Methods and Calculi for Deduction

W. Bibel, E. Eder

Contents

Abstract

The presentation has three goals in mind. For one, it gives an introduction to various calculi for first-order logic which should provide the reader with a good sense of them. Secondly, it aims at clarifying the relationships among these calculi, including resolution, especially in view of automating deduction. To be specific, we consider Hilbert and Gentzen calculi, tableau calculi, connection calculi, and resolution including its extensions. With such a broader view of automated deduction in mind several issues present themselves in a new light. The third goal, therefore, is a presentation of the issues relevant for the automation of deduction. In particular, we discuss several aspects of controlling the search for a proof, we consider restrictions as well as extensions of first-order logic, and we present proof methods and heuristics for special theories.

1 Introduction

Logic has been presented in Chapter 1 as a language to formalize problems arising in subareas of Intellectics[1] (and of other fields, for that matter). In Chapter 2 it is shown that logic also supplies a relationship (denoted by $\models$) among different statements (or *formulas*) in that language. Note that there are actually many logics in the literature. As a default, we mean *first-order* logic unless explicitly specified otherwise. The formulas related this way to the empty set of formulas (ie. $\emptyset \models F$) play a particularly important role. They are called *valid* formulas. It turns out that in many cases a problem can be reformulated as a logical formula to be proven valid. For this reason one often speaks of "theorem proving" where in effect solving problems, as those just referred to, is the issue.

Deduction, the topic of this chapter, is concerned with syntactic ways of capturing the relationship just mentioned. A syntactic form of $\models$ is denoted by $\vdash$. A logic along with such a syntactically defined relation is called a (logical or deductive) *calculus*.

Intellectics aims at solving problems, or proving validity, automatically. Calculi provide a way of testing the validity of formulas in a purely mechanical way. This does not necessarily mean, however, that any such calculus is well-suited for the automation of this test. While discussing calculi in general in this chapter, a major concern in these discussions will therefore be their suitability for automation.

There are many different ways of characterizing $\models$ syntactically. Some are closely related to each other so as to form a family of calculi, in which case we will also talk of a (deductive) *method* to refer to the entire family.

The most widely used method (in this sense) is resolution (along with its many refinements). The entire subsequent Chapter 4 of this volume is devoted to it for its importance and popularity. It is still included in the present chapter for completeness, but for obvious reasons is treated here (in Section 2.9) only very briefly.

But there are more such methods different from resolution. Most of them were known even before the discovery of resolution. Hence the question naturally arises which of these is most suitable for computational purposes in view of possible deductive tasks such as proving validity. A closely related question to ask is "What *makes* a logical formula valid?" [And89]. Also one might ask how many different methods there are. Is there perhaps only a single deductive method such that all known calculi are members of the family defined by it?

These questions lie at the heart of the material presented in the present chapter in which we study most of those methods. We attempt to give answers not only to these particular questions but also to those concerning the relationships and differences between various calculi, most of the time with the task of theorem proving in mind.

The focus of our study is calculi for first-order logic with function symbols and without (a special treatment of) equality. Only occasionally shall we touch upon extensions to include equality, higher-order features, and so forth. The calculi will be presented in Section 2 with an emphasis on the question cited above. After a few historical remarks we discuss Frege–Hilbert calculi, Gentzen natural deduction and sequent calculi, tableau calculi, connection and connection structure calculi, and resolution. This review is completed by a few pointers to several further calculi that for reasons of space had to be omitted.

Thereafter, in Section 3, we proceed in a topic-oriented way. First the conversion of a formula of full first-order logic into its clausal form is presented. It is superior from a computational point of view to the popular one presented in detail in the next chapter. Then a Frege–Hilbert calculus, Gentzen's natural deduction and sequent calculus, the tableau calculus, a connection calculus, a connection structure calculus, and resolution are compared in a pairwise way in terms of their complexity.

Each of these calculi opens a different perspective on the nature of validity. Thus the

[1]Intellectics is the name preferred by the authors for the fields of Artificial Intelligence and Cognitive Science which will be used throughout this chapter. See [Bib92] for more explanations.

point of studying them all is to get a deeper understanding by combining the insights from these different viewpoints and from these derive improvements for *any* of these calculi. This is also to say that we do not expect one of the methods to be *intrinsically* better than all others. While significant computational differences have been discovered, these might well be removed by subsequent modifications. So, while believing in the potential of further significant improvements, we do share the view that "other syntactic methods ... will fare only about as well as resolution systems" [Ble77] in the sense that any advantage in some calculus can probably always be incorporated in some way into a different method as well (note what we understand by the word 'method'). Put in a different way, there might actually be a single deductive method, so that all calculi do belong to the same family. The connection structure calculus is a step towards identifying such a method (if it exists).

For the task of finding proofs in any of these calculi a search strategy has to be added, a topic treated in Section 4. Considerations of this nature are influenced by what one would regard as an ideal proof system. Our view in this respect is influenced by what we call an adequate proof system. With this view in mind, two fundamental concepts in automated deduction, namely linear chains and hinged loops, are discussed and their various disguises pointed out. On this basis the complex question of controlling the proof search is discussed. Besides presenting the usual methodes of control (top-down, bottom-up, and so forth), we identify the principle of compression as fundamental for an enhanced proof search. Simplifications, theory connections, and sorts are viewed from this perspective. Finally, in addition to all these purely syntactic controlling techniques, methods for semantic guidance of proof search are presented.

Section 5 briefly points to a number of techniques developed for restricted parts of logic such as datalogic, constraints, Horn clause logic, each of which has its importance for certain applications. We also discuss there extensions of first-order logic, such as higher-order and modal logic.

A further important topic in automated deduction is the special treatment of particular theories such as equality, induction, set theory, analysis, arithmetic, geometry, and so forth. It is in such applications with their specialized techniques that additionally heuristics become especially useful. Section 6 is devoted to this kind of material. It is shorter not because of its lesser importance but because one chapter is simply not sufficient to cover all material in an equally adequate way. We complete the chapter with some conclusions.

It should be obvious to the reader that this wide range of topics indeed cannot be covered exhaustively in a single chapter. In consequence, there are many omissions. Particularly unfortunate is the lack of a more detailed treatment of working systems since these, after all, provide the motivation for what we are doing here.

The question has sometimes been raised what the motivation of automated deduction actually is, given that the formalisms involved all seem computationally infeasible. One of the working hypotheses underlying the work on deduction is that there is, in principle, no difference between man and machine with regard to discovering proofs and deductions. So, since men do discover proofs in spite of the complexity involved in the formalisms, machines should be able to do so as well. While progress is slow, experience supports this stance, or at any rate does not refute it. Whatever the final answer to this question will be, the results of this research have become so useful in applications that the efforts are justified regardless. PROLOG is the best-known among these results, but by far not the only one.

Deduction often is also considered too narrow and too rigid for being useful in practical applications. Therefore, in Section 6.6 we briefly make the point that ultimately deduction will be the most general form of automated reasoning.

2 A survey of logical calculi

There is a confusing abundance of logical calculi. This section is meant not only to provide an overview of many of them, but also to give a handle for understanding their purpose and nature.

The first thing in talking about logical calculi is to decide which *logic* we want to deal with. Logic is concerned with the experienced phenomenon that accepting certain statements as valid (the "true premises" or "axioms") implicitly makes us accept other statements as well. In other words there is some *logical relationship* between a given set $\mathcal{A}$ of statements and other statements T. In modern terms one might say, T is in the theory of $\mathcal{A}$. Logic thus is concerned with this relationship and automated deduction with its computational aspects.

In this characterization we have already made two important assumptions. First, we assume the existence of a uniform language which clarifies in a precise way what statements are under consideration (eg. do they include those made in a literary discourse?). Second, it suggests that there be one single such relationship (that R. Brachman refers to as "the myth of the one logic"). As we said in the introduction, we mostly restrict our discussion in the present chapter to first-order logic which may be regarded as providing core features of any logic. This way we avoid the need for justifying these two assumptions which, while being important issues, would lead us too far away from our topic. Since we thus restrict our attention to first-order logic, all the calculi that we survey are characterizing validity in a sound and complete way.

We begin with a quick look at the history of logical calculi up to and beyond the point when Frege published his seminal paper "Begriffsschrift" [Fre79].[2]

2.1 The prehistory of logical calculi

The existence of some sort of a logical relationship among statements was recognized even long before Aristotle. However, the earliest document that deals with many aspects of this relationship in some depth indeed is Aristotle's *Organon*, a collection of his writings grouped together after his death in 322 B.C. The influence of this treatise can be felt even in our days.

It has laid the foundation of the concepts necessary for a development of a formal logical language, has captured the notion of demonstrative (as well as dialectic) arguments along with that of premises (both required for proofs), has studied particular arguments (or rules of inference) in terms of so-called *syllogisms*, has presented a theory of meaning and truth, has even treated modal statements, to mention some of the important contributions of this seminal opus. Obviously, each of these issues is of great relevance for automated deduction.

To allow at least one glimpse at this œuvre, we mention that there are three syllogistic figures (or schemes) from which particular syllogisms may be constructed. In a sense these categorical arguments resemble the (non–) inheritance relations used in modern inheritance nets. The first[3] two such particular syllogisms (or moods), later (mnemonically[4]) called *Barbara* and *Celarent*, may be presented as

If every M is L and every S is M, then every S is L
If no M is L and every S is M, then no S is L

There are altogether fourteen such basic syllogisms (apart from derivative patterns).

[2] For obvious reasons this section provides an extremely compressed account of that history. The reader with a historic interest is referred to the comprehensive treatment provided in [KK84, Wil52] from where much of the following information is taken.

[3] in Aristotle's original numbering

[4] The mnemonic meaning of vowels and consonants in these three syllable words composing a meaningful Latin text is quite elaborate; eg. a stands for universal affirmative, e for the universal negative, etc.

Aristotle's achievement is so outstanding that we tend (unjustifiedly[5]) to ignore to a large part the further developments of logic during the subsequent nearly two thousand years (that indeed included "Dark Ages") until Leibniz (1646–1716) who, like Aristotle, "deserves to be ranked among the greatest of all logicians" [KK84]. There are a number of ideas in his work that have a strong bearing on automated deduction; in fact, he may be regarded as the first who anticipated the mechanization of formal reasoning in a rather comprehensive way.

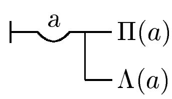
There is first his plan for an ideal language, the *lingua philosophica* or *characteristica universalis* which, however, he never realized in any detail. Such a language would provide, second, the framework for a *calculus rationator*, ie. a reasoning calculus. In addition and within this context, Leibniz pursued the idea of an *encyclopaedia*, ie. a systematic collection of knowledge of all kinds. It is needless to point out that these three may be seen as the basic components of a knowledge-based system of our days.

In this grand design he was probably influenced by the *Ars Magna* of Raymond Lull (1235–1315), a system for combining concepts in a mechanical fashion so as to exhaust all alternatives. In this Leibniz also has taken up Thomas Hobbes' *Computatio sive Logica* (1655) that suggested our thinking to be reducible to a species of calculation. While Hobbes did not make any attempt to work out such a project even as a thought design, Leibniz apparently developed a fairly clear idea what such a system would have to look like. He even speculated that, when perfected, such a system would allow men of good will desiring to settle a controversy on any subject whatsoever to take their pens in their hands and say *Calculemus*, ie. let's simply compute it.

Leibniz tried to present logic as a calculus that could be mechanized like the arithmetic calculus. Although ultimately he aimed at a general calculus of structures that was meant even to include a logic of discovery, he was able to work out details for two specific structures only. One is a calculus for identity and inclusion with which he anticipated Boole's *Mathematical Analysis of Logic* (1847), ie. Boolean algebra or propositional logic, presented in the latter work for the first time as a calculus in Leibniz's sense, now in perfection. The other is a geometrical calculus of similarity and congruence with which, again, he anticipated Grassmann's *Ausdehnungslehre* (1844), ie. a geometry without the use of co-ordinates.

