Part II

Uniform Proof Procedures

Chapter 5

Formalizing the Notion of Proof

- This chapter defines 'proof' formally and introduces clausal form and two rules of inference: resolution and paramodulation.
- Section 5.1 defines the resolution rule of inference in easy stages.
- Section 5.2 introduces Kowalski form.
- Section 5.3 defines the paramodulation rule of inference.

Now we have developed our formal notation for expressing mathematical knowledge we can return to the question of how it can be manipulated in a computer program. That is, we can continue our look at *procedures* for 'doing' Mathematics. In section 2.2.4 we described a decision procedure for identifying correct propositional arguments. But in section 3.4 we saw that there were practical difficulties in extending this procedure to arguments involving the internal structure of propositions. So in this chapter we develop an alternative technique: instead of trying to show that the conclusion of an argument is a logical consequence of its hypothesis, we will try to *prove* the conclusion from the hypothesis.

In order for a computer to prove theorems it is necessary that it should be able to represent proofs internally. So we must extend our notation for expressing mathematical knowledge to include proofs.

Definition 5.1 *A proof is a sequence of formulae, each of which is either an axiom or follows from earlier formulae by a rule of inference.*

This definition captures the idea that to prove a theorem: we start with some axioms; apply a rule of inference to derive a lemma from these axioms; add the lemma to the original axioms; and and continue recursively until we arrive at the theorem we wanted. For instance,

1. $X = X$	(axiom)
2. $U = V \rightarrow (W = V \rightarrow U = W)$	(axiom)
3. $X = X \rightarrow (W = X \rightarrow X = W)$	(from 2. by substitution)
4. $W = X \rightarrow X = W$	(from 3. by modus ponens)

The skill of a mathematician consists in picking an appropriate rule of inference and applying it in such a way that he arrives quickly at the theorem he was trying to prove. This aspect is not covered by the definition, but it is the main theme of the rest of this book.

The theorem proving procedures described in this book follow the following pattern. The hypotheses, Hyp, of the theorem to be proved are added as temporary additional axioms to those, Ax, of the theory we are working in. A proof by is then sought, by adding the negation of the conclusion of the theorem, $Conc$, as an additional temporary axiom, and trying to prove the formula, f. If this succeeds then we have proved

$$Ax \wedge Hyp \wedge \neg Conc \rightarrow f$$

which is equivalent to

$$Ax \rightarrow (Hyp \rightarrow Conc)$$

Procedures based on this pattern are called *refutation systems*. We will see that refutation systems can direct the search for a proof, to some extent, by eliminating some of the more ridiculously inappropriate steps.

In this chapter we introduce two rules of inference, for deriving new theorems from old: *Resolution* and *Paramodulation*. Resolution is the Sheffer stroke of rules of inference: in a refutation system, it is the only rule of inference that is necessary in order to find proofs to all correct arguments. We say that resolution is *complete*. The proof of this, called the *completeness theorem*, is too technical for this introductory chapter, but is given in chapter 16. Paramodulation is introduced as an alternative to the axioms of equality. Both rules of inference apply only to First Order Logic, i.e. variable functions, variable predicates, variable functionals, etc are not allowed. In this chapter then, and until further notice, we restrict ourselves to First Order Logic.

Resolution and Paramodulation apply to only a restricted class of first order formulae, those in *clausal form*. This is not a significant restriction, since all mathematical theories and theorems can be translated into an equivalent clausal form. Again, the proof of this is too technical for an introductory chapter, and is relegated to chapter 15.

Clausal form is a normal form for logical formulae, in which some of the redundancy of the representation is eliminated. The use of a normal form for formulae, assists the computer to find proofs, by eliminating some duplication of effort. It helps avoid the situation where the computer develops two, apparently different, but essentially identical proofs, consisting of sequences of pairwise equivalent formulae. The equivalent formulae translate into the same normal form and the apparent difference between the proofs, disappears.

Definition 5.2 *Clausal Form.*

A set of formulae is in clausal form *iff each formula is a* clause. *A* clause *is a disjunction of literals,*

$$L_1 \vee ... \vee L_n$$

where each literal *is either a proposition or a negated proposition, e.g.*

$$x = 0 \vee x > 0 \vee \neg x = 2$$

is a clause consisting of 3 literals, $x = 0$, $x > 0$ *and* $\neg x = 2$.

Note that clauses contain no quantifiers. Universal quantification is indicated by leaving variables free. Existential quantification is indicated by the introduction of new functions and constants called and .

5.1 The Resolution Rule

What is this resolution rule, about which we have heard so much? Let me start by explaining it in the simplest case, when no substitutions are required, and then gradually deal with more complex cases until we reach resolution in its full glory.

5.1.1 Stage 1 – Variable Free Resolution

Suppose we have two clauses, one containing the literal P and the other the literal $\neg P$, where C' and C'' are the respective remaining disjoined literals. The *resolution* rule enables us to deduce $C' \vee C''$, i.e.

$$\begin{array}{l} 1.\ C' \vee P \\ 2.\ C'' \vee \neg P \\ \hline 3.\ C' \vee C'' \end{array}$$

1 and 2 are called the *parents* of the *resolvant*, 3. P and $\neg P$ are called *complementary literals.*

Exercise 14 *Check that 3 is a logical consequence of 1 and 2, using the techniques of section 3.4.*

It may help you verify this rule in your own mind if you rewrite 1 as $\neg C' \to P$ and 2 as $P \to C''$, deduce $\neg C' \to C''$ and rewrite this to $C' \vee C''$.

For any formula A, $A \vee f$ is equivalent to A, by the semantic tree for $\vee$. Hence we adopt the convention that a disjunction of no literals is equivalent to f. In the definition of resolution above, either C' or C'' or both may contain no literals. We say that they are *empty.* If both C' and C'' are empty then we can derive the *empty clause*, denoted by f. This is the aim of a refutation system, and hence having C' and/or C'' empty is a Good Thing.

Example

The two clauses:

$$\neg y = y \vee \neg x = y \vee y = x \text{ and } \neg y = x$$

contain the complementary literals, $y = x$ and $\neg y = x$, where C' is $\neg y = y \vee \neg x = y$ and C'' is empty. Thus we can resolve the clauses to form:

$$\neg y = y \vee \neg x = y$$

5.1.2 Stage 2 – Binary Resolution

Now the parent clauses may not contain complementary literals, but it may be possible to apply substitutions to each of them so that the resulting clauses do, i.e. we may have clauses $C' \vee P'$ and $C'' \vee \neg P''$ such that $P'\phi' \equiv P''\phi''$ for some ϕ' and ϕ'', where $\equiv$ to means 'is identical with'. In this case we can deduce $(C' \vee C'')\phi'\phi''$. It is usual to combine ϕ' and ϕ'' into a single substitution, ϕ. Since a major objective of resolution is to minimize the amount of substitution going on, we will want ϕ to be the most general substitution such that $P'\phi \equiv P''\phi$. Fortunately, there is only one such, most general ϕ (up to permutations of variables) and there is a procedure for calculating it, given P' and P'', called *unification*. ϕ is called the *most general unifier* of P' and P''. Details of the unification procedure are too technical for this introductory chapter, and are given in chapter 17.

So the rule now is:

1. $C' \vee P'$
2. $C'' \vee \neg P''$

---

3. $(C' \vee C'')\phi$

where ϕ is the most general unifier of P' and P''.

This is called the *binary resolution rule.*

Example

The two clauses:

$$\neg U = V \vee \neg W = V \vee U = W \qquad \text{and} \qquad \neg y = x$$

contain the literals, $U = W$ and $\neg y = x$, respectively, which can be made complementary by the substitution $\{y/U, x/W\}$. Thus we can resolve them to form:

$$\neg y = V \vee \neg x = V$$

The newly derived clause should have its variables *standardized apart* before it is used as a parent of a new resolution, that is, its variables should be given new names, e.g.

$$\neg y = V' \vee \neg x = V'$$

The reason for standardizing apart is to stop spurious coincidences preventing unification, e.g. $p(X)$ and $p(X + 1)$ will not unify, whereas $p(X)$ and $p(Y + 1)$ will.

5.1.3 Stage 3 – Full Resolution

One last wrinkle. It may be possible to eliminate, not just one, but several literals from the parent clause, at a stroke. Suppose the parent clauses are $C' \vee P'_1 \vee ... \vee P'_m$ and $C'' \vee \neg P''_1 \vee ... \vee \neg P''_n$ and ϕ is a most general unifier of $P'_1, ..., P'_m, P''_1, ..., P''_n$ then we may deduce $(C' \vee C'')\phi$, i.e.

$$\begin{array}{l} 1.\ C' \vee P'_1 \vee ... \vee P'_m \\ \underline{2.\ C'' \vee \neg P''_1 \vee ... \vee \neg P''_n} \\ 3.\ (C' \vee C'')\phi \end{array}$$

where ϕ is the most general unifier of all P's and P''s.

