

## Chapter 9

# Relational Resolution

### §9.1 Introduction

The *relational resolution principle* is a rule of inference for Relational Logic analogous to the propositional resolution principle for propositional logic. Using the relational resolution principle alone (without axiom schemata or other rules of inference), it is possible to build a reasoning program that is sound and complete for all of relational logic. The search space using the resolution principle is smaller than the search space for generating relational proofs.

In our tour of resolution, we look first at unification, which allows us to "unify" expressions by substituting terms for variables. We then move on to a definition of clausal form extended to handle variables. The resolution principle follows. We then look at some applications. Finally, we examine strategies for making the procedure more efficient.

### §9.2 Unification

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

A *substitution* is a finite mapping of variables to terms. In what follows, we write substitutions as sets of replacement rules, like the one shown below. In each rule, the variable to which the arrow is pointing is to be replaced by the term from which the arrow is pointing. In this case,  $x$  is to be replaced by  $a$ ,  $y$  is to be replaced by  $f(b)$ , and  $z$  is to be replaced by  $v$ .

$$\{x \leftarrow a, y \leftarrow f(b), z \leftarrow v\}$$

The variables being replaced together constitute the *domain* of the substitution, and the terms replacing them constitute the *range*. For example, in the preceding substitution, the domain is  $\{x, y, z\}$ , and the range is  $\{a, b, v\}$ .

A substitution is *pure* if and only if all replacement terms in the range are free of the variables in the domain of the substitution. Otherwise, the substitution is *impure*. The substitution shown above is pure whereas the one shown below is impure.

$$\{x \leftarrow a, y \leftarrow f(b), z \leftarrow x\}$$

The result of applying a substitution  $\sigma$  to an expression  $\phi$  is the expression  $\phi\sigma$  obtained from the original expression by replacing every occurrence of every variable in the domain of the substitution by the term with which it is associated.

$$\begin{aligned}
q(x, y)\{x \leftarrow a, y \leftarrow f(b), z \leftarrow v\} &= q(a, f(b)) \\
q(x, x)\{x \leftarrow a, y \leftarrow f(b), z \leftarrow v\} &= q(a, a) \\
q(x, w)\{x \leftarrow a, y \leftarrow f(b), z \leftarrow v\} &= q(a, w) \\
q(z, v)\{x \leftarrow a, y \leftarrow f(b), z \leftarrow v\} &= q(v, v)
\end{aligned}$$

Note that, if a substitution is pure, application is idempotent, i.e. applying a substitution a second time has no effect.

$$\begin{aligned}
q(x, x, y, w, z)\{x \leftarrow a, y \leftarrow f(b), z \leftarrow v\} &= q(a, a, f(b), w, v) \\
q(a, a, f(b), w, v)\{x \leftarrow a, y \leftarrow f(b), z \leftarrow v\} &= q(a, a, f(b), w, v)
\end{aligned}$$

However, this is not the case for impure substitutions, as illustrated by the following example. Applying the substitution once leads to an expression with an  $x$ , allowing for a different answer when the substitution is applied a second time.

$$\begin{aligned}
q(x, x, y, w, z)\{x \leftarrow a, y \leftarrow f(b), z \leftarrow x\} &= q(a, a, f(b), w, x) \\
q(a, a, f(b), w, x)\{x \leftarrow a, y \leftarrow f(b), z \leftarrow v\} &= q(a, a, f(b), w, a)
\end{aligned}$$

Given two or more substitutions, it is possible to define a single substitution that has the same effect as applying those substitutions in sequence. For example, the substitutions  $\{x \leftarrow a, y \leftarrow f(u), z \leftarrow v\}$  and  $\{u \leftarrow d, v \leftarrow e\}$  can be combined to form the single substitution  $\{x \leftarrow a, y \leftarrow f(d), z \leftarrow e, u \leftarrow d, v \leftarrow e\}$ , which has the same effect as the first two substitutions when applied to any expression whatsoever.

Computing the *composition* of a substitution  $\sigma$  and a substitution  $\tau$  is easy. There are two steps. (1) First, we apply  $\tau$  to the range of  $\sigma$ . (2) Then we adjoin to  $\sigma$  all pairs from  $\tau$  with different domain variables.

As an example, consider the composition shown below. In the right hand side of the first equation, we have applied the second substitution to the replacements in the first substitution. In the second equation, we have combined the rules from this new substitution with the non-conflicting rules from the second substitution.

$$\begin{aligned}
&\{x \leftarrow a, y \leftarrow f(u), z \leftarrow v\}\{u \leftarrow d, v \leftarrow e, z \leftarrow g\} \\
&= \{x \leftarrow a, y \leftarrow f(d), z \leftarrow e\}\{u \leftarrow d, v \leftarrow e, z \leftarrow g\} \\
&= \{x \leftarrow a, y \leftarrow f(d), z \leftarrow e, u \leftarrow d, v \leftarrow e\}
\end{aligned}$$

It is noteworthy that composition does not necessarily preserve substitutional purity. The composition of two impure substitutions may be pure, and the composition of two pure substitutions may be impure.

