CHAPTER 4
Resolution

IN THIS CHAPTER we describe an inference procedure based on a simple yet extremely powerful rule of inference known as the *resolution principle.* Because it uses just one rule of inference, the procedure is simple to analyze and implement; yet it is known to be both sound and, in a sense, complete. Section 4.1 introduces the variant of predicate calculus used by resolution; Section 4.2 defines the critical concept of unification, and Section 4.3 describes the resolution principle itself. Section 4.4 introduces the resolution procedure. Section 4.5 shows how the procedure can be used in determining satisfiability, Section 4.6 shows how it can be used in answering true-or-false questions, and Section 4.7 shows how it can be used to answer fill-in-the-blank questions. Sections 4.8 and 4.9 offer examples. Section 4.10 discusses the issues of soundness and completeness. The final section shows how resolution can be used in proving results from statements about equality.

4.1 Clausal Form

The resolution procedure takes as argument a set of expressions in a simplified version of predicate calculus, called *clausal form.* The symbols, terms, and atomic sentences of clausal form are the same as those in ordinary predicate calculus. Instead of logical and quantified sentences, however, clausal form has literals and clauses.

```
Procedure Convert (x)
1     Begin x <- Implications_out(x),
2           x <- Negations_in(x),
3           x <- Standardize_variables(x),
4           x <- Existentials_out(x),
5           x <- Universals_out(x),
6           x <- Disjunctions_in(x),
7           x <- Operators_out(x),
8           x <- Rename_variables(x)
      End
```

Figure 4.1 Conversion to clausal form.

A *literal* is an atomic sentence or the negation of an atomic sentence. An atomic sentence is a *positive* literal, and the negation of an atomic sentence is a *negative* literal.

A *clause* is a set of literals representing their disjunction. For example, the sets {On(A,B)} and {¬On(A,B),Above(A,B)} are both clauses. The first states that the block named A is on the block named B. The second clause states that either A is not on B or it is above B. A *Horn clause* is a clause with at most one positive literal.

At first glance, clausal form may appear very restrictive, but this is illusory. For any sentence in predicate calculus, there is a set of clauses that is equivalent to the original sentence in that the sentence is satisfiable if and only if the corresponding set of clauses is satisfiable. The procedure defined in Figure 4.1 sketches a method for converting an arbitrary closed sentence into its clausal form.

In the first step, we eliminate all occurrences of the $\Rightarrow$, $\Leftarrow$, and $\Leftrightarrow$ operators by substituting equivalent sentences involving only the $\neg$, $\wedge$, and $\vee$ operators.

- $\phi \Rightarrow \psi$ is replaced by $\neg\phi \vee \psi$.
- $\phi \Leftarrow \psi$ is replaced by $\phi \vee \neg\psi$.
- $\phi \Leftrightarrow \psi$ is replaced by $(\neg\phi \vee \psi) \wedge (\phi \vee \neg\psi)$.

In the second step, negations are distributed over other logical operators until each such operator applies to a single atomic sentence. The following replacement rules do the job.

- $\neg\neg\phi$ is replaced by ϕ.
- $\neg(\phi \wedge \psi)$ is replaced by $\neg\phi \vee \neg\psi$.
- $\neg(\phi \vee \psi)$ is replaced by $\neg\phi \wedge \neg\psi$.

- $\neg\forall\nu\ \phi$ is replaced by $\exists\nu\ \neg\phi$.
- $\neg\exists\nu\ \phi$ is replaced by $\forall\nu\ \neg\phi$.

In the third step, we rename variables so that each quantifier has a unique variable; i.e., the same variable is not quantified more than once within the same sentence. For example, we can replace the formula `(∀x P(x,x)) ∧ (∃x Q(x))` by `(∀x P(x,x)) ∧ (∃y Q(y))`.

In the fourth step, we eliminate all existential quantifiers. The method for doing this is a little complicated, and we describe it in two stages.