This is called the *full resolution rule.*

Example

Consider the two clauses:

$$\neg U = V \vee \neg W = V \vee U = W \qquad \text{and} \qquad \neg y = X \vee \neg Y = x$$

The first clause contains the literal $U = W$, and the second the literals $\neg y = X$ and $\neg Y = x$. These 3 propositions can be unified with the substitution, $\{y/U, x/W, x/X, y/Y\}$. Thus we can resolve the two clauses to form:

$$\neg y = V \vee \neg x = V$$

Exercise 15 *Apply the full resolution rule, in all possible ways, to the following pairs of clauses*

1.	$p \vee q \vee \neg r$	$s \vee \neg p$
2.	$p(X) \vee q(f(X))$	$\neg r(Y) \vee \neg p(f(Y))$
3.	$p(X, a) \vee p(f(a), Y) \vee q(X, Y)$	$r(Z) \vee \neg p(f(Z), Z)$

NB You can apply the rule to pair 3 in three different ways.

5.1.4 Factoring

In some Resolution systems, resolution is divided into two rules: binary resolution and *factoring*, i.e.

$$\begin{array}{l} \underline{1.\ C \vee P_1 \vee ... \vee P_n} \\ 2.\ (C \vee P_1)\phi \end{array}$$

where ϕ is the most general unifier of all the Ps.

This is just a special case of the substitution rule.

Example

Consider the clause:

$$\neg y = X \vee \neg Y = x$$

This contains the literals $\neg y = X$ and $\neg Y = x$, which can be unified with the substitution $\{x/X, y/Y\}$. Thus, with $\neg y = X$ as P_1, $\neg Y = x$ as P_2 and C empty, we can factor the clause to form:

$$\neg y = x$$

Everything we have to say about Resolution systems applies to both versions – the one rule or two rule systems.

5.2 Kowalski Form

Representing clauses as disjunctions of literals is convenient for theoretical discussions, but when we come to do actual resolution proofs (see chapter 6), a variant of clausal form due to Bob Kowalski and called *Kowalski form* will be more natural.

Definition 5.3 *Kowalski Form.*
A clause,

$$\neg P_1 \vee ... \vee \neg P_m \vee Q_1 \vee ... \vee Q_n$$

where $P_1, ..., P_m,\ Q_1, ..., Q_n$ *are propositions and* m *and* n *are* ≥ 0 *is in the, logically equivalent,* Kowalski form, *when it written as:*

$$P_1 \wedge ... \wedge P_m \rightarrow Q_1 \vee ... \vee Q_n$$

That is, all the unnegated literals are collected in a disjunction of propositions on the right of an implication arrow, called the consequent, and all the negated literals are collected in a conjunction to the left of the arrow, called the antecedent.

Some authors draw the arrow pointing to the left with the propositions switched round, i.e.

$$Q_1 \vee ... \vee Q_n \leftarrow P_1 \wedge ... \wedge P_m$$

We prefer the former notation since it preserves the format of the implication arrow.

The meaning of $\rightarrow$ must be extended to deal with the cases where either m or n is 0. When m is 0 we will interpret

$$\rightarrow Q_1 \vee ... \vee Q_n$$

as meaning

$$Q_1 \vee ... \vee Q_n$$

When n is 0 we will interpret

$$P_1 \wedge ... \wedge P_m \rightarrow$$

as meaning

$$\neg\{P_1 \wedge ... \wedge P_m\}$$

When both m and n are 0, we will interpret the empty clause, $\rightarrow$, as f, i.e false.

Exercise 16 *Go back to section 4.2 and check how many of the axioms are clauses in Kowalski form.*

In this form the full resolution rule is.

$$\begin{array}{l} 1.\ H' \rightarrow C' \vee P'_1 \vee ... \vee P'_m \\ 2.\ H'' \wedge P''_1 \wedge ... \wedge P''_n \rightarrow C'' \\ \hline 3.\ (H' \wedge H'' \rightarrow C' \vee C'')\phi \end{array}$$

where ϕ is the most general unifier of the P's and P''s.

To see that this rule is true note that applying ϕ to 1 gives a formula equivalent to

$$H'\phi \wedge \neg C'\phi \rightarrow P$$

where $P \equiv P'_i\phi \equiv P''_j\phi$.

Similarly, applying ϕ to 2 gives a formula equivalent to

$$P \rightarrow \neg H''\phi \vee C''\phi$$

From these we can deduce

$$H'\phi \wedge \neg C'\phi \rightarrow \neg H''\phi \vee C''\phi$$

which is equivalent to 3.

Example

Consider the two clauses:

$$U = V \wedge W = V \rightarrow U = W \qquad \text{and} \qquad y = X \wedge Y = x \rightarrow$$

The first clause contains the proposition $U = W$ in its consequent. and the second the propositions $y = X$ and $Y = x$ in its antecedent. These 3 propositions can be unified with the substitution, $\{y/U, x/W, x/X, y/Y\}$. Thus, with $U = V \wedge W = V$ as H' and C', H'' and C'' empty, we can resolve the two clauses to form:

$$y = V \wedge x = V \rightarrow$$

An especially important class of clauses, both because they arise so often in practice and because simplified theoretical results apply to them are the *Horn clauses*, named after Alfred Horn, who originally isolated them. These are clauses with at most one unnegated literal, i.e those clauses in one of the four forms:

Implication Clause
$P_1 \wedge ... \wedge P_n \rightarrow P$

Goal Clause
$P_1 \wedge ... \wedge P_n \rightarrow$

Assertion Clause
$\rightarrow P$

Empty Clause $\rightarrow$

Exercise 17 *Go back to section 4.2 and check how many of the axioms are Horn clauses.*

5.3 The Paramodulation Rule

We have seen in section 4.2 that the equality axioms are a common ingredient of mathematical theories. Unfortunately, their inclusion in a set of clauses input to a Resolution theorem prover is a source of many difficulties, one of which is illustrated below.

Suppose that we have derived

$$f(a) = b \rightarrow \text{ and } \rightarrow c = a$$

and wish to substitute c for a in $f(a) = b$ to form

$$f(c) = b \rightarrow$$

Using the resolution rule this requires that the term, a, which is to be replaced must be isolated in an equation of its own, which can then be resolved with $\rightarrow c = a$. This requires several unpacking steps: each one of which reduces the depth of the term, followed by a re-orientation and then the main resolution, namely

- resolving with transitivity to derive $f(a) = V \wedge V = b \rightarrow$
- and resolving with substitution to derive $a = V2 \wedge f(V2) = b \rightarrow$.
- The term is now isolated, but needs reorienting using symmetry to $V2 = a \wedge f(V2) = b \rightarrow$, before it can be resolved with $\rightarrow c = a$.
- The resolution with $\rightarrow c = a$ can now take place producing $f(c) = b \rightarrow$.

Thus a lot of rather unnatural looking steps are required to effect a rather simple substitution of equals for equals. One way round this problem is to make the substitution of equals for equals an inference rule of logic, i.e. to introduce a rule

$$\begin{array}{l} 1.\ C(T) \\ 2.\ T = S \\ \hline 3.\ C(S) \end{array}$$

where $C(T)$ is a clause containing a term T and S is another term. This is the simplest case of a rule we will call *paramodulation*.

We will encounter a further problem with the equality axioms: their tendency to increase dramatically the branching rate of search trees. In chapter 7 we will consider this problem and whether paramodulation offers a solution to it.

The first wrinkle we will add to paramodulation is allow only one occurrence of T to be replaced by S, instead of all of them. Several occurrences can then be replaced by repeated application. We will indicate that C contains a distinguished occurrence of T by using square brackets, i.e., $C[T]$.

The second wrinkle is to recognize that the literal defining the substitution, T=S, is unlikely to appear in a clause on its own. It may have other disjuncts and these must be inherited by the derived clause.

The third wrinkle is to allow the substitution literal to be either way round, i.e. either $T = S$ or $S = T$. The paramodulation rule now looks like

$$\begin{array}{l} 1.\ C[T] \\ 2.\ T = S \vee D \ (\mathit{or}\ S = T \vee D) \\ \hline 3.\ C[S] \vee D \end{array}$$

Finally we will let the two occurrences of T in 1. and 2. be unifiable rather than identical.

$$\begin{array}{l} 1.\ C[T'] \\ 2.\ T = S \vee D \ (\mathit{or}\ S = T \vee D) \\ \hline 3.\ (C[S] \vee D)\theta \text{ where } \theta \text{ is the most general unifier of} \\ \qquad\qquad\qquad T' \text{ and } T. \end{array}$$

The paramodulation rule can be used to replace nearly all the equality axioms, that is if we allow derivations using paramodulation as well as binary resolution and factoring then the equality axioms of , and can be omitted. The only axiom we must retain is , $X = X$. In addition we must add a family of new axioms, called the *functional reflexive axioms*, namely

$f(x_1, ..., x_n) = f(x_1, ..., x_n)$
for each n-ary function f

The functional reflexive axioms are something of an embarrassment to researchers in theorem proving: repeated attempts to prove the completeness of paramodulation + resolution without them have failed; but everyone's conviction is that they are really superfluous. The paramodulation completeness proof is complicated enough even with the functional reflexive axioms and we omit it (it is given in [58]). The conviction that completeness can be proved without these extra axioms is called the *Paramodulation conjecture*. For an attempted proof of the result, now believed to be faulty, see [57].

5.4 Summary

We have defined a formal notion of proof as a sequence of formulae, starting with axioms and continuing with theorems derived from earlier members of the sequence by rules of inference. We have defined two such rules of inference: resolution and paramodulation. The first of these rules is sufficient to prove all correct arguments. The proof of this is given in chapter 16. The second is an alternative to the equality axioms.