This problem does not occur if the substitutions are composable. A substitution  $\sigma$  and a substitution  $\tau$  are *composable* if and only if the domain on  $\sigma$  and the range of  $\tau$  are disjoint. Otherwise, they are *noncomposable*.

For example, the substitutions shown below are composable. It makes no difference that  $x$  appears in both substitutions.

$$\{x \leftarrow a, y \leftarrow b, z \leftarrow v\} \{x \leftarrow u, v \leftarrow b\}$$

By contrast, the following substitutions are noncomposable. Here,  $x$  occurs in both the domain of the first substitution and the range of the second substitution, violating the definition of composability.

$$\{x \leftarrow a, y \leftarrow b, z \leftarrow v\} \{u \leftarrow d, v \leftarrow x\}$$

The importance of composability is that it ensures preservation of purity. The composition of composable pure substitutions must be pure. In the sequel, we look only at compositions of composable pure substitutions.

A substitution  $\sigma$  is a *unifier* for an expression  $\phi$  and an expression  $\psi$  if and only if  $\phi\sigma = \psi\sigma$ , i.e. the result of applying  $\sigma$  to  $\phi$  is the same as the result of applying  $\sigma$  to  $\psi$ . If two expressions have a unifier, they are said to be *unifiable*. Otherwise, they are *nonunifiable*.

The expressions  $p(x,y)$  and  $p(a,v)$  are unifiable because they have a unifier, viz.  $\{x \leftarrow a, y \leftarrow b, v \leftarrow b\}$ . The results of applying this substitution to the two expressions are shown below.

$$\begin{aligned} p(x,y)\{x \leftarrow a, y \leftarrow b, v \leftarrow b\} &= p(a,b) \\ p(a,v)\{x \leftarrow a, y \leftarrow b, v \leftarrow b\} &= p(a,b) \end{aligned}$$

Note that, although this substitution unifies the two expressions, it is not the only unifier. We do not have to substitute  $b$  for  $y$  and  $v$  to unify the two expressions. We can equally well substitute  $c$  or  $d$  or  $f(c)$  or  $f(w)$ . In fact, we can unify the expressions without changing  $v$  at all by simply replacing  $y$  by  $v$ .

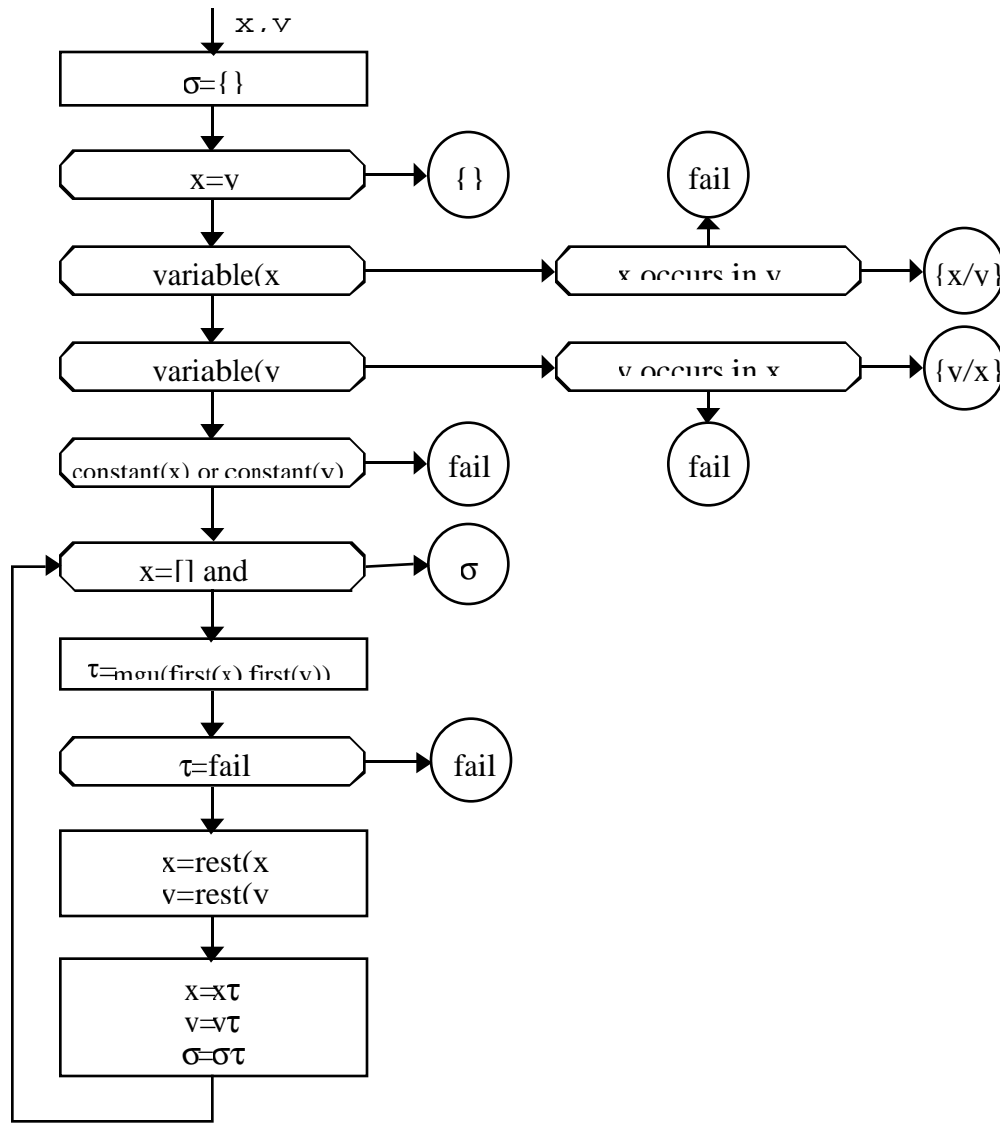
In considering these alternatives, it should be clear that some substitutions are more general than others are. We say that a substitution  $\sigma$  is *as general as or more general than* a substitution  $\tau$  if and only if there is another substitution  $\delta$  such that  $\sigma\delta = \tau$ . For example, the substitution  $\{x \leftarrow a, y \leftarrow v\}$  is more general than  $\{x \leftarrow a, y \leftarrow f(c), v \leftarrow f(c)\}$  since there is a substitution  $\{v \leftarrow f(c)\}$  that, when applied to the former, gives the latter.