With his *Begriffsschrift — Eine der arithmetischen nachgebildete Formelsprache des reinen Denkens* (1879) Gottlob Frege (1848–1925) explicitly referred to Leibniz' concept of a characteristica universalis (ie. a universal language) included in Frege's system. This language Begriffsschrift (ie. conceptual language) basically is what today we call first-order logic. Frege intended to achieve with it the more general goal of subsuming arithmetic within logic. While he himself did not fully succeed in this respect, as Bertrand Russell discovered an inconsistency in his formalism, he still believed in the goal as such, to which solutions have been presented more recently [Fef84, Per85, Gil89].

Frege's symbolism has not found the appreciation of logicians who have regarded it as too awkward. We nevertheless want to give a taste of it for the reader. What today might be expressed as $\vdash \forall a(\Lambda(a) \rightarrow \Pi(a))$, Frege pictured as follows.



This expression may be regarded as consisting of the following different parts. There is the symbol $\vdash$ at the leftmost end, which is the only fragment still in use today. Then comes the concavity serving the role of a universal quantifier, with the quantified variable on top of it. The right part shows Frege's way of picturing conditionals with the conclusion on top of the premise. Negation can be added by a small vertical line at (and below) the horizontal line (not demonstrated by this example). It can be placed wherever required for the appropriate scope; the same holds for the quantifier, ie. the concavity. Arbitrary nesting is possible. Since

[5]For instance, it was Chrysippus (280-207), rather than Aristotle, who supplied the best account of propositional logic.

all other logical symbols can be defined in terms of the ones just mentioned, this provides a full first-order language. A way of introducing definitions (using identity) is given in addition. The Begriffsschrift supplies not only a language, but also a calculus built on top of it of a kind as presented in the following subsection.

The book [KK84] refers to 1879 as the most important date in the history of logic. In any case it is among the few outstanding milestones marking the fields of logic in general as well as of automated deduction in particular. Frege then provided the first really comprehensive system of formal logic that contains all the essentials of modern logic. Especially his discovery of quantifiers to bind variables has been considered one of the greatest intellectual inventions of the nineteenth century.

Another important contribution by Frege of relevance for automated deduction is what is variously called the "logistic thesis", or the "Frege–Russell thesis", viz., that mathematics is derivable from, or an extension of, logic. It was presented in extensive detail in the famous *Principia Mathematica* [WR13]. Hilbert [HB39], in slight contrast, put more emphasis on the axiomatization of mathematics. From there it is but a small step (in time) to Herbrand [Her30], who studied also under Hilbert, and to Gentzen [Gen35] who completed the basis on which automated deduction could be built.

2.2 Frege and Hilbert calculi

In the literature the notion of *Frege–Hilbert calculi* (sometimes only called Hilbert calculi) is used for a type of calculus that is characterized by a number of axiom schemes and a number of rules. Each axiom is a formula and each rule allows to derive new formulas from given ones. A *derivation* in a Frege–Hilbert calculus is a finite sequence of formulas such that each formula of the sequence is either an axiom or is obtained from previous formulas of the sequence by application of a rule. A *derivation of a formula* is a derivation whose last member is that formula. Usually there are many axiom schemes and just a very few rules. Frege–Hilbert calculi were first introduced by Gottlob Frege in [Fre79]. They were popularized by David Hilbert [HB39]. As an example of a Frege–Hilbert calculus we give a calculus taken from [Gen35]. Its axiom schemes and its rules are shown in Table 1.

Here A, B, C, F denote formulas, x denotes a bound variable, a denotes a free variable, and t is a term not containing any bound variables. By $F\{x\backslash t\}$ we denote the formula obtained from F by replacing every free occurrence of x with t. In the rules R2 and R3, the variable a is called the *eigenvariable*. It is subject to the *eigenvariable condition* that a must not occur in the conclusion. R1 is the well-known rule of modus ponens. The formulas above the horizontal line in a rule are called *premises*, the one below *conclusion*. An example of a derivation in this calculus is given in Table 2.

Frege–Hilbert calculi are specially suited for proof-theoretic considerations because each object occurring as a member of a derivation is just a formula whose truth is asserted at the respective step in the derivation. Also, since the number of rules is usually small, such calculi are convenient for proving meta-theorems. For automated deduction they seem to be less adequate, especially because of the modus ponens rule. Namely, the most simple way to prove a formula automatically using a Frege-Hilbert calculus is to follow a top-down approach (see Section 4.5). One starts from the formula to be proved and applies the rules in the backward direction until one arrives at formulas that are axioms. Now, if you want to prove a formula B, the problem is that a backward application of modus ponens yields the formulas A and $A \to B$ for an infinite number of formulas A. So the branching factor of the search tree may be infinite.[6] We shall meet the same problem again in connection with the cut-rule in sequent calculi.

[6]Alternatively, A can be treated as a variable for a formula which in the course of proof search may be instantiated by formula unification.

Table 1. A Frege–Hilbert calculus

The Axiom Schemes:

A1	$A \vee \neg A$
A2	$A \to A$
A3	$A \to (B \to A)$
A4	$(A \to (A \to B)) \to (A \to B)$
A5	$(A \to (B \to C)) \to (B \to (A \to C))$
A6	$(A \to B) \to ((B \to C) \to (A \to C))$
A7	$(A \wedge B) \to A$
A8	$(A \wedge B) \to B$
A9	$(A \to B) \to ((A \to C) \to (A \to (B \wedge C)))$
A10	$A \to (A \vee B)$
A11	$B \to (A \vee B)$
A12	$(A \to C) \to ((B \to C) \to ((A \vee B) \to C))$
A13	$(A \to B) \to ((A \to \neg B) \to \neg A)$
A14	$\neg A \to (A \to B)$
A15	$\forall x F \to F\{x \backslash t\}$
A16	$F\{x \backslash t\} \to \exists x F$

The Rules:

R1	$\dfrac{A \qquad A \to B}{B}$	
R2	$\dfrac{A \to F\{x \backslash a\}}{A \to \forall x F}$	with eigenvariable condition
R3	$\dfrac{F\{x \backslash a\} \to A}{\exists x F \to A}$	with eigenvariable condition

Table 2. A derivation of $P \to (P \to \forall x(P \vee Qx))$ in a Frege–Hilbert calculus

F1	$P \to P \vee Qa$	by A10
F2	$P \to \forall x(P \vee Qx)$	from F1 by R2
F3	$(P \to \forall x(P \vee Qx)) \to (P \to (P \to \forall x(P \vee Qx)))$	by A3
F4	$P \to (P \to \forall x(P \vee Qx))$	from F2 and F3 by R1

2.3 Natural deduction

Despite their advantages, Frege–Hilbert calculi do not reflect the way mathematicians proceed when proving theorems. Mathematicians make assumptions, prove new formulas from these assumptions, and finally eliminate these assumptions. For example, in order to prove an implication $A \rightarrow B$, a mathematician would first assume that A holds, then prove B, and finally, by elimination of the assumption A, conclude that $A \rightarrow B$ holds. Gerhard Gentzen's calculus of *natural deduction* [Gen35] is based on this idea.

In a Frege–Hilbert calculus, a derivation starts from valid formulas as axioms, and from these axioms new valid formulas are derived according to given rules. In contrast to this, in a natural deduction calculus, the derivation starts from assumptions from which new formulas are derived that depend on these assumptions. There are rules for making a derived formula independent of assumptions that have been made.

In the field of automated theorem proving, natural deduction calculi have been used by several authors for the implementation of deductive systems. Some of these systems (such as AUTOMATH, NuPRL, and so forth) are discussed later in Section 5.5. Natural deduction style presentations of proofs have also been used to formulate a framework for a wide class of logics (see [Bur87] and [HHP87]).

Here we shall present the calculus of natural deduction in a setting which is used in present-day natural deduction theorem proving systems, and which best shows the close relationship between natural deduction and the sequent calculus. This relationship was established by Gerhard Gentzen in [Gen35]. In contrast to the form of the natural deduction calculus presented in this section, Gentzen's original calculus uses a tree representation of derivations in connection with structure sharing. But the two kinds of representation are easily seen to be equivalent.

In our setting of natural deduction, the objects that occur as elements in a derivation are strings of the form $F_1, \ldots, F_k \Longrightarrow H$, called *judgements*. Here, $F_1, \ldots, F_k$ and H are formulas, and the intuitive meaning of the judgement $F_1, \ldots, F_k \Longrightarrow H$ is that H holds under the assumption that $F_1, \ldots, F_k$ hold. We use capital Greek letters to denote sequences of formulas separated by commas. So "$\Gamma \Longrightarrow H$" will denote a judgement. We shall say that the members of Γ are the *assumptions* of the judgement and that H is the *conclusion* of the judgement. In Table 3 we give the axioms and rules of the calculus. For each logical symbol there is at least one introduction rule and one elimination rule. $\perp$ is the falsum.

The meaning of a rule

$$\frac{H_1 \qquad \ldots \qquad H_n}{H}$$

is that we can conclude a judgement $\Gamma \Longrightarrow H$ from the judgements $\Gamma_1 \Longrightarrow H_1, \ \ldots, \ \Gamma_n \Longrightarrow H_n$ if, for all $i = 1, \ldots, n$, each member of Γ_i is also a member of Γ. If the rule in addition has a formula A_i in square brackets above some premise H_i then it has the same meaning just explained except that only those members of Γ_i which are different from A_i are required to be in Γ. For example, $\vee$-E means that if $\Gamma_1 \Longrightarrow A \vee B$ and $\Gamma_2, A \Longrightarrow C$ and $\Gamma_3, B \Longrightarrow C$ and $\Gamma_i \subseteq \Gamma$ for $i = 1, 2, 3$, then $\Gamma \Longrightarrow C$. Thus the rules $\neg$-I, $\vee$-E, $\rightarrow$-I and $\exists$-E eliminate assumptions in this way. In the rules $\forall$-I and $\exists$-E, the *eigenvariable a* is required not to occur in the judgement $\Gamma \Longrightarrow H$ and, in the case of $\exists$-E, also not in the first premise H_1 (ie. $\exists x F$).

A *derivation* is a finite sequence of judgements such that each member of the sequence is either an axiom or obtained from previous members of the sequence by application of a rule. A *derivation of a formula F* is a derivation whose last member is the judgement $\Longrightarrow F$. As an example we present a derivation of the formula $\exists x \forall y Pxy \rightarrow \forall y \exists x Pxy$[7] taken from [Gen35]. First we consider how a mathematician would prove this formula.

1. Assume $\exists x \forall y Pxy$.

[7]Since all our examples use names of length 1 (not counting indices) to denote variables and constants, we may dispense with parentheses in the absence of function symbols as well as for unary predicates or function symbols.

Table 3. Gentzen's system of natural deduction

Axioms:

ASSUMPTION	$A \Longrightarrow A$
TERTIUM NON DATUR	$\Longrightarrow A \vee \neg A$

Rules:

EX FALSO QUODLIBET $\dfrac{\bot}{\mathrm{D}}$

$\neg$-I	$\dfrac{\begin{matrix}[A]\\ \bot\end{matrix}}{\neg A}$	$\neg$-E	$\dfrac{A \qquad \neg A}{\bot}$
$\wedge$-I	$\dfrac{A \qquad B}{A \wedge B}$	$\wedge$-E	$\dfrac{A \wedge B}{A} \qquad \dfrac{A \wedge B}{B}$
$\vee$-I	$\dfrac{A}{A \vee B} \qquad \dfrac{B}{A \vee B}$	$\vee$-E	$\dfrac{A \vee B \qquad \begin{matrix}[A]\\ C\end{matrix} \qquad \begin{matrix}[B]\\ C\end{matrix}}{C}$
$\to$-I	$\dfrac{\begin{matrix}[A]\\ B\end{matrix}}{A \to B}$	$\to$-E	$\dfrac{A \qquad A \to B}{B}$
$\forall$-I	$\dfrac{F\{x\backslash a\}}{\forall x F}$ with eigenvariable condition	$\forall$-E	$\dfrac{\forall x F}{F\{x\backslash t\}}$
$\exists$-I	$\dfrac{F\{x\backslash t\}}{\exists x F}$	$\exists$-E	$\dfrac{\exists x F \qquad \begin{matrix}[F\{x\backslash a\}]\\ C\end{matrix}}{C}$ with eigenvariable condition

2. Let a be such an x. Then $\forall y Pay$.
3. Now let b be an arbitrary object. Then Pab holds.
4. Thus $\exists x Pxb$ holds.
5. Since b was arbitrary, $\forall y \exists x Pxy$ holds.
6. So, we now have that $\forall y \exists x Pxy$ holds under the assumption $\exists x \forall y Pxy$.
7. Therefore $\exists x \forall y Pxy \rightarrow \forall y \exists x Pxy$ holds.