To prove theorems by resolution, they and the axioms must be put into clausal form. Most of the axioms of section 4.2 are already in clausal form. A procedure for putting other formulae into clausal form is given in chapter 15.

Application of the resolution rule involves the unification of two or more literals, i.e. the calculation of the most general substitution which unifies the two literals to a common instantiation. A procedure for making this calculation is given in chapter 17.

Chapter 6

Searching for a Refutation

- In this chapter we consider how to use resolution and paramodulation to search for a refutation.
- Section 6.1 introduces the linear format for representing refutations as paths and defines the linear and input refinements.
- Section 6.2 shows how to represent the search for a refutation as a tree.
- Section 6.3 defines Lush Resolution.
- Section 6.4 defines subsumption and discusses loop prevention.
- Section 6.5 defines and contrasts forwards and backwards search.
- Section 6.6 discusses the difficulties with non-Horn clauses and introduces case analysis and ancestor resolution.
- Section 6.7 defines various search strategies: depth first 6.7.1, breadth first 6.7.2 and heuristic 6.7.3.

Now we have some rules of inference and some axioms, we can consider putting them together to generate some proofs. In this chapter we will consider the problems of getting a computer program to do this.

We will start by considering a simple resolution proof: the proof of the axiom for = from the axiom and a slightly twisted version of the axiom (see section 4.2.1). The reflexivity and twisted transitivity axioms, put in Kowalski form are:

1. $\rightarrow X = X$
2. $U = V \wedge W = V \rightarrow U = W$

These are just as in section 4.2.1 except that the variables have been standardized apart and the second literal of the transitivity axiom has had its parameters switched around.

The conjecture we want to prove is the symmetry axiom $X = Y \rightarrow Y = X$. To prove a conjecture in a refutation system it must be: negated, put in , added to the axioms, and the derived. To ensure that the is done correctly, the conjecture must first have all its free variables bound by s, e.g.

$$\forall X\, \forall Y \;\; X = Y \rightarrow Y = X$$

This process is called forming the . The conjecture can then be negated.

$$\neg \forall X\, \forall Y \;\; X = Y \rightarrow Y = X$$

To put this formula in clausal form: the negation sign must be brought inside the universal quantifiers, changing them to s;

$$\exists X\, \exists Y \;\; \neg\{X = Y \rightarrow Y = X\}$$

the existentially bound variables, X and Y, must be replaced with new constants, x and y, respectively;

$$\neg\{x = y \rightarrow y = x\}$$

the negation sign must be taken inside the arrow;

$$\neg\{\neg x = y \vee y = x\}$$
$$\neg\neg x = y \wedge \neg y = x$$
$$x = y \wedge \neg y = x$$

and the formula separated into two clauses in Kowalski form.

3. $\rightarrow x = y$ and
4. $y = x \rightarrow$

A general procedure for transforming a formula into clausal form is given in chapter 15 together with a proof that a formula has model iff its clausal form has a model. These give a technical justification of the above steps.

An intuitive justification is as follows. In order to prove a conjecture of the form $P \rightarrow Q$, we assert $\rightarrow P$ as a new axiom and set up $Q \rightarrow$ as a goal to be proved. Free variables in conjectures, like X and Y, are regarded as 'typical' objects, for which, if P is true then Q is true. In a refutation system, such 'typical' objects cannot be represented by free variables, or else there is a danger that a faulty 'proof' will be found. For instance, if clause 4 were

$4'$. $Y = X \rightarrow$

then the empty clause could be derived by resolution with either

1. $\rightarrow X = X$ or even $\rightarrow 0 = 0$

without reference to the other axioms. To prevent this kind of faulty 'proof' we must represent 'typical' objects by constants, but not existing constant, like 0 or 3. We must create new constants, x and y, about which nothing is already known. x and y are called *Skolem constants* after Thoralf Skolem who first invented this device. We will adopt the convention of using the lower case version of a existential variable when converting it into a Skolem constant.

We are now in a position to derive a proof of the conjecture. We have four axioms: 1, 2, 3 and 4. 4 and 2 can be resolved to form

5. $y = V' \wedge x = V' \rightarrow$

and 3 can be resolved with this to form

6. $y = y \rightarrow$

$$\begin{tikzcd} 4.y = x \rightarrow & \circ \arrow[d, "2."] \\ 5.y = V' \wedge x = V' \rightarrow & \circ \arrow[d, "3."] \\ 6.y = y \rightarrow & \circ \arrow[d, "1."] \\ 7. \rightarrow & \circ \end{tikzcd}$$

Figure 6.1: The Refutation as a Path

Finally, 1 can be resolved with 6 to form

7. $\rightarrow$

the empty clause.

Notice how this refutation takes the result of each resolution and resolves this in turn with one of the original clauses. We can represent this situation with the in figure 6.1 where the s are labelled with the goal clause and the subsequently derived clauses and the s are labelled with the numbers of the original clauses.

6.1 Following Your Nose

This kind of refutation is very natural. It models, in a simple way, the 'mental set' of a real mathematician who takes the original conclusion of a conjecture and, working backwards, worries away at it with axioms and hypothesis. It is called *Linear, Input Resolution*. *Linear* because we keep using the last resolvent as the parent of the next resolution and so can represent the refutation as a linear path: we do not resolve 4 and 2 and then, say 1 and 2 *Input* because we always resolve the last resolvent with one of the original, input clauses, not with a derived clause, e.g. having derived 6 we may resolve it with 1, 2, 3 or 4, but not with 5.

Linear, Input Resolution is an example of a *refinement* of full resolution. Since it excludes various possibilities it is inherently more efficient than full resolution, i.e. it involves less search. It is complete for sets of Horn clauses (like those in our example), i.e. it will find a refutation if there is one. This follows form the Horn clause completeness of Lush Resolution below. For non-Horn clauses we must relax the Input constraint, but we can keep the Linear one. A non-Input resolution, e.g. 6 with 5, is called *ancestor resolution*, because 6 resolves with one of its ancestors higher up the path.

$$\begin{tikzcd}[row sep=large]
4. y = x \rightarrow & \circ \arrow[d, "2."] \\
5. y = V' \wedge x = V' \rightarrow & \circ \arrow[d, "1."] \\
8. x = y \rightarrow & \circ \arrow[d, "3."] \\
9. \rightarrow & \circ
\end{tikzcd}$$

Figure 6.2: Another Refutation Path

$$\begin{tikzcd}[row sep=large]
4. y = x \rightarrow & \circ \arrow[d, "2."] \\
5. y = V' \wedge x = V' \rightarrow & \circ \arrow[d, "1."] \\
10. y = x \rightarrow & \underset{\mathrm{x}}{\circ}
\end{tikzcd}$$

Figure 6.3: A Loop

6.2 Representing Choice

Of course, the path in figure 6.1 is not the only possible one. Figure 6.2 is another one (which succeeds) and figure 6.3 another (which represents a loop).

We can fit all these paths together to make a new kind of , a *search tree*.

Once again trees come to our aid as a descriptive device. This time to describe the process of searching for a refutation.

Exercise 18 *Represent the applications of clause 2 to clause 5 by labelling the two unlabelled arcs.*

6.3 AND choices and OR choices

Consider the choices represented by the five way branching from clause 5 in figure 6.4. There are really two kinds of choice packed together here:

- the choice of which literal (or in general, which group of literals) to resolve away;
- the choice of which clause to resolve the literal away with.

If the first literal is chosen then it can be resolved away in two ways: with 1 to yield

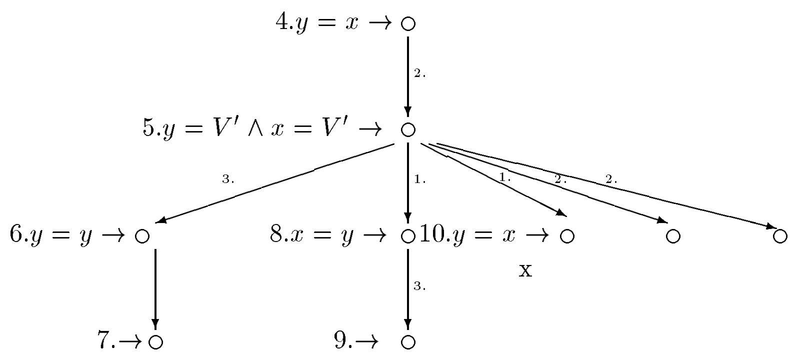

Figure 6.4: A Search Tree

$$x = y \rightarrow$$

or with 2 to yield

$$y = V'' \wedge V' = V'' \wedge x = V' \rightarrow$$

If the second literal is chosen then it can be resolved away in three ways: with 1, 2 or 3.

These two kinds of choice are essentially different.

- The choice of literal is only a question of order: all literals must eventually be resolved away, we may only choose what order to do this in. We call such a choice an *AND* choice, because we must first choose one clause AND then another.

- The choice of which clause to resolve with is a genuine choice. If this branch of the tree leads to the empty clause then the other branch does not need to be developed. We call such a choice an *OR* choice, because we may choose either one clause OR another.

AND choices are innocuous. It does not affect the completeness of Linear Input Resolution in what order the literals of a clause are resolved away. So we can pick an order, either at random or exercising extreme cunning, and stick to it. This means that branch points representing AND choices can be pruned from our search tree. The resulting tree for our running example is given in figure 6.5. The literal selected to be resolved away next is underlined.