$$\{x \leftarrow a, y \leftarrow v\} \{v \leftarrow f(c)\} = \{x \leftarrow a, y \leftarrow f(c), v \leftarrow f(c)\}$$

In resolution, we are interested only in unifiers with maximum generality. A *most general unifier*, or *mgu*,  $\sigma$  of two expressions has the property that it is as general as or more general than any other unifier.

Although it is possible for two expressions to have more than one most general unifier, all of these most general unifiers are structurally the same, i.e. they are unique up to variable renaming. For example,  $p(x)$  and  $p(y)$  can be unified by either the substitution  $\{x \leftarrow y\}$  or the substitution  $\{y \leftarrow x\}$ ; and either of these substitutions can be obtained from the other by applying a third substitution. This is not true of the unifiers mentioned earlier.

One good thing about our language is that there is a simple procedure for computing a most general unifier of any two expressions if it exists. The following figure gives a flowchart for this procedure.



The procedure assumes a representation of expressions as lists of subexpressions. For example, the expression  $p(a, f(b), z)$  can be thought of as a list with four elements, viz. the relation constant  $p$ , the object constant  $a$ , the term  $f(b)$ , and the variable  $z$ . The term  $f(b)$  can in turn be thought of as a list of two elements, viz. the function constant  $f$  and the object constant  $b$ .

The procedure begins with an empty substitution  $\sigma$ . If the two expressions it is given as argument are identical, it returns the empty substitution as an answer. If they are not identical and both expressions are constants, then it fails. If either is a variable, then it checks whether the other expression contains that variable. If the variable occurs within the expression, it fails; otherwise, it binds the variable to the expression and returns the resulting substitution. The only remaining possibility is that the two

expressions are compound. In this case, it iterates across the expressions, calling itself recursively on each pair of components. If the recursive call succeeds with a substitution  $\tau$ , then  $\sigma$  is updated to be  $\sigma\tau$ , and  $\tau$  is substituted in the remaining components of the two expressions. If the recursive call fails, the procedure reports failure to its caller. If the procedure gets to the ends of the two expressions, it succeeds and returns  $\sigma$  as value.

One especially noteworthy part of the unification procedure is the test for whether a variable occurs within an expression before the variable is bound to that expression. This test 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 makes them look alike.

### §9.3 Clausal Form

As with propositional resolution, relational resolution works only on expressions in *clausal form*. The definitions here are analogous. A *literal* is either a relational sentence or a negation of a relational sentence. A *clause* is a set of literals and, as in propositional logic, represents a disjunction of the literals in the set. A clause set is a set of clauses and represents a conjunction of the clauses in the set.

The procedure for converting relational sentences to clausal form is similar to that for Propositional Logic. Some of the rules are the same. However, there are a few additional rules to deal with the presence of variables and quantifiers. The conversion rules are summarized below and should be applied in order.

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

$$\begin{aligned}\phi_1 \Rightarrow \phi_2 &\rightarrow \neg\phi_1 \vee \phi_2 \\ \phi_1 \Leftarrow \phi_2 &\rightarrow \phi_1 \vee \neg\phi_2 \\ \phi_1 \Leftrightarrow \phi_2 &\rightarrow (\neg\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \neg\phi_2)\end{aligned}$$

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

$$\begin{aligned}\neg\neg\phi &\rightarrow \phi \\ \neg(\phi_1 \wedge \phi_2) &\rightarrow \neg\phi_1 \vee \neg\phi_2 \\ \neg(\phi_1 \vee \phi_2) &\rightarrow \neg\phi_1 \wedge \neg\phi_2 \\ \neg\forall v.\phi &\rightarrow \exists v.\neg\phi \\ \neg\exists v.\phi &\rightarrow \forall v.\neg\phi\end{aligned}$$

In the third step (Standardize variables), 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. The following transformation is an example.

$$\forall x.(p(x) \Rightarrow \exists x.q(x)) \rightarrow \forall x.(p(x) \Rightarrow \exists y.q(y))$$