The corresponding derivation by natural deduction goes as follows.

F1	$\exists x \forall y Pxy \Longrightarrow \exists x \forall y Pxy$	by Assumption
F2	$\forall y Pay \Longrightarrow \forall y Pay$	by Assumption
F3	$\forall y Pay \Longrightarrow Pab$	from F2 by $\forall$-E
F4	$\forall y Pay \Longrightarrow \exists x Pxb$	from F3 by $\exists$-I
F5	$\forall y Pay \Longrightarrow \forall y \exists x Pxy$	from F4 by $\forall$-I
F6	$\exists x \forall y Pxy \Longrightarrow \forall y \exists x Pxy$	from F1 and F5 by $\exists$-E
F7	$\Longrightarrow \exists x \forall y Pxy \rightarrow \forall y \exists x Pxy$	from F6 by $\rightarrow$-I

There is a one-to-one correspondence between the seven proof steps in the informal proof given above and the seven steps in this natural deduction derivation.

For the interested reader comprehensive sources for material on natural deduction are [Pra65] and [Gal86].

2.4 The sequent calculus

In this section we present Gerhard Gentzen's sequent calculus which was introduced in [Gen35]. This calculus is particularly interesting because it has had great impact on proof theory as well as on the development of calculi which are adequate for automated theorem proving. In a sequent calculus the objects derived during a derivation, ie. the members of a derivation, are so-called *sequents*. They play the role that formulas play in Frege–Hilbert calculi and that judgements play in natural deduction calculi. A *sequent* is a string of characters of the form

$$A_1, \ldots, A_\mu \Longrightarrow B_1, \ldots, B_\nu$$

where $\mu, \nu \geq 0$ and $A_1, \ldots, A_\mu, B_1, \ldots, B_\nu$ are formulas. The formulas $A_1, \ldots, A_\mu$ form the *antecedent*, and the formulas $B_1, \ldots, B_\nu$, the *succedent* of the sequent. Informally, the sequent has the same meaning as the formula

$$A_1 \wedge \cdots \wedge A_\mu \rightarrow B_1 \vee \cdots \vee B_\nu.$$

We use capital Latin letters as syntactic variables to denote formulas and capital Greek letters to denote finite sequences $A_1, \ldots, A_\mu$ of formulas separated by commas. So, every sequent has the form $\Gamma \Longrightarrow \Theta$. We denote free variables by a, b, c and bound variables by x, y, z. Now, the sequent calculus has as axioms the sequents consisting of one formula in the antecedent and the same formula in the succedent. In addition there are two kinds of inference rules, structural and operational inference rules. For each propositional connective and each quantifier, there is one operational rule for its introduction in the antecedent and one operational rule for its introduction in the succedent.

In the rules $\forall$-IS and $\exists$-IA, the variable a is called the *eigenvariable*. It is subject to the *eigenvariable condition* that a must not occur in the conclusion, ie. neither in F nor in any formula of Γ or Θ. Moreover, for all rules there is the restriction that no free variable occurs bound and no bound variable occurs free in any formula of any of the occurring sequents. In the rules $\forall$-IA and $\exists$-IS, the term t must not contain any bound variables.

Table 4. Gentzen's sequent calculus

AXIOMS

$$D \Longrightarrow D$$

Structural inferences:

THINNING

$$\frac{\Gamma \Longrightarrow \Theta}{D, \Gamma \Longrightarrow \Theta} \qquad \frac{\Gamma \Longrightarrow \Theta}{\Gamma \Longrightarrow \Theta, D}$$

CONTRACTION

$$\frac{D, D, \Gamma \Longrightarrow \Theta}{D, \Gamma \Longrightarrow \Theta} \qquad \frac{\Gamma \Longrightarrow \Theta, D, D}{\Gamma \Longrightarrow \Theta, D}$$

INTERCHANGE

$$\frac{\Delta, D, E, \Gamma \Longrightarrow \Theta}{\Delta, E, D, \Gamma \Longrightarrow \Theta} \qquad \frac{\Gamma \Longrightarrow \Theta, E, D, \Lambda}{\Gamma \Longrightarrow \Theta, D, E, \Lambda}$$

CUT

$$\frac{\Gamma \Longrightarrow \Theta, D \qquad D, \Delta \Longrightarrow \Lambda}{\Gamma, \Delta \Longrightarrow \Theta, \Lambda}$$

Operational inferences:

$\neg$-Introductions: $\neg$-IA $\neg$-IS

$$\frac{\Gamma \Longrightarrow \Theta, A}{\neg A, \Gamma \Longrightarrow \Theta} \qquad \frac{A, \Gamma \Longrightarrow \Theta}{\Gamma \Longrightarrow \Theta, \neg A}$$

$\wedge$-Introductions: $\wedge$-IA $\wedge$-IS

$$\frac{A, \Gamma \Longrightarrow \Theta}{A \wedge B, \Gamma \Longrightarrow \Theta} \quad \frac{B, \Gamma \Longrightarrow \Theta}{A \wedge B, \Gamma \Longrightarrow \Theta} \qquad \frac{\Gamma \Longrightarrow \Theta, A \qquad \Gamma \Longrightarrow \Theta, B}{\Gamma \Longrightarrow \Theta, A \wedge B}$$

$\vee$-Introductions: $\vee$-IA $\vee$-IS

$$\frac{A, \Gamma \Longrightarrow \Theta \qquad B, \Gamma \Longrightarrow \Theta}{A \vee B, \Gamma \Longrightarrow \Theta} \qquad \frac{\Gamma \Longrightarrow \Theta, A}{\Gamma \Longrightarrow \Theta, A \vee B} \quad \frac{\Gamma \Longrightarrow \Theta, B}{\Gamma \Longrightarrow \Theta, A \vee B}$$

$\rightarrow$-Introductions: $\rightarrow$-IA $\rightarrow$-IS

$$\frac{\Gamma \Longrightarrow \Theta, A \qquad B, \Delta \Longrightarrow \Lambda}{A \rightarrow B, \Gamma, \Delta \Longrightarrow \Theta, \Lambda} \qquad \frac{A, \Gamma \Longrightarrow \Theta, B}{\Gamma \Longrightarrow \Theta, A \rightarrow B}$$

$\forall$-Introductions: $\forall$-IA $\forall$-IS

$$\frac{F\{x\backslash t\}, \Gamma \Longrightarrow \Theta}{\forall x F, \Gamma \Longrightarrow \Theta} \qquad \frac{\Gamma \Longrightarrow \Theta, F\{x\backslash a\}}{\Gamma \Longrightarrow \Theta, \forall x F}$$

with eigenvariable condition

$\exists$-Introductions: $\exists$-IA $\exists$-IS

$$\frac{F\{x\backslash a\}, \Gamma \Longrightarrow \Theta}{\exists x F, \Gamma \Longrightarrow \Theta} \qquad \frac{\Gamma \Longrightarrow \Theta, F\{x\backslash t\}}{\Gamma \Longrightarrow \Theta, \exists x F}$$

with eigenvariable condition

The formula D in the cut rule is called the *cut formula*. A *derivation* in the sequent calculus is a finite sequence of sequents such that each sequent of the sequence is either an axiom or obtained from previous sequents of the sequence by application of a rule. The last sequent of the derivation is also called the *end sequent* of the derivation. A *derivation of a formula F* is a derivation whose end sequent is $\Longrightarrow F$.

The major reason that the sequent calculus is of great interest for proof theory as well as for automated theorem proving is the fact that it is complete also without the cut rule (Gentzen's Hauptsatz or cut elimination theorem [Gen35]). We then speak of the *cut-free sequent calculus*. The rules of the cut-free sequent calculus have the *subformula property*, ie. every formula occurring in a premise of a rule instance is a subformula — modulo some kind of substitution — of a formula occurring in the conclusion of this rule instance. In proof theory this property is used to prove the consistency of sequent calculi. For automated theorem proving the subformula property guarantees a finite branching factor of the search tree when the rules of a cut-free sequent calculus are applied in the backward direction in order to prove a given formula. So, by inverting a cut-free sequent calculus, we obtain calculi suitable for automated theorem proving such as the tableau and connection calculi. Such calculi are *analytic* in the sense that they work by splitting a given formula into subformulas (modulo substitution). The completeness of the cut-free sequent calculus holds for pure first-order predicate logic, but not, for example, for logic with induction. In fact the cut rule is needed for any calculus including induction which is a main reason why automated theorem proving in logic with induction is rather difficult.

Kurt Schütte's calculus of positive and negative parts [Sch77] is a variant of the sequent calculus. While in the sequent calculus we have three levels of language, the level of formulas (object level), the level of sequents, and the meta level, Schütte has merged the level of sequents into the level of formulas. His negative parts of a formula correspond to the members of the antecedent of a sequent, and the positive parts of a formula[8] correspond to the members of the succedent of a sequent. Also some connectives are defined in terms of others. This permits reduction of the number of rules to three and makes the calculus as suited for proof theoretic investigations as Frege–Hilbert calculi while retaining the suitability for automated theorem proving enjoyed by the sequent calculus.

2.5 The tableau calculus

The *tableau calculus* originates from Evert Beth's *semantic tableaux* [Bet55, Bet59] and was further developed by Raymond Smullyan [Smu71]. This calculus together with its variants is also called the method of *analytic tableaux*.[9] For historical reasons the tableau calculus is formulated as a refutation calculus which directly establishes the unsatisfiability of a given formula rather than its validity. But since a formula is valid iff its negation is unsatisfiable, and vice versa, there is no difference in principle between proof and refutation calculi. In fact, the tableau calculus could as well be formulated as a proof calculus for proving the validity of a formula.

The idea behind the tableau calculus is that, in order to prove a formula F, we assume $\neg F$ and try to derive a contradiction by successively splitting F into its subformulas (modulo some kind of substitution) until we obtain a formula together with its negation. As an example, assume we want to prove $\neg\forall x Px \vee (Pa \wedge Pb)$. We proceed as shown in Figure 1.

At each step in the proof, the truth or falsehood of a formula is asserted, and at a later step this formula may split into its immediate subformulas to derive their truth or falsehood, this way analyzing the formula (whence *analytic*). If a formula whose truth is asserted has the form of a disjunction then splitting of that formula produces a case analysis. The same

[8] Recently some authors prefer to speak of parts with positive and negative *polarity*.

[9] Any calculus that starts with the formula to be proven, reducing it until some termination criterium is satisfied, is called *analytic*. In contrast, a *synthetic* calculus derives the formula to be proven from axioms.

1. Assume $\neg\forall x Px \lor (Pa \land Pb)$ is false.
2. Then $\neg\forall x Px$ is false, and
3. $Pa \land Pb$ is false.
4. $\forall x Px$ is true by 2.

Since $Pa \land Pb$ is false, there are two cases, 5. and 6.

5. Pa is false.	6. Pb is false.
7. Pa is true by 4.	8. Pb is true by 4.
Contradiction between 5. and 7.	Contradiction between 6. and 8.