OR choices are not innocuous. They are a (the?) major source of difficulty in automatic theorem proving. We will address this difficulty in section 6.7.

Thus if our original clauses are all Horn clauses we may apply Linear Input Resolution with the following additional restriction; that a literal be selected from the current clause and that any resolutions between the current clause and the input clauses must involve that selected literal. The resulting Resolution restriction is called *Lush Resolution* [32]. Lush stands for **L**inear Resolution with **u**nrestricted **s**election function for **H**orn clauses. In the case of non-Horn clauses the choice of selected literal is restricted to be one of the most recently introduced. This restriction is called *SL Resolution* [37]. SL stands for **S**elected **L**iteral.

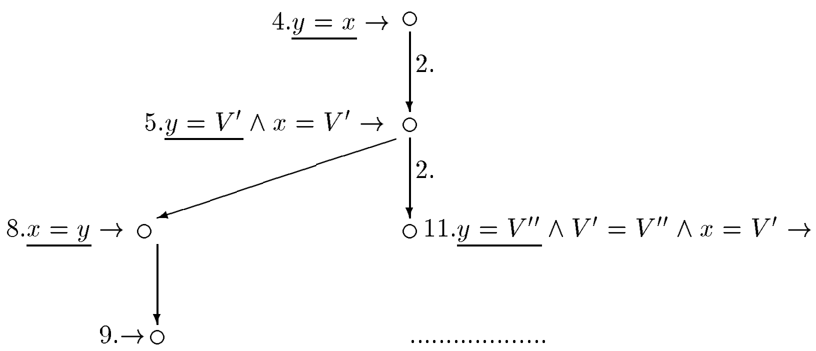

Figure 6.5: A Lush Resolution Search Tree

6.4 Preventing Looping

In figure 6.3 we give an example of a loop: the derivation of a clause identical to one derived previously. There is no point continuing to develop this clause, since it can only lead to clauses which could already have been derived from its predecessor: we will merely be duplicating work. We will certainly want to build a loop checker into our theorem prover, i.e. a piece of program which detects loops and prevents them from being developed further.

In fact, we can go further than this. Suppose the clauses

$$y = x \rightarrow \tag{i}$$

and

$$Y = x \rightarrow \tag{ii}$$

have been derived. Now since (i) is an instantiation of (ii) anything derivable from (i) can also be derived from (ii). Thus (i) is a kind of loop and further derivations from it should be blocked. (i) is said to be *subsumed* by (ii) and the resolution restriction of blocking further development of subsumed clauses is called *subsumption checking*.

The clauses

$$y = x \wedge a > b \rightarrow \tag{iii}$$

and

$$y = x \rightarrow \tag{iv}$$

exhibit another kind of subsumption relationship. Any derivation of the empty clause from (iii) can be pruned into a derivation of the empty clause from (iv) by deleting occurrences of $a > b$ and those resolutions required to resolve it and its descendants away. Thus (iii) cannot produce any refutations that cannot be produced better from (iv) and further derivations from it can be blocked without loosing completeness.

Definition 6.1 *In general, a clause, C, subsumes a clause, D, if there is a substitution θ such that*

$$D \leftrightarrow C\theta \vee E$$

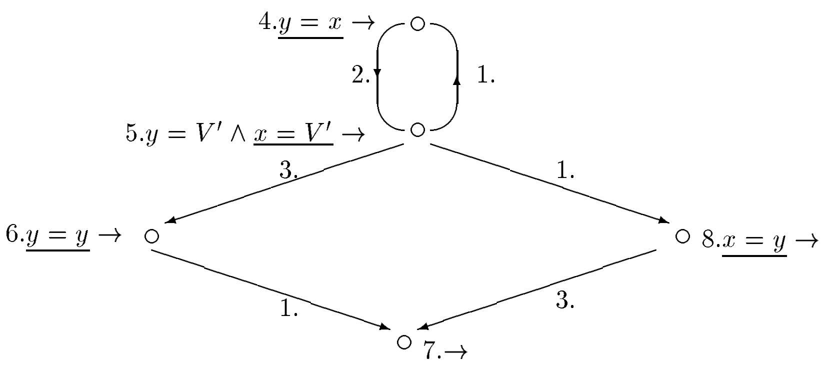

Figure 6.6: A Search Graph

For instance,

$$\neg y{=}x \leftrightarrow \neg(Y{=}x)\{y/Y\} \vee f$$
$$\neg y{=}x \vee \neg a > b \leftrightarrow \neg(y{=}x)\{\} \vee \neg a > b$$

Subsumption checking preserves the completeness of most Resolution restrictions provided care is taken when the subsuming and subsumed clauses are identical. In this case the second clause derived must be the one whose further development is blocked or the looping may continue. Otherwise, it does not matter whether the subsumed or subsuming clause is derived first.

Because we have chosen to represent the process of searching for a refutation with search *trees*, a clause which can be derived in more than one way is represented by more than one node. If we decide to identify nodes in the tree labelled by the same clause, then the resulting structure is called a *search graph*. [1] Figure 6.6 is a graphical representation of the tree in figure 6.4.

When a loop check is built into the theorem proving program the graph representation of the search is particularly appropriate.

6.5 Choosing Where to Start

In all our example refutation paths we started with 4 as the *top clause*, so this is the of the tree in figure 6.4. Starting with 4 seemed like a good idea. Since 4 was inherited from the conjecture, it seemed bound to be involved in any refutation. Using it as the top clause guaranteed that we didn't try to make a refutation involving just 1, 2 and 3. In fact, all four clauses are bound to be involved, so it would have been as good to start with any one of them.

–1,2,3,4″ form a *minimally unsatisfiable set* of clauses. That is, they are unsatisfiable, but dropping any one of them will result in a satisfiable set. It is not possible to derive the empty clause from a satisfiable set of clauses, so 1, 2, 3 and 4 must all be involved in any refutation.

Suppose that the theorem prover were provided with an additional axiom, e.g.

[1] Contrary to common expectations the branches of a wooden tree can be made to rejoin by careful grafting. Maybe this is the origin of the term graph.

5. $\to X + 1 > X$

It would not have been a good idea to use 5 as top clause, because no other clause will resolve with it, and so the empty clause will not be derived. Put in another way, 5 is not a member of a minimally unsatisfiable subset of –1,2,3,4,5″. The top clause should always be chosen from a minimally unsatisfiable set. If a minimally unsatisfiable set cannot be detected then it is necessary to try each clause in turn as top clause. Only if all these refutation attempts fail can we be sure that the conjecture is not a theorem.

If a minimally unsatisfiable set can be detected then we only need try one of its members as top clause. If this refutation attempt fails then the conjecture is not a theorem. This is called the *set of support* strategy.

The usual trick for choosing the top clause, is the one we used, of choosing a clause inherited from the conjecture, e.g. 3 or 4. However, such a clause is not guaranteed to belong to a minimally unsatisfiable set, because the conjecture may not be theorem, or the clause we choose may not play a role in any proof. A trick like this, which is not *guaranteed* to succeed, but probably will, is called a *heuristic*.

Exercise 19 *Draw a Lush Resolution search tree, of at least size 6, with clause 3, $\to x = y$, as top clause.*

When a goal clause, like 4, is chosen as top clause, the resulting refutation is called a *backwards proof* and the search process a *backwards search*, because they start from the conclusion of the theorem and work backwards to the axioms and hypothesis of the theorem.

When an assertion, inherited either from the theorem, like 3, or an axiom, like 1, is used as the top clause the refutation and search process are called a *forwards proof* and a *forwards search*, respectively.

6.6 Non-Horn Clauses, Case Analysis and Ancestor Resolution

To demonstrate the incompleteness of Lush Resolution when there are non-Horn clauses, consider the following example.

1. $\to natural(n) \vee non\_neg(n)$
2. $natural(n) \to non\_neg(n)$
3. $non\_neg(n) \to natural(n)$
4. $natural(n) \wedge non\_neg(n) \to$

$natural(n)$ means n is a natural number. $non\_neg(n)$ means n is a non-negative number. The clauses can be interpreted as a request to show that n is both a natural number and a non-negative number (clause 4) under the assumptions that: it is either one or the other (clause 1); if it is natural then it is non-negative (clause 2); and if it is non-negative then it is natural (clause 3). Clearly, this should be possible.

However, clause 1 is a non-Horn clause. Let us choose the goal clause, 4, as top clause and consider the partial refutation in figure 6.7

$$
\begin{array}{rc}
4. natural(n) \wedge \underline{non-neg(n)} \rightarrow & \circ \\
 & \Big\downarrow 2. \\
5. \underline{natural(n)} \wedge \underline{natural(n)} \rightarrow & \circ \\
 & \Big\downarrow 1. \\
6. \rightarrow \underline{non-neg(n)} & \circ \\
 & \Big\downarrow 3. \\
7. \rightarrow \underline{natural(n)} & \circ
\end{array}
$$

Figure 6.7: A Non-Horn Clause Partial Proof

Clause 7 will only resolve with input clauses 2 or 4. Resolution with 2 will result in $\rightarrow non\_neg(n)$, which is a loop. Resolution with 4 will result in $non\_neg(n) \rightarrow$, which is a symmetrical situation to the one we now face. Similar situations occur on all the other branches and so the empty clause cannot be derived.

