, all such binary resolutions must occur simultaneously, thus yielding no intermediate clauses. For example, UR-resolution applied to the clauses

\[-\text{MARRIEDTO}(x,y) \mid -\text{MOTHER}(x,z) \mid \text{FATHER}(y,z).\]
\[\text{MARRIEDTO}((\text{Thelma},\text{Pete}).\]
\[-\text{FATHER}(\text{Pete},\text{Steve}).\]

yields

\[-\text{MOTHER}(\text{Thelma},\text{Steve}).\]
Inference Rules

as the result. In practice, of course, to attempt an application of UR-resolution, you must rename the variables in the set of clauses under consideration so that no variable appears in more than one clause in the set. The name UR-resolution is derived from “unit resulting”, since the object of using the rule is to produce unit clauses.

3.4.4 Hyperresolution

The object of an application of hyperresolution is to produce a positive clause from a set of clauses one of which is negative or mixed while the remaining are positive clauses. Formally, hyperresolution is that inference rule that applies to a set of clauses one of which must be a negative clause or a mixed clause, the remaining must be positive clauses and their number must be equal to the number of negative literals in the negative or mixed clause, and that requires the result of successful application to be a positive clause. The positive clauses are called satellites, while the clause with the negative literals is called a nucleus. The number of negative literals in the nucleus must equal the number of satellites, which are not necessarily distinct clauses. The literals of the nucleus must be paired with a literal in the corresponding satellite such that each negative literal has the predicate of its paired literal and such that each literal unifies with its paired literal. For hyperresolution to succeed, a simultaneous replacement of all variables in all clauses in the set under consideration must exist that, when applied to each pair, makes that pair identical except for sign. The result of a successful application is obtained by applying the most general replacement that exists and that satisfies the given criterion, canceling all of the negative literals and their counterparts in the positive clauses, and taking the or of the remaining literals, after the replacement of variables has been made, in all clauses.

A successful application of hyperresolution can be viewed as a sequence of binary resolutions in which each is required to involve exactly one positive clause. However, all such binary resolutions must occur simultaneously, thus yielding no intermediate clauses. For example, hyperresolution applied to the clauses

\[-\text{MARRIEDTO}(x,y) \mid -\text{MOTHER}(x,z) \mid \text{FATHER}(y,z).
\text{MARRIEDTO}(\text{Thelma, Pete}) \mid \text{OLDERTHAN}(\text{Thelma, Pete}).
\text{MOTHER}(\text{Thelma, Steve}).\]

yields

\[\text{FATHER}(\text{Pete, Steve}) \mid \text{OLDERTHAN}(\text{Thelma, Pete}).\]

as the result.

The need for hyperresolution is explained by the fact that hyperresolution can yield a positive nonunit clause, while UR-resolution can yield only unit clauses. On the other hand, UR-resolution can yield negative clauses, while hyperresolution
cannot. In some cases, hyperresolution and UR-resolution produce the same clause. The name hyperresolution indicates that more is occurring than occurs in (ordinary) binary resolution, for hyperresolution combines the actions of a number of binary resolution steps into one step. As with UR-resolution, the variables in the clauses are renamed so that no variable appears in more than one clause.

3.4.5 Paramodulation

The object of an application of paramodulation is to cause an equality substitution to take place from one clause into another. Paramodulation is that inference rule applied to a pair of clauses and requiring that at least one of the two contains a positive equality literal, and yielding a clause in which the equality substitution corresponding to the equality literal has occurred. The clause containing the equality literal is called the from clause, and the clause into which the equality substitution is made is called the into clause. By using this inference rule, equality is “built in”; none of the usual equality axioms need be present other than that for reflexivity, namely, \( \text{EQUAL}(x,x) \). A successful application of paramodulation requires the selection of a positive equality literal (by convention one whose predicate begins with \( \text{EQUAL} \) or \( \text{EQ} \)) in the from clause, the selection of one of its two arguments, the selection of a term in the into clause, and the successful unification of the selected argument and the selected term.

Obtaining the result of a successful application of paramodulation first requires applying to each of the two clauses the replacement of variables dictated by the successful unification of the selected argument and term. Second, for the selected term that has the variable replacement applied to it, paramodulation requires substituting the other argument of the selected equality literal after applying the variable replacement. Finally, the inference rule requires deleting the selected positive equality literal and forming the or of the remaining literals in the from clause and all the literals in the into clause that result after applying the preceding steps (of course ignoring duplicate literals). Paramodulation is a generalization of equality substitution.

For example, paramodulation applies in the following situation and with the following result.

**Fact:** A person’s father is older than the person:
\( \text{OLDERTHAN(father(x),x)} \).
**Fact:** Jack’s father is Ralph:
\( \text{EQUALP(father(Jack),Ralph)} \).
**Conclusion:** Ralph is older than Jack:
\( \text{OLDERTHAN(Ralph,Jack)} \).

The first of the three clauses is the into clause, the second the from clause, and the third the result of a successful application of paramodulation. Paramodulation
Inference Rules

also includes ordinary equality substitution, as exhibited by the following example. (Although the concept of sister cannot ordinarily be represented with a function because it does not always denote a unique object, it is represented as such here for convenience.)

Fact: Ted’s sister’s husband is Bob:
EQUALP(husband(sister(Ted)),Bob).
Fact: Ted’s sister is Mary:
EQUALP(sister(Ted),Mary).
Conclusion: Mary’s husband is Bob:
EQUALP(husband(Mary),Bob).

The first clause is the into clause, the second the from clause, and the third the result of successfully applying paramodulation.

As the first example shows, paramodulation goes beyond equality substitution and even includes the case, illustrated with the next example, in which variables must be replaced in both the from and the into clauses before the substitution can take place. This case occurs when the required unification forces a nontrivial replacement of variables simultaneously in both the from and the into clause. Thus, paramodulation can yield clauses that might not otherwise be obtained by ordinary equality substitution. For example, from the clauses

EQUAL(sum(0,x),x).
EQUAL(sum(y,minus(y)),0).

paramodulating from the first into the second yields

EQUAL(minus(0),0).

as the result. A somewhat more complex illustration is provided by applying paramodulation to the clauses

EQUAL(sum(x,minus(x)),0).
EQUAL(sum(y,sum(minus(y),z)),z).

where the first clause is the from clause, and the second the into. The result of the paramodulation is a clause that says that \( y + 0 = -(-y) \). The actual detailed paramodulation is left as an exercise for you. An additional example, concerning relationships between people, which illustrates the value of permitting nontrivial variable replacement in both the from and the into clauses, can be found in Section 3.10. In practice, of course, to apply paramodulation, you rename the variables in the two clauses so that no variable appears in both clauses. The name paramodulation is derived from the close relationship of the inference rule to demodulation, a concept discussed in Section 3.7.
3.4.6 Other Inference Rules

**Negative hyperresolution** is that inference rule that interchanges the roles of positive and negative in hyperresolution. **Unit resolution** is that restriction of binary resolution that requires that at least one of the two clauses be a unit clause. (OTTER does not offer the use of the inference rule unit resolution; however, since the source code for the program is included on the CD-ROM, you have the opportunity of adding this feature to your copy of OTTER.) **Factoring** is that rule that takes a single clause, selects two of its literals that are alike in predicate and in sign, attempts to unify the two literals, and applies the replacement of variables that corresponds to a successful unification to produce the new clause. While most inference rules operationally require that the variables be renamed in all of the clauses being considered by the rule so that no two clauses have a variable in common, factoring requires that the variables be left as is. The preceding three inference rules are additional examples of what can be offered by an automated reasoning program.