In the fourth step (Existentials out), 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. The constant used to replace the existential variable in this case is called a *Skolem constant*. The following example assumes that  $a$  is not used anywhere else.

$$\exists x.p(x) \rightarrow p(a)$$

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. Any function defined in this way is called a *Skolem function*. The following example illustrates this transformation. It assumes that  $f$  is not used anywhere else.

$$\forall x.(p(x) \wedge \exists z.q(x, y, z)) \rightarrow \forall x.(p(x) \wedge q(x, y, f(x, y)))$$

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

$$\forall x.(p(x) \wedge q(x, y, f(x, y))) \rightarrow p(x) \wedge q(x, y, f(x, y))$$

In the sixth step (Disjunctions in), 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 rules.

$$\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \rightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$$

$$(\varphi_1 \wedge \varphi_2) \vee \varphi_3 \rightarrow (\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3)$$

$$(\varphi_1 \vee \varphi_2) \vee \varphi_3 \rightarrow \varphi_1 \vee (\varphi_2 \vee \varphi_3)$$

$$(\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \rightarrow \varphi_1 \wedge (\varphi_2 \wedge \varphi_3)$$

In the seventh step (Operators out), we eliminate operators by separating any conjunctions into its conjuncts and writing each disjunction as a separate clause.

$$\varphi_1 \wedge \dots \wedge \varphi_n \rightarrow \varphi_1, \dots, \varphi_n$$

$$\varphi_1 \vee \dots \vee \varphi_n \rightarrow \{\varphi_1, \dots, \varphi_n\}$$

In the final step (Rename variables), we rename variables so that no variable appears in more than one clause. The following transformations illustrate.

$$\begin{aligned} p(x) &\rightarrow p(x) \\ q(x) &\rightarrow q(y) \end{aligned}$$

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 labeled lines are the results of the corresponding steps of the conversion procedure.

$$\begin{aligned} &\exists y.(g(y) \wedge \forall z.(r(z) \Rightarrow f(y,z))) \\ \text{I} &\exists y.(g(y) \wedge \forall z.(\neg r(z) \vee f(y,z))) \\ \text{N} &\exists y.(g(y) \wedge \forall z.(\neg r(z) \vee f(y,z))) \\ \text{S} &\exists y.(g(y) \wedge \forall z.(\neg r(z) \vee f(y,z))) \\ \text{E} &g(\text{greg}) \wedge \forall z.(\neg r(z) \vee f(\text{greg},z)) \\ \text{A} &g(\text{greg}) \wedge (\neg r(z) \vee f(\text{greg},z)) \\ \text{D} &g(\text{greg}) \wedge (\neg r(z) \vee f(\text{greg},z)) \\ \text{O} &\{g(\text{greg})\} \\ &\{\neg r(z), f(\text{greg},z)\} \\ \text{R} &\{g(\text{greg})\} \\ &\{\neg r(z), f(\text{greg},z)\} \end{aligned}$$

Here is another example. In this case, the starting sentence is almost the same. The only difference is the leading  $\neg$ , but the result looks quite different.

$$\begin{array}{l}
\neg\exists y.(g(y) \wedge \forall z.(r(z) \Rightarrow f(y, z))) \\
\text{I} \quad \neg\exists y.(g(y) \wedge \forall z.(\neg r(z) \vee f(y, z))) \\
\text{N} \quad \neg\exists y.(g(y) \wedge \forall z.(\neg r(z) \vee f(y, z))) \\
\forall y.\neg(g(y) \wedge \forall z.(\neg r(z) \vee f(y, z))) \\
\forall y.(\neg g(y) \vee \neg\forall z.(\neg r(z) \vee f(y, z))) \\
\forall y.(\neg g(y) \vee \exists z.(\neg(\neg r(z) \vee f(y, z)))) \\
\forall y.(\neg g(y) \vee \exists z.(\neg\neg r(z) \wedge \neg f(y, z))) \\
\forall y.(\neg g(y) \vee \exists z.(r(z) \wedge \neg f(y, z))) \\
\text{S} \quad \forall y.(\neg g(y) \vee \exists z.(r(z) \wedge \neg f(y, z))) \\
\forall y.(\neg g(y) \vee \exists z.(r(z) \wedge \neg f(y, z))) \\
\text{E} \quad \forall y.(\neg g(y) \vee (r(g(y)) \wedge \neg f(y, g(y)))) \\
\text{A} \quad \neg g(y) \vee (r(g(y)) \wedge \neg f(y, g(y))) \\
\text{D} \quad (\neg g(y) \vee r(g(y))) \wedge (\neg g(y) \vee \neg f(y, g(y))) \\
\text{O} \quad \{\neg g(y) \vee r(g(y))\} \\
\quad \quad \{\neg g(y) \vee \neg f(y, g(y))\} \\
\text{R} \quad \{\neg g(y) \vee r(g(y))\} \\
\quad \quad \{\neg g(z) \vee \neg f(z, g(z))\}
\end{array}$$

In Propositional Logic, the clause set corresponding to any sentence is logically equivalent to that sentence. In relational logic, this is not necessarily the case. For example, the clausal form of the sentence  $\exists x.p(x)$  is  $\{p(a)\}$ . This is not logically equivalent. Since the clause exists in a language with an additional object constant, there are interpretations that satisfy the sentence but not the clause. On the other hand, the clause is *effectively* equivalent to the sentence in the sense that it is satisfiable if and only if the sentence is satisfiable. As we shall see, in resolution, this effective equivalence is all we need.