Exercise 20 *Develop the complete, Lush Resolution, search tree for clauses 1–4, using 4 as the top clause. Check that the tree is finite, but does not contain the empty clause.*

The partial refutation in figure 6.7 has several interesting features.

- Note the use of full resolution to resolve away two occurrences of $natural(n)$ simultaneously.
- Note also that clause 6 and 7 are not goal clauses, i.e. they have non-empty consequents. If the top clause is a goal clause and the input clauses are Horn clauses then this situation cannot arise. It has happened here because of the resolution with the non-Horn clause 1.

One way to deal with the incompleteness of Lush Resolution is to allow *case analysis* by splitting clause 1 into two clauses,

$1a.$ $\rightarrow natural(n)$
$1b.$ $\rightarrow non\_neg(n)$

and seeking two Lush Resolution refutations, one from clauses $1a$, 2, 3 and 4, and one from clauses $1b$, 2, 3 and 4. The resulting refutations are shown in figure 6.8.

An alternative technique is to relax the input restriction to allow ancestor resolution, i.e. the resolution of the current clause with on of its own ancestors. This allows us to resolve clause 7 in figure 6.7 with its ancestor, clause 5, to derive the empty clause. The result is exhibited in figure 6.9. One way to think of this refutation is as a combination of the two case analysis refutations of figure 6.8. The steps of the two case analysis refutations are input resolutions with 2 and $1a$ and with 3 and $1b$. Inverting the second pair, and merging $1a$ and $1b$ to give 1, gives a sequence of resolutions with 2, 1 and 3. These are the

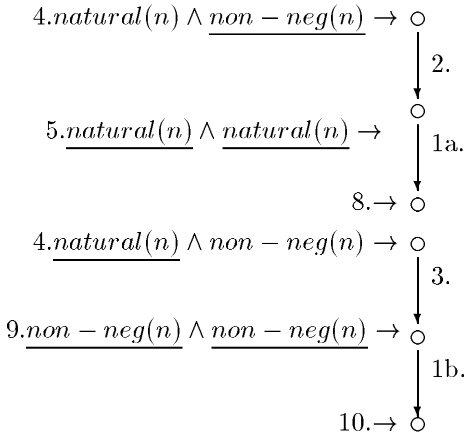

Figure 6.8: Proof by Case Analysis

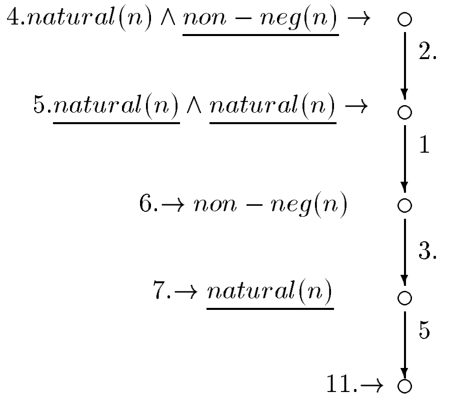

Figure 6.9: An Ancestor Resolution Refutation

steps of the ancestor resolution refutation. The use of ancestor resolution in the last step can be read as the ability to assume $\neg natural(n)$ in the second case, since the case where $natural(n)$ has already been dealt with in the first case.

6.7 How to Make OR Choices

In a search tree, every node with more than one daughter represents a choice. Following one path may lead to success (the empty clause). Following another may lead to failure (only loops and non-empty clauses without daughters). How can we choose? Or rather how can we instruct our computer programs to choose?

We could let them use random number generators – a sort of computational coin toss. The programs would then be simulating one of the 'mysterious' processes mentioned in the introduction – luck! (Certainly the easiest of the 'mysterious' processes to simulate.) Or we could try to find heuristics to guide the search, and simulate instead the remaining 'mysterious' processes, of experience and intuition. Finding such heuristics is certainly the hardest problem in automatic theorem proving!

6.7.1 Depth First Search

Let us start by following up the ideas of 'mental set' discussed in section 6.1. We claimed that a natural way to search for a refutation was to take the last clause derived and employ it as the parent of the next resolution. This suggests a *search strategy* in which a path is pursued until this development is blocked in some way: either by success, the derivation of the empty clause: or by failure, a loop having been detected or no further resolution being possible. In the case of failure we will then want to *back-up* and try some other path. If we stick to our desire to model mental set we will want to retreat as little as possible – only as far as the last choice point – then plunge on down again.

The search strategy just described is called *depth first search*. It is illustrated in figure 6.10. Each node is labelled with a letter indicating the order in which it was developed. (The actual clauses are irrelevant here and are omitted.) A cross represents some kind of failure, causing back-up. Back-up from node F passed through nodes E and C, back to B, indicating that no further choices were available at E and C.

Depth first search leaves mostly unanswered the central question of automatic theorem proving: how the choices are to be made. It gives no grounds for the choice of developing C before G, but once this is made it forces the choice of F before H. The unforced choices may be made arbitrarily or with extreme cunning: the search will still be depth first provided two criteria are met.

- The current path is pursued until further progress is blocked.
- Back-up is always to the latest choice point.

6.7.2 Breadth First Search

Not only does depth first search fail to settle the central question, it fails to answer two simpler questions.

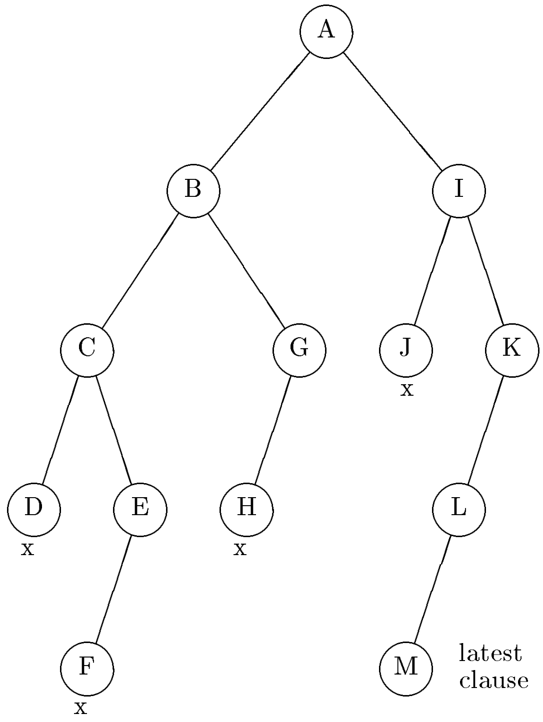

Figure 6.10: Depth First Search

- *Complete Search*
  How can the tree be searched so that every node is eventually reached? In depth first search we may pursue an infinite branch indefinitely and never back-up to the rest of the tree. Thus we may choose a complete inference system (e.g. Linear Resolution), but still invalidate our guarantee of being able to prove all theorems, if we then choose an *incomplete* search strategy like depth first search.

- *Shortest Proofs*
  How can the tree be searched so that the shortest refutation is found first? In depth first search we may find a solution on one branch, but a shorter refutation may be lurking, undiscovered, on another, unsearched branch.

One answer to both these questions is provided by *breadth first search*. In breadth first search all the nodes at depth 1 are developed first, then all the nodes at depth 2 and so on (see the illustration in figure 6.11 in which the labels on the nodes again illustrate the order in which they were developed).

A breadth first search strategy develops the search tree in stages: at stage n all nodes of depth n are developed. Since there are only a finite number of nodes with depth n, stage n will terminate eventually, and stage $n+1$ will commence. Hence,

- All nodes will eventually be developed. A node of depth n, will be developed during stage n.

- The first refutation to be found will be shorter than any other. If the length of the shortest refutation is n then it will be found at stage n

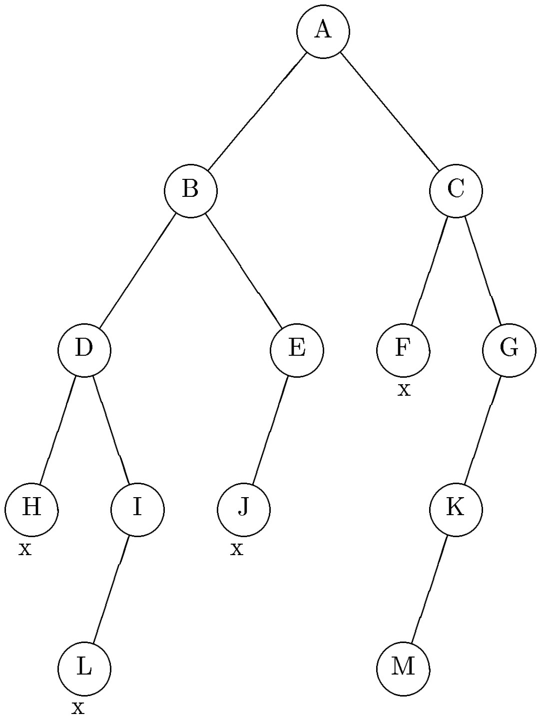

Figure 6.11: Breadth First Search

before any other refutation is found.

These arguments confirm that breadth first search answers the questions about complete search and shortest refutations. It also provides a partial answer to the central question – each node is developed during stage n – the only remaining choice is the order of development of nodes within each stage.