If an existential quantifier does not occur within the scope of a universal quantifier, we simply drop the quantifier and replace all occurrences of the quantified variable by a new constant; i.e., one that does not occur anywhere else in our database. Thus, if we have never before used the object constant `A`, we can replace `∃x P(x)` by `P(A)`. The constant used to replace the existential variable in this case is called a *Skolem constant.*

If an existential quantifier is within the scope of any universal quantifiers, there is the possibility that the value of the existential variable depends on the values of the associated universal variables. Consequently, we cannot replace the existential variable with a constant. Instead, the general rule is to drop the existential quantifier and to replace the associated variable by a term formed from a new function symbol applied to the variables associated with the enclosing universal quantifiers. For example, if `F` is a new function symbol, we can replace `∀x∀y∃z P(x,y,z)` with the sentence `∀x∀y P(x,y,F(x,y))`. Any function defined in this way is called a *Skolem function.*

In the fifth step, we drop all universal quantifiers. Because the remaining variables at this point are universally quantified, this does not introduce any ambiguities.

In the sixth step, we put the expression into *conjunctive normal form*; i.e., a conjunction of disjunctions of literals. This can be accomplished by repeated use of the following rule.

- $\phi \vee (\psi \wedge \chi)$ is replaced by $(\phi \vee \psi) \wedge (\phi \vee \chi)$.

In the seventh step, we eliminate operators by writing the conjunction obtained in the sixth step as a set of clauses. For example, we replace the sentence `P ∧ (Q ∨ R)` with the set consisting of the singleton clause `{P}` and the binary clause `{Q,R}`.

In the final step, we rename variables so that no variable appears in more than one clause. This process is called *standardizing the variables apart.*

As an example of this conversion process, consider the problem of transforming the following expression to clausal form. The initial expression appears on the top line, and the expressions on the numbered lines are the results of the corresponding steps of the conversion procedure.

initial: `∀x (∀y P(x,y)) ⇒ ¬(∀y Q(x,y) ⇒ R(x,y))`
step 1: `∀x ¬(∀y P(x,y)) ∨ ¬(∀y ¬Q(x,y) ∨ R(x,y))`

```
step 2: ∀x (∃y ¬P(x,y)) ∨ (∃y Q(x,y) ∧ ¬R(x,y))
step 3: ∀x (∃y ¬P(x,y)) ∨ (∃z Q(x,z) ∧ ¬R(x,z))
step 4: ∀x ¬P(x,F1(x)) ∨ (Q(x,F2(x)) ∧ ¬R(x,F2(x)))
step 5: ¬P(x,F1(x)) ∨ (Q(x,F2(x)) ∧ ¬R(x,F2(x)))
step 6: (¬P(x,F1(x)) ∨ Q(x,F2(x))) ∧
        (¬P(x,F1(x)) ∨ ¬R(x,F2(x)))
step 7: {¬P(x,F1(x)), Q(x,F2(x))}
        {¬P(x,F1(x)), ¬R(x,F2(x))}
step 8: {¬P(x1,F1(x1)), Q(x1,F2(x1))}
        {¬P(x2,F1(x2)), ¬R(x2,F2(x2))}
```

4.2 Unification

Unification is the process of determining whether two expressions can be made identical by appropriate substitutions for their variables. As we shall see, making this determination is an essential part of resolution.

A *substitution* is any finite set of associations between variables and expressions in which (1) each variable is associated with at most one expression, and (2) no variable with an associated expression occurs within any of the associated expressions. For example, the following set of pairs is a substitution in which the variable `x` is associated with the symbol `A`, the variable `y` is associated with the term `F(B)`, and the variable `z` is associated with the variable `w`.

$$\{x/A, y/F(B), z/w\}$$

Each variable has at most one associated expression, and no variable with an associated expression occurs within any of the associated expressions.

By contrast, the following set of pairs is not a substitution.

$$\{x/G(y), y/F(x)\}$$

The variable `x`, which is associated with `G(y)`, occurs in the expression `F(x)` associated with `y`; the variable `y` occurs in the expression `G(y)` associated with `x`.

We often speak of the terms associated with the variables in a substitution as *bindings* for those variables; the substitution itself is called a *binding list*; and the variables with bindings are said to be *bound*.