### §9.3 Resolution Principle

The relational resolution principle is analogous to that of propositional resolution. The only difference is the use of unification to unify literals before applying the rule.

A simple version of the Resolution Principle for Relational Logic is show below. Given a clause with a literal  $\phi$  and a second clause with a literal  $\neg\psi$  such that  $\phi$  and  $\psi$  have a most general unifier  $\sigma$ , we can derive a conclusion by applying  $\sigma$  to the clause consisting of the remaining literals from the two original clauses.

$$\begin{array}{c}
\{\phi_1, \dots, \phi, \dots, \phi_m\} \\
\{\psi_1, \dots, \neg\psi, \dots, \psi_n\} \\
\hline
\{\phi_1, \dots, \phi_m, \psi_1, \dots, \psi_n\}\sigma \\
\text{where } \sigma = mgu(\phi, \psi)
\end{array}$$

Consider the example shown below. The first clause contains the positive literal  $p(a,y)$  and the second clause contains a negative occurrence of  $p(x,f(x))$ . The substitution  $\{x \leftarrow a, y \leftarrow f(a)\}$  is a most general unifier of these two expressions. Consequently, we can collect the remaining literals  $r(y)$  and  $q(g(x))$  into a clause and apply the substitution to produce a conclusion.

$$\frac{\{p(a,y), r(y)\} \quad \{\neg p(x, f(x)), q(g(x))\}}{\{r(f(a)), q(g(a))\}}$$

Unfortunately, this simple version of the Resolution Principle is not quite good enough. Consider the two clauses shown below. Given the meaning of these two clauses, it should be possible to resolve them to produce the empty clause. However, the two atomic sentences do not unify. The variable  $x$  must be bound to  $a$  and  $b$  at the same time.

$$\{p(a, x)\} \\ \{\neg p(x, b)\}$$

Fortunately, this problem can easily be fixed by extending the Resolution Principle slightly as shown below. Before trying to resolve two clauses, we select one of the clauses and rename any variables the clause has in common with the other clause.

$$\frac{\{\varphi_1, \dots, \varphi, \dots, \varphi_m\} \quad \{\psi_1, \dots, \neg\psi, \dots, \psi_n\}}{\{\varphi_1\tau, \dots, \varphi_m\tau, \psi_1, \dots, \psi_n\}\sigma}$$

where  $\sigma = mgu(\varphi\tau, \psi)$   
 where  $\tau$  is a variable renaming on  $\varphi$

This extension of the Resolution Principle solves the problem. Unfortunately, it is still not quite adequate. A little later, we look at a second and final extension. However, we first look at some examples of the Resolution Principle in action.

As a more extensive example of resolution, once again consider the problem of Harry and Ralph introduced in the preceding chapter. We know that every horse can outrun every dog. Some greyhounds can outrun every rabbit. Greyhounds are dogs. The relationship of being faster is transitive. Harry is a horse. Ralph is a rabbit. We desire to prove that Harry can outrun Ralph.

$$\begin{aligned}
& \forall x. \forall y. (h(x) \wedge d(y) \Rightarrow f(x, y)) \\
& \forall y. (g(y) \wedge \forall z. (r(z) \Rightarrow f(y, z))) \\
& \forall y. (g(y) \Rightarrow d(y)) \\
& \forall x. \forall y. \forall z. (f(x, y) \wedge f(y, z) \Rightarrow f(x, z)) \\
& h(harry) \\
& r(ralph)
\end{aligned}$$

We desire to prove that Harry is faster than Ralph. In order to do this, we negate the desired conclusion.

$$\neg f(harry, ralph)$$

To do the proof, we take the premises and the negated conclusion and convert to clausal form. The resulting clauses are shown below. Note that the second premise has turned into two clauses.

1.  $\{\neg h(x), \neg d(y), f(x, y)\}$
2.  $\{g(greg)\}$
3.  $\{\neg r(z), f(greg, z)\}$
4.  $\{\neg g(y), d(y)\}$
5.  $\{\neg f(x, y), \neg f(y, z), f(x, z)\}$
6.  $\{h(harry)\}$
7.  $\{r(ralph)\}$
8.  $\{\neg f(harry, ralph)\}$

From these clauses, we can derive the empty clause, as shown in the following proof.

9.  $\{d(greg)\}$
10.  $\{\neg d(y), f(harry, y)\}$
11.  $\{f(harry, greg)\}$
12.  $\{f(greg, ralph)\}$
13.  $\{\neg f(greg, z), f(harry, z)\}$
14.  $\{\neg f(greg, z), f(harry, z)\}$
15.  $\{f(harry, ralph)\}$
16.  $\{\}$

That is pretty much it for resolution. However, there is one more minor technicality that must be addressed to finish the story. As stated, even with the extension mentioned above, the rule is not quite good enough. For example, given the clauses

$\{p(x), p(y)\}$  and  $\{\neg p(u), \neg p(v)\}$ , we should be able to infer the empty clause  $\{\}$ ; however, this is not possible with the preceding definition.