Thus breadth first search provides answers to all our questions, but not very intelligent answers. The strategy has no psychological plausibility. People never search for refutations like this. Besides the abandonment of 'mental set' – the flipping about from one path to another – this method has huge memory overheads. At stage n, all the nodes of depth n must be remembered. The number of such nodes usually grows exponentially! For instance, if each node has approximately 10 daughters there will be of the order of a million nodes at depth 6. This is quite a problem for computers too. They soon get bogged down in the memory overhead.

6.7.3 Heuristic Search

Depth first and breadth first search are both *uninformed* search strategies: that is, they both search the search tree without regard to the particular problem being solved. In this section we consider how knowledge about the problem can be used to influence the direction of the search.

When a human is searching for a refutation he sometimes has a feeling he is on the right track and sometimes that he is hopelessly lost. Some steps seem

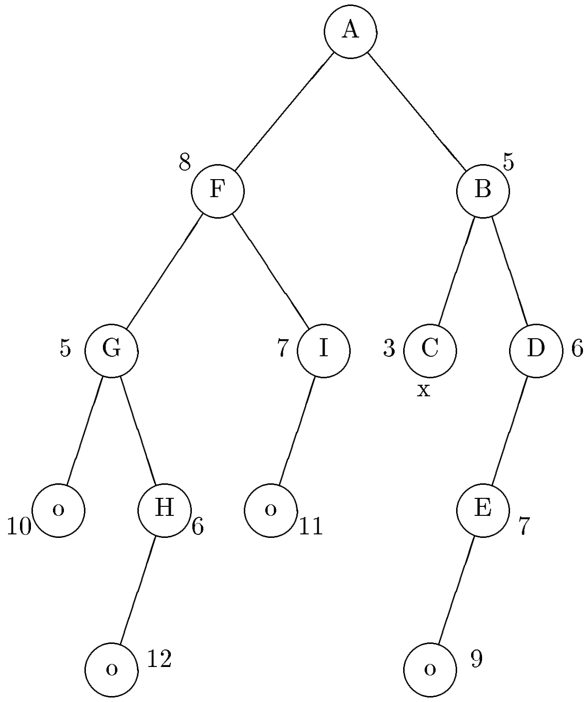

Figure 6.12: Heuristic Search Tree

to take him closer to the goal and some further from it. How can we capture these feelings and use them to guide the search?

One simple and crude technique is to associate a number with each node of the search tree, called its *score*, where the smaller the score the better the node. Whenever we have a choice, of which node to search next, we choose the node with the smallest score. This search strategy is called *heuristic search*. It is illustrated in figure 6.12. The score of each node is written beside it and the letters, as usual, indicate the order in which the nodes are developed. Lower case o, indicates a node which is *open*, i.e. the undeveloped daughter of a developed node. Note that, at each stage, the next node developed is always the open node with the lowest score. In programming heuristic search, open nodes are usually stored on a list, called an *agenda*, in increasing order of score. The node to be developed next is, thus, the first node on the agenda.

Exercise 21 *Which node should be developed next in figure 6.12?*

Where do the numerical scores come from? The score is assigned to a node by applying to it an *evaluation function*. The evaluation function is heuristic. It may be general purpose or special purpose, i.e. it may be useful for guiding the search for the refutation of any conjecture from any set of axioms or it may exploit special properties of the domain it is used in.

Length of clause is an example of a general purpose evaluation function. The *length* of a clause is the number of literals it contains. The idea of this evaluation function is that the empty clause has length 0 and that the shorter the clause the fewer literals have to be resolved away to derive the empty clause – thus the shorter the better. Suppose the current clause were

$$y = V' \wedge x = V' \rightarrow$$

and we can resolve the first literal with the reflexivity axiom to produce

$$x = y \rightarrow \tag{v}$$

or with the twisted transitive axiom to produce

$$y = V'' \wedge V' = V'' \wedge x = V' \rightarrow \tag{vi}$$

Now clause (vi) has length 3, whereas clause (v) only has length 1 and hence will be developed first by heuristic search. Conjectures can be found in which this would be the correct choice and others for which it would be the wrong choice, but on the whole it is sensible practice.

A more elaborate evaluation function could be defined by assigning a subscore to each literal based on its complexity and then summing these subscores. The depth of a literal considered as an expression tree is one simple measure of its complexity. This is sometimes called the *depth of function nesting*. The depth of $x = y$ is 1 and the depth of $x + 1 = 2 \cdot y - 1$ is 3, so that the score of

$$x = y \wedge x + 1 = 2 \cdot y - 1 \rightarrow$$

is 4.

We could get more elaborate still, by assigning different subscores to different functions and predicates, e.g. $=$ might rate 1, $+$ and $-$ might rate 1 and $\cdot$ might rate 2; expressing a conviction that literals containing $\cdot$ are hardest to resolve away and those containing only $+$ are easier, etc. Such an evaluation function is special purpose because it relies, in part, on heuristics specific to the domain of arithmetic.

To find out more about heuristic search; including what conditions must be placed on the evaluation function to guarantee that shortest refutations are always found first; see [51].

6.8 Summary

We have been considering how to search for the refutation of a theorem by applying rules of inference to formulae in different ways. Such a search process can be described with the aid of search trees and search graphs.

In particular, we have looked at refining resolution by only considering refutations which obey the linear, input, selected literal or subsumption restrictions. In the case of non-Horn clauses we saw that the input restriction was too strong and that ancestor resolution had to be allowed. We saw that an additional rule of inference, paramodulation, made refutations involving equality shorter and more natural.

We also looked at various search strategies including: backwards versus forwards search; and depth first, breadth first and heuristic search.

Further Reading Suggestions for Chapters 5 and 6

The last two chapters have only scratched the surface of the immense amount of work on uniform proof procedures. [16] gives a more comprehensive survey. [51] is a general introduction to Artificial Intelligence problem solving techniques.

Chapter 7

Criticisms of Uniform Proof Procedures

- This chapter reviews and evaluates the contribution of resolution type proof procedures to mathematical reasoning.
- Section 7.1 lists the ways in which mathematical logic has contributed to mathematical reasoning.
- Section 7.2 uses a group theory example to illustrate the combinatorial explosion.
- Section 7.3 describes various uniform techniques for avoiding or overcoming the combinatorial explosion, including a discussion of whether additional rules of inference, e.g. paramodulation, reduce the amount of search (7.3.1), and a discussion of ways of disguising how bad things are (7.3.2).
- Section 7.4 begins a discussion non-uniform guidance techniques, starting with an analysis of a human proof of the group theory example.
- Section 7.5 formalizes the knowledge used in the human proof.
- Section 7.6 generalizes the last two sections into a new methodology for mathematical reasoning.

The previous chapters have described some of the work done in 'Automatic Theorem Proving' during the 1960s, especially the resolution and paramodulation rules of inference and the formalization of mathematical knowledge using the Predicate Calculus. The time has now come to take stock: to consider how effective these techniques are in enabling us to build artificial mathematicians; and to present some of the criticisms of such 'uniform proof techniques' levelled by other workers in Artificial Intelligence.

7.1 The Contribution of Logic

Automatic Theorem Proving was based on the solid foundation of earlier work in Mathematical Logic. This is not surprising: both fields had similar aims.

Mathematical Logic was concerned to justify mathematical activity, that is to analyse and explain the existing proofs of Mathematics; relating fields together, discovering and correcting inconsistencies and omissions etc. Automatic Theorem Proving was concerned to explain how proofs could be discovered in the first place. Though similar, these aims are different, and we cannot expect that Logic should provide all the answers, or even that it should suggest the right approach, to the problem of explaining proof discovery.

Let us start by listing what I take to be the major contributions of Mathematical Logic to Automatic Theorem Proving.

1. The notation of Logic (see chapters 2, 3 and 4) gives us a precise language, suitable for describing mathematical knowledge to a computer.

2. The semantics of this language. This enables us to relate the notation to actual pieces of mathematics and to eliminate faulty representations (see especially section 4.3).

3. The definition of a proof as a sequence of applications of rules of inference to axioms or previously proved theorems. This enables us to be precise about what we are hoping to discover. Before the advent of Logic a proof was any convincing argument.

4. Hilbert's claim that any mathematical activity can be regarded as finding proofs in some formal system. For instance, equation solving can be regarded as theorem proving where our main interest is in the substitutions for variables on route. This gives theorem proving a central role in mathematical reasoning.

5. The standard axiomatizations of many mathematical theories. In section 4.2 we gave the axiomatizations of equality, groups and arithmetic and used some of these axioms to demonstrate the workings of the resolution and paramodulation rules.

We will see that the value of these contributions diminishes as we move down the list. Numbers 4 and 5 are especially suspect and two major themes of the chapters ahead will be: the investigation of areas of mathematical reasoning other than theorem proving; and the provision of non-standard axiomatic theories.

However, they have provided a useful first approximation to a solution. They have enabled us to design theorem proving systems and prove elementary theorems artificially. To improve on this and move on to a second approximation we must turn our attention to an actual resolution proof attempt and see in what ways it goes wrong.

7.2 A Resolution Proof and the Combinatorial Explosion

Consider the theorem in group theory that

> Every group of exponent 2 is abelian.

This problem can be formalised in a way suitable for solution by a Resolution theorem prover by taking the clausal form of

(a) the axioms of equality,

(b) the remaining axioms of group theory,

(c) the axiom, $X \circ X = e$,

(d) and the negation of the commutativity axiom, $A \circ B = B \circ A$. (This is the goal clause, $a \circ b = b \circ a \rightarrow$.)

and trying to derive the empty clause from them using the resolution rule.

