3.6 Bibliographical and Historical Remarks

The axiom schemata introduced in this chapter are rather standard and follow the treatment by Enderton [Enderton 1972]. The equivalence of provability and logical implication was first proved by Gödel [Gödel 1930]; proofs appear in textbooks on logic. The incompleteness of any finite axiomatization of arithmetic also was proved by Gödel [Gödel 1931]. Although this result is extremely important in mathematical logic, it does not (as some people have claimed [Lucas 1961]) preclude the possibility that machines will be able to reason as well as people. People cannot prove the consistency of complex systems in this way either!

Exercises

1. *Derivability.* The law says that it is a crime to sell an unregistered gun. Red has several unregistered guns, and all of them were purchased from Lefty. Using the rules of inference given in this text, derive the conclusion that Lefty is a criminal.

2. *Inference procedures.* Define an inference procedure based on modus ponens in which the search is done in depth-first fashion.

3. *Distinctions and confusions.* Distinguish the following three statements.

   a. $P \Rightarrow Q$

   b. $P \models Q$

   c. $P \vdash Q$

4. *Proofs.* Give a formal proof of the sentence $\forall x\ P(x) \Rightarrow R(x)$ from the premises $\forall x\ P(x) \Rightarrow Q(x)$ and $\forall x\ Q(x) \Rightarrow R(x)$. Note that the generalization theorem does not solve this problem. We need to use a generalized axiom schema.

5. *Substitution.* Show that, if it is possible to prove $\phi \Leftrightarrow \psi$, then it is possible to prove $\chi \Leftrightarrow \chi_{\phi/\psi}$, where $\chi_{\phi/\psi}$ is the sentence obtained by substituting ψ for ϕ in χ.

6. *Generalization on constants.* Prove that, if $\Delta \vdash \phi$ and α is an object constant that occurs in ϕ but not in Δ, then $\Delta \vdash \forall \nu\ \phi_{\alpha/\nu}$, where ν is a variable that does not occur in Δ or ϕ and where $\phi_{\alpha/\nu}$ is the expression resulting from the consistent replacement of α by ν in ϕ.

7. *Existential instantiation.* Prove that, if the object constant α does not occur in ψ or Δ and it is possible to prove ψ from Δ and ϕ, then it is possible to prove ψ from Δ and $\exists \nu\ \phi_{\alpha/\nu}$. Hint: Use Exercise 6.

NILSON
logical Journal of AI

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)
    Begin x <- Implications_out(x),
          x <- Negations_in(x),
          x <- Standardize_variables(x),
          x <- Existentials_out(x),
          x <- Universals_out(x),
          x <- Disjunctions_in(x),
          x <- Operators_out(x),
          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 ⇒, ⇐, and ⇔ operators by substituting equivalent sentences involving only the ¬, ∧, and ∨ 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

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.

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

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.

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

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).

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

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*

```
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.

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

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.

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

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.	{¬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

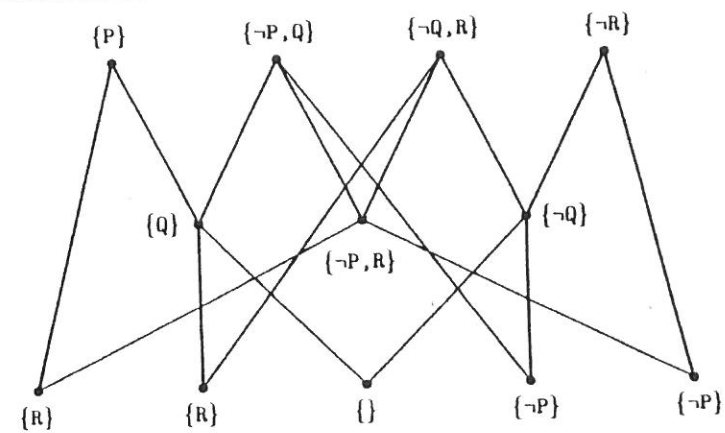

Figure 4.4 Three-level resolution graph.

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

formula to be proved. As in preceding examples, the Δ notation indicates that the associated clause is in the initial database.

1.	{F(Art,Jon)}	Δ
2.	{F(Bob,Kim)}	Δ
3.	{¬F(x,y),P(x,y)}	Δ
4.	{¬P(Art,Jon)}	Γ
5.	{P(Art,Jon)}	1, 3
6.	{P(Bob,Kim)}	2, 3
7.	{¬F(Art,Jon)}	3, 4
8.	{}	4, 5
9.	{}	1, 7

We often refer to the formula we are trying to prove as a *goal* and to the clauses that result from its negation as *goal clauses*. In this previous example, there is just one goal clause. The negation and conversion of more complex questions can lead to several goal clauses, all of which must be added to the database. In some cases, several or all of these goal clauses must be used to derive a result.

Suppose, for example, that we knew nothing about Art or John and we wanted to prove the simple tautology that either Art is the father of Jon or he is not. The goal in this case is the disjunction F(Art,Jon) ∨ ¬F(Art,Jon). Negating this sentence and converting to clausal form leads to the first two clauses in the following resolution trace. These two clauses can be resolved with each other directly to produce the empty clause and so prove the result.

1.	{¬F(Art,Jon)}	Γ
2.	{F(Art,Jon)}	Γ
3.	{}	1, 2

In addition to answering true-or-false questions from databases, resolution is useful in proving mathematical theorems and program correctness. Examples are given in Section 4.9 and in the Exercises.

4.7 Fill-in-the-Blank Questions

In Section 4.6, we saw how to use resolution in answering true-or-false questions (e.g., *Is Art one of Jon's parents?*). In this section, we show how resolution can be used to answer fill-in-the-blank questions as well (e.g., *Who is Jon's parent?*).

A fill-in-the-blank question is a predicate-calculus sentence with free variables specifying the blanks to be filled in. The goal is to find bindings

for the free variables such that the database logically implies the sentence obtained by substituting the bindings into the original question. For example, to ask about Jon's parent, we would write the question P(x,Jon). Using the database from the previous section, we see that Art is an answer to this question, since the sentence P(Art,Jon) is logically implied by the database.

An *answer literal* for a fill-in-the-blank question ϕ is a terin of the form Ans($\nu_1, \ldots, \nu_n$), where the variables $\nu_1, \ldots, \nu_n$ are the free variables in ϕ. To answer ϕ, we form a disjunction from the negation of ϕ and its answer literal and convert to clausal form. For example, the negation of P(x,Jon) is combined with its answer literal Ans(x) to form the the disjunction ¬P(x,Jon) ∨ Ans(x), which leads to the clause {¬P(x,Jon),Ans(x)}.

We then use resolution as described in Section 4.4, except that we change the termination test. Rather than waiting for the empty clause to be produced, the procedure halts as soon as it derives a clause consisting of only answer literals. The following resolution trace shows how we compute the answer to *Who is Jon's father?*

1.	{F(Art,Jon)}	Δ
2.	{F(Bob,Kim)}	Δ
3.	{¬F(x,y),P(x,y)}	Δ
4.	{¬P(z,Jon),Ans(z)}	Γ
5.	{P(Art,Jon)}	1, 3
6.	{P(Bob,Kim)}	2, 3
7.	{¬F(w,Jon),Ans(w)}	3, 4
8.	{Ans(Art)}	4, 5
9.	{Ans(Art)}	1, 7

If this procedure produces only one answer literal, the terms it contains constitute the only answer to the question. In some cases, the result of a fill-in-the-blank resolution depends on the refutation by which it is produced. In general, several different refutations can result from the same query. In some cases, as in this one, the answers may be the same; in other cases, there may be a difference.

Suppose, for example, that we knew the identities of both the father and mother of Jon and that we asked *Who is one of Jon's parents?* The following resolution trace shows that we can derive two answers to this question.

1.	{F(Art,Jon)}	Δ
2.	{M(Ann,Jon)}	Δ
3.	{¬F(x,y),P(x,y)}	Δ
4.	{¬M(u,v),P(u,v)}	Δ
5.	{¬P(z,Jon),Ans(z)}	Γ

6.	{P(Art,Jon)}	1, 3
7.	{P(Ann,Jon)}	2, 4
8.	{¬F(s,Jon),Ans(s)}	3, 5
9.	{¬M(t,Jon),Ans(t)}	4, 5
10.	{Ans(Art)}	5, 6
11.	{Ans(Ann)}	5, 7
12.	{Ans(Art)}	1, 8
13.	{Ans(Ann)}	2, 9

Unfortunately, we have no way of knowing whether or not the answer statement from a given refutation exhausts the possibilities. We can continue to search for answers until we find enough of them. However, due to the undecidabilty of logical implication, we can never know in general whether we have found all the possible answers.

Another interesting aspect of fill-in-the-blank resolution is that in some cases the procedure can result in a clause containing more than one answer literal. The significance of this is that no one answer is guaranteed to work, but one of the answers must be correct.

The following resolution trace illustrates this fact. The database in this case is a disjunction asserting that either Art or Bob is the father of Jon, but we do not know which man is. The goal is to find the father of John. After resolving the goal clause with the database disjunction, we get a clause that can once again be resolved with the goal clause yielding a clause with two answer literals.

1.	{F(Art,Jon),F(Bob,Jon)}	Δ
2.	{¬F(x,Jon),Ans(x)}	Γ
3.	{F(Bob,Jon),Ans(Art)}	1, 2
4.	{Ans(Art),Ans(Bob)}	2, 3

In such situations, we can continue searching in hope of finding a more specific answer. However, given the undecidability of logical implication, we can never know in general whether we can stop and say that no more specific answer exists.

4.8 Circuits Example

One advantage of describing a circuit in predicate calculus is that we can use automated deduction procedures, such as resolution, to reason about the circuit in a variety of ways. For example, we can simulate the behavior of the circuit for given values of the inputs, we can diagnose its failures, and we can generate tests to ensure that it is working properly.

The first step in performing any of these tasks is to convert the description to clausal form. Consider the circuit described in Figure 2.3. The structural description of the circuit is easily transformed, since the sentences are all atomic.

1. {Xorg(X1)}
2. {Xorg(X2)}
3. {Andg(A1)}
4. {Andg(A2)}
5. {Org(O1)}

6. {Conn(I(1,F1),I(1,X1))}
7. {Conn(I(2,F1),I(2,X1))}
8. {Conn(I(1,F1),I(1,A1))}
9. {Conn(I(2,F1),I(2,A1))}
10. {Conn(I(3,F1),I(2,X2))}
11. {Conn(I(3,F1),I(1,A2))}
12. {Conn(O(1,X1),I(1,X2))}
13. {Conn(O(1,X1),I(2,A2))}
14. {Conn(O(1,A2),I(1,O1))}
15. {Conn(O(1,A1),I(2,O1))}
16. {Conn(O(1,X2),O(1,F1))}
17. {Conn(O(1,O1),O(2,F1))}

Since the behavior of each of the components is described by a simple implication, there is one clause for each sentence in the behavioral description. Recall that the function (denoted by) I maps an integer and a device into the corresponding input port of the device, the function O maps an integer and a device into the corresponding output port, and the relation V is true of a port and a signal if and only if the specified port is carrying the specified signal.

18. {¬Andg(d),¬V(I(1,d),1),¬V(I(2,d),1),V(O(1,d),1)}
19. {¬Andg(d),¬V(I(n,d),0),V(O(1,d),0)}

20. {¬Org(d),¬V(I(n,d),1),V(O(1,d),1)}
21. {¬Org(d),¬V(I(1,d),0),¬V(I(2,d),0),V(O(1,d),0)}

22. {¬Xorg(d),¬V(I(1,d),y),¬V(I(2,d),z),y=z,V(O(1,d),1)}
23. {¬Xorg(d),¬V(I(1,d),z),¬V(I(2,d),z),V(O(1,d),0)}

24. {¬Conn(x,y),¬V(x,z),V(y,z)}

We also need to express the fact that the two possible digital values are not equal to each other. If there were a large or infinite number of possible values, such inequalities would have to be handled by procedural attachment; since there are only two, however, the following two clauses are sufficient.

CHAPTER 5
Resolution Strategies

ONE OF THE DISADVANTAGES of using the resolution rule in an unconstrained manner is that it leads to many useless inferences. Some inferences are redundant in that their conclusions can be derived in other ways. Some inferences are irrelevant in that they do not lead to derivations of the desired result.

As an example, consider the resolution trace in Figure 5.1. Clauses 9, 11, 14, and 16 are redundant; clauses 10 and 13 are redundant; clauses 12 and 15 are redundant; all these redundancies lead to subsequent redundancies at the next level of deduction. We can remove duplicate clauses and thereby prevent the propagation of redundant conclusions. However, their initial generation is an indication of inefficiency in the unconstrained use of the resolution principle.

This chapter presents a number of strategies for eliminating useless work. In reading the chapter, it is important to bear in mind that we are concerned here not with the order in which inferences are done, but only with the size of a resolution graph and with ways of decreasing that size by eliminating useless deductions.

5.1 Deletion Strategies

A *deletion strategy* is a restriction technique in which clauses with specified properties are eliminated before they are ever used. Since those clauses

1.	{P,Q}	Δ
2.	{¬P,R}	Δ
3.	{¬Q,R}	Δ
4.	{¬R}	Γ
5.	{Q,R}	1, 2
6.	{P,R}	1, 3
7.	{¬P}	2, 4
8.	{¬Q}	3, 4
9.	{R}	3, 5
10.	{Q}	4, 5
11.	{R}	3, 6
12.	{P}	4, 6
13.	{Q}	1, 7
14.	{R}	6, 7
15.	{P}	1, 8
16.	{R}	5, 8
17.	{}	4, 9
18.	{R}	3, 10
19.	{}	8, 10
20.	{}	4, 11
21.	{R}	2, 12
22.	{}	7, 12
23.	{R}	3, 13
24.	{}	8, 13
25.	{}	4, 14
26.	{R}	2, 15
27.	{}	7, 15
28.	{}	4, 16
29.	{}	4, 18
30.	{}	4, 21
31.	{}	4, 23
32.	{}	4, 26

Figure 5.1 Example of unconstrained resolution.

are unavailable for subsequent deduction, this can lead to computational savings.

A literal occurring in a database is *pure* if and only if it has no instance that is complementary to an instance of another literal in the database.

A clause that contains a pure literal is useless for the purposes of refutation, since the literal can never be resolved away. Consequently, we can safely remove such a clause. Removing clauses with pure literals defines a deletion strategy known as *pure-literal elimination*.

The database that follows is unsatisfiable. However, in proving this we can ignore the second and third clauses, since they both contain the pure literal S.

{¬P,¬Q,R}

{¬P,S}

{¬Q,S}

{P}

{Q}

{¬R}

Note that, if a database contains no pure literals, there is no way we can derive any clauses with pure literals using resolution. The upshot is that we do not need to apply the strategy to a database more than once, and in particular we do not have to check each clause as it is generated.

A *tautology* is a clause containing a pair of complementary literals. For example, the clause {P(F(A)),¬P(F(A))} is a tautology. The clause {P(x),Q(y),¬Q(y),R(z)} also is a tautology, even though it contains additional literals.

As it turns out, the presence or absence of tautologies in a set of clauses has no effect on that set's satisfiability. A satisfiable set of clauses remains satisfiable, no matter what tautologies we add. An unsatisfiable set of clauses remains unsatisfiable, even if we remove all tautologies. Therefore, we can remove tautologies from a database, because we need never use them in subsequent inferences. The corresponding deletion strategy is called *tautology elimination*.

Note that the literals in a clause must be exact complements for tautology elimination to apply. We cannot remove nonidentical literals, just because they are complements under unification. For example, the clauses {¬P(A),P(x)}, {P(A)}, and {¬P(B)} are unsatisfiable. However, if we were to remove the first clause, the remaining set would be satisfiable.

In *subsumption elimination*, the deletion criterion depends on a relationship between two clauses in a database. A clause Φ *subsumes* a clause Ψ if and only if there exists a substitution σ such that $\Phi\sigma \subseteq \Psi$. For example, {P(x),Q(y)} subsumes {P(A),Q(v),R(w)}, since there is a substitution {x/A,y/v} that makes the former clause a subset of the latter.

If one member in a set of clauses is subsumed by another member, then the set remaining after eliminating the subsumed clause is satisfiable if and

only if the original set is satisfiable. Therefore, subsumed clauses can be eliminated. Since the resolution process itself can produce tautologies and subsuming clauses, we need to check for tautologies and subsumptions as we perform resolutions.

5.2 Unit Resolution

A *unit resolvent* is one in which at least one of the parent clauses is a *unit clause*; i.e., one containing a single literal. A *unit deduction* is one in which all derived clauses are unit resolvents. A *unit refutation* is a unit deduction of the empty clause {}.

As an example of a unit refutation, consider the following proof. In the first two inferences, unit clauses from the initial set are resolved with binary clauses to produce two new unit clauses. These are resolved with the first clause to produce two additional unit clauses. The elements in these two sets of results are then resolved with each other to produce the contradiction.

1. {P,Q}	Δ
2. {¬P,R}	Δ
3. {¬Q,R}	Δ
4. {¬R}	Γ
5. {¬P}	2, 4
6. {¬Q}	3, 4
7. {Q}	1, 5
8. {P}	1, 6
9. {R}	3, 7
10. {}	6, 7
11. {R}	2, 8
12. {}	5, 8

Note that the proof contains only a subset of the possible uses of the resolution rule. For example, clauses 1 and 2 can be resolved to derive the conclusion {Q,R}. However, this conclusion and its descendants are never generated, since neither of its parents is a unit clause.

Inference procedures based on unit resolution are easy to implement and are usually quite efficient. It is worth noting that, whenever a clause is resolved with a unit clause, the conclusion has fewer literals than the parent does. This helps to focus the search toward producing the empty clause and thereby improves efficiency.

Unfortunately, inference procedures based on unit resolution generally are not complete. For example, the clauses {P,Q}, {¬P,Q}, {P,¬Q}, and {¬P,¬Q} are inconsistent. Using general resolution, it is easy to derive the

empty clause. However, unit resolution fails in this case, since none of the initial propositions is a single literal.

On the other hand, if we restrict our attention to Horn clauses (i.e., clauses with at most one positive literal), the situation is much better. In fact, it can be shown that there is a unit refutation of a set of Horn clauses if and only if it is unsatisfiable.

5.3 Input Resolution

An *input resolvent* is one in which at least one of the two parent clauses is a member of the initial (i.e., input) database. An *input deduction* is one in which all derived clauses are input resolvents. An *input refutation* is an input deduction of the empty clause {}.

As an example, consider clauses 6 and 7 in Figure 5.1. Using unconstrained resolution, these clauses can be resolved to produce clause 14. However, this is not an input resolution, since neither parent is a member of the initial database.

Note that the resolution of clauses 1 and 2 is an input resolution but not a unit resolution. On the other hand, the resolution of clauses 6 and 7 is a unit resolution but not an input resolution. Despite differences such as this one, it can be shown that unit resolution and input resolution are equivalent in inferential power in that there is a unit refutation from a set of sentences whenever there is an input refutation and vice versa.

One consequence of this fact is that input resolution is complete for Horn clauses but incomplete in general. Again, the unsatisfiable set of propositions {P,Q}, {¬P,Q}, {P,¬Q}, and {¬P,¬Q} provides an example of a deduction on which input resolution fails. An input refutation must (in particular) have one of the parents of {} be a member of the initial database. However, to produce the empty clause in this case, we must resolve either two single literal clauses or two clauses having single-literal factors. None of the members of the base set meet either of these criteria, so there cannot be an input refutation for this set.

5.4 Linear Resolution

Linear resolution (also called *ancestry-filtered resolution*) is a slight generalization of input resolution. A *linear resolvent* is one in which at least one of the parents is either in the initial database or is an ancestor of the other parent. A *linear deduction* is one in which each derived clause is a linear resolvent. A *linear refutation* is a linear deduction of the empty clause {}.

Linear resolution takes its name from the linear shape of the proofs it generates. A linear deduction starts with a clause in the initial database (called the *top clause*) and produces a linear chain of resolutions such as that shown in Figure 5.2. Each resolvent after the first one is obtained from

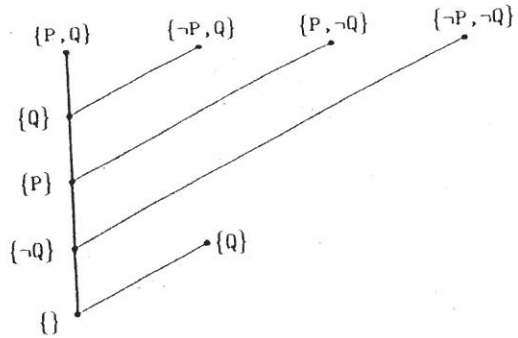

Figure 5.2 Chain of resolutions in a linear deduction.

the last resolvent (called the *near parent*) and some other clause (called the *far parent*). In linear resolution, the far parent must either be in the initial database or be an ancestor of the near parent.

Much of the redundancy in unconstrained resolution derives from the resolution of intermediate conclusions with other intermediate conclusions. The advantage of linear resolution is that it avoids many useless inferences by focusing deduction at each point on the ancestors of each clause and on the elements of the initial database.

Linear resolution is known to be refutation complete. Furthermore, it is not necessary to try every clause in the initial database as top clause. It can be shown that, if a set of clauses Γ is satisfiable and $\Gamma \cup \{\phi\}$ is unsatisfiable, then there is a linear refutation with ϕ as top clause. So, if we knows that a particular set of clauses is consistent, one need not attempt refutations with the elements of that set as top clauses.

A *merge* is a resolvent that inherits a literal from each parent such that this literal is collapsed to a singleton by the most general unifier. The completeness of linear resolution is preserved even if the ancestors that are used are limited to merges. Note that, in this example, the first resolvent (i.e., clause {Q}) is a merge.

5.5 Set of Support Resolution

If we examine resolution traces such as the one shown in Figure 5.1, we notice that many conclusions come from resolutions between clauses contained in a portion of the database that we know to be satisfiable. For

example, in Figure 5.1, the set Δ is satisfiable, yet many of the conclusions in the trace are obtained by resolving elements of Δ with other elements of Δ. As it turns out, we can eliminate these resolutions without affecting the refutation completeness of resolution.

A subset Γ of a set Δ is called a *set of support* for Δ if and only if $\Delta - \Gamma$ is satisfiable. Given a set of clauses Δ with set of support Γ, a *set of support resolvent* is one in which at least one parent is selected from Γ or is a descendant of Γ. A *set of support deduction* is one in which each derived clause is a set of support resolvent. A *set of support refutation* is a set of support deduction of the empty clause {}.

The following trace is a set of support refutation for the example in Figure 5.1, with the singleton set {¬R} as the set of support. The clause {¬R} resolves with {¬P,R} and {¬Q,R} to produce {¬P} and {¬Q}. These then resolve with clause 1 to produce {Q} and {P}, which resolve to produce the empty clause.

1.	{P,Q}	Δ
2.	{¬P,R}	Δ
3.	{¬Q,R}	Δ
4.	{¬R}	Γ
5.	{¬P}	2, 4
6.	{¬Q}	3, 4
7.	{Q}	1, 5
8.	{P}	1, 6
9.	{R}	3, 7
10.	{}	6, 7
11.	{R}	2, 8
12.	{}	5, 8

Obviously, this strategy would be of little use if there were no easy way of selecting the set of support. Fortunately, there are several ways this can be done at negligible expense. For example, in situations where we are trying to prove conclusions from a consistent database, the natural choice is to use the clauses derived from the negated goal as the set of support. This set satisfies the definition as long as the database itself is truly satisfiable. With this choice of set of support, each resolution must have a connection to the overall goal, so the procedure can be viewed as working "backward" from the goal. This is especially useful for databases in which the number of conclusions possible by working "forward" is larger. Furthermore, the goal-oriented character of such refutations often makes them more understandable than refutations using other strategies.

5.6 Ordered Resolution

Ordered resolution is a very restrictive resolution strategy in which each clause is treated as a linearly ordered set. Resolution is permitted only on the first literal of each clause; i.e., the literal that is least in the ordering. The literals in the conclusion preserve the order from their parent clauses with the literals from the positive parent followed by the literals from the negative parent (i.e., the one with the negated atom).

The following trace is an example of an ordered refutation. Clause 5 is the only ordered resolvent of clauses 1 through 4. Clauses 1 and 3 do not resolve, since the complementary literals are not first in each clause. Clauses 2 and 4 do not resolve for the same reason, nor do clauses 3 and 4. Once clause 5 is generated, it resolves with clause 3 to produce clause 6, which resolves with clause 4 to produce the empty clause.

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

Ordered resolution is extremely efficient. In this case, the empty clause is produced at the third level of deduction, and the inference space through that level of deduction includes only three resolvents. By comparison, general resolution through that level results in 24 resolvents.

Unfortunately, ordered resolution is not refutation complete. However, if we restrict our attention to Horn clauses, refutation completeness is guaranteed. Furthermore, we can get refutation completeness in the general case by considering resolvents in which the remaining literals from the positive parent follow the remaining literals from the negative parent, as well as the other way around.

5.7 Directed Resolution

Directed resolution is the use of ordered resolution in an important but restricted set of deductions. In directed resolution, the query takes the form of a conjunction of positive literals, and the database consists entirely of *directed clauses*. A directed clause is a Horn clause in which the positive literal occurs either at the beginning or the end of the clause. The goal is to find bindings for the variables so that the conjunction resulting from the substitution of these bindings is provable from the database.

In looking at directed resolution, we can use a bit of syntactic sugar. Since all the clauses are directional, we can write them in *infix form*. We write clauses with the positive literal at the end using the ⇒ operator. We write clauses in which the positive literal is at the beginning using the reverse implication operator ⇐. We let the literal in a positive unit clause represent the clause as a whole. We write the negative literals in clauses without positive literals as the antecedents of either implication operator.

$$\{\neg\phi_1,\ldots,\neg\phi_n,\psi\} \leftrightarrow \phi_1,\ldots,\phi_n \Rightarrow \psi$$
$$\{\psi,\neg\phi_1,\ldots,\neg\phi_n\} \leftrightarrow \psi \Leftarrow \phi_1,\ldots,\phi_n$$
$$\{\neg\phi_1,\ldots,\neg\phi_n\} \leftrightarrow \phi_1,\ldots,\phi_n \Rightarrow$$
$$\{\neg\phi_1,\ldots,\neg\phi_n\} \leftrightarrow \Leftarrow \phi_1,\ldots,\phi_n$$

The distinguishing feature of directed resolution is the directionality of the clauses in the database. Some clauses give rise to forward resolution, in which positive conclusions are derived from positive data. Other clauses give rise to backward resolution, in which negative clauses are derived from other negative clauses. As suggested by the preceding equivalences, the directionality of a clause is determined by the position of the positive literal in the clause.

A *forward clause* is one in which the positive literal comes at the end. In directed resolution, forward clauses give rise to forward resolution. To see why this is so, consider the following proof. Using ordered resolution on the first two clauses leads to the conclusion P(A), and then this conclusion is resolved with the negative unit to derive the empty clause. Putting the positive literal at the end makes it possible to work forward to the positive intermediate conclusion (clause 4), but makes it impossible to work backward from the negative clause (clause 3).

1. {¬M(x),P(x)}	M(x) ⇒ P(x)
2. {M(A)}	M(A)
3. {¬P(z)}	P(z) ⇒
4. {P(A)}	P(A)
5. {}	{}

Symmetrically, if the positive literal is put at the front of a clause, the clause is *backward*. If we rewrite the previous clauses in this way, we get the opposite behavior. In the following proof, the negative clause is resolved with the first clause to produce the intermediate negative conclusion {¬M(z)}, then this result is resolved with the second clause to derive the empty clause.

1. {P(x),¬M(x)}	P(x) ⇐ M(x)
2. {M(A)}	M(A)
3. {¬P(z)}	⇐ P(z)