The good news is that we can solve this problem with one slight modification to our definition. If a subset of the literals in a clause  $\Phi$  has a most general unifier  $\gamma$ , then the clause  $\Phi'$  obtained by applying  $\gamma$  to  $\Phi$  is called a *factor* of  $\Phi$ . For example, the literals  $p(x)$  and  $p(f(y))$  have a most general unifier  $\{x \leftarrow f(y)\}$ , 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 a complete definition for the *resolution principle*. Suppose that  $\Phi$  and  $\Psi$  are two clauses. If there is a literal  $\phi$  in some factor  $\Phi'$  of  $\Phi$  and a literal  $\neg\psi$  in some factor  $\Psi'$  of  $\Psi$  such that  $\phi$  and  $\psi$  have a most general unifier  $\gamma$ , then we say that the two clauses  $\Phi$  and  $\Psi$  *resolve* and that the new clause  $((\Phi' - \{\phi\}) \cup (\Psi' - \{\neg\psi\}))\gamma$  is a *resolvent* of the two clauses.

$$\begin{array}{l}
 \Phi \\
 \Psi \\
 \hline
 ((\Phi' - \{\phi\}) \cup (\Psi' - \{\neg\psi\}))\sigma \\
 \text{where } \phi \in \Phi', \text{ a factor of } \Phi \\
 \text{where } \neg\psi \in \Psi', \text{ a factor of } \Psi \\
 \text{where } \sigma = mgu(\phi, \psi)
 \end{array}$$

Using this enhanced definition of resolution, we can solve the problem mentioned above. Once again, consider the premises  $\{p(x), p(y)\}$  and  $\{\neg p(u), \neg p(v)\}$ . The first premise has the factor  $\{p(x)\}$ , and the second has the factor  $\{\neg p(u)\}$ , and these two factors resolve to the empty clause in a single step.

Variable renaming like this 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.

A *resolution derivation* of a clause  $\phi$  from a set  $\Delta$  of clauses is a sequence of clauses terminating in  $\phi$  in which each item is either (1) a member of  $\Delta$  or (2) the result of applying the resolution principle to early items in sequence. A sentence  $\phi$  is *provable* from a set of sentences  $\Delta$  by resolution if and only if there is a derivation of the empty clause from the clausal form of  $\Delta \cup \{\neg\phi\}$ .

Although it is a little more complicated than propositional resolution, relational resolution shares with it the nice properties of soundness and completeness. If a conclusion is provable from a set of premises using resolution, then the premises logically entail the conclusion; and, if a set of premises logically entails a conclusion, then the conclusion is provable using resolution.

## §9.4 Applications

### 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 unsatisfiability, all we need do is to use resolution to derive consequences from the set to be tested, terminating whenever the empty clause is generated.

Demonstrating that a set of clauses is unsatisfiable can be used to demonstrate that a sentence is logically entailed by a set of sentences. Suppose we wish to show that the set of sentences  $\Delta$  logically entails the formula  $\psi$ . We can do this by finding a proof of  $\psi$  from  $\Delta$ ; i.e., by establishing  $\Delta \vdash \psi$ . By the refutation theorem, we can establish that  $\Delta \vdash \psi$  by showing that  $\Delta \cup \{\neg\psi\}$  is unsatisfiable. Thus, if we show that the set of formulas  $\Delta \cup \{\neg\psi\}$  is unsatisfiable, we have demonstrated that  $\Delta$  logically entails  $\psi$ .

Let us look at this technique from the standpoint of models. If  $\Delta \models \psi$ , then all the models of  $\Delta$  also are models of  $\psi$ . 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  $\Delta$  is satisfiable. Let  $i$  be an interpretation that satisfies  $\Delta$ ;  $i$  does not satisfy  $\neg\psi$ , because, if it did,  $\Delta \cup \neg\psi$  would be satisfiable. Therefore,  $i$  satisfies  $\psi$ . (An interpretation must satisfy one of either  $\psi$  or  $\neg\psi$ .) Since this holds for arbitrary  $i$  satisfying  $\Delta$ , it holds for all  $i$  satisfying  $\Delta$ . Thus, all models of  $\Delta$  are also models of  $\psi$ , and  $\Delta$  logically entails  $\psi$ .

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

### 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.

- |    |                             |         |
|----|-----------------------------|---------|
| 1. | $\{f(art, jon)\}$           | Premise |
| 2. | $\{f(bob, kim)\}$           | Premise |
| 3. | $\{\neg f(x, y), p(x, y)\}$ | Premise |
| 4. | $\{\neg p(art, jon)\}$      | Goal    |
| 5. | $\{p(art, jon)\}$           | 1,3     |
| 6. | $\{\}$                      | 4,5     |

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 ( $father(art, jon) \vee \neg father(art, jon)$ ). 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. | $\{\neg f(art, jon)\}$ | Goal |
| 2. | $\{f(art, jon)\}$      | Goal |
| 3. | $\{\}$                 | 1,2  |

In addition to answering true-or-false questions from databases, resolution is useful in proving mathematical theorems and program correctness.