The complete set of clauses is:

1. $\rightarrow X1 = X1.$
2. $X2 = Y2 \rightarrow Y2 = X2.$
3. $X3 = Y3 \wedge Y3 = Z3 \rightarrow X3 = Z3.$
4. $X4 = Y4 \wedge Z4 = W4 \rightarrow X4 \circ Z4 = Y4 \circ W4.$
5. $X5 = Y5 \rightarrow i(X5) = i(Y5).$
6. $\rightarrow X6 = e \circ X6$
7. $\rightarrow X7 = X7 \circ e.$
8. $\rightarrow e = X8 \circ i(X8)$
9. $\rightarrow e = i(X9) \circ X9.$
10. $\rightarrow X10 \circ (Y10 \circ Z10) = (X10 \circ Y10) \circ Z10.$
11. $\rightarrow X \circ X = e.$
12. $a \circ b = b \circ a \rightarrow .$

The empty clause can be derived from these clauses after 42 resolutions starting with number 12 as the top clause. In figure 7.1 we give the very beginning of the search tree generated by Lush Resolution.

From this it can be seen that the search tree rapidly grows as the depth increases. In fact, the average number of daughters per parent, or the *branching rate*, is 3. At this stage only the equality axioms are applicable. Later on some of the group theory axioms get a look in and the branching rate goes up.

Even with this, rather conservative, estimate of a branching rate of 3, there will be 3^{42} nodes at depth 42, where the first empty clause can be found. This is a very big number! It is of the order of 10^{21}. That number of clauses could not be stored in any one of todays digital computers, so a breadth first search strategy is obviously out of the question. A depth first or a heuristic search is feasible, but is unlikely to find the proof unless it receives very cunning guidance. There are too many losing choices available. This rapid growth of the size of the search tree with the depth is called the *combinatorial explosion*. It is a stick with which to beat uniform proof procedures.

This particular theorem has a special place in my heart. When I first got the opportunity to feed theorems to an automatic theorem prover this was the theorem I fed. It seemed about the right level of difficulty. Not trivial, but easy enough to be the sort of theorem to give to beginning students of group theory. Imagine my surprise when the Lush Resolution theorem prover I was using had found no proof after 3/4 hour (I turned it off to put it out of its misery). I printed out the partial search tree to see what had gone wrong. All the steps were perfectly legal, but that was about all you could say for them! It was clear

Figure 7.1: Lush Resolution Search Tree to Depth 2

that without guidance the theorem prover would do the most amazingly silly things.

My experience is pretty standard. An unguided theorem prover can prove trivial theorems by sheer brute strength, but collapses under the weight of anything the least bit difficult. It is hard to imagine, in advance, what silly things it might try to do. We humans tend to be so prejudiced in favour of the proof we know is in there, that we overlook most of the blind alleys.

Let us look at some of the silly things which are going on even in the little bit of search tree in figure 7.1.

- The path marked in heavy type consists of repeated applications of the transitivity axiom. The effect of these is to break equations into two, inserting a new variable in the openings thus created. We saw in section 5.3 that such a move is occasionally essential. However, repeated applications to no purpose are silly. They create a situation a little like looping, but which, unfortunately, cannot be picked up by a conventional subsumption check.

- The application of clause 4 to the top clause creates the goals $a = b$ and $b = a$. These are certainly unprovable, since a and b are arbitrary, and hence unequal, constants. It is silly to pursue this branch any longer. (In chapter 10 we will describe a mechanism for automatically pruning such branches.)

7.3 Attempts to Guide Search

Clearly, a theorem prover which is going to tackle non-trivial theorems needs some guidance. A variety of techniques have been explored for providing such guidance. Let us look at some.

The most common way is to introduce some completeness preserving restriction to basic resolution. We have already seen some of these: The Linear and Input restrictions and the selection of literals; The use of set of support and the deletion of subsumed clauses. In fact we employed all of these in the search tree of figure 7.1 and still ran into trouble.

Another technique is to expand the inference system in the hope of finding shorter proofs or of decreasing the branching rate by being able to delete axioms. The introduction of the paramodulation rule was such an expansion. Let us see what effect its introduction has on our group theory example.

7.3.1 Paramodulation

The use of paramodulation telescopes our 42 step resolution proof into a 10 step proof. This is because, as shown in section 5.3, several resolution steps are required to make an equality substitution, and these steps collapse into one paramodulation step. The actual paramodulation plus resolution proof is given in figure 7.2.

A reduction from 42 steps to 10 steps is very good news and encourages us to think that the the automatic proof of the theorem may now be within the grasp of an automatic theorem prover.

$$\underline{a \circ b} = b \circ a \rightarrow$$

paramodulation
from 6

$$\underline{e} \circ (a \circ b) = b \circ a \rightarrow$$

paramodulation
from 11

$$(\underline{X} \circ X) \circ (a \circ b) = b \circ a \rightarrow$$

paramodulation
from 7

$$((X \circ \underline{e}) \circ X) \circ (a \circ b) = b \circ a \rightarrow$$

paramodulation
from 11

$$(\underline{(X \circ (Y \circ Y))} \circ X) \circ (a \circ b) = b \circ a \rightarrow$$

paramodulation
from 10

$$(\underline{((X \circ Y) \circ Y) \circ X}) \circ (a \circ b) = b \circ a \rightarrow$$

paramodulation
from 10

$$\underline{((X \circ Y) \circ (Y \circ X)) \circ (a \circ b)} = b \circ a \rightarrow$$

paramodulation
from 10

$$(X \circ Y) \circ \underline{((Y \circ X) \circ (a \circ b))} = b \circ a \rightarrow$$

paramodulation
from 11

$$\underline{(b \circ a) \circ e} = b \circ a \rightarrow$$

paramodulation
from 7

$$\underline{b \circ a = b \circ a} \rightarrow$$

resolution
with 1

$$\rightarrow$$

Figure 7.2: A Paramodulation and Resolution Proof

The bad news is that the branching rate is also dramatically increased from about 3 to about 12! So we have gone from a deep thin tree to a shallow bushy tree. Since 12^{10} is less than 3^{42}, this seems like it might be an improvement, however the branching rates of 3 and 12 were only very crude estimates so we cannot be sure without a more careful analysis and we certainly cannot deduce anything about the general case from this. And 12^{10} is still a very big number.

There is a marked improvement in the naturalness and readability of the paramodulation proof and this *is* a positive step. It makes it much easier to bring to bear our intuitions about how the proof should be guided if the theorem prover is making steps comparable to our own.

7.3.2 Cheating Techniques

All the above methods are *uniform* ones: that is, they can be used when trying to prove any theorem from any set of axioms. They do not rely on any domain dependant proof techniques. Uniform methods are generally regarded as having failed. New inference systems, either restrictions or extensions, may offer some slight improvement, may bring another trivial theorem within reach, but they do not substantially affect the situation.

Nevertheless there is always someone, who after enormous effort locating and stamping out some source of inefficiency, believes that his new system offers a panacea. In the literature one may find glowing reports of such innovations. The inadequacies of such systems are easy to overlook or (horrors!) to coverup. The reports of successful experiments with the new system may not always be what they seem!

To forearm you against possible misrepresentations, either deliberate or unintentional, (and, as an unwitting side effect, to show you how to cheat yourself) we record below some of the most common cheating techniques.

- Displaying only proofs obtained by the system and not the search tree generated in the process. As we have seen above, it is often possible to trade a deep narrow tree for a short wide tree, but this is not automatically an improvement.

- Feeding to the theorem prover only those axioms known to be required in the proof. Irrelevant axioms can dramatically increase the size of the search tree.

- Generating only that part of the search tree which lies within some arbitrary limits, but which is known to include the proof. Typically, the sought proof is examined to see: how long it is; what the maximum depth of function nesting is and what the maximum length of clauses is: and then the tree is only searched within those limits.

- Of course, if such techniques are applied uniformly and without pre-examination of the sought proof then they are legitimate (although they will always lose completeness). One could imagine, for instance, such a technique being proposed as a psychological theory of the memory limitations of human mathematicians.

- The program implementing the new system, may be rejigged for each new proof by a variety of other techniques, e.g. changing the criteria by which literals are selected and clauses chosen.

We would not want to suggest that these techniques are peculiar to Automatic Theorem Proving. Their analogues can be found in all areas of Artificial Intelligence and, one gathers, it most other branches of Science.

So if uniform methods have failed, what are we to do?

7.4 Analysing Human Proofs

One way forward is to turn our attention to the way in which humans go about solving problems. Experienced human mathematicians prove theorems, like our group theory example, which appear to involve huge combinatorial explosions. How do they do it? In particular, I found a proof of that group theory example, which was 42 resolutions long. How did I do it?

Well I did not start from the statement of the problem as a set of 12 clauses and then resolve away. I started from my statement given in section 7.2 above, namely

Every group of exponent 2 is abelian.

This description of the problem uses a different language from that of the standard axiomatization of group theory – it suggests a different axiomatization. An axiomatization, perhaps, in which the the search tree is not so big?

Abelianness (or commutativity) is a significant concept in group theory, related, in my mind, to other concepts like *commutators* and *commutator subgroups*.