A substitution can be *applied* to a predicate-calculus expression to produce a new expression (called a *substitution instance*) by replacing all bound variables in the expression by their bindings. Variables without bindings are left unchanged. In contrast to the usual functional notation, it is customary to write $\phi\sigma$ to denote the substitution instance obtained by applying the substitution σ to the expression ϕ. For example, applying the preceding legitimate substitution to the expression on the left in the

following equation results in the expression shown on the right. Note that both occurrences of the variable x are replaced by A and that variable v, having no associated expression, is simply left alone.

$$\mathtt{P(x,x,y,v)\{x/A,y/F(B),z/w\}} = \mathtt{P(A,A,F(B),v)}$$

A substitution τ is *distinct* from a substitution σ if and only if no variable bound in σ occurs anywhere in τ (although variables with bindings in τ may occur in σ). Now, consider a substitution σ and a distinct substitution τ. The *composition* of τ with σ (again, written backward as $\sigma\tau$) is the substitution obtained by applying τ to the terms of σ and then adding to σ the bindings from τ. In the following example, the bindings for x and y are plugged into the binding for w in the first substitution, and then the bindings from the second substitution are added to the resulting set of associations.

$$\mathtt{\{w/G(x,y)\}\{x/A,y/B,z/C\}} = \mathtt{\{w/G(A,B),x/A,y/B,z/C\}}$$

A set of expressions $\{\phi_1,\ldots,\phi_n\}$ is *unifiable* if and only if there is a substitution σ that makes the expressions identical; i.e., $\phi_1\sigma = \cdots = \phi_n\sigma$. In such a case, σ is said to be a *unifier* for the set. For example, the substitution {x/A,y/B,z/C} unifies the expression P(A,y,z) and the expression P(x,B,z) to yield P(A,B,C).

$$\mathtt{P(A,y,z)\{x/A,y/B,z/C\}} = \mathtt{P(A,B,C)} = \mathtt{P(x,B,z)\{x/A,y/B,z/C\}}$$

Although this substitution unifies the two expressions, it is not the only unifier. We do not have to substitute C for z to unify the two expressions. We can equally well substitute D or F(C) or F(w). In fact, we can unify the expressions without changing z at all. In looking at these alternatives, it is worth noting that some substitutions are more general than others are; e.g., the substitution {z/F(w)} is more general than {z/F(C)} is. We say that a substitution σ is as general as or more general than a substitution τ if and only if there is another substitution δ such that $\sigma\delta = \tau$. It is interesting to consider unifiers with maximum generality. A *most general unifier*, or *mgu*, γ of ϕ and ψ has the property that, if σ is any unifier of the two expressions, then there exists a substitution δ with the following property.

$$\phi\gamma\delta = \phi\sigma = \psi\sigma$$

An important property of any most general unifier is that it is unique up to variable renaming. The substitution {x/A} is a most general unifier for the expressions P(A,y,z) and P(x,y,z). The less general unifier, {x/A,y/B,z/C}, can be obtained by composing the most general one with the substitution {y/B,z/C}. Because of this property, we often speak of *the* most general unifier of two expressions.

```
Recursive Procedure Mgu (x,y)
    Begin x=y ==> Return(),
          Variable(x) ==> Return(Mguvar(x,y)),
          Variable(y) ==> Return(Mguvar(y,x)),
          Constant(x) or Constant(y) ==> Return(False),
          Not(Length(x)=Length(y)) ==> Return(False),
          Begin i <- 0,
                g <- [],
          Tag   i=Length(x) ==> Return(g),
                s <- Mgu(Part(x,i),Part(y,i))
                s=False ==> Return(False),
                g <- Compose(g,s),
                x <- Substitute(x,g),
                y <- Substitute(y,g),
                i <- i + 1,
                Goto Tag
          End
     End

    Procedure Mguvar (x,y)
        Begin Includes(x,y) ==> Return(False),
              Return([x/y])
        End
```

Figure 4.2 Procedure for computing the most general unifier.

Figure 4.2 presents a simple recursive procedure for computing the most general unifier of two expressions. If two expressions are unifiable, the procedure returns the most general unifier. Otherwise, it returns `False`.

The procedure assumes that an expression is a constant, a variable, or a structured object. The predicate `Variable` is true of variables, and the predicate `Constant` is true of constants. A structured object consists of a function constant, relation constant, or operator and some number of arguments. The `Length` of a structured object is equal to the number of arguments. The top-level function constant, relation constant, or operator in a structured object is its zeroth `Part`, and the arguments are the other parts. For example, the expression `F(A,G(y))` can be represented as a structured object of length 2. The zeroth part is the constant `F`, the first part is the constant `A`, and the second part is the term `G(y)`.