### Fill-in-the-Blank Questions

In the preceding section, 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 entails the sentence obtained by substituting the bindings into the original question. For example, to ask about Jon's parent, we would write the question  $parent(x, jon)$ . Using the database from the previous section, we see that *Art* is an answer to this question, since the sentence  $parent(art, jon)$  is logically entailed by the database.

An *answer literal* for a fill-in-the-blank question  $\phi$  is a sentence  $goal(v_1, \dots, v_n)$ , where the variables  $v_1, \dots, v_n$  are the free variables in  $\phi$ . To answer  $\phi$ , we form an implication from  $\phi$  and its answer literal and convert to clausal form. For example, the literal  $parent(x, jon)$  is combined with its answer literal  $goal(x)$  to form the rule ( $parent(x, jon) \Rightarrow goal(x)$ ), which leads to the clause  $\{\neg parent(x, jon), goal(x)\}$ .

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.	$\{f(art, jon)\}$	Premise
2.	$\{f(bob, kim)\}$	Premise
3.	$\{\neg f(x, y), p(x, y)\}$	Premise
4.	$\{\neg p(z, jon), goal(z)\}$	Goal
5.	$\{\neg f(w, jon), goal(art)\}$	3,4
6.	$goal(art)$	1,5

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, leading to multiple answers.

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.	$\{f(art, jon)\}$	Premise
2.	$\{m(ann, jon)\}$	Premise
3.	$\{\neg f(x, y), p(x, y)\}$	Premise
4.	$\{\neg m(x, y), p(x, y)\}$	Premise
5.	$\{\neg p(z, jon), goal(z)\}$	Goal
6.	$\{\neg f(z, jon), goal(z)\}$	3,5
7.	$\{\neg m(z, jon), goal(z)\}$	4,5
8.	$goal(art)$	1,6
9.	$goal(ann)$	2,7

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 undecidability 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 a parent of John. After resolving the goal clause with the sentence about fathers and parents, we resolve the result with the database disjunction,

obtaining a clause that can be resolved a second time yielding a clause with two answer literals. This answers indicates not two answers but rather uncertainty as to which is the correct answer.

1.  $\{f(\text{art}, \text{jon}), f(\text{bob}, \text{jon})\}$  Premise
2.  $\{\neg f(x, y), p(x, y)\}$  Premise
3.  $\{\neg p(z, \text{jon}), \text{goal}(z)\}$  Goal
4.  $\{\neg f(z, \text{jon}), \text{goal}(z)\}$  2,3
5.  $\{f(\text{art}, \text{jon}), \text{goal}(\text{bob})\}$  1,4
6.  $\{\text{goal}(\text{art}), \text{goal}(\text{bob})\}$  4,5

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.

## §9.5 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.

This section presents a number of strategies for eliminating useless work. In reading this material, 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.

### Pure Literal Elimination

A literal occurring in a clause set is *pure* if and only if it has no instance that is complementary to an instance of another literal in the clause set. 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 clause set that follows is unsatisfiable. However, in proving this we can ignore the second and third clauses, since they both contain the pure literal  $s$ .

$$\{\neg p, \neg q, r\}$$

$$\{\neg p, s\}$$

$$\{\neg q, s\}$$

$$\{p\}$$

$$\{q\}$$

$$\{\neg r\}$$

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.

### **Tautology Elimination**

A *tautology* is a clause containing a pair of complementary literals. For example, the clause  $\{p(f(a)), \neg p(f(a))\}$  is a tautology. The clause  $\{p(x), q(y), \neg q(y), r(z)\}$  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 non-identical literals, just because they are complements under unification. For example, the clauses  $\{\neg p(a), p(x)\}$ ,  $\{p(a)\}$ , and  $\{\neg p(b)\}$  are unsatisfiable. However, if we were to remove the first clause, the remaining set would be satisfiable.

### **Subsumption Elimination**

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

If one member in a set of clauses subsumes 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.

### **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.	$\{p, q\}$	Premise
2.	$\{\neg p, r\}$	Premise
3.	$\{\neg q, r\}$	Premise
4.	$\{\neg r\}$	Premise
5.	$\{\neg p\}$	2, 4
6.	$\{\neg q\}$	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  $\{p, q\}$ ,  $\{\neg p, q\}$ ,  $\{p, \neg q\}$ , and  $\{\neg p, \neg q\}$  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.

## 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.

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. The unsatisfiable set of propositions  $\{p,q\}$ ,  $\{\neg p,q\}$ ,  $\{p, \neg q\}$ , and  $\{\neg p,\neg q\}$  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.

## 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 resolution. Each resolvent after the first one is obtained from 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  $\Gamma$  is satisfiable and  $\Gamma \cup \{\phi\}$  is unsatisfiable, then there is a linear refutation with  $\phi$  as top clause. So, if we know that a particular set of clauses is consistent, we 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.

## Set of Support Resolution

If we examine resolution traces, 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 many cases, the set of premises is satisfiable, yet many of the conclusions are obtained by resolving premises with other premises rather than the negated conclusion. As it turns out, we can eliminate these resolutions without affecting the refutation completeness of resolution.