Especially since the source code is included on the CD-ROM, the program OTTER can be enhanced to offer other combinations and variations, such as requiring that the result of hyperresolution be restricted to be a unit clause. Such a restriction combines some of the properties of hyperresolution with some of those of UR-resolution. A similar restriction that can be imposed on various inference rules is that the result of a successful application be accepted only if the number of literals in the conclusion is less than some fixed number. Thus, rather than being forced to choose from among the inference rules given here, you can mix and match according to what you suspect might aid a reasoning program in its attempt to answer a question or solve a problem. The choice of inference rule can have an overwhelming effect on the performance of a reasoning program. By offering many inference rules from which to choose, a reasoning program offers you the opportunity of sharply increasing the effectiveness of using such a program.

The formulation of inference rules that offer possibly far more reasoning power continues to be a fruitful area for research. From recent research (Veroff and Wos 1992), you now have access to a class of rules called **linked inference rules**, many of which generalize the more well known rules such as binary resolution, UR-resolution, and hyperresolution. Among such rules, OTTER offers the use of **linked UR-resolution**. Where standard UR-resolution requires that each literal to be “removed” from the nucleus be unified with a unit satellite, linked UR-resolution generalizes the standard rule by also permitting such a removal with a **chain** of clauses (appropriately constrained) that includes one or more nonunit clauses. Where standard hyperresolution requires that each negative literal of the nucleus be removed with the use of a satellite consisting of positive literals only, **linked hyperresolution** also permits such a removal with a chain of clauses (appropriately constrained) that includes one or more clauses consisting of both positive and negative literals. In contrast to standard paramodulation, **linked paramodulation** permits, in a single reasoning step, repeated unification with the selected **into term** and its descendants.
Of course, as you probably expect, linked paramodulation also captures other types of equality-oriented reasoning; for example, in a single step of linked paramodulation, a substitution into a term might be followed by a substitution into a superterm containing that term. Since the current version of OTTER offers the use of neither linked hyperresolution nor linked paramodulation, again you have the opportunity of enhancing your copy by relying on your programming skills.

3.5 The Empty Clause

The empty clause is that clause that contains no literals. If a reasoning program finds this clause, then the program has found a contradiction. In this book, frequently no need exists for the empty clause; contradiction is found usually when two unit clauses are obtained that are alike in predicate, are opposite in sign, and are unifiable. When such a condition has been obtained, unit conflict has been found. By definition, the empty clause is a positive clause, but is also a negative clause. The empty clause is needed in connection with some applications of hyperresolution.

For example, if a reasoning program obtains clauses that say that Steve is strictly older than Pete and that Pete is strictly older than Steve, then rather quickly a contradiction is found. In such a situation, the key clause is

\[-\text{OLDER}(x,y) \mid -\text{OLDER}(y,x).\]

With hyperresolution as the sole rule of inference, an automated reasoning program will, in the presence of the clauses

\[
\text{OLDER}(\text{Steve}, \text{Pete}). \\
\text{OLDER}(\text{Pete}, \text{Steve}).
\]

deduce the empty clause. Such a deduction informs a reasoning program that a contradiction has been found and, therefore, that the assignment has been completed.

3.6 Proof by Contradiction

A proof by contradiction consists of a number of steps, starting with a given set of facts and properties, such that each step that is not a given step follows from earlier steps and is obtained by applying a specific inference rule, and such that the final step is in direct contradiction with an earlier step. This definition can be modified to cover the case in which the final step is the empty clause, but typically the contradiction is between two unit clauses.

By far the most common termination condition for a reasoning program that succeeds in the assigned task is that of finding a proof by contradiction. The usual way of using a reasoning program to prove that some property follows from a given
set of properties, facts, and definitions is to assume the desired conclusion false. If the desired conclusion does follow from the remaining clauses, then assuming it false must in principle eventually lead to a contradiction. Although the precise discussion of this topic is left for Chapter 8, the following example illustrates what can go wrong.

Assume that you have chosen binary resolution as your standard approach to attacking a new problem and that, in fact, it is the only inference rule being employed. Also assume that you are asked to demonstrate the usefulness of an automated reasoning program to a new user and that the new user has selected a problem that, with a straightforward approach to representation, yields the following clauses.

\[ P(x) \lor P(y). \]
\[ \neg P(x) \lor \neg P(y). \]

Although this set of clauses is inconsistent—a proof by contradiction can be obtained starting with the set and using the appropriate inference rules—a reasoning program will not find a proof of the inconsistency under the given conditions. (A set that is inconsistent is technically called an unsatisfiable set.) The failure to find a proof is caused by the omission, for this problem, of a needed second inference rule. Without the second inference rule, a reasoning program will simply deduce clauses such as

\[ P(y) \lor \neg P(z). \]

and quickly exhaust the set of (distinct) deducible clauses. To complete the task of finding a proof, and thus provide the desired demonstration, the inference rule factoring is needed. Do you see what results with that addition?

What happens is that a reasoning program yields

\[ P(x). \]

from the first of the two clauses, and

\[ \neg P(x). \]

from the second. As defined in this book, a proof is thus obtained.

### 3.7 Demodulation

Demodulation is the process of rephrasing or rewriting or simplifying or canonicalizing expressions by automatically applying unit equality clauses designated for this purpose. Its use is virtually required in a large fraction of the applications of automated reasoning. A unit equality that is designated to be used with demodulation is called a demodulator. Typically, all terms of all newly generated clauses are examined for possible demodulation. A term is demodulated if a demodulator exists such that one of the arguments of the demodulator can be unified with the
Demodulation

term without replacing any of the variables in the term. A common convention is to consider only the first argument of the demodulator for unification with the term under consideration. The demodulated term is obtained by replacing the term by the second argument of the demodulator after the variable replacement is applied corresponding to the successful unification. Then the original version of the clause is replaced by the demodulated version. (This replacement, and hence the discarding of the original version of the clause, is one difference between demodulation and paramodulation. The other important difference is, of course, that demodulation permits variable substitution—when unification is being attempted—only in the equivalent of the “from” clause.) When a new demodulator is found by a reasoning program, depending on the instructions given to the program, all clauses that have been previously retained are examined for possible demodulation with the new demodulator. Such a process is called back demodulation. Typically, one successful application of demodulation is followed immediately by further attempts to apply other demodulators.

To succeed, demodulation requires that the variables of the term under consideration for demodulation remain as they are. Thus,

\[ \text{EQUALP}(\text{father}(\text{Pete}), \text{Steve}). \]

demodulates no term in the clause

\[ \text{OLDER}(\text{father}(x), x). \]

since the variables of the second clause would be affected nontrivially by the results of any unification. Paramodulation would, however, yield

\[ \text{OLDER}(\text{Steve}, \text{Pete}). \]

if applied to the two clauses and, of course, without deleting a clause.

Demodulation has a close relationship to paramodulation both in origin and in purpose. Each, if successful, causes an equality substitution to take place. While demodulation requires the equality literal to be in a unit clause, paramodulation does not. Another vital difference is that, while demodulation allows a nontrivial variable replacement only in the argument of the equality literal, paramodulation allows a nontrivial replacement of variables in both the argument of the equality literal and in the term into which the substitution is being attempted. Two additional differences have already been mentioned. First, while demodulation discards the original version of the clause if demodulation is successful, with paramodulation both the original version and the new clause are kept. Second, one successful application of demodulation immediately triggers further attempts at demodulation, while paramodulation stops after a single application of equality substitution.
3.8 Subsumption

Subsumption is the process for discarding a clause that duplicates or is less general (and hence logically weaker) than another clause available to a reasoning program. Subsumption is used to address the obstacle of redundancy. One clause subsumes a second clause if the variables in the first can be replaced in such a manner that the resulting clause is a (not necessarily proper) subclause of the second. Thus,

\[ \text{OLDER}(\text{father}(x), x). \]

subsumes

\[ \text{OLDER}(\text{father}(\text{Ann}), \text{Ann}). \]

On the other hand,

\[ \neg \text{WIFE}(\text{Kim}, \text{Bob}) \mid \text{FEMALE}(\text{Kim}). \]

does not subsume

\[ \neg \text{WIFE}(x, y) \mid \text{FEMALE}(x). \]

for the second of these two clauses is more general than the first, so the test for the first subsuming the second fails. Note that the second clause, on the other hand, does subsume the first.

A different example is provided by

\[ \neg \text{WIFE}(\text{Kim}, \text{Bob}) \mid \text{FEMALE}(\text{Kim}) \mid \text{FEMALE}(\text{Kim}). \]

in which the second clause subsumes the first. One clause can subsume another regardless of the number of literals in either. When a newly generated clause is discarded because a previously retained clause subsumes it, the process is called forward subsumption. When a newly generated clause is used to discard previously retained clauses by subsumption, the process is called back subsumption. In many cases, you need not use back subsumption; in contrast, the use of forward subsumption is virtually a requirement, unless you are willing to let your automated reasoning program drown in redundant information.

Although I only briefly touch on the concept of ancestor subsumption here, with a fuller treatment of its value given in Section 7.5.2, I note that this variant of subsumption offers substantial assistance when the object is to find a shorter proof than that in hand. First, note that the clause A properly subsumes the clause B if and only if A subsumes B but B does not subsume A. Second, note that the derivation length of the clause A is the number of distinct steps in the deduction of A, not including those that are among the input; thus the derivation length is equal to the number of applications of the inference rule or rules used to deduce A. Finally, by definition, the clause A ancestor-subsumes the clause B if and only if (1)
A properly subsumes B or (2) A and B are alphabetic variants and the derivation length of A is less than or equal to that of B. When OTTER is instructed to use ancestor subsumption, the deduction of a second copy of an already-retained clause causes the program to compare the two derivation lengths. If the length of the copy is strictly less than that of the already-retained clause, then, if back subsumption is in use, the copy is retained in the set of support list with the original clause purged from the database.

Of the various techniques offered by an automated reasoning program for seeking shorter proofs, the use of ancestor subsumption is the most direct and practical. The other techniques that can be used to seek shorter proofs include a breadth-first search, weighting, and demodulation. However, the pursuit of shorter proofs is far more complicated than it might at first appear. Indeed, although the fact is not particularly obvious, you can be misled into believing that progress is being made when the program finds a substantially shorter proof of a sought-after lemma or intermediate step. There can exist two proofs of A such that (1) the deduced clause C is present in both, (2) the derivation length of C in the first proof is substantially less than it is in the second proof, and (3) the length of the first proof (of A) is substantially longer than the length of the second proof. Such can occur when the derivation (or subproof) of C in the second proof contains a number of steps absent from its derivation in the first proof, and those steps are used repeatedly to complete the shorter proof of A. In other words, the absence in the first proof of those repeatedly used steps forces the program to use far more expensive steps in their place, where expense is measured in terms of derivation length. (For those who wish to pursue the preceding discussion in far more depth, I recommend my latest book featuring numerous experiments (Wos 1996a).)

Therefore, the use of ancestor subsumption—although far more effective than any other technique currently known—does not guarantee answering the question of whether the shortest possible proof has been found. In contrast, the use of a breadth-first search without the use of any type of subsumption does in principle offer such a guarantee; but, in the vast majority of cases, an unrestrained use of such a search is totally impractical. You might find it interesting to note that, when a breadth-first search is in use, the first proof that is found is not necessarily the shortest that can be found. You might find it amusing to construct an appropriate example of such an occurrence. Since shorter proofs are of interest for their elegance and for their relevance to finding more efficient circuits, algorithms, and computer code, and since (as correctly suggested by the preceding observations) the problem of seeking shorter proofs is complex and intriguing, you might enjoy its study with your copy of OTTER.
3.9 Strategy

Strategy is a means for controlling an automated reasoning program’s attack on a problem. Some strategies, called restriction strategies, require a reasoning program to avoid ever considering certain combinations of clauses. Other strategies, called direction strategies, direct a reasoning program in its choice of which clause to focus on next. In addition, processes such as subsumption can be classed as pruning strategies, while processes such as demodulation can be classed as canonicalization strategies. Without strategy, an automated reasoning program would usually, even for the simplest of problems, deduce too many conclusions to be effective.

3.9.1 The Set of Support Strategy

The set of support strategy prohibits application of an inference rule to a set of clauses unless at least one of the clauses has support. (The vignette in Section 10.3 is devoted to the set of support strategy.) A clause has support if it is an input clause and has been designated as having support, or if it is obtained by application of an inference rule to a set of clauses one of which has support. In other words, having support is inherited in the sense that in each step of the derivation of the clause, at least one of the clauses participating in the step has support. Technically, therefore, it is the occurrence of a clause that has support, for a clause may be deduced in more than one way.

The recommended choices for the initial or input set of support are either the set of clauses that constitute the special hypothesis together with those that constitute the denial (of the theorem to be proved or objective to be reached), or simply those constitute the denial. Notice that either of these choices has the property that the complement is expected to be a consistent set of clauses. (A consistent set of clauses is technically called a satisfiable set.) By constraining a reasoning program from applying inference rules to sets of clauses completely contained in this consistent set, a reasoning program is prevented from exploring the domain of inquiry as a whole.

From an intuitive viewpoint, the set of support strategy restricts a reasoning program from simply expanding a set of consistent clauses, a move that makes sense because (usually) a proof by contradiction is being sought. Thus, the strategy takes advantage of the satisfiability of the set of clauses that constitute the complement of the typically chosen set of support. Should the initial set of support be chosen unwisely, then all proofs might be blocked. This topic is reserved for Chapter 8. The set of support strategy is a restriction strategy. The strategy is offered by OTTER; in fact, when you use this program, you must include one or more clauses in the list(sos), or OTTER will be unable to draw any conclusions.
3.9.2 Weighting

Weighting is the procedure for assigning priorities to terms, clauses, and concepts. Weighting can be used to reflect your knowledge and intuition about how a reasoning program should proceed. For example, in a puzzle such as the jobs puzzle in which each person has a specific job, you can choose the appropriate weights to cause a reasoning program to focus on one person. You can assign a weight to Roberta in such a way that the program will always choose (as the next clause on which to base its reasoning) a clause in which the term “Roberta” occurs in preference to any other clause. On the other hand, you can assign a weight to Roberta so that every clause containing the term Roberta is given the merest of consideration, or even immediately discarded as undesirable. Weighting is both a direction and a restriction strategy. OTTER offers weighting; if you choose not to provide any weight templates to affect the program’s actions, OTTER will assign priorities based on symbol count.

3.9.3 Unit Preference Strategy

Briefly, the unit preference strategy is the strategy that causes an automated reasoning program to prefer for application of the inference rule binary resolution a set of clauses one of which is a unit clause. Further, if a unit clause is found, an automated reasoning program employing this strategy seeks a clause with the fewest possible literals and containing a literal that unifies with the unit clause but is of opposite sign. If no such pair exists, the unit preference strategy causes a reasoning program to prefer a set in which one of the clauses has a number of literals as few as possible. (OTTER does not offer the use of this strategy; however, since the source code for the program is included on the CD-ROM, you have the opportunity of adding this feature to your copy of OTTER. You also have the opportunity of generalizing the strategy to direct the application of other inference rules.)

The motivation for the unit preference strategy is the seeking of unit clauses, and especially of unit conflict. Since applying the strategy (when using binary resolution) to a pair of clauses one of which is a unit clause must yield a shorter clause than the participating nonunit, a reasoning program employing the unit preference strategy is proceeding in the direction of generating unit clauses. Unit clauses play an extremely important role in automated reasoning, independent of their potential for establishing proof by contradiction.

For example, with the unit preference strategy, if binary resolution is the only inference rule being used, then the first set of pairs of clauses to be examined (if such pairs exist) is that in which at least one of the clauses is a unit clause. Within that set of pairs, the strategy prefers a pair in which the nonunit clause has two literals rather than a pair in which the nonunit clause has three. If no unit clauses are available, a reasoning program using this strategy will seek a clause with two literals. If a clause with two literals is found, the strategy prefers that the other
nonunit clause also have two literals rather than, say, three or more. Finally, when new unit clauses are deduced and retained, again pairs including one of the unit clauses will be preferred. The unit preference strategy is a direction strategy.