The definition uses several subroutines that are undefined in Figure 4.2. `Substitute` takes as argument an expression and a substitution represented as a set of bindings and returns the expression that results from applying

the substitution to the expression. `Compose` takes as argument two substitutions and returns their composition. The predicate `Includes` takes as argument a variable and an expression and returns `True` if and only if the variable is contained in the expression.

The use of `Includes` in `Mguvar` is called an *occur check*, since it is used to check whether or not the variable occurs within the term with which it is being unified. Without this check, the algorithm would find that expressions such as `P(x)` and `P(F(x))` are unifiable, even though there is no substitution for `x` that could ever make them look alike.

4.3 Resolution Principle

The idea of resolution is simple. If we know that `P` is true or `Q` is true and we also know that `P` is false or `R` is true, then it must be the case that `Q` is true or `R` is true. The general definition is a little complicated, and we introduce it in three stages.

Resolution without regard to variables is the simplest case. Given a clause containing a literal ϕ and another clause containing the literal $\neg\phi$, we can infer the clause consisting of all the literals of both clauses without the complementary pair.

$$\frac{\begin{array}{ll}\Phi & \text{with } \phi \in \Phi \\ \Psi & \text{with } \neg\phi \in \Psi\end{array}}{(\Phi - \{\phi\}) \cup (\Psi - \{\neg\phi\})}$$

As an example, consider the following deduction. The first premise asserts that either `P` or `Q` is true. The second premise states that either `P` is false or `R` is true. From these premises, we can infer by resolution that either `Q` is true or `R` is true. The Δ notation on the right indicates that the associated clauses are in the initial database, and the numbers indicate the clauses from which the associated clause is derived.

1. {P,Q} Δ
2. {¬P,R} Δ
3. {Q,R} 1, 2

Since clauses are sets, there cannot be two occurrences of any literal in a clause. Therefore, in drawing a conclusion from two clauses that share a literal, we merge the two occurrences into one, as in the following example.

1. {P,Q} Δ
2. {¬P,Q} Δ
3. {Q} 1, 2

If either of the clauses is a singleton set, we see that the number of literals in the result is less than the number of literals in the other clause. From the clause {¬P,Q} and the singleton clause {P}, we can derive the singleton clause {Q}. Note the correspondence between this deduction and that of modus ponens, illustrated on the right.

1.	{¬P,Q}	Δ	1. P ⇒ Q	Δ
2.	{P}	Δ	2. P	Δ
3.	{Q}	1, 2	3. Q	1, 2

Resolving two singleton clauses leads to the *empty clause*; i.e., the clause consisting of no literals at all, as shown below. The derivation of the empty clause means that the database contains a contradiction.

1.	{P}	Δ
2.	{¬P}	Δ
3.	{}	1, 2

Unfortunately, our simple definition of resolution is too simple. It provides no way to instantiate variables. Fortunately, we can solve this problem by redefining the resolution principle using the notion of unification.

Suppose that Φ and Ψ are two clauses. If there is a literal ϕ in Φ and a literal $\neg\psi$ in Ψ such that ϕ and ψ have a most general unifier γ, then we can infer the clause obtained by applying the substitution γ to the union of Φ and Ψ minus the complementary literals.

$$\frac{\begin{array}{ll}\Phi & \text{with } \phi \in \Phi \\ \Psi & \text{with } \neg\psi \in \Psi\end{array}}{((\Phi - \{\phi\}) \cup (\Psi - \{\neg\psi\}))\gamma \quad \text{where } \phi\gamma = \psi\gamma}$$

The following deduction illustrates the use of unification in applying the resolution rule. In this case, the first disjunct of the first sentence unifies with the negation of the first disjunct of the second sentence, with mgu {x/A}.

1.	{P(x),Q(x,y)}	Δ
2.	{¬P(A),R(B,z)}	Δ
3.	{Q(A,y),R(B,z)}	1, 2

If two clauses resolve, they may have more than one resolvent because there may be more than one way in which to choose ϕ and ψ. Consider the following deductions. In the first, ϕ = P(x,x) and ψ = P(A,z), and the mgu is {x/A},{z/A}. In the second, ϕ = Q(x) and ψ = Q(B), and the mgu is {x/B}. Fortunately, two sentences can have at most a finite number of resolvents.