A subset  $\Gamma$  of a set  $\Delta$  is called a *set of support* for  $\Delta$  if and only if  $\Delta - \Gamma$  is satisfiable. Given a set of clauses  $\Delta$  with set of support  $\Gamma$ , a *set of support resolvent* is one in which at least one parent is selected from  $\Gamma$  or is a descendant of  $\Gamma$ . 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, with the singleton set  $\{\neg r\}$  as the set of support. The clause  $\{\neg r\}$  resolves with  $\{\neg p, r\}$  and  $\{\neg q, r\}$  to produce  $\{\neg p\}$  and  $\{\neg q\}$ . These then resolve with clause 1 to produce  $\{q\}$  and  $\{p\}$ , which resolve to produce the empty clause.

- |     |                 |         |
|-----|-----------------|---------|
| 1.  | $\{p, q\}$      | Premise |
| 2.  | $\{\neg p, r\}$ | Premise |
| 3.  | $\{\neg q, r\}$ | Premise |
| 4.  | $\{\neg r\}$    | Support |
| 5.  | $\{\neg p\}$    | 2, 4    |
| 6.  | $\{\neg q\}$    | 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.

## Historical Notes

The resolution principle was introduced by Robinson [Robinson1965], based on earlier work by Prawitz [Prawitz1960] and others. Books by Chang and Lee [Chang1973], Loveland [Loveland1978], Robinson [Robinson1979], and Wos et al. [Wos1984a] describe resolution theorem proving methods and systems. A useful collection of theorem-proving papers can be found in the volumes by Siekmann and

Wrightson [Siekmann1983a, Siekmann1983b]. See also the reviews by Loveland [Loveland1983] and by Wos [Wos1985].

Our procedure for converting sentences into clausal form is based on work by Davis and Putnam [Davis1960]. Resolution also can be accomplished on formulas not in clausal form (see [Manna1979, Stickel1982]).

A unification algorithm and a proof of correctness is presented in Robinson [Robinson1965]. Several variations have appeared since. Raulefs et al. [Raulefs1978] survey unification and matching. Paterson and Wegman [Paterson1976] present a linear-time (and linear-space) unification algorithm. Unification has become increasingly important generally in computer science and in computational linguistics [Shieber1986]. It is a fundamental operation performed in the computer language prolog [Clocksin1981] [Sterling1986].

The use of an answer literal in resolution was first proposed by Green [Green1969b], and was investigated in more detail by Luckham and Nilsson [Luckham1971]. The idea of procedural attachment is extremely important for improving the efficiency of theorem-proving systems. Work by Weyhrauch [Weyhrauch1980] explains this technique, which he calls *semantic attachment*, in terms of having a *partial model* of the sentences. Semantic attachment is an excellent candidate for the important bridge that is needed between declarative knowledge and procedural knowledge in complex AI systems. Stickel [Stickel1985] shows how semantic attachment is related to what he calls *theory resolution*.

The soundness and completeness of resolution were originally proved by Robinson [Robinson1965]. The concept of excess literals is due to Bledsoe [Bledsoe1977].

Many restriction strategies for resolution refutations are discussed in detail by Loveland [Loveland1978], by Chang and Lee [Chang1973], and by Wos et al. [Wos1984a].

Ordered resolution is similar to *lock resolution*, which was originally proposed by Boyer [Boyer1971], and to *SL-resolution*, which was explored by Kowalski [Kowalski1971]. Depth-first backward resolution is the strategy used in prolog [Clocksin1981, Sterling1986], as well as in numerous expert systems. Moore [Moore1975] was one of the first people to point out the efficiencies to be gained by choosing the appropriate direction for reasoning. Treitel and Genesereth explored the problem of automatically determining optimal directionality [Treitel1987]. The adjacency theorem for optimal literal ordering was proved by Smith and Genesereth [Smith1985]. A variety of additional strategies for resolution are discussed in [1971, 1972, Minker 1973, 1979, Smith 1986][Kowalski1970].

Although not discussed in this book, it is often helpful to pre-compute all the possible resolutions that can be performed among a set of clauses and to store the results in a *connection graph*. The actual search for a refutation can then be described in terms of operations on this graph. The use of connection graphs was first proposed by Kowalski [Kowalski1975]. Other authors who have used various forms of connection graphs are Stickel, Chang and Slagle, and Stickel.

Several extremely efficient resolution refutation systems have been written that are able to solve large, nontrivial reasoning problems, including some open problems in

mathematics. A typical challenge problem for testing and illustrating the features of theorem-proving programs is the so-called *Schubert steamroller* problem.

Several other non-resolution theorem-proving systems also have been developed. Examples include those of Bledsoe, and of Boyer and Moore. Shankar used the Boyer-Moore theorem prover in verifying steps in the proof of Godel's incompleteness.

### Exercises

1. *Resolution*. If a course is easy, some students are happy. If a course has a final, no students are happy. Use resolution to show that, if a course has a final, the course is not easy.
2. *Substitution*. Composition of substitution does not preserve either purity or impurity.
  - (a) Give an example of two impure substitutions that, when composed, yield a pure substitution.
  - (b) Give an example of two pure substitutions that, when composed, yield an impure substitution.