Definition 7.1 *If G is group, a commutator is a term $i(A) \circ i(B) \circ A \circ B$ where A and B are in G.*

The commutator subgroup, G', of G is generated by "$i(A) \circ i(B) \circ A \circ B$ for all A and B in G".

The link between these concepts is a theorem that

G is abelian iff G' is trivial, i.e. $G' = \{e\}$

This theorem in turn is related to a common technique for proving a group abelian, namely to show that G' is trivial by proving

$i(a) \circ i(b) \circ a \circ b = e$ for typical elements a and b in G

The 'commonness' of this technique was a piece of guidance information which caused me to select it as the technique to try first.

Not all groups are abelian, so clearly the idempotency of G must be used (more guidance information). Idempotency is another important concept in group theory and is also related to other key concepts and theorems, in this case the key concept is that of *inverse* and the key theorem that

Every element of a group of exponent 2 is its own inverse,

i.e. $X = i(X)$ for all X in G. If we use this theorem our goal is to prove that:

$a \circ b \circ a \circ b = e$ for typical elements a and b in G

Now comes the only hard part of the proof, noticing that the idempotency of G can be used again on the element $a \circ b$.

$$a \circ b \circ a \circ b = (a \circ b) \circ (a \circ b) = e$$

Now we have *a* proof, but can we translate it into a resolution proof from the clauses 1-12? To do this we must recover the elementary proof that

G is abelian if G' is trivial

and modify it.

The standard proof of this result is:

$$\begin{aligned} a \circ b &= e \circ a \circ b \\ &= b \circ i(b) \circ a \circ b \\ &= b \circ e \circ i(b) \circ a \circ b \\ &= b \circ a \circ i(a) \circ i(b) \circ a \circ b \\ &= b \circ a \circ e \\ &= b \circ a \end{aligned}$$

Our proof above involved substituting a and b for $i(a)$ and $i(b)$. Doing this yields:

$$\begin{aligned} a \circ b &= e \circ a \circ b \\ &= b \circ b \circ a \circ b \\ &= b \circ e \circ b \circ a \circ b \\ &= b \circ a \circ a \circ b \circ a \circ b \\ &= b \circ a \circ e \\ &= b \circ a \end{aligned}$$

It was fairly straightforward to transform this proof into the 10 step paramodulation proof. The trickiest bit was locating the implicit applications of associativity and inserting applications of clause 10. Then, the 42 step resolution proof was constructed by a laborious replacement of paramodulation steps by applications of equality axioms.

What can we learn from this exercise?

- That we should look at alternative axiomatizations of mathematical theories, which utilize a language closer to the one with which we ourselves describe the problem. It may be that the search tree in this new axiomatization is smaller. But even if it is not, it will certainly be easier to express our intuitions about guiding search in such a language.

- That guidance information is available, given a language to express it in. We saw: an example where this information was domain specific, concerning common techniques for proving that a group is abelian; and an example which was more general purpose, that one should try to use all the hypotheses of the theorem.

7.5 Alternative Axiomatization

Can we use logical notation to formalize the reasoning described in the previous section? What would that look like?

The theorem on which the whole proof was based was:

G is abelian iff G' is trivial

Clearly this is a double implication between two propositions the predicates of which are 'abelian' and 'trivial'. We will also want to represent the relationship between G and G'. Since G', the commutator subgroup, is uniquely determined by G, we should represent the relationship with a function, 'commutator_subgroup'. So the theorem can be formalized as:

$$abelian(G) \leftrightarrow trivial(commutator\_subgroup(G)).$$

Our original statement also contained a definition of 'trivial'

i.e. $G' = \{e\}$.

Again this can be represented as a double implication between $trivial(G')$ and $G' = \{e\}$ Actually there is a catch. Each group, G, has a different identity element, so e should be replaced by $e(G)$. But this has repercussions. 'trivial' is not such a trivial concept as it at first appeared – it is dependant on a particular e and therefore on a particular group. It should have G as an additional parameter. Thus the axiomatization so far is:

$$abelian(G) \leftrightarrow trivial(commutator\_subgroup(G), G) \tag{i}$$

$$trivial(G', G) \leftrightarrow G' = \{e(G)\}$$

We also gave a definition of commutator subgroup above, as consisting of all commutators of a group. Can we formalize this? We will need to discuss the typical element of the commutator subgroup and how it relates to two typical elements of the group. For this we will want some functions, , el and el', which take a group as parameter and represent typical elements of that group. The definition of commutator subgroup can then be represented as:

$$\begin{aligned} &el(commutator\_subgroup(G)) \\ &\qquad = i(el(G)) \circ i(el'(G)) \circ el(G) \circ el'(G) \end{aligned}$$

$el(G)$ and $el'(G)$ are rather like skolem functions. We know nothing about their values other than that they are elements of the group G.

Converting what we have so far into clausal form results in the clauses

1. $trivial(commutator\_subgroup(G1), G1) \rightarrow abelian(G1)$
2. $abelian(G2) \rightarrow trivial(commutator\_subgroup(G2), G2)$
3. $G3' = \{e(G3)\} \rightarrow trivial(G3', G3)$
4. $trivial(G4', G4) \rightarrow G4' = \{e(G4)\}$
5. $\rightarrow el(commutator\_subgroup(G5)) =$
   $i(el(G5)) \circ i(el'(G5)) \circ el(G5) \circ el'(G5)$

To this we may add our conjecture

Every group of exponent 2 is abelian

negated and put in clausal form

6. $\rightarrow exponent(g, 2)$
7. $abelian(g) \rightarrow$

Now taking 7 as top clause we can resolve it with 1 to form

8. $trivial(commutator\_subgroup(g), g) \rightarrow$

and this with 3 to form

9. $commutator\_subgroup(g) = \{e(g)\} \rightarrow$

At this stage we are stuck. We would like to combine 9 in some way with 5, but no resolutions or paramodulations are possible. There is a gap. If we consult the proof we are modelling we will see that the gap is filled by the knowledge that to show a group consists of one individual, show that a typical element of it is that individual, i.e.

10. $el(G10) = X10 \rightarrow G10 = \{X10\}$

Now 10 resolves with 9 to give

11. $el(commutator\_subgroup(g)) = e(g) \rightarrow$

and 11 paramodulates into 5 to give

12. $i(el(g)) \circ i(el'(g)) \circ el(g) \circ el'(g) = e(g) \rightarrow$

Exercise 22 *Continue in this way representing the knowledge used in the proof, converting it into clausal form and trying to form a proof by resolution and paramodulation. Did you uncover any gaps? Can you fill them?*

But this was not the only type of reasoning going on as we made the proof. Most of the time was spent discussing the theorems available. We said that (i) above was a theorem

$\rightarrow$ theorem((i)).

We said (i) was a common technique for proving the abelianness of groups

$\rightarrow$ common_technique((i), abelianness)

We gave a standard proof of (i)

$\rightarrow$ proof((i),$a \circ b = e \circ a \circ b = ... = b \circ a$)

Whenever we pop up a level like this and start discussing and reasoning about the level below, it is called *meta-level* reasoning. The lower level is called the *object-level*. In chapter 12 we will discuss meta-level reasoning and how it can induce and guide a proof at the object-level.

7.6 A New Methodology

Subsequent chapters of this book will elaborate this second approximation to the automation of mathematical reasoning: we will describe some attempts to apply the methodology of the last two sections, namely to analyse the proofs of experienced human mathematicians and to try and formalize what we see there.

Thus, instead of trying to describe uniform, domain independent techniques for guiding search – techniques which are derived from work in Mathematical Logic and which have failed to produce successful theorem provers – we will seek guidance techniques in actual proofs and then try to generalize these.

In rejecting the uniform proof procedure methodology of the 1960s we must be careful not to throw out the baby with the bath water. Resolution theorem provers gained us a number of advances.

- They enabled us to specify a problem and the axioms from which its solution must be built without having to specify the solution itself.
- This was achieved with the aid of a language for the representation of knowledge, a semantics for defining the meaning of this language and a definition of proof.
- They contributed a number of procedures for manipulating this language - especially pattern matching procedures for applying the axioms to the problem and search techniques to enable the computer to change its mind and attempt a different solution.
- Occasionally we were able to prove theorems about our procedures, e.g. that they were sound or complete. When we can do it, this is a good trick. It is often a great help in correcting or improving our procedures. However, there is a danger here, of which we should beware, of designing procedures we can prove theorems about, rather than ones which are successful.

The principle deficiency of uniform proof procedures was the lack of a way of guiding the search. Control knobs were provided whereby the search could be directed. These were the selection of which literal to resolve away and the choice of which clause to resolve it with. But almost no advice was provided about how these control knobs were to be twiddled. Worst of all the control knobs were not designed to meet the needs of people who knew how they wanted the search to be directed. They were not designed at all, but emerged as a by-product of the construction of the theorem provers.

We will want to see what sort of control advice emerges from our analyses so that we can design the search control knobs ergonomically.

7.7 Summary

In this chapter we have seen that uniform proof procedures, like SL Resolution, are inadequate to prove non-trivial theorems. General purpose search guiding and techniques have only a minor effect on curbing the combinatorial explosion. If we are build expert artificial mathematicians, then we must use domain specific guidance information. A useful source of such information is the analysis of human generated proofs. The results of this methodology are the subject of part III of this book.

Part III

Guiding Search