1.	{P(x,x),Q(x),R(x)}	Δ
2.	{¬P(A,z),¬Q(B)}	Δ
3.	{Q(A),R(A),¬Q(B)}	1, 2
4.	{P(B,B),R(B),¬P(A,z)}	1, 2

Unfortunately, even this definition is not quite enough. For example, given the clauses {P(u),P(v)} and {¬P(x),¬P(y)}, we should be able to infer the empty clause {}—i.e., a contradiction—and this is impossible with the preceding definition. Fortunately, we can solve this problem with one final modification to our definition.

If a subset of the literals in a clause Φ has a most general unifier γ, then the clause Φ' obtained by applying γ to Φ is called a *factor* of Φ. For example, the literals P(x) and P(F(y)) have a most general unifier {x/F(y)}, so the clause {P(F(y)),R(F(y),y)} is a factor of {P(x),P(F(y)),R(x,y)}. Obviously, any clause is a trivial factor of itself.

Using the notion of factors, we can give our official definition for the *resolution principle.* Suppose that Φ and Ψ are two clauses. If there is a literal ϕ in some factor Φ' of Φ and a literal $\neg\psi$ in some factor Ψ' of Ψ such that ϕ and ψ have a most general unifier γ, then we say that the two clauses Φ and Ψ *resolve* and that the new clause, $((\Phi' - \{\phi\}) \cup (\Psi' - \{\neg\psi\}))\gamma$, is a *resolvent* of the two clauses.

$$\begin{array}{ll} \Phi & \text{with } \phi \in \Phi' \\ \Psi & \text{with } \neg\psi \in \Psi' \\ \hline ((\Phi' - \{\phi\}) \cup (\Psi' - \{\neg\psi\}))\gamma & \text{where } \phi\gamma = \psi\gamma \end{array}$$

Standardizing variables apart can be interpreted as a trivial application of factoring. In particular, our definition allows us to rename the variables in one clause so that there are no conflicts with the variables in another clause. Situations in which there are nontrivial factors are extremely rare in practice, and none of the clauses in our subsequent examples contain any nontrivial factors. Consequently, except for variable renaming, we ignore factors in the remainder of our discussion.

4.4 Resolution

A *resolution deduction* of a clause Φ from a database Δ is a sequence of clauses in which (1) Φ is an element of the sequence, and (2) each element is either a member of Δ or the result of applying the resolution principle to clauses earlier in the sequence.

For example, the following sequence of clauses is a resolution deduction of the empty clause from the set of clauses labeled Δ. The clause in line 5 is derived from the clauses in lines 1 and 2; the clause in line 6 is derived

from the clauses in lines 3 and 4; and the conclusion (line 7) is derived by resolving these two conclusions (lines 5 and 6) with each other.

1.	{P}	Δ
2.	{¬P,Q}	Δ
3.	{¬Q,R}	Δ
4.	{¬R}	Δ
5.	{Q}	1, 2
6.	{¬Q}	3, 4
7.	{}	5, 6

Figure 4.3 outlines a nondeterministic procedure for resolution. There is a termination condition in the first line that varies from use to use. The next few sections describe several uses with different termination conditions. If the termination condition is not satisfied, the procedure selects clauses `Phi` and `Psi`, adds their resolvents to the clause set `Delta`, and repeats. The `Resolvents` subroutine is assumed to compute all the resolvents of the two clauses and to standardize their variables apart from those in the rest of the database; e.g., by using new variable names.

This procedure could be used to generate the previous resolution deduction. In this case, we made the right choices for `Phi` and `Psi` at each point, but we might just as well have chosen other resolutions. Figure 4.4 shows the graph of possible resolutions from the initial database, expanded out to three levels of deduction. A graph of this sort is called a *resolution graph.*

One of the problems with inference graphs such as the one in Figure 4.4 is that they are difficult to lay out in two dimensions. Fortunately, we can encode such graphs in linear form. A *resolution trace* is a sequence of annotated clauses separated into *levels.* The first level contains just the clauses in the initial database. Each subsequent level contains all clauses with at least one parent at the previous level. As with proofs, the annotations specify the clauses from which they are derived. For example, the following resolution trace captures the information from the resolution graph in Figure 4.4.