Fig. 1. An informal proof of $\neg\forall x Px \lor (Pa \land Pb)$ by contradiction

1. $\neg(\neg\forall x Px \lor (Pa \land Pb))$
2. $\neg\neg\forall x Px$
3. $\neg(Pa \land Pb)$
4. $\forall x Px$

5. $\neg Pa$	6. $\neg Pb$
7. Pa	8. Pb

Fig. 2. A closed tableau for $\neg(\neg\forall x Px \lor (Pa \land Pb))$

happens with conjunctions whose falsehood has been asserted. So, the derivation takes the form of a tree, called a *tableau*. The tableau for our example is shown in Figure 2, with the numbers 1–8 added to show the correspondence with the informal proof of Figure 1.

We now describe the tableau calculus in a more formal way. In the tableau method the formulas are categorized according to their syntactical structure. We distinguish five classes of formulas, namely literals, α-, β-, γ- and δ-*formulas*, as shown in Table 5. Other logical operators such as implication are assumed to be defined by those shown in the table. For each α-formula α we define two formulas α_1 and α_2, and for each β-formula β we define two formulas β_1 and β_2. For each γ-formula γ and each variable-free term t we define a formula $\gamma(t)$, and for each δ-formula δ and each constant a we define a formula $\delta(a)$, according to Table 5.

The rules of the tableau calculus used to successively construct a tableau for a given formula, are

α-RULE
$$\frac{\alpha}{\alpha_1} \quad \text{and} \quad \frac{\alpha}{\alpha_2}$$

β-RULE
$$\frac{\beta}{\beta_1 \mid \beta_2}$$

α	α_1	α_2
$\neg\neg F$	F	F
$F \wedge G$	F	G
$\neg(F \vee G)$	$\neg F$	$\neg G$

β	β_1	β_2
$F \vee G$	F	G
$\neg(F \wedge G)$	$\neg F$	$\neg G$

γ	$\gamma(t)$
$\forall x F$	$F\{x\backslash t\}$
$\neg\exists x F$	$\neg F\{x\backslash t\}$

δ	$\delta(a)$
$\exists x F$	$F\{x\backslash a\}$
$\neg\forall x F$	$\neg F\{x\backslash a\}$

Table 5. α-, β-, γ- and δ-formulas

γ-RULE $\quad \dfrac{\gamma}{\gamma(t)}$ where t is a variable-free term

δ-RULE $\quad \dfrac{\delta}{\delta(a)}$ where a is a new constant.

Definition 2.5.1. Let $\mathcal{T}$ be a tree whose nodes are marked with formulas. Let $\mathcal{T}'$ be a tree obtained by appending to the end of some branch of $\mathcal{T}$ the formula α_1 or the formula α_2 as its sole successor, where α is an α-formula occurring somewhere on that branch. Then we say that $\mathcal{T}'$ is obtained from $\mathcal{T}$ by *application of the α-rule*. The concepts of application of the γ- and δ-rule are defined in an analogous way. In the case of the β-rule $\mathcal{T}'$ is obtained by appending both β_1 and β_2 as two different successors (so that the tree splits here in two different branches).

Definition 2.5.2. A *tableau* for a formula F is a tree obtained by repeated application of α-, β-, γ- and δ-rules from the trivial tree consisting only of its root marked with F. A branch of a tableau is said to be *closed* if it contains a formula and its negation. A tableau is said to be *closed* if each of its branches is closed. A *refutation* of a formula F in the tableau calculus is a closed tableau for F. A *proof* of F in the tableau calculus is a refutation of $\neg F$.

For example, the tableau depicted in Figure 2 is a proof of $\neg\forall x Px \vee (Pa \wedge Pb)$. Informally, the closed property of the tableau means that for each case in the analysis represented by the tableau, a contradiction is derived.

As we shall see in Section 3, the tableau calculus is essentially an inverted cut-free sequent calculus in the sense that the rules of the sequent calculus are applied in the backward direction.

The tableau calculus in conjunction with the feature of unification discussed below in Section 2.6 is suitable for automation. For a fuller discussion of the automation possibilities of tableaux we refer the reader to Fitting's book [Fit90].

2.6 Unification

In the sequence of calculi presented so far, we may observe an increasing suitability for the automation of deduction. In particular, we have seen that the tableau calculus is an analytic calculus which takes the given formula and breaks it up into its subformulas modulo some kind of substitution of variables. At each stage of the proof process there are only a finite number of ways to perform the next inference step provided we find some way of determining this substitution. This is the point where unification, defined in detail in Chapter 4, enters the stage in automated deduction. It can be incorporated already in the tableau calculus, as

$$\neg(\exists u \exists v P(u, f(v)) \vee \exists x \neg P(g(x), x))$$
$$|$$
$$\neg\exists u \exists v P(u, f(v))$$
$$|$$
$$\neg\exists x \neg P(g(x), x)$$

Fig. 3.

$$\neg(\exists u \exists v P(u, f(v)) \vee \exists x \neg P(g(x), x))$$
$$|$$
$$\neg\exists u \exists v P(u, f(v))$$
$$|$$
$$\neg\exists x \neg P(g(x), x)$$
$$|$$
$$\neg\exists v P(r, f(v))$$
$$|$$
$$\neg P(r, f(s))$$
$$|$$
$$\neg\neg P(g(t), t)$$

Fig. 4.

we now demonstrate with a simple example, and is a fundamental ingredient in virtually all calculi for automated deduction, including those discussed further below.

Suppose we want to prove

$$\exists u \exists v\ P(u, f(v)) \vee \exists x\ \neg P(g(x), x)$$

using the tableau calculus. We have to find a closed tableau for the negation of this formula. In the first two inference steps we apply the α-rule to the root of the tableau and obtain the tableau shown in Figure 3. In order to close this tableau the γ-rule has to be applied twice to the formula $\neg\exists u \exists v\ P(u, f(v))$ and once to the formula $\neg\exists x\ \neg P(g(x), x)$. This means we have to replace the variable u with some term r and the variable v with some term s in the formula $\neg P(u, f(v))$ which is obtained from $\neg\exists u \exists v\ P(u, f(v))$ by omitting the quantifiers. Thus we get the formula $\neg P(r, f(s))$. We say that $\neg P(r, f(s))$ is obtained from $\neg P(u, f(v))$ by application of a *substitution*. Similarly, we have to replace the variable x with some term t in $\neg\neg P(g(x), x)$ to obtain the formula $\neg\neg P(g(t), t)$. The resulting tableau is shown in Figure 4.

This tableau is closed if and only if the two atomic formulas $P(r, f(s))$ and $P(g(t), t)$ are identical. In our example the problem of proving the given formula in the tableau calculus has thus been reduced to the problem of finding a substitution[10] $\sigma = \{u\backslash r, v\backslash s, x\backslash t\}$ such that $P(u, f(v))\sigma = P(g(x), x)\sigma$. In other words, σ is a unifier of the two atomic formulas $P(u, f(v))$ and $P(g(x), x)$. The substitution $\{u\backslash g(f(v)), x\backslash f(v)\}$ is such a unifier of the two literals and is in fact a most general one. These concepts will bee introduced in detail in the Chapter 4. A brief definition is also contained in the subsequent section.

2.7 The connection method

The connection method has its roots in the calculi discussed in the preceding subsections. In a sense it may in fact be regarded as a compressed version of those formalisms designed for the purposes of automated deduction, in which as much redundancy is eliminated as possible. These roots may be traced in the papers [Bib74, BS75], while a comprehensive treatment may be found in [Bib87a]. The method has been developed independently by Andrews [And76, And81, And89] with different terminology (see [Bib83] for the detailed correspondences and differences).

Like its predecessors, the connection method can handle arbitrary formulas of first-order predicate logic directly. They need not be in clausal (or clause or normal) form as in resolution. However, using the transformation given in Section 3.1, any formula of first-order predicate logic can be transformed in a natural way to a formula in clausal form. This transformation suggests that only negligible disadvantages result from it for the proof search in comparison with that carried out on the original formula (but no formal comparison has been published). We may therefore restrict our attention to the clausal form versions of the connection method which makes it easier to compare with resolution.

The connection method may be presented as an affirmative method for proving the validity of a formula in disjunctive normal form while resolution (discussed in Section 2.9 as well as in the next chapter) is a refutational method for proving the unsatisfiability of a formula in conjunctive normal form. As already pointed out in the previous subsection, this difference is however an absolutely negligible one since both methods could as well be presented in either way with no difference whatsoever in the computational consequences.

Let us consider a simple example, namely the formula

$$Ms \leftarrow \forall x(Hx \rightarrow Mx) \wedge Hs$$

which, with its obvious mnemonic abbreviations, may be read as "Socrates is mortal since all humans are mortal and Socrates is a human".

For simplicity let us use the simple usual transformation of such a formula into clause form as described in detail in Chapter 4 (see Section 4.2.2). In brief, propositional operators other than negation, conjunction, and disjunction are replaced by these, negation is eliminated except in literals, and quantifiers are eliminated by skolemization and by convention. In the affirmative case, the resulting formula is put into disjunctive normal form, in the refutational case into conjunctive normal form. Applying all these transformations to our formula affirmatively yields

$$(Ms) \vee (Hx \wedge \neg Mx) \vee (\neg Hs)$$

Clause form represents such a formula as a set of clauses where clauses are sets of literals. In our example the clauses are

$$\{Ms\}, \{Hx, \neg Mx\}, \{\neg Hs\}$$

In comparison the transformation refutationally yields the clauses

$$\{\neg Ms\}, \{\neg Hx, Mx\}, \{Hs\}$$

As we see, the only difference lies in a switch of signs, ie. positive (unnegated) literals are becoming negative (negated) ones, and vice versa. For the purposes of presentation we use a

[10] By $\{x_1\backslash t_1, \ldots, x_n\backslash t_n\}$ we denote the substitution replacing all occurrences of the variable x_i with the term t_i for $i = 1, \ldots, n$.

two-dimensional display of clauses in which the clause structure will occasionally be emphasized by the use of brackets. In simple cases like the present one the mere spacing should make the inherent structure clear to the eye. So in the affirmative case our example in form of a matrix reads as follows.

$$\begin{array}{ccc} & \neg Mx & \neg Hs \\ Ms & Hx & \end{array}$$

Here the clauses are represented in the form of a matrix whose columns represent the clauses (or disjuncts). In other words, conjunction is represented by a vertical listing, disjunction by a horizontal one. This conforms with standard forms of two-dimensional representation also for refutational approaches such as resolution, in which the refutational clause form above would read

$$\begin{array}{cc} Hs & \\ Mx & \neg Hx \\ & \neg Ms \end{array}$$

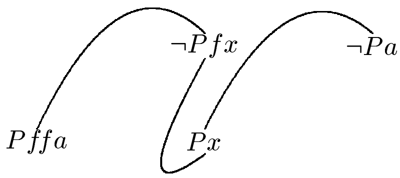
If, in this latter representation of this particular example, the negation sign is replaced by a right-to-left arrow then we get the form used in PROLOG. A comparison of the two matrices shows that, apart from the sign switches already mentioned, the former representation yields the latter by way of a 90^o counter-clockwise rotation. Because of this simple relationship it does not really matter which representation we prefer. For historical reasons[11] we retain the affirmative form and turn our attention now to the first of these two matrices.

A *path through* such a matrix is a set of literals (literal occurrences, to be precise), one taken from each clause. There are exactly two paths in our example, viz. $\{Ms, \neg Mx, \neg Hs\}$ and $\{Ms, Hx, \neg Hs\}$. A *connection* in a matrix is an unordered pair of literals (ie. literal occurrences) with the same predicate symbol, but different signs. There are two connections in our example, viz. $\{Ms, \neg Mx\}$ and $\{Hx, \neg Hs\}$. A connection is called *complementary* if both literals are identical except for the negation sign. In our example this is only the case after substitution of the variable x by the constant s. A *mating* is a set of connections (such as the two connections in our example). A mating is said to *span* a matrix (or is called spanning) if each path contains a connection from the mating (which is the case for our two connections). A mating is called *complementary* if each of its connections is complementary. A *substitution* is a mapping of a set of variables into a set of terms. A *unifier* of a set of terms is a substitution rendering all its elements equal. A unifier is called *most general* if any other unifier may be obtained from it by applying some additional substitution.[12]

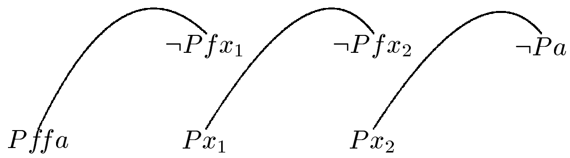
If, for a given matrix M, there exists a mating C that spans M, and a substitution σ such that $C\sigma$ is complementary, then M is a valid formula. As we just noted, the premise of this theorem is true for our example, hence we may conclude its validity. The following example demonstrates an additional feature not present in the previous one.

[11] Note that the history of logic begins centuries before resolution was invented.

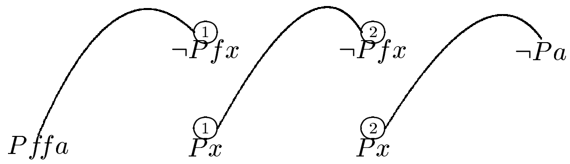
[12] For details concerning the last three concepts see Chapter 4.



Here we have illustrated the three connections of the matrix with connecting lines as we will mostly do in the paper. While this mating spans the matrix, there is no substitution that makes it complementary. Yet the formula is valid. The point is that we need account for the possibility of considering multiple instances of clauses and specify for the connections in which of the instances the literal occurrence is contained. In the present case we need two instances of the middle clause, the left connection enters the first instance, the right connection leaves the second instance, and the vertical connection connects the bottom literal of the first instance with the top literal of the second. For illustration one may depict this in the following way of an *expanded* (or *amplified*) matrix obtained by *duplication* of one of its clauses.

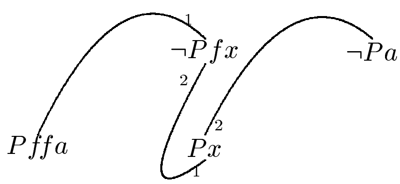


Here we have used the obvious idea to distinguish the two copies of the one clause by indexing the variables since these are the ones to be treated differently in determining an appropriate substitution. We may as well associate the index with each copy as a whole, or with its connected literals for that matter, for achieving the same effect. This is illustrated in the following picture.



In trying to find a substitution making all connections complementary, the effect of the indices associated with the ends of the connections, ie. with the connected literals, is the transfer of the index to each of the variables of the connected literal (as explicated in the preceding picture).

The last picture makes obvious that the explicit representation of the two clause copies is completely redundant since both clauses are absolutely identical (the relevant information being carried in association with the connections). So we use one copy only as shown in the following picture. One may think of obtaining it from the previous one by merging the two copies, thereby pulling the connections along in a natural way.



Matrices with indices associated with their connections (or clauses) will be referred to as *indexed matrices*. The notion of a path through an indexed matrix derives in a natural way from that of a path through the same matrix in its expanded form. In consequence the same is true for the concept of a spanning mating. Since matrices are just representations of formulas all notions discussed so far apply to formulas as well. With all these preliminaries we may now state the fundamental theorem on which the connection method is based.

Theorem 2.7.1. *A formula F is valid iff there exists an indexed form of F, a mating C that spans it, and a substitution σ such that $C\sigma$ is complementary.*

For a detailed proof see III.6.4 in [Bib87a]. To give an idea of the "if"-case, consider the indexed form of F in its expanded form with the substitution σ already applied. A transformation of this (affirmatively represented) matrix to conjunctive normal form would result in a set of conjuncts that is identical with the set of paths through the matrix. Each conjunct has to be valid to establish validity for the entire conjunction. Since each conjunct consists of a disjunction of literals, the spanning property implies that two of them are complementary ones, so that validity becomes obvious. Note that these arguments are not restricted to formulas in normal form (as we mentioned in the outset of this subsection).

On the basis of this characterization it is now obvious that the present example as well is valid. We take two indices for the second clause and the mating as in the (last) picture along with the substitution that makes it complementary. This mating spans the indexed matrix. For instance, take $\{Pffa, (Px)_1, (\neg Pfx)_2, \neg Pa\}$ out of the four possible paths to see that it contains the connection $\{(Px)_1, (\neg Pfx)_2\}$; similarly for the other three.

While this theorem characterizes the connection *method*[13], it leaves ample room for deriving specific connection *calculi* from it. So the connection method characterizes a family of calculi of the sort mentioned in Section 1. Any particular calculus that somehow checks the set of paths of the given formula in the way prescribed by the theorem is considered a family member.

In a brief look into this family, let us begin with the remark that not all deductive calculi may naturally be regarded as connection calculi. For instance resolution, discussed in the next subsection (and in great detail in the next chapter), generates at each step a new clause (viz. the resolvent) to be added to the previous clauses, so that the paths to be tested after one or more resolution steps differ from those in the original matrix, and in fact they do so in a rather complex way. Thus clause graph resolution discussed in detail in Chapter 4, and often also called connection graph resolution (or even the connection graph method), is not a connection calculus (despite the confusing occurrence of the word "connection" in this second name). However, there is a way to consider even resolution as a connection calculus, which is through consolution discussed in Section 2.8.

Model elimination [Lov69] historically may be regarded as the first detailed "connection calculus" (although it was of course developed from a rather different perspective). To illustrate how this particular calculus would be applied in searching for a proof, consider again our present example.

We select any clause, say the goal clause $Pffa$ (and any literal — or subgoal — in it, but there is no choice here). Next we select a connection containing this subgoal, viz. $\{Pffa, (\neg Pfx)_1\}$, and a substitution σ that unifies the connected literals ($\sigma = \{x_1 \backslash ffa\}$ in the present case). This step is called an *extension* step. In Section 4.2 a calculus using this rule as its essential proof mechanism is called an *extension* calculus. After performing this extension step, we have verified that all those paths through the matrix that contain this connection are in fact complementary so that only the remaining ones yet have to be tested. These include all paths that contain the previous subgoal $Pffa$ and the remainder of the clause connected with it, here the literal $(Px)_1\sigma$. From there we iterate what we have just done for the previous subgoal; and so forth. Briefly, this next iteration might attempt a connection with the right-most clause which fails since the terms do not unify. Alternatively, only a second instance of the middle clause may be taken into consideration, from where the proof completes in a straightforward way.

More details for this kind of calculus are provided in the Sections 4.2 and 4.3. There are

[13]Some authors use the term *matrix method* instead; but the method works for any formula representation, not only matrices.

many ways to improve this particular calculus in terms of its computational properties, as there are many alternate ways of designing connection calculi other than those of the model elimination type. Historically, the first proof procedure in the spirit of a connection calculus is the one by Prawitz [Pra60]. It uses explicit expansions and a form of unification realized by sets of equations.

Also the Davis–Putnam procedure [DP60] can be understood as a connection calculus. It is restricted to the case of normal form formulas in propositional logic for which it works fairly well. For this and for historical reasons, we will briefly explain its main features.

The procedure applies the simplification rules *tautology*, *purity*, *unit*, and *subsumption* discussed in Section 4.4. In addition it uses the following rule (cf. [Lov78]).

Splitting rule Choose a literal P such that P and its complement $\neg P$ both are present in the clause set and replace the set of clauses by the following two sets of clauses: (i) the set of clauses with all clauses containing P removed and all occurrences of $\neg P$ removed; (ii) the set of clauses with all clauses containing $\neg P$ removed and all occurrences of P removed.

This rule may be regarded as a combination of *factoring* (again see Section 4.4), Prawitz' matrix reduction [Pra70], and purity (in this sequence). Matrix reduction is defined by the following rule.

Matrix reduction Choose two clauses, one containing a literal P, the other its complement $\neg P$. In order to show that all paths through the matrix are complementary, it is sufficient to verify it for the following two subsets of paths: (i) the set of paths containing the chosen occurrence of P, but not that of $\neg P$; (ii) the set of paths containing the chosen occurrence of $\neg P$, but not that of P.

A comparison of these two rule definitions, deliberately phrased in a rather similar way, reveals their close relationship just mentioned. The path sets in the matrix reduction rule may as well be represented as sets of clauses. But the definition in terms of path sets makes its nature as a connection calculus explicit. Since all simplification rules may be regarded as reductions of path sets (see Section 4.4), this nature is inherited by the Davis–Putnam procedure through the relationship thus described.

Prawitz' matrix reduction may easily be generalized to first-order logic [SC77], while the splitting rule is of limited value in this more general context. Matrix reduction has been generalized in two different directions by [Bib79] and by [MR91]. The first allows more than just one propositional variable to be taken into consideration at each step. The second allows the application to non-normal form matrices.

From some point of view Maslov's inverse method [Mas68] may be seen as a connection calculus, but we will present it in the context of resolution within the next subsection. A number of further connection calculi are presented in Chapter IV of[Bib87a] where also further references to the respective literature may be found.

While the connection method may use Skolem functions as do other methods, it also offers an alternative that dispenses with them and instead uses a relation among the terms originally present in the formula (see Section IV.8 in [Bib87a] for the details). This alternative reflects the close relationship with the calculi discussed in the preceding subsections as the relation has its origin in the proof tree relation of these calculi.

2.8 Consolution

A rather simple, yet powerful connection calculus is *consolution* [Ede91]. The idea behind this calculus is the following. In order to prove the validity of a given formula using the connection method, all paths through its matrix have to be checked for complementarity. At each stage of a connection proof a certain set of paths has already been checked for complementarity

$$\begin{tikzcd}[column sep=small]
\{\{P\},\{Q\}\} \arrow[dr, dash] & & \{\{\neg P\},\{Q\}\} \arrow[dl, dash] & \\
& \{\{P,Q\},\{Q,\neg P\},\{Q\}\} \arrow[dr, dash] & & \{\{\neg Q\}\} \arrow[dl, dash] \\
& & \{\} &
\end{tikzcd}$$

Fig. 5. A consolution proof

whereas all the remaining paths still have to be processed this way. In consolution at each stage of a proof process, this remaining set of paths yet to be checked for complementarity is explicitly represented. It is coded in the form of partial paths which represent the set of all their extensions through the whole matrix. Let us consider an example which for simplicity is taken from propositional logic.

Suppose we want to prove the validity of the formula

$$(P \wedge Q) \vee (\neg P \wedge Q) \vee \neg Q.$$

In Figure 5, a proof tree of this formula by consolution is shown. Each node of the tree is marked with a set of paths in the matrix

$$\begin{array}{ccc} P & \neg P & \neg Q \\ Q & Q & \end{array}$$

representing the given formula. Note the distinction between a path *through* a matrix as defined further above, and a subset of such a path called a (partial) path *in* the matrix [Bib87a]. A leaf of the proof tree is marked with the set $\mathcal{P}_c$ of all one-element paths through some clause c of the formula, ie. $\mathcal{P}_c = \{\{L\} \mid L \in c\}$. This establishes a correspondence of leaves and the clauses of the given formula.

The inference rule, also called *consolution*, takes the set of paths from each premise and combines these sets into a new set, called the *consolvent*. This combination can be regarded to consist of two parts. The first part is defined as the product $\mathcal{PQ}$ of $\mathcal{P}$ and $\mathcal{Q}$ determined in the following way.

$$\mathcal{PQ} = \{p \cup q \mid p \in \mathcal{P} \text{ and } q \in \mathcal{Q}\}$$

For example, if $\mathcal{P} = \{\{P\},\{Q\}\}$ and $\mathcal{Q} = \{\{\neg P\},\{Q\}\}$ then $\mathcal{PQ} = \{\{P,\neg P\},\{P,Q\},\{Q,\neg P\},\{Q\}\}$. Note that this is equivalent to distributing $\wedge$ over $\vee$ and omitting duplicates from disjunctions. The second part of the inference rule consists in simplifications of $\mathcal{PQ}$. One such simplification is the elimination of complementary paths. To continue the illustration of our example, this means that the complementary path $\{P,\neg P\}$ is removed from $\mathcal{PQ}$. Actually, both parts are to be seen as a single operation, a remark which bears its relevance on the efficiency of implementation. But for the ease of the reader's understanding we will continue to make the distinction.

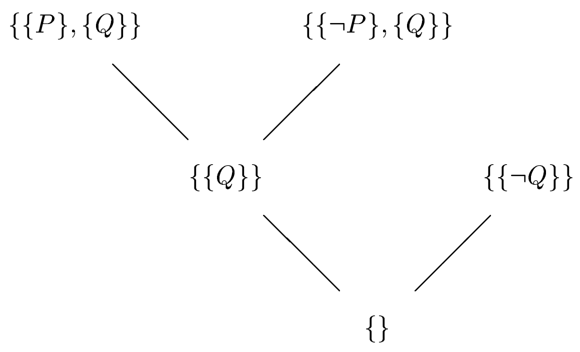


Fig. 6. A consolution proof with shortening of paths

The proof in our example is completed by applying consolution once again to the result of the previously illustrated step and the remaining leaf $\{\{\neg Q\}\}$. The resulting empty set is the criterion of a successful derivation (as in resolution). To summarize, for a formula F in disjunctive normal form, a *proof* of F is a derivation of the empty set from the sets $\mathcal{P}_c$ $(c \in S)$ with the consolution rule, where S is the set of clauses of F.

While this explains the essentials of the calculus, the following additional details complete its description. Above we have used a single simplification which is elimination of complementary paths. This is all needed for completeness and soundness of the calculus. For efficiency, it is necessary to incorporate at least the following simplification (since otherwise a huge number of paths will quickly be generated in practice). Any path p in $\mathcal{P}\mathcal{Q}$ may be replaced by any subset of p. Both simplifications may be applied simultaneously. So in our present example, the first consolvent $\{\{P,Q\},\{Q,\neg P\},\{Q\}\}$ may be further reduced, for instance to $\{\{Q\}\}$. This reduction corresponds to the fact that $Q \models (P \vee Q)$ and $Q \models (Q \vee \neg P)$ and $Q \models Q$. Note that three distinct paths have been replaced by a single one in this case. In general, the number of paths may be reduced considerably this way. There is a drawback, however, since the path sets associated with a node may be used more than once as a premise in a consolution step while this would not be the case, had we not allowed this additional simplification. If we incorporate this simplification into the derivation given in Figure 5, then a consolution proof of the same formula results which is shown in Figure 6. In summary, any consolution calculus must, among its simplifications within the consolution rule, include the elimination of complementary paths. While it is not absolutely necessary, the shortening of paths is understood to be always included in consolution. As an aside we mention that such an inclusion amounts to an extension of the basic calculus. A further extension of consolution might include the matrix reduction (as defined further above in this section) as a simplification which deletes more than just the complementary paths.

Consolution is lifted to the first-order level in exactly the same way as resolution (see next section and Chapter 4) by use of unification as described in the previous section. In detail, variables in the premises of a consolution step are separated apart (by renaming) and a substitution determined by unification is applied to the consolvent in order to make paths complementary or in order to merge two merge paths into a single one (as in the simplification of shortening of paths).

As a further remark we point out that the sets resulting by consolution from paths in a given matrix are not always paths in the matrix as in the example shown above. Rather they may also be the union of such paths even in propositional logic. For illustration, the reader may also think of paths in a matrix that has multiple occurrences of clauses. As in any connection calculus (or in any deductive method for that matter) a number of additional instances of the given clauses may be needed for the proof in the first-order case.

2.9 Resolution and the inverse method

As we already pointed out in Section 1 resolution is the best known and most widely used method for automated deduction. Many refinements are known for it. They provide the basis for numerous implementations. Because of the importance attested by this popularity, this volume devotes an entire chapter to resolution and its refinements. Here we briefly describe the resolution rule for completeness and for comparison only.

As in the previous section we restrict the discussion to normal form following common practice. As opposed to the affirmative form of proofs in the connection method, resolution for historical reasons uses a refutational form of proofs establishing unsatisfiability (rather than validity) of formulas. This has the consequence that we are dealing here with the conjunctive rather than disjunctive normal form. A formula thus is represented as a finite set of clauses, and the idea is to derive new clauses which are logical consequences of these given clauses until the empty clause appears. The empty clause represents a contradiction.

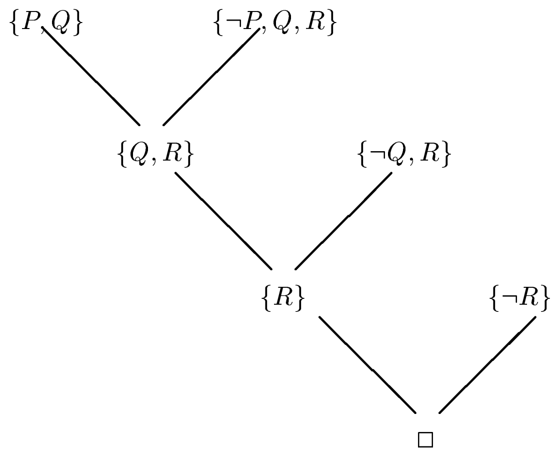
In propositional logic the resolution rule is as follows. From two clauses $c \cup \{P\}$ and $d \cup \{\neg P\}$ the clause $c \cup d$, called the *resolvent* of these two *parent clauses*, is derived. Here, c and d are sets of literals, and P is an atomic formula. The literals P and $\neg P$ are said to be the *literals resolved upon*. As an example of a refutation of a formula by resolution, suppose we want to prove the unsatisfiability of the formula

$$(P \vee Q) \wedge (\neg P \vee Q \vee R) \wedge (\neg Q \vee R) \wedge \neg R.$$

Its set of clauses is

$$\{\{P, Q\}, \{\neg P, Q, R\}, \{\neg Q, R\}, \{\neg R\}\}$$

and the following tree represents a resolution refutation of this set.

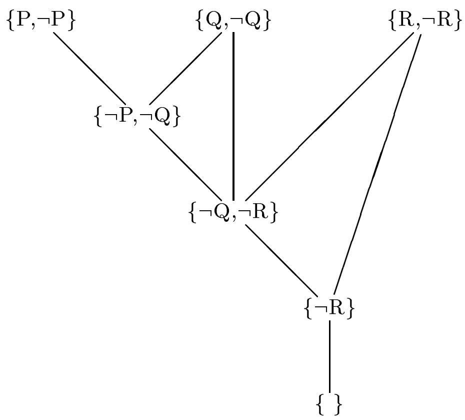


Resolution is *lifted* to first-order predicate logic by weakening the condition that the literals resolved upon be complementary. It is only required that they are complementary after application of a suitable substitution which is determined by unification similarly as discussed in Section 2.6. For a more detailed introduction to resolution we refer the reader to the next chapter.

Let us refer once again to the set of clauses

$$\{\{P, Q\}, \{\neg P, Q, R\}, \{\neg Q, R\}, \{\neg R\}\}$$

just used for the explanation of resolution in order to introduce a method which in some sense may be regarded dual to resolution, viz. Maslov's inverse method [Mas68]. The following is a derivation for our example using the inverse method.



While resolution initially operates on the given set of *clauses*, the inverse method operates on the *connections* in these clauses; further while each resolution step is triggered by one particular *connection*, a step in the inverse method is determined by a particular *clause*. This explains the duality between clauses and connections, the role of which is exchanged in the inverse method as compared to resolution [Dav73, Zam90].

For instance consider the first step in the depicted derivation. Its premises are the two connections $\{P, \neg P\}, \{Q, \neg Q\}$ and it is determined by the first clause $\{P, Q\}$ in the given set, which is a subset of the union of the premises. The complement of the subset in this union is the "resolvent" of the step which is $\{\neg P, \neg Q\}$ in the case of this first step. The general rule is

$$c_1 \cup \{L_1\}, \ldots, c_n \cup \{L_n\} \vdash c_1 \cup \ldots \cup c_n$$

whereby $\{L_1, \ldots, L_n\}$ is a clause from the given set and $c_i \cup \{L_i\}$ are connections or previously derived sets of literals.

From a different viewpoint, the inverse method may as well be regarded as a connection calculus (it focuses on connections after all) which, however, uses a control strategy different from model elimination (discussed in the previous subsection). Starting with some clause such as $\{P, Q\}$ in the example, rather than considering the subgoals of a clause determined by a selected connection (as explained for model elimination in the previous subsection), it first solves all subgoals of the start clause by considering a connection for each of its literals, encoding the remaining task in the set of connected literals not in the start clause. Lifting of all these considerations to the first-order level is done as with resolution.

We mention once again that there are many restricted versions of resolution (as well as of the inverse method), usually called *refinements*. We refer to Chapter 4 as well as to Volume 4, Chapter 8 for some of these.

2.10 The connection structure method

The connection structure method introduced in [Ede92] is a method generalizing the connection method. In order to illustrate the basic idea underlying connection structures let us consider the following example.

$$\begin{array}{ccc} & Px & \\ \neg Pa & & Pf^8a \\ & \neg Pfx & \end{array}$$

f^8a stands for $ffffffffa$. A connection proof using the extension calculus mentioned in Section 2.7 needs eight instances of the second clause and the set of connections shown in Figure 7

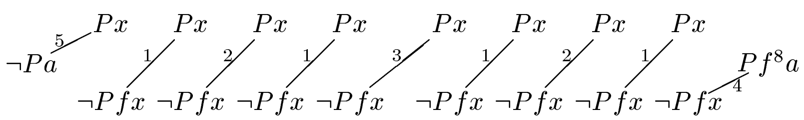


Fig. 7. An expanded matrix representing a connection proof

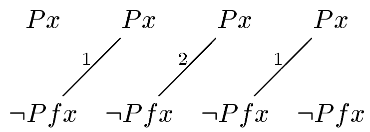


Fig. 8. A submatrix of the expanded matrix

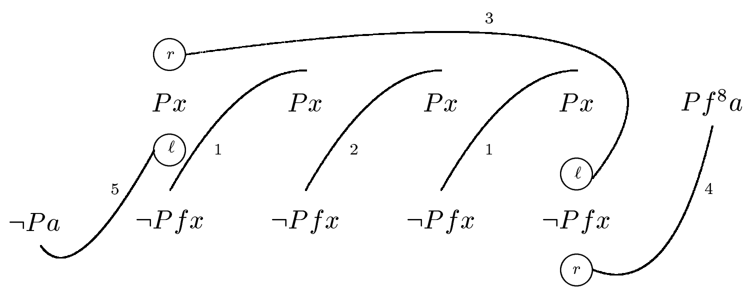


Fig. 9. The expanded matrix after the first compactification step

(with numbers attached to the connections to be ignored for the moment). More precisely, the different instances of the clause $\{Px, \neg Pfx\}$ are to be understood with indices attached to the variable x. This proof representation is extremely redundant as we can see from a closer look at the proof. The part shown in Figure 8 occurs twice in this expanded matrix. For a more compact proof representation one would wish to represent it only once which is what the connection structures achieve. Think of merging the right occurrence of this part into the left one in order to make them identical. Somehow the fact that in the previous matrix the literal $\neg Pa$ is connected with the *left* occurrence of this part rather than with the right one must be encoded. In the subsequent picture this is done by associating the label ℓ (for "left") with the right end of the connection labeled with 5. Similarly, the left end of the connection containing Pf^8a and labeled with 4 is associated with an r (for "right"). Finally, the left end of the connection labeled with 3 is associated with an ℓ and the right end with an r. Altogether we obtain the picture of Figure 9. In this proof the part

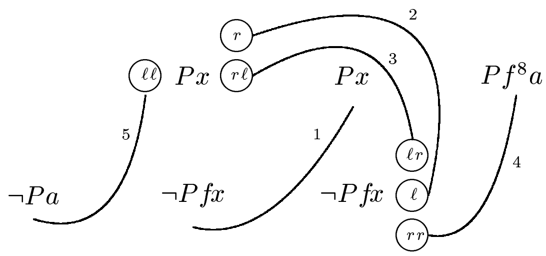


Fig. 10. The expanded matrix after the second compactification step

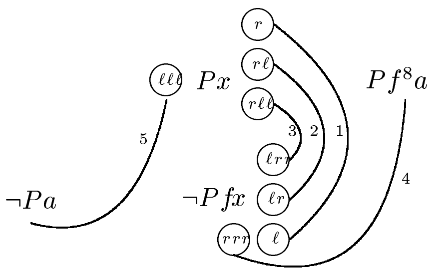
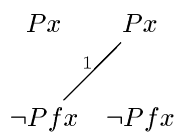


Fig. 11. The expanded matrix after the third compactification step



occurs twice again (ie. four times in the original matrix). As before we merge its right occurrence into the left one. Thereby labels are associated with the ends of the connections as before. Each end of a connection in the left occurrence of this part is associated with an ℓ and each end in the right occurrence with an r. If the end has already been associated with a label from the previous operation then the ℓ or r is appended. The result is shown in the Figure 10. This matrix still contains one clause twice so that the compaction can be performed once again resulting in Figure 11. Note that there is a lack in uniformity insofar as the connections 4 and 5 have ends without associated labels. We should actually have started this process with dividing the original matrix into three parts, namely into the leftmost one, ie. the clause $\neg Pa$, the rightmost one, ie. Pf^8a, and the middle one which has been compacted the way just explained. Therefore our final matrix in its fully compacted form actually looks as shown in Figure 12.

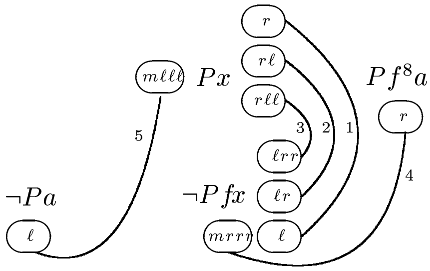


Fig. 12. The compact representation of Figure 14

The representation is incomplete in the sense that it does not yet contain the information about the order of the merging steps. The full information required is contained in the representation shown in Figure 13. It shows the given set S of clauses as leaves of a directed acyclic graph (dag) which is called a *connection structure*. Its inner nodes are marked with the connections depicted in Figure 12. Its edges are marked with selectors such as ℓ, m, and r for "left", "middle", and "right". These edges represent the hierarchy of the merging operations performed above. More precisely, each node represents a submatrix of the expanded matrix. The root represents the entire matrix and the leaves its clauses. As an example of an inner node, the node labeled $(\ell r.\neg Pfx, r\ell.Px)$ represents the submatrix shown in Figure 8. If a node n is below a node m, then the matrix represented by n is a submatrix of the matrix represented by m. If m and n are connected by an edge marked with ℓ then the lower node represents a left-hand submatrix, and similarly for the other selectors.

As we said above, the connection structure shown in Figure 13 is a refined form of the compact version shown in Figure 12 of the expanded matrix. This corresponing expanded matrix is shown in Figure 14. It is the same matrix as our original expanded matrix depicted in Figure 7 except that it now has indices attached to its variables distinguishing the different instances as required by the connection method. This expanded matrix can be read from the corresponding connection structure in Figure 13 in the following way. Each of its clauses corresponds to a unique branch from the root of the dag to a leaf of the dag marked with a clause of which it is an instance. The variables of this instance are indexed by the selector sequence along the branch. Note that there are exactly ten branches representing the ten clauses in the expanded matrix.

For a literal L and a selector sequence s let us denote by L_s the literal obtained from L by replacing each variable x with x_s. The connections shown in Figure 14 are coded by the pairs marking the nodes in the connection structure of Figure 13. These pairs have the form $(s.K, s'.L)$ where s and s' are selector sequences and K and L are literals. One such pair $(s.K, s'.L)$ encodes a whole set of connections, namely all connections $\{K_{ts}, L_{ts'}\}$ such that t is a selector sequence leading from the root of the connection structure to the respective node. In Figure 14, connections coded by the same pair, have been marked with the same number. For example, the pair $(\ell r.\neg Pfx, r\ell.Px)$ marking the third node from the bottom in Figure 13, can be reached from the root by any of the two selector sequences $m\ell$ and mr. Thus it encodes the two connections $\{\neg Pfx_{m\ell\ell r}, Px_{m\ell r\ell}\}$ and $\{\neg Pfx_{mr\ell r}, Px_{mrr\ell}\}$ marked with 2 in Figure 14.

A connection structure allows a concise representation of a set of instances of clauses and of a set of connections. In fact, it may represent a number of instances of clauses and connections which is exponential in its size. Moreover, whereas in the extension calculus there is only a very restricted possibility of the use of lemmata, the connection structure calculus allows multiple use of lemmata as does resolution. Each node in the dag corresponds to a lemma in this sense.

In the *connection structure method*, for a given set of clauses, a connection structure is constructed step by step starting from its leaves as roots of a dag. Moreover, each node of

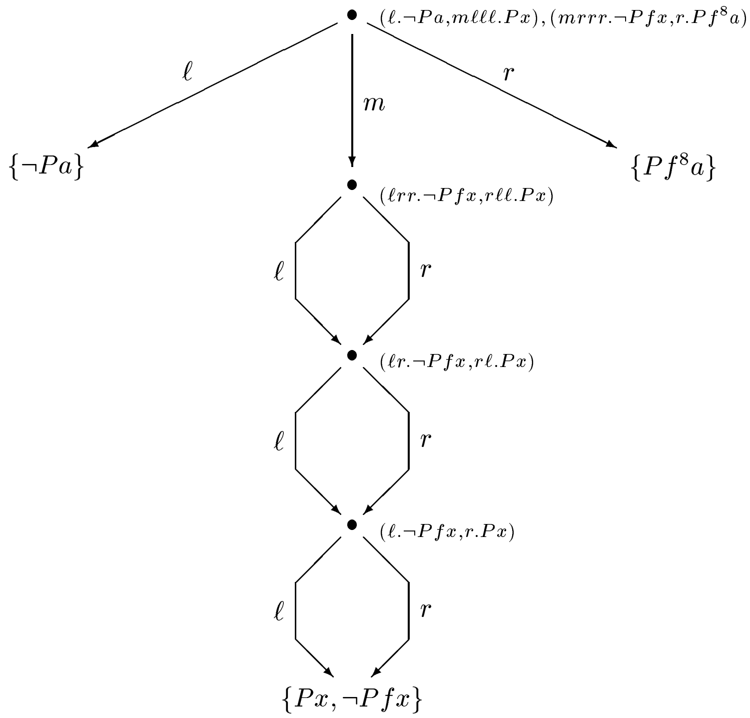


Fig. 13. A connection structure

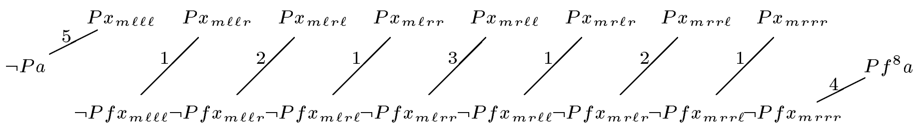


Fig. 14. The expanded matrix of the connection structure of Figure 13

this connection structure is annotated with a substitution and with a set of paths in a set of variants of the clauses of the given matrix. Hereby, each leaf (marked with a clause c) is annotated with the empty substitution ε and with the set $\mathcal{P}_c$ of (one-element) paths through c defined in Section 2.8.

At each step a new node is appended to one or more roots of the dag (called old roots in the following) as a new root instead of those with edges to them marked with distinct (otherwise arbitrary) selectors. Thereby, the substitution annotated with the new root is computed as follows. From each substitution annotated with an old root a new substitution is obtained by replacing each variable x with x_s where s is the selector marking the edge from the new root to the present old one. So the desired substitution results as the merge of all these new substitutions and the unifier of a consolution step 2.8. The parents of this step are the sets of paths annotated to the old roots with their annotated substitutions applied to them. An annotated connection structure is a *derivation* for the given set of clauses if there exists a most general unifier associated with its root and if the set of paths associated with its root is empty.

After the first inference step applied to our example, we have the two lower nodes of the connection structure shown in Figure 13 but without annotations as a new root and its two (identical) successors. The corresponding expanded matrix is

$$\begin{array}{cc} Px_\ell & Px_r \\ \\ \neg Pfx_\ell & \neg Pfx_r \end{array}$$

.

The most general unifier is $\{x_r \leftarrow fx_\ell\}$, and the set of paths not yet checked to be complementary is $\{\{Px_\ell, Px_r\}, \{Px_\ell, \neg Pfx_r\}, \{\neg Pfx_\ell, \neg Pfx_r\}\}$. Now, there is a rule of the connection structure calculus that allows us to shorten any path. For example, we can shorten the paths $\{Px_\ell, Px_r\}$ and $\{Px_\ell, \neg Pfx_r\}$ to merge them to one path. We then get the path set $\{\{Px_\ell\}, \{\neg Pfx_\ell, \neg Pfx_r\}\}$. In the next step we use this mgu and this path set to compute the new mgu and the new path set, and so on.

The completeness and soundness of the connection structure calculus is shown in [Ede92]. We mention that in the connection structure calculus it is also allowed to draw factorization links, ie. links between unifiable literals of identical rather than opposite sign. They are attached as labels to the nodes the same way that connections are. Factorization links may reduce the number of paths to be checked for complementarity considerably.

As a final remark we mention that connection structures contain more information than the path sets in consolution. They encode not only derived knowledge but also the history of a derivation

2.11 Extension and renaming

There is a powerful extension of any proof calculus. In the case of resolution it is called *extended resolution* or *resolution with extension*.

Extension is a rule that introduces the concept of definitions to a calculus. Intuitively speaking, an extension step consists of a definition of a new predicate P in terms of one or two predicates Q (or Q and R). The predicates Q and R are assumed to be given predicates, or predicates that have been defined in an earlier extension step. The extension rule was introduced by Tseitin [Tse70] for propositional logic and then generalized to first-order logic in [Ede92].

First we define a junctorial[14] extension step which can be regarded as an extension of the set of clauses by the definition of a literal J as $K \vee L$. The three clauses of S_0 in the

[14]A propositional connective is sometimes called a junctor, especially in German.

following definition are the result of transforming the equivalence $J \leftrightarrow K \vee L$ to clausal form. If applied in an iterative way, this step allows the abbreviation of a larger formula by a single symbol. Further handling of this symbol is obviously easier than handling the entire abbreviated expression from where the power of such extensions arises. Then we define a quantorial extension step which again can be regarded as adding the definition of J as $\exists x K$. Again, the two clauses of S_0 are the result of transforming the equivalence $J \leftrightarrow \exists x K$ to clausal form. The quantorial extension is the natural generalization of the junctorial extension as the existential quantifier may be regarded as a generalized disjunction. Extension covers both concepts at the same time. One of the roles of quantorial extension is similar to the role of Skolemization which is to remove complicated nestings of quantifiers.

Definition 2.11.1. A set of clauses S' is obtained from a set of clauses S by *junctorial extension* if $S' = S \cup S_0$ where[15]

$$S_0 = \{\{\neg J, K, L\}, \{J, \neg K\}, \{J, \neg L\}\},$$

J has the form $P(x_1, \ldots, x_k)$, and the following three conditions hold.

1. P is a predicate symbol not occurring in S, possibly preceded by a negation sign.
2. The variables $x_1, \ldots, x_k$ are pairwise distinct.
3. Every variable occurring in K or in L is in $\{x_1, \ldots, x_k\}$.

Definition 2.11.2. A set of clauses S' is obtained from a set of clauses S by *quantorial extension* if $S' = S \cup S_0$ where

$$S_0 = \{\{J, \bar{K}\}, \{\bar{J}, L\}\},$$

J has the form $P(x_1, \ldots, x_k)$, and there is a variable x such that the following four conditions hold.

1. P is a predicate symbol not occurring in S, possibly preceded by a negation sign.
2. The variables $x_1, \ldots, x_k$ and the variable x are pairwise distinct.
3. Every variable occurring in K is in $\{x_1, \ldots, x_k, x\}$.
4. There is an n-ary function symbol f not occurring in S such that L is obtained from K by replacing all occurrences of x in K by $f(x_1, \ldots, x_k)$.

The function f is called the *Skolem function* of the quantorial extension step.

Definition 2.11.3. A set of clauses S' is obtained from a set of clauses S by *extension* if S' is obtained from S by junctorial extension or by quantorial extension. S' is obtained from S by a step of *extended resolution* if S' is obtained from S either by adding a resolvent of clauses of S or by extension. A *refutation* of a set of clauses S is a finite sequence $(S_0, \ldots, S_n)$ where $S_0 = S$, each S_j is obtained from S_{j-1} by a step of extended resolution, and $\Box \in S_n$.

The extension rule is a rather powerful rule which allows in many cases for much shorter refutations than resolution alone. However, reasonable strategies for the extension rule are difficult to develop. The situation is similar as in the sequent calculus with the cut rule. The definition just given for the case of resolution carries over to other calculi. In any case one can always let the extension steps precede any other steps of the calculus in question.

A technique related with extension is renaming. Assume we have established the validity of some formula. To keep things simple, think of the formula $P \vee \neg P$. Further assume, we are faced with another problem consisting in establishing the validity of $Q \vee \neg Q$. We notice

[15] For readability we allow a slight sloppiness in our notation in that we understand $\neg\neg A$ as A.

that by renaming the predicate symbol in this formula we obtain the previous one for which the proof is already given. This is the idea behind the rule of *renaming* which is to substitute in a formula some of the predicate symbols consistently by others. The savings may seem negligible in our trivial example. In more complicated ones this rule is a rather powerful one. For instance, it has been shown in [Bib90] that with renaming there are linear proofs for the class of pigeonhole formulas which otherwise needs superpolynomial proofs.

A generalization of renaming is the substitution rule. Here predicate symbols may be substituted by nominal forms [Sch77] so that atomic formulas are replaced by arbitrary formulas. Both renaming and substitution may be regarded as a higher-order logic feature (see Section 5.5). Namely, in our example we would rather consider the formula $\forall P\,(P \vee \neg P)$. If the validity of this higher-order formula has been proved, then the usual rule of $\forall$–instantiation has the same effect as renaming and substitution.

2.12 Further calculi and summary

There are numerous further systems for establishing the validity of propositional or first-order formulas. Let us just mention a few of them.

Well-known is the *truth-table method* (for the propositional case) with which, for every possible truth-value assignment to the propositional variables, the value of the formula is computed. Further there is the method of *semantic trees* [Lov78] (for first-order logic) based on the same idea, but organized in a more formula-oriented way. Shannon's normal form (eg. see [Sha38, Koh70, Mor82]) (for the propositional case) amounts to a semantic tree coded in a slightly different form. The importance of factored forms (see Section 4.4) of formulas has recently been recognized also in the hardware community [MB88].

A huge amount of literature has been produced on the topic of rewriting. It is covered in Chapter 5 of the present volume. The point of interest in the present context is the possibility of representing first-order logic within the framework of rewriting which has been shown in [HD83].

While the variety of methods leaves plenty of room for detailed comparison, those not covered in this section do not seem to add any important novel idea to our topic.

Throughout the paper we are dealing with *classical* logic in which the *tertium non datur* — formulas are true or not true, and no third alternative exists — holds. In *intuitionistic* logic this principle is repudiated and a strictly constructive logic is demanded. In Section 5.5 we will briefly mention that nevertheless the deductive methods discussed in Section 2.7 (among others) apply to this logic as well. There also linear logic is briefly discussed.

We already mentioned in Section 2.6, the presentation of calculi in this entire section proceeded from the less to the better computationally suitable ones. The Frege–Hilbert calculi seem hardly useful for automated deduction. At the other end, there is plenty of evidence that calculi based on the connection (structure) method or on resolution are better suited for implementations. As we will see in the subsequent section, there are very close relations among all of these calculi. Therefore, in view of implementations the focus should be on these latter calculi. The others have their merits for other purposes. For instance, natural deduction calculi play an important role in the presentation of proofs for users (see Section 6.4). Extension and renaming are unexplored topics in terms of implementations.

3 Relations among logical calculi

As we pointed out in Section 1, understanding a given calculus in terms of another one deepens our understanding of the nature of validity. In the present section some results of this nature are presented. They refer to the question of how a proof in one calculus can be transformed to a proof in another calculus.

When comparing the complexities of theorem proving in two different calculi, a distinction has to be made between the complexity of a proof and the complexity of proof search. In this section we concentrate on the complexities of proofs, although the results also bear relevance upon comparison of complexities of proof search for suitable search strategies.

For a common basis we restrict our discussion to logic in clause form most of the time. Since from a computational point of view this is a serious restriction unless care is taken with the transformation, we begin with the presentation of a transformation that is suitable in this sense.

3.1 An efficient transformation to clausal form

In Section 2.7 we have briefly indicated the transformation to normal (or clausal or clause) form which is discussed in detail in Chapter 4. Every formula F of first-order predicate logic is transformed this way to a formula in clausal form which is satisfiable if and only if F is satisfiable. Note that the structure of the formula F is disrupted. Also, the length of the resulting formula grows exponentially wrt. the length of F as the following formula of propositional logic shows.

$$F \equiv (A_1 \wedge B_1) \vee \cdots \vee (A_n \wedge B_n).$$

Multiplying this formula out into conjunctive normal form using the distributive law yields 2^n clauses.

However, there is another transformation to clausal form that does not suffer from this exponential explosion. In fact, the length of the formula increases in this alternative at most by a factor given by the maximal depth of nesting of quantifiers in the formula. The number of subformulas increases only by a constant factor. The idea of this transformation to *definitional form* is to introduce, for each subformula G of F, a new name. Syntactically the name is a new predicate symbol P_G whose arity equals the number of free variables of G. To make this more precise, let L_G be the formula $P_G(x_1, \ldots, x_k)$ where $x_1, \ldots, x_k$ are the free variables of G. We introduce a formula D_G, intuitively serving as a definition defining L_G as an abbreviation for G, as follows. If G is atomic then let D_G be the universal closure of $L_G \leftrightarrow G$. If G is non-atomic, $G \equiv \circ(G_1, \ldots, G_n)$, then

$$D_G \stackrel{\text{def}}{=} \forall x_1 \ldots \forall x_k (L_G \leftrightarrow \circ(L_{G_1}, \ldots, L_{G_k})).$$

Here $\circ$ denotes either a propositional connective or a quantifier prefix. Let D be the conjunction of all D_G taken over all subformulas G of F.

Then the formula F is valid iff $D \to L_F$ is valid, and F is satisfiable iff $D \wedge L_F$ is satisfiable. Moreover, $D \models L_F \leftrightarrow F$. Now, the point is that $D \to L_F$ or $D \wedge L_F$, respectively, is a formula which is already very close to clausal form. Its transformation to clausal form using the standard procedure described in Chapter 4 leads only to a moderate increase of its length. The total space as well as time requirements for the whole transformation of F are at most of the order of the product of the length of F and the maximal depth of nesting of quantifiers in F. Also, the structure of the formula is not disrupted as by the usual transformation to clausal form. Therefore the transformation to definitional form and similar transformations to normal form are also called *structure-preserving* in the literature.

We illustrate the transformation with an example. Let F be the formula $\exists x(Rx \iff \forall y Ry)$. The subformulas of F are $G_0, \ldots, G_4$ where

$$G_0 \equiv \exists x(Rx \iff \forall y Ry)$$

$$\begin{array}{lcl} G_1 & \equiv & Rx \iff \forall y Ry \\ G_2 & \equiv & Rx \\ G_3 & \equiv & \forall y Ry \\ G_4 & \equiv & Ry \end{array}$$

Then $D_{G_0}, \ldots, D_{G_4}$ are given by

$$\begin{array}{l} D_{G_0} \equiv \quad P_{G_0} \iff \exists x P_{G_1} x \\ D_{G_1} \equiv \forall x (P_{G_1} x \iff (P_{G_2} x \iff P_{G_3})) \\ D_{G_2} \equiv \forall x (P_{G_2} x \iff Rx) \\ D_{G_3} \equiv \quad P_{G_3} \iff \forall y P_{G_4} y \\ D_{G_4} \equiv \forall y (P_{G_4} y \iff Ry). \end{array}$$

Each D_{G_i} is then transformed to a set of clausal form (conjunctive normal form) in the usual way. For example, from D_{G_3} we obtain $\forall y (\neg P_{G_3} \vee P_{G_4} y) \wedge (P_{G_3} \vee \neg P_{G_4} a)$ where a is a Skolem constant. So D_{G_3} yields the two clauses $\{\neg P_{G_3}, P_{G_4} y\}$ and $P_{G_3}, \neg P_{G_4} a$. Similarly, we get a number of clauses (at most 4) from each D_{G_i}. If we want to prove the validity of F using resolution, for example, we add the unit clause $\{\neg P_{G_0}\}$ to these clauses and try to refute the resulting set of clauses. If we want to prove the unsatisfiability of F, we have to add the unit clause $\{P_{G_0}\}$ instead and again try to refute the resulting set of clauses.

For complex formulas this transformation is more efficient than the standard one. The point is that each D_G has a simple structure no matter how complex F is, and the number of literals generated from each D_G is always small. Also, it can usually be seen immediately by polarity considerations that one or more clauses obtained from D_G are redundant. These clauses need not be generated in the transformation.

The idea of transforming a formula to clausal form by introducing definitions has been known to logicians for a long time. Similar transformations have been used by Thoralf Skolem in the proof of the Löwenheim-Skolem theorem in [Sko20][16] and by Kurt Gödel in the proof of his completeness theorem in [Göd30]. In 1966, G.S. Tseitin introduced the structure-preserving transformation to clausal form described above for the case of propositional logic. An English translation of his paper can be found in [Tse70]. Stephen Cook and Robert Reckhow used a similar transformation in [CR74b], [Rec76], and [CR79], as a translation between two languages of propositional logic in order to obtain simulation results for Frege systems with substitution and for extended Frege Systems.

In the field of automated theorem proving, the structure-preserving transformations to normal form began to become known and applied in actual implementations starting with [HLO+80]. Greenbaum, Nagasaka, Rorke and Plaisted demonstrate in [GNOP82] the structure-preserving transformation to clausal form on an example (without giving its explicit definition). A first formal definition of such a transformation for full first-order logic was given in [Ede85a] where also an estimate of its complexity can be found (see also [Ede87]). Similar transformations were given by Plaisted & Greenbaum [PG86] and by Poole [Poo84]. In [BdlT90], Boy de la Tour shows that, in some cases, an even slightly better behaviour in terms of the number of clauses generated, is obtained by hybrids between the standard and the structure-preserving transformations to clausal form.

Transformations to normal form reduce the problem of deduction in full first-order logic to the case of normal form fomulas which usually is much simpler. On the other hand, it is important in practice to make sure that the efficiency of the whole proof procedure does not suffer too much through the use of such a transformation. It has already been mentioned above that the structure-preserving transformations to clausal form increase the complexity

[16] Actually, the definitional form resulting from a given formula by the transformation described in the present section is more closely related to Skolem's normal form of the formula as presented in his paper than it is to what is usually called the Skolem normal form of the formula [Bib87a] today.

of a formula, measured by the number of its subformulas, only by a constant factor [Ede85a]. Moreover, if we use these transformations in conjunction with the extension calculus (see 2.7) of the connection method then the size of the search tree increases also by a constant factor only (while this fact is obvious, no formal proof exists in the literature). So the price to be paid in terms of efficiency for the comfort of using a simpler calculus is rather modest. While one might expect a similar modest increase for other calculi as well, details would have to be worked out.

3.2 Gentzen's Transformations of Derivations

The Frege–Hilbert calculus, the natural deduction calculus, and the sequent calculus (with the cut rule) introduced in