3.9.4 Other Strategies

Two additional direction strategies merit attention, each offered by OTTER. In the first strategy, called the queue strategy, the choice of the clause on which to focus next is based purely on the order in which clauses are retained; therefore, the weight of a clause is irrelevant to making the choice. In other words, with this strategy, clauses are taken first-come first-served; the strategy treats the retained clauses as a queue and is a breadth-first search strategy in the following sense. By definition, the level of an input clause is 0, and the level of a deduced clause is one greater than the maximum of the levels of its (immediate) parents. Excluding input clauses, use of the queue strategy causes the automated reasoning program to focus first on all retained clauses of level 1, then on all those of level 2, and so on.

If the set of support strategy is in use, the queue strategy is constrained by requiring that the clause chosen as the focus of attention (to drive the reasoning) be from among those in the set of support list. With OTTER, since you must include at least one clause in the initial set of support (otherwise no conclusions will be drawn), you can defeat the intent of the set of support strategy by placing all input clauses in list(sos). Once chosen as the focus of attention, the clause is immediately moved from the set of support to the usable (formerly called axioms) list of clauses, and the chosen inference rules are then applied. Regardless of which strategy is in use, OTTER attempts to complete an application of each chosen inference rule by selecting any needed clauses from the usable (or axioms) list.

As you can see, in sharp contrast to the use of weighting, the use of the queue strategy permits an automated reasoning program to focus on clauses with high weight before clauses with much lower weight, if the high-weight clauses are retained earlier. You might wish access to a strategy that gives you the best of both worlds, allowing the program to choose (as the focus of attention) clauses with high weight as often or nearly as often as choosing clauses with low weight. If that is your desire, then the second of the two strategies (featured in Sections 7.5.2 and 7.7)—that called the ratio strategy—will serve you well. To use the ratio strategy, you choose an integer $k$ and assign it as the value for the pick\_ratio. The program then chooses clauses (as the focus of attention) in the order $k$ by weight, 1 by queue (in effect), $k$ by weight, and so on. For example, if you choose 1 as the value for $k$, the program will alternate between focusing on a clause because of its weight and focusing on a clause that is earliest among the clauses remaining in the set of support list. In other words, an assignment of the value 1 to the pick\_ratio equally blends weighting with the queue strategy.
3.10 An Automated Reasoning Program in Action

I now illustrate, with one mundane example, a number of the concepts that have been discussed in this chapter, and also show how you might use an automated reasoning program. Assume that certain specific and certain somewhat general facts are available, selected from some hypothetical database of knowledge about relationships. For example, suppose that one of the somewhat general facts is that, if someone is Brian’s sibling, then the father of that person and Brian have Brian’s last name.

\[
(1) \neg \text{SIBLING}(x, \text{Brian}) \mid \text{EQUAL}(\text{last}(\text{father}(x, \text{Brian})), \text{last}(\text{Brian})).
\]

Suppose another of the somewhat general facts is that, if Rick is someone’s sibling, then the father of Rick and that person have Rick’s last name.

\[
(2) \neg \text{SIBLING}(\text{Rick}, y) \mid \text{EQUAL}(\text{last}(\text{father}(\text{Rick}, y)), \text{last}(\text{Rick})).
\]

Among the specific facts is that Brian is Rick’s sibling.

\[
(3) \text{SIBLING}(\text{Brian}, \text{Rick}).
\]

Of course, you know that Brian and Rick have the same last name, but how might an automated reasoning program prove it?

First, you ordinarily would write a clause that amounts to assuming the conclusion false, that says that Brian and Rick do not have the same last name.

\[
(4) \neg \text{EQUAL}(\text{last}(\text{Brian}), \text{last}(\text{Rick})).
\]

After all, since a proof is the objective, and since proof by contradiction is the usual form of proof, clause 4 is a good beginning.

Next comes hidden information, or easily overlooked information. As noted in earlier chapters—at least in the beginning of your use of a reasoning program—all potentially useful properties of concepts in the problem had best be supplied to a reasoning program. In the problem at hand, a property of sibling that might be needed is that if \(x\) is the sibling of \(y\), then \(y\) is the sibling of \(x\). Therefore, the following clause is added.

\[
(5) \neg \text{SIBLING}(x, y) \mid \text{SIBLING}(y, x).
\]

The value of this clause can be quickly seen. The literal of the unit clause 3 and the first literal of clause 2 share the predicate SIBLING. However, the argument Rick appears as the second argument in the one clause, and the first argument in the other. Noticing this potential difficulty points to the need for an additional clause, one that permits a reasoning program to cope with the differing position of the argument Rick. The predicate SIBLING is symmetric, and clause 5 will suffice. Clause 6
(6) $\neg$EQUAL($x$, $y$) $\lor$ EQUAL($y$, $x$).

also is a candidate for a clause that might well be needed. Clause 6 is suggested by the fact that clause 4 is likely to participate in the final step, that of unit conflict, and the other participant may not have the arguments in the required order to match clause 4.

Since equality literals are present, you might correctly conjecture that paramodulation is one of the inference rules to be employed. Since clause 3 gives information specific to the problem, and since clause 3 would interact with clause 5 were the appropriate inference rule present, a clue exists for choosing another inference rule to employ. Clauses 3 and 5 will produce new information if considered by binary resolution, by UR-resolution, or by hyperresolution. A further clue is provided by the presence of clause 4. Clause 4 is present as the denial of the desired result, and clause 4 would interact with clause 6 were the appropriate inference rule present. Clauses 4 and 6 produce new information either with binary resolution or with UR-resolution as the inference rule, but not with hyperresolution. So hyperresolution can be at least temporarily eliminated from consideration. Binary resolution is eliminated because the object of the problem suggests that a specific unit clause is being sought, and because the inference rule produces too much information. Binary resolution is an inference rule that you should avoid in most cases, for conclusions produced from its application often correspond to steps that are smaller than necessary. Finally, you should (in the majority of cases) use as few inference rules as possible to increase the effectiveness of an automated reasoning program’s attack on a problem. Thus, UR-resolution is the choice for the moment.

With the choice of representation and of inference rules made (at least for the moment), the question of strategy arises. In general, without strategy an automated reasoning program wanders from the goal. The recommended approach with this example is to use the set of support strategy. Clauses 3 and 4 are placed in the set of support, 3 because it is “special” to the problem, and 4 because it is the “denial” of the conclusion. The remaining four clauses are not placed in the set of support because they are general information.

An automated reasoning program employing the chosen set of support, paramodulation, and UR-resolution and starting with clauses 1 through 6 might obtain the following proof by contradiction.

From clauses 4 and 6:
(7) $\neg$EQUAL(last(Rick), last(Brian)).

From clauses 3 and 5:
(8) SIBLING(Rick, Brian).

From clauses 8 and 1:
(9) EQUAL(last(father(Rick, Brian)), last(Brian)).

From clauses 8 and 2:
(10) EQUAL(last(father(Rick, Brian)), last(Rick)).
From clauses 10 and 9 (by paramodulation):
(11) \( \text{EQUAL(last(Rick),last(Brian))} \).

From clauses 11 and 7, unit conflict

The given approach is successful in proving that Rick and Brian have the same last name, which was the goal. Further, an intuitive method has been shown for choosing representation, inference rules, and strategy. Finally, although clause 4 did not participate in the last step of the given proof, the conjecture that it might lead to good moves.

This simple example can be used to illustrate two additional points brought out in this chapter. If paramodulation is applied to clause 2 as the from clause with clause 1 as the into clause, the result

\(-\text{SIBLING(Rick,Brian)} \mid \text{EQUAL(last(Rick),last(Brian))}\)

is obtained. First, note that this last application of paramodulation requires a non-trivial variable replacement in both the from and the into clauses. Second, note that the resulting clause exhibits the collapsing of identical literals. Although I recommend in most cases avoiding paramodulation from nonunit clauses and avoiding paramodulation into nonunit clauses, occasionally a reasoning program must consider such applications.

As you may have surmised, once paramodulation is chosen for use, clause 6 is not needed, for paramodulation builds in equality. Of course, although it was not included in this simple illustration,

\(\text{EQUAL}(x,x)\).

should be included whenever equality is present in a problem. Many proofs are completed by discovering clauses such as

\(-\text{EQUAL}(a,a)\).

where \(a\) is some constant in the problem. If clause 6 were not present, could you complete the proof that Rick and Brian have the same last name?

### 3.11 OTTER and Earlier Automated Theorem-Proving Programs

Almost all of the features discussed in this book are offered by OTTER. If you wish access to those features not offered, you can test your programming skills by enhancing your own copy of the program, for all of the C code is present on the CD-ROM. Those missing features will in no way hinder you from obtaining (with your own copy of OTTER) essentially all of the results presented throughout this book.

Historically, many of those results were first obtained with the automated reasoning program AURA (AUtomated Reasoning Assistant) designed at Argonne National Laboratory and at Northern Illinois University. With much assistance from
AURA, open questions in mathematics and in formal logic were answered, circuits were designed and validated, various puzzles were solved, and assorted tasks were completed (Wos and Winker 1984). However, because AURA (unlike OTTER) was written in IBM 360/370 assembly language and PL/I, one of its unfortunate features is its lack of portability, in contrast to OTTER.

Another feature, shared by both AURA and OTTER, that you may find tedious is the requirement that typically you set many flags, choose various values for a number of control parameters, and comply with various other conventions. However, as you experiment with your copy of OTTER and master its use, you will discover its great versatility and power. Most fortunately, a mastery of the program is not required for you to find it a valuable aid in research and for applications.

A (possibly) monumental programming task that, when completed, would provide a most valuable enhancement to OTTER is to give the program the property that AURA possessed of having the capacity to perform as a set of reasoning programs. Each such reasoning program, called an environment, can function quite independently from the others. For example, one inference rule can be used for one task in one environment, while a different inference rule is used for a different task in a separate environment; see (Smith 1988).

If you enjoy program enhancement, another feature whose addition to OTTER would prove most useful is induction. Since induction plays such a vital role in certain types of mathematics and also in program verification, access to it would make OTTER even more useful. Whether induction should be treated as the Boyer-Moore program treats it is a challenging research question. Note that, for program verification, there currently exists no program that has been used successfully as often as the Boyer-Moore program has (Boyer and Moore 1998). Note also that their program is not designed for attacking the variety of questions and problems that can be answered and solved with OTTER.

If you have read of the automated reasoning program ITP—for example, in the my first book (Wos, Overbeek, Lusk, and Boyle 1992)—or have used that program, you might naturally wonder about its relation to OTTER. OTTER is written in approximately 28,000 lines of C; ITP was written in approximately 60,000 lines of Pascal. OTTER does not (currently) offer interactive use; ITP did. OTTER does not offer a built-in interface to Prolog; ITP did. OTTER is still being enhanced; ITP is not. In addition to running on a workstation (through the use of the included C code), a version of OTTER (found on the enclosed CD-ROM) runs on an IBM-compatible personal computer, while another runs on the Macintosh.; ITP did not offer such versions. Finally and (from a practical viewpoint) most important, OTTER completes assignments 10 to 70 times faster than did ITP.

Because of OTTER’s portability, versatility, and power, this program can be used effectively on many computers and in many contexts. The style in which it is written permits rather easy modification and enhancement. Researchers throughout the world are now using OTTER profitably, making copies of it for colleagues,
extending it in various ways, and—so it appears—immensely enjoying its many features.

Exercises

7. Consider the twelve clauses that were produced in problem 5, found before Section 3.4.
   a. Give a proof using hyperresolution as the only inference rule.
   b. Give a proof using UR-resolution as the only inference rule.
   c. Give a proof using binary resolution alone that is distinct from both of your preceding two proofs.

8. BNF grammars are occasionally used to describe the structure of sentences. For example,
   \[
   \text{<sentence>} ::= \text{<noun-phrase><verb-phrase>''.''}
   \]
   represents a rule that states that a <sentence> can be formed by a <noun-phrase>, followed by a <verb-phrase>, followed by a period. Similarly,
   \[
   \text{<noun-phrase>} ::= \text{<noun>} \mid \text{<article><noun>}
   \]
describes the possible ways to form a valid <noun-phrase>. Here the vertical bar separates the two alternatives. Thus, a <noun-phrase> can simply be <noun>, or it can be formed by an <article> followed by a <noun>. The possible replacements for <noun> might be given by
   \[
   \text{<noun>} ::= ''\text{cat}'' \mid ''\text{dog}'' \mid ''\text{man}''
   \]
This simple grammar can be completed with the following rules.
   \[
   \text{<article>} ::= ''\text{a}'' \mid ''\text{the}''
   \text{<verb-phrase>} ::= \text{<verb>} \mid \text{<verb><noun-phrase>}
   \text{<verb>} ::= ''\text{walks}'' \mid ''\text{runs}''
   \]
Now consider the sentence “The man walks the dog”. You can show that the sentence can be formed using the rules in this simple grammar.

The man walks the dog. [Given sentence]
   <article> man walks the dog.
   <article> <noun> walks the dog.
   <noun-phrase> walks the dog.
   <noun-phrase> <verb> the dog.
   <noun-phrase> <verb> <article> dog.
   <noun-phrase> <verb> <article> <noun>.
   <noun-phrase> <verb> <noun-phrase>.
Automated Reasoning in Full

<noun-phrase> <verb-phrase>.

Here, each step consists of a single replacement that can be made according to the rules of the grammar. The problem of finding the structure of a sentence can be described in the following way. The notation used here is consistent with the grammar and with the input acceptable to ITP; however, with OTTER, since quoting a string does not have the same effect as it does with ITP, you must replace "." with the word "period".

a. Encode a sentence as a list of symbols. Thus, the short sentence given earlier would be encoded as

\[ l('the', l('man', l('walks', l('the', l('dog', '.')))))) \]

b. Add a clause that indicates that anything that can be rewritten to the symbol "sentence" is a valid sentence.

\[(1) \text{VALID(sentence)}.\]

c. The substitution rules can be specified by equality clauses in the following way.

\[(2) \text{EQUAL}(l(\text{nounphrase}, l(\text{verbphrase}, '.')), \text{sentence}).\]

\[(3) \text{EQUAL}(\text{noun}, \text{nounphrase}).\]

\[(4) \text{EQUAL}(l(\text{article}, l(\text{noun}, x)), l(\text{nounphrase}, x)).\]

\[(5) \text{EQUAL}('cat', \text{noun}).\]

\[(6) \text{EQUAL}('dog', \text{noun}).\]

\[(7) \text{EQUAL}('man', \text{noun}).\]

\[(8) \text{EQUAL}('a', \text{article}).\]

\[(9) \text{EQUAL}('the', \text{article}).\]

\[(10) \text{EQUAL}(\text{verb}, \text{verbphrase}).\]

\[(11) \text{EQUAL}(l(\text{verb}, l(\text{nounphrase}, x)), l(\text{verbphrase}, x)).\]

\[(12) \text{EQUAL}('walks', \text{verb}).\]

\[(13) \text{EQUAL}('runs', \text{verb}).\]

d. The denial of the statement that the sentence is valid is

\[(14) \neg\text{VALID}(l('the', l('man', l('walks', l('the', l('dog', '.'))))))).\]

The problem is now to find a contradiction by showing that

\[\neg\text{VALID(sentence)}.\]

can be deduced from clauses 2 through 14. The inference rule to be used in such a case is paramodulation. First, using paramodulation, give the sequence of clauses that can be deduced to arrive at the contradiction. Next, show how the clause
OTTER and Earlier Automated Theorem-Proving Programs

EQUAL(l(article, l(noun, l(verbphrase, "."))), sentence).

can be deduced using paramodulation.

9. This problem uses the notions developed in problem 8. Here the objective is to formulate the problem of determining whether a sequence of symbols represents a valid arithmetic expression. Suppose that the following expressions are valid.

\[ a + b, (a + b), (a - b), (a - b), +a, (a * a), -b \]

Suppose that the following expressions are not well formed.

\[ a + b, (b), aa, (+ b) \]

Write a simple grammar that describes valid expressions. Formulate in clauses the problem of showing that \((a + b - +a)\) is valid.

10. Can you avoid using paramodulation and simply use demodulation as an inference rule for problems 8 and 9?

11. Twelve clauses were used in the answer to problem 5. According to the recommendations in Section 3.9.1, which of those clauses should be put into the set of support? Which of the proofs given in the answer to problem 7 could be produced using the specified set of support?

12. Both of the last two proofs given in the answer to problem 7 are proofs using binary resolution. Which of these satisfies the requirements of the unit preference strategy?

3.11.1 The Basic Argonne Paradigm

At this point, you may well wonder about the basic paradigm on which OTTER is based and, more generally, about the paradigm that is known in some circles as the Argonne paradigm for the automation of reasoning. Indeed, I have often been asked about the sequence of operations and the implementation of OTTER and about other approaches to obtaining high performance. More recently, with the growing emphasis on parallelism, I have been asked about techniques for implementing a parallel version of the program. In response to these questions, I shall discuss in this section the basic Argonne paradigm for the automation of reasoning. I shall also present two ways of implementing OTTER on multiprocessor computers. To begin, here is a brief overview of the algorithm used by OTTER.

The basic sequential algorithm takes as input two lists of clauses. The first list normally contains clauses that “characterize the general theory”; in particular, the clauses are general statements known to be true. The first list of clauses is called the usable list (formerly, axioms list). A second list of clauses contains the specific
assertions about the theorem under consideration and those that are believed to introduce the contradiction. The objective is to show that the clauses in this second list cannot all be true, given that the clauses in the initial usable list are true. (As you will see, the term “initial usable list” is used because clauses are added to the list during the attempt to complete the given assignment.) This second list of clauses is called the set of support list. Intuitively, in contrast to the usable list, the set of support list can be thought of as the waiting list, for clauses on that list are waiting to be selected as the focus of attention. The usable list is denoted by list(usable) and the set of support list by list(sos). The basic algorithm used to seek a proof by contradiction is given in the following subsection.

### 3.11.2 A Sequential Theorem-Proving Algorithm

While you cannot find two unit clauses that conflict, perform the following steps.

1. Select a clause from list(sos) and call it G; historically such a clause was called the given clause, as it is in the OTTER users manuals, but more often than not it is referred to as the clause in focus or the focal clause. Move G from list(sos) to list(usable).

2. Use each of the user-chosen inference rules to deduce clauses that have G as a parent, along with any other clauses from list(usable). Specifically, G must be one of the parents, and you can use whatever other clauses you like from list(usable). Call the set of deduced clauses “new-clauses”. Then, while new-clauses is not empty,

   a. Select a clause C1 from the new-clauses.
   b. Simplify C1 by demodulating all terms in C1, producing a new clause C2, and discarding C1.
   c. If C2 is subsumed by any clause in list(usable) or in list(sos), discard C2. This process is called forward subsumption.
   d. If C2 is not subsumed, but it is too complex, discard it. The criterion used to delete such clauses is based on the strategy of weighting the clauses and discarding those that are “too heavy”.
   e. If C2 still appears to be worth keeping, then delete all clauses in list(usable) and list(sos) that are subsumed by C2. This process is called back subsumption.
   f. If C2 itself is a new equality clause containing a single literal and if it should become a new demodulator, add the clause to the list of demodulators called list(demodulators) and look for clauses in list(usable), list(sos), and list(demodulators) that can be simplified by C2. For each such clause, (1) delete the clause from the list it is in, (2) demodulate it, and (3) add the demodulated clause to new-clauses. This is called back demodulation.
(g) If $C_2$ has not been deleted, add $C_2$ to list(sos).

The complexity of this algorithm rests with the objective of gradually expanding the set of significant new clauses, while attempting to eliminate any unnecessary redundancy. Such an attempt has proved to be critical to obtaining proofs of many of the more difficult theorems.

Now consider two basic approaches to implementing variations of this algorithm on parallel computers. The goal in both cases is to attain high performance by allowing multiple processors to coordinate activities, and to perform these operations in parallel. Efficient implementation of this algorithm is extremely difficult. The difficulty centers on the fact that almost all of the steps in the algorithm require coordinating access to list(usable) and list(sos), since both lists are updated from several points in the algorithm.

I shall begin the discussion with a description of an approach that will work for parallel computers in which the memory is shared—computers in which a single copy of list(usable) and list(sos) can be maintained and updated. Then, I shall consider an implementation on computers in which the processors do not share memory.

### 3.11.3 Implementing the Algorithm on Shared-Memory Multiprocessors

Several motivations exist for implementing an automated reasoning system on a shared-memory computer. First, sharing of the central data structures of clauses allows immediate and rapid access to clauses during the inference process. Second, the shared-memory model enables rapid synchronization among the processes sharing these structures. Third, and most important, the shared-memory approach facilitates automated load balancing.

The basic idea is to parallelize the outer “while” loop of the sequential algorithm by considering multiple focal (given) clauses simultaneously. For convenience, first consider a simplified version of the algorithm, in which no deletion of clauses by back subsumption or back demodulation takes place. The complete version of the parallel algorithm will be described afterwards.

The difficulty in having multiple processes deduce new clauses and perform forward demodulation and subsumption tests is that two copies of the same clause might enter the permanent clause space, each deduced by a different process. A solution to this difficulty is to use an intermediate list (arbitrarily called K) as a holding area. New clauses that pass steps a, b, c, and d of the algorithm given in the preceding section are put in K, from which they are removed by a single process that repeats steps a–d on the clause before adding it to the clause database. Thus the work to be done is separated into two tasks: A and B. At any given moment, multiple processes will be executing task A, but at most one process will be executing task B. Each instance of task A is associated with a given clause. Task
Fig. 3.1 Flow of data in a simplified parallel theorem-proving algorithm

B is executed only when the set $K$ is nonempty. See Figure 3.1.

Task A (focal clause):
  Generate new clauses
  For each new clause
    rewrite it
      subsumption test
      filter
      If it survives
        lock $K$
        put new clause in $K$
        unlock $K$
      end if
  end for

Task B:
  While $K$ is not empty
    lock $K$
    select a clause from $K$
    unlock $K$
    redo rewrite, subsumption test, filter
    If it survives
      integrate new clause into database
      put new clause in list(sos)
    end if
  end while
Note that no separate process is dedicated to task B. Rather, there is created a uniform pool of processes, each of which performs the following loop.

While it is not time to stop
    If $K$ is nonempty and no process is already doing it
        do Task B
    else
        If list(sos) is not empty
            select a new focal clause from list(sos)
            do Task A (focal clause)
        end if
    end if
end while

The loop continues until some process detects unit conflict or until list(sos) is empty and all processes are waiting for a clause to appear there.

You might think that task B would become a bottleneck, since it is not allowed to share its workload with other processes. It is indeed possible for such a bottleneck to occur, but in most cases it does not. Usually, after an initial start-up period, most newly generated clauses are subsumed by the (parallel) task A and never make it into $K$. Thus $K$ rarely becomes nonempty; when it does, it is quickly emptied by the next available process.

In the complete version of the parallel algorithm, back subsumption tests are done on newly deduced clauses, and new rewrite rules may be deduced during the run. When new rewrite rules are applied to existing clauses, they cause both deletions and new additions. Coping with deletions complicates the algorithm, since the data structures are in continuous use by multiple instances of task A. This problem is solved by having task B handle actual deletions. Back subsumption and back demodulation processes can run in parallel with all other processes. Instead of actually deleting clauses from the database, however, these processes place their identifiers in shared lists, where task B (executed by only one process at a time) can find them and carry out the actual deletions. This algorithm is shown schematically in Figure 3.2.

The parallel algorithm under discussion was implemented in a program called ROO, developed at Argonne National Laboratory and the Australian National University (Lusk and McCune 1992). ROO is completely compatible with OTTER in terms of input files and obtains near-linear speedups on a variety of problems. In some cases, superlinear speedups were achieved, because ROO’s simultaneous consideration of multiple focal clauses substantially reorders the search space.
3.11.4 Implementing the Algorithm on Distributed-Memory Machines

An alternative approach to shared memory is the distributed-memory model. In this section, a fairly straightforward approach is presented to implementing the basic deduction algorithm on distributed-memory multiprocessors, in which processes communicate by sending messages.

This approach has been used successfully to obtain performance improvements of at least a factor of 20 on difficult theorems. However, more sophisticated approaches almost certainly are possible that might well lead to far greater improvements.

The approach presented here was used by A. Jindal in designing PARROT, a parallel implementation based on the sequential OTTER code (Jindal, Overbeek, and Kabat 1992). (Other researchers continue to explore parallelization of deduction strategies; see, for example, (Bonacina 1994) and (Suttner 1999).) The approach is founded on three basic observations.

(1) On long runs, the sequential algorithm often reaches a state in which the vast majority (over 90 percent, and often over 99 percent) of deduced clauses are eliminated fairly rapidly by forward subsumption or weighting. During
some periods of execution, a substantial amount of activity appears to be taking place: new clauses are kept, old ones are back subsumed or back demodulated, and the contents of list(usable) and list(sos) list are expanded and contracted substantially. These “active periods”, however, tend to be overshadowed by long, relatively unproductive periods, which consume the vast majority of the execution time.

(2) Most of the rapid subsumption can be done from clauses in list(usable), although a significant percentage is still subsumed by clauses in list(sos).

(3) The set of support list often becomes quite large relative to list(usable).

Consider an example problem in which a set of processes is started with the configuration presented in Figure 3.3. The top level of processes are the Generators (G1, G2, ...). They maintain local copies of list(usable), but do not have copies of list(sos). The Distributor sends copies of each clause to the Generators as it is moved from list(sos) to list(usable), along with an indication of which Generator is responsible for deducing new clauses from that focal (given) clause. In addition, the Distributor sends commands to delete back-subsumed or back-demodulated clauses, as well as clauses that are to be added as new demodulators. Each Generator passes on a message that flows through to the Distributor every time it completes processing a focal (given) clause, which allows the Distributor to maintain an overview of which Generators need more work.

Each Generator passes on a message that flows through to the Distributor every time it completes processing a focal (given) clause, which allows the Distributor to maintain an overview of which Generators need more work.

The Generators send newly deduced clauses to the Subsumers (S1, S2, ...). Each Subsumer takes input from a set of Generators. It maintains a nonredundant copy of all of the clauses that flow through it. As clauses are received from each Generator, it determines whether the clause is subsumed by another clause that it has already passed on to the Distributor; if not, it passes the clause on to the

---

**Fig. 3.3** Flow of data in a simplified message-passing theorem-proving algorithm
Finally, the Distributor coordinates the entire search. It does this by maintaining its version of list(usable) and list(sos), which is thought of as the “master copy”. It maintains its view of the work assigned to each Generator—an attempt is made to keep work flowing to the Generators so that they do not run out of work. It must perform a final forward-subsumption check on new clauses arriving from the subsumers, perform back subsumption and back demodulation, and check for the completion of a proof.

The details of setting up this process network and coordinating the processes can become complex; however, the basic scheme is fairly straightforward. The weak point in this approach is that the Distributor will certainly become a bottleneck. The real issue is: How much parallelism can be exploited on common problems before the Distributor becomes the limiting factor? Too little research has been done to make an accurate assessment. Research in the future may well center on how to effectively use hundreds of processors in a distributed system. Dramatic improvements are possible: On most problems requiring more than just a few minutes of processing, speedups of at least 20–30 are possible.

3.12 Answers to Exercises

(1) The first set of exercises tests your ability to evaluate expressions using logical operators.

(a) This statement is false because “2 + 4 > 6” is false.
(b) This statement is true. “P or Q” is true when both P and Q are true and, of course, if just one of the two is true.
(c) This statement is true. Since “3 – 2 = 2” is false, the and of it with anything must be false. not of a false statement is true.
(d) The statement is true. Note that, at least here, no attempt is made to determine whether the conclusion must be true because the hypothesis is true. No causal connection is implied.
(e) The statement is true because the hypothesis “1 – 2 = 0” is false.
(f) This statement is also true. When the hypothesis is false, it does not matter whether the conclusion is true.
(g) This statement is true because both sides have the same value, namely, true.
(h) This statement is false.
(i) This statement is true because both sides have the same value, false.

(2) Here are the answers to the question on which translations can be made.

(a) Yes, this translation follows from the fact that not (not (x)) translates to x.
(b) This translation is not permitted, although both expressions evaluate to true. Here, the focus is on translations that do not depend on the truth or falsity of the atomic propositions, where expressions such as “1 + 2 = 3” are called atomic. The stand taken here is quite arbitrary because both expressions represent the same value, namely, true. The intent is merely to give you experience with translations that work, even when you cannot determine the truth or falsity of the atomic propositions.

(c) This translation can be made by using the rule that `\( \text{not} (\text{not} \, x) \)` translates to `x`.

(d) This translation cannot be made.

(e) This translation can be made using the rule that “\( \text{if} \, P \, \text{then} \, Q \)” may be translated to “\( (\text{not} \, P) \, \text{or} \, Q \)”.

(f) This translation can also be made, but it requires two steps. The first is the one given in part e. The second is based on the rule that “\( P \, \text{or} \, Q \)” may be translated to “\( Q \, \text{or} \, P \)”.

(g) This translation can be made. The statement is of the form “\( \text{not} \, (\text{if} \, P \, \text{then} \, Q) \)”. This translates to “\( (\text{not} \, P) \, \text{or} \, Q \)” (which translates to “\( (\text{not} \, (\text{not} \, P)) \, \text{and} \, (\text{not} \, Q) \)”)

(f) Again, although both expressions evaluate to false, the translation cannot be made (independent of knowing how the atomic propositions evaluate).

(i) Here there is no question: The translation cannot be made.

(3) In the following answers, the order of the literals is, of course, irrelevant.

(a) \(-\text{MOTHER(Mary, Sam)} \mid -\text{SISTER(Linda, Mary)} \mid \text{AUNT(Linda, Sam)}\).

(b) The use of variables (always assumed to be universally quantified) produces a clause similar to the preceding, with constants replaced by variables.

\(-\text{MOTHER(x, y)} \mid -\text{SISTER(z, x)} \mid \text{AUNT(z, y)}\).

(c) In this one, do not be misled by the fact that the statement, given the normal interpretation of the symbols, is not true. It still can be translated to

\(-\text{MOTHER(x, y)} \mid \text{SISTER(x, z)} \mid -\text{AUNT(z, y)}\).

which is not always true.

(d) This statement can be represented with two separate clauses.

\(-\text{MOTHER(x, y)} \mid \text{PARENT(x, y)}\).
Automated Reasoning in Full

\[-FATHER(x,y) | \text{PARENT}(x,y)\].

(e) This translates to
\[-\text{POSITIVE}(x) | -\text{NEGATIVE}(x)\].

(4) The following answers all focus on removing an existential quantifier by introducing a function.

(a) \text{GREATERTHAN}(f(x),x).
(b) \text{GREATERTHAN}(y,a).
(c) \text{EQUAL}(f(x,y),\text{sum}(x,y)).

Here the \text{EQUAL} and \text{sum} are the symbols for equality and addition. Any others would be just as acceptable.

(d) \text{-LT}(x,z) | \text{LT}(f(x),z).
(e) \text{GT}(\text{sum}(y,g(y)),a) | \text{EQUAL}(\text{sum}(y,g(y)),a).
(f) In this case, two clauses are produced by the translation.

\text{GT}(\text{sum}(y,g(y)),a).

and

\text{LT}(\text{diff}(y,g(y)),a).

Again, the \text{diff} is just the name that is chosen to represent subtraction.

(5) The following clauses result from the assumptions.

1. \text{-INHOUSE}(x) | \text{CAT}(x).  
2. \text{-GAZER}(x) | \text{SUITABLEPET}(x).  
3. \text{-DETESTED}(x) | \text{AVOIED}(x).  
4. \text{-CARNIVORE}(x) | \text{PROWLER}(x).  
5. \text{-CAT}(x) | \text{MOUSEKILLER}(x).  
6. \text{-TAKESTOME}(x) | \text{INHOUSE}(x).  
7. \text{-GIRAFFE}(x) | -\text{SUITABLEPET}(x).  
8. \text{-MOUSEKILLER}(x) | \text{CARNIVORE}(x).  
9. \text{TAKESTOME}(x) | \text{DETESTED}(x).  
10. \text{-PROWLER}(x) | \text{GAZER}(x).  

The denial of the theorem—the denial of what is to be proved—is represented with two clauses.

11. \text{GIRAFFE}(a).  
12. \text{-AVOIED}(a).  

(6) You can use ISA by simply changing \text{CLASS}(x) to ISA(x,class). Thus, the first clause would be

(1) \text{-ISA}(x,\text{inhouse}) | \text{ISA}(x,\text{cat}).
The differences in the two formulations are just cosmetic.

(7) The three proofs are the following.

(a) By hyperresolution

\[
\text{From clauses 3 and 9:} \\
(13) \ TAKESOME(x) \mid AVOIDED(x).
\]

\[
\text{From clauses 13 and 12:} \\
(14) \ TAKESOME(a).
\]

\[
\text{From clauses 14 and 6:} \\
(15) \ INHOUSE(a).
\]

\[
\text{From clauses 15 and 1:} \\
(16) \ CAT(a).
\]

\[
\text{From clauses 16 and 5:} \\
(17) \ MOUSEKILLER(a).
\]

\[
\text{From clauses 17 and 8:} \\
(18) \ CARNIVORE(a).
\]

\[
\text{From clauses 18 and 4:} \\
(19) \ PROWLER(a).
\]

\[
\text{From clauses 19 and 10:} \\
(20) \ GAZER(a).
\]

\[
\text{From clauses 20 and 2:} \\
(21) \ SUITABLEPET(a).
\]

Now clauses 21, 7, and 11 taken together are contradictory. Specifically, the empty clause (see Section 3.5) can be deduced from these three clauses by using hyperresolution.

(b) With UR-resolution, the following proof can be obtained.

\[
\text{From clauses 12 and 3:} \\
(13) \ \neg\text{DETESTED}(a).
\]

\[
\text{From clauses 13 and 9:} \\
(14) \ TAKESOME(a).
\]

The rest of the proof is similar to the proof by hyperresolution. The only difference is that

\[
\text{From clauses 21 and 7:} \\
(22) \ \neg\text{GIRAFFE}(a).
\]

\[
\text{can be deduced, which, with clause 11, gives unit conflict and the desired contradiction.}
\]

(c) With binary resolution, the following is a proof.

\[
\text{From clauses 1 and 5:} \\
(13) \ \neg\text{INHOUSE}(x) \mid \text{MOUSEKILLER}(x).
\]

\[
\text{From clauses 13 and 8:}
\]
(14) \(-\text{INHOUSE}(x) \mid \text{CARNIVORE}(x)\).
From clauses 14 and 6:
(15) \(-\text{TAKESTOME}(x) \mid \text{CARNIVORE}(x)\).
From clauses 9 and 3:
(16) \text{TAKESTOME}(x) \mid \text{AVOIDED}(x).
From clauses 15 and 16:
(17) \text{CARNIVORE}(x) \mid \text{AVOIDED}(x).
From clauses 4 and 10:
(18) \text{GAZER}(x) \mid \neg\text{CARNIVORE}(x).
From clauses 17 and 18:
(19) \text{GAZER}(x) \mid \text{AVOIDED}(x).
From clauses 19 and 12:
(20) \text{GAZER}(a).
From clauses 20 and 2:
(21) \text{SUITABLEPET}(a).
From clauses 21 and 7:
(22) \neg\text{GIRAFFE}(a).

Here clauses 22 and 11 are contradictory. Of course, there are many different ways to construct such a proof using binary resolution.

Note that none of these proofs make use of the fact that a cat is not a giraffe. But that was not given to you in the puzzle!

(8) The proof proceeds exactly as does the nine-step derivation given in the statement of the problem. For example, the first deduced clause is

\(\neg\text{VALID}(\text{l(article},1(\text{‘‘man’’},1(\text{‘‘walks’’},1(\text{‘‘the’’}, 1(\text{‘dog’’},1(\dots)))))).\)

and is obtained by paramodulating from clause 9 into clause 14. The remaining steps are left to you to see that the derivation works.

Note that the concept of equality is being used in a somewhat strange way. Essentially, equality is used here as an equivalence relation. All derivable sentences are “equal” under this interpretation of the symbol. Similarly, “a” and “the” are equal, since they are both equal to article. The clause

\(\text{EQUAL}(\text{l(article},1(\text{noun},1(\text{verbphrase},1(\dots))))),\text{sentence}).\)

can be deduced by paramodulating from clause 2 into clause 4.

(9) Several possible grammars would be considered correct. The one chosen is

\[
\begin{align*}
<\text{expr}> & ::= <\text{expr}><\text{bop}><\text{expr}> | <\text{uop}><\text{expr}> \\
& \quad | \ <\text{letter}> \ | \ ‘’(\langle<\text{expr}>\langle‘’\rangle\rangle, ‘’)’. \\
<\text{letter}> & ::= ‘’a’’ | ‘’b’’ \\
<\text{bop}> & ::= ‘’+’’ | ‘’-’’ | ‘’*’’ \\
<\text{uop}> & ::= ‘’+’’ | ‘’-’’
\end{align*}
\]
The clauses to represent the given problem are simply

(1) VALID(l(expr,end)).
(2) EQUAL(l(expr,l(bop,l(expr,x))),l(expr,x)).
(3) EQUAL(l(uop,l(expr,x))),l(expr,x)).
(4) EQUAL(letter,expr).
(5) EQUAL(l(''(',l(expr,l('')''),x))),l(expr,x)).
(6) EQUAL(''a'',letter).
(7) EQUAL(''b'',letter).
(8) EQUAL(''++'',bop).
(9) EQUAL(''--'',bop).
(10) EQUAL(''*''',bop).
(11) EQUAL(''+''',uop).
(12) EQUAL(''-''',uop).
(13) -VALID(l(''(',l(''a'',l(''*'',l(''b'',l(''--'',l(''++'',1('')''),end))))))).

Notice the use of “end” to mark the end of the list. Why is it used?

(10) No. Demodulation discards the into parent. Hence, if there are multiple ways to rewrite a term, only one will be chosen. In each of the problems, there are points where multiple rewrites could occur, and each of the possibilities must be considered. Demodulation should be restricted (normally) to just simplification and normalization. However, in some places, it is used to perform well-defined computations such as counting symbols or sorting a list.

(11) In this problem, clause 11 is the “special” hypothesis, and clause 12 denies the conclusion. Hence, the set of support would normally contain either clauses 11 and 12, or just clause 12. The proof by UR-resolution is the only one that conforms to the set of support strategy.

(12) The proof by UR-resolution conforms to the unit preference strategy, but the third proof does not.