1.	{P}	Δ
2.	{¬P,Q}	Δ
3.	{¬Q,R}	Δ
4.	{¬R}	Δ
5.	{Q}	1, 2
6.	{¬P,R}	2, 3
7.	{¬Q}	3, 4
8.	{R}	3, 5
9.	{R}	1, 6
10.	{¬P}	4, 6

```
Procedure Resolution (Delta)
    Repeat Termination(Delta) ==> Return(Success),
           Phi <- Choose(Delta), Psi <- Choose(Delta),
           Chi <- Choose(Resolvents(Phi,Psi)),
           Delta <- Concatenate(Delta,[Chi])
    End
```

Figure 4.3 The resolution procedure.

11. $\{\neg P\}$ 2, 7
12. $\{\}$ 5, 7

We can generate resolution traces mechanically as follows. We store the database as a list of clauses, with two pointers initialized to the head of the list. We let the first pointer range over the list until it reaches the second pointer, after which the first pointer is reinitialized to the head of the list and the second pointer is advanced to the next element in the list. For each combination of pointers, we compute the resolvents of the corresponding clauses and add them to the end of the list. This procedure in effect searches the inference graph in a breadth-first fashion.

Although it is not part of the definition of resolution, it is common to augment resolution procedures (or indeed any deduction procedure) with

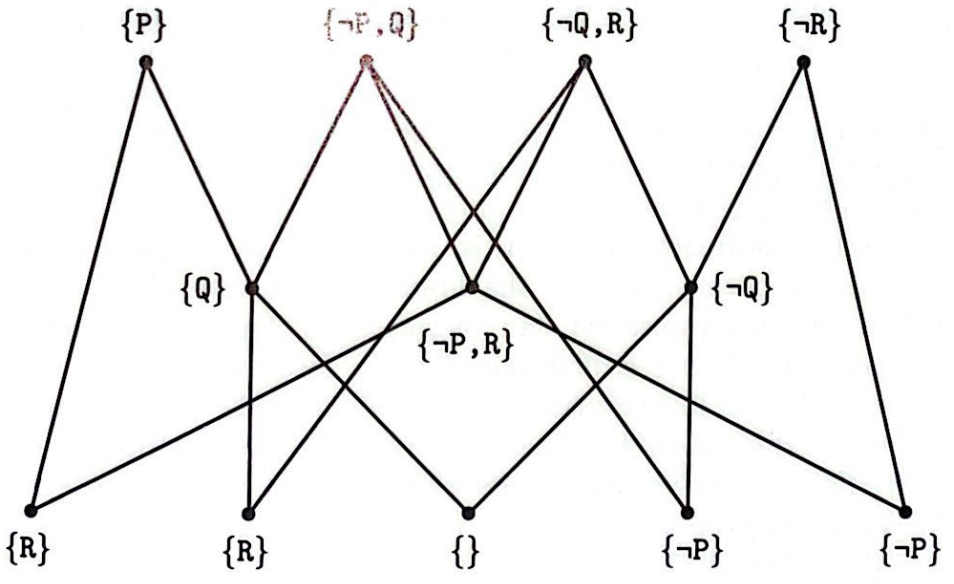

Figure 4.4 Three-level resolution graph.

various instances of *procedural attachment.* This is especially useful when the machine running the procedure has special programs for evaluating the truth of certain literals under their standard interpretations. Typically, evaluations are performed for ground instances. For example, if the predicate symbol > stands for the *greater than* relation between numbers, it is a simple matter to evaluate ground instances such as 7>3 when they occur, whereas we probably would not want to include in the base set a table of numbers that satisfy this relation.

It is instructive to look more closely at what is meant by "evaluating" an expression such as 7>3. Predicate-calculus expressions are linguistic constructs that denote objects, functions, or relations in a domain. Such expressions can be interpreted with reference to a model that associates linguistic entities with appropriate domain entities.

Given a model, we can use any finite processes for interpretation with respect to it as a way of deciding the truth or falsity of sentences. Unfortunately, models and interpretation processes are not, in general, finite—but we often can use partial models. In our inequality example, we can associate with the predicate symbol > a computer program that compares numbers within the finite domain of the program. Let us call this program `Greaterp`. We say that the program `Greaterp` is *attached* to the predicate symbol >. We can associate the linguistic symbols 7 and 3 (i.e., numerals) with the computer data objects 7 and 3, respectively. We say that 7 is attached to 7 and that 3 is attached to 3, and the computer program and arguments represented by `Greaterp(7,3)` are attached to the linguistic expression 7>3. Then we can run the program to determine that 7 is indeed greater than 3.

We also can attach procedures to function symbols. For example, an addition program can be attached to the function symbol +. In this manner, we can establish a connection or procedural attachment between executable computer code and some of the linguistic expressions in our predicate-calculus language. Evaluation of attached procedures can be thought of as a process of interpretation with respect to a partial model. When it can be used, procedural attachment reduces the search effort that would otherwise be required to prove theorems.

A literal is evaluated when it is interpreted by running attached procedures. Typically, not all the literals in a set of clauses can be evaluated, but the clause set can, nevertheless, be simplified by such evaluations. If a literal is determined to be false, then the occurrence of just that literal in the clause can be eliminated. If a literal in a clause is determined to be true, the entire clause can be eliminated without affecting the unsatisfiability of the rest of the set. The clause {P(x),Q(x),7<3} can be replaced by {P(x),Q(x)}, since 7<3 is false. The clause {P(x),Q(x),7>3} can be eliminated, since the literal 7>3 is true. Attachment of linguistic objects to semantic elements is an important idea with general application in AI.

4.5 Unsatisfiability

The simplest use of resolution is in demonstrating unsatisfiability. If a set of clauses is unsatisfiable, then it is always possible by resolution to derive a contradiction from the clauses in the set. In clausal form, a contradiction takes the form of the empty clause, which is equivalent to a disjunction of no literals. Thus, to automate the determination of unsatisfiablity, all we need do is to use resolution to derive consequences from the set to be tested, terminating whenever the empty clause is generated.

The derivation presented in the Section 4.4 is a good example of using resolution to demonstrate unsatisfiability. Since resolution generates the empty clause, the initial set is unsatisfiable.

Demonstrating that a set of clauses is unsatisfiable can also be used to demonstrate that a formula is logically implied by a set of formulas. Suppose we wish to show that the set of formulas Δ logically implies the formula ψ. We can do this by finding a proof of ψ from Δ; i.e., by establishing $\Delta \vdash \psi$. By the refutation theorem (Chapter 3), we can establish $\Delta \vdash \psi$ by showing that $\Delta \cup \{\neg\psi\}$ is inconsistent (unsatisfiable). Thus, if we show that the set of formulas $\Delta \cup \{\neg\psi\}$ is unsatisfiable, we have demonstrated that Δ logically implies ψ.

Let us look at this technique from the standpoint of models. If $\Delta \models \psi$, then all the models of Δ also are models of ψ. Hence, none of these can be models of $\neg\psi$, and thus $\Delta \cup \neg\psi$ is unsatisfiable. Conversely, suppose $\Delta \cup \neg\psi$ is unsatisfiable but that Δ is satisfiable. Let I be an interpretation that satisfies Δ; I does not satisfy $\neg\psi$, because, if it did, $\Delta \cup \neg\psi$ would be satisfiable. Therefore, I satisfies ψ. (An interpretation must satisfy one of either ψ or $\neg\psi$.) Since this holds for arbitrary I satisfying Δ, it holds for all I satisfying Δ. Thus, all models of Δ are also models of ψ, and Δ logically implies ψ.

To apply this technique of establishing logical implication by establishing unsatisfiability using resolution, we first negate ψ and add it to Δ to yield Δ'. We then convert Δ' to clausal form and apply resolution. If the empty clause is produced, the original Δ' was unsatisfiable, and we have demonstrated that Δ logically entails ψ. This process is called a *resolution refutation*; it is illustrated by examples in the following sections.

4.6 True-or-False Questions

One application of proving logical implication through resolution refutation is in answering true-or-false questions. As an example, consider the following resolution trace. The database includes the facts that Art is the father of Jon, that Bob is the father of Kim, and that fathers are parents. To prove that Art is a parent of Jon, we negate the formula representing this fact to get clause 4, which states that Art is not a parent of Jon. The Γ notation indicates that the associated clause is derived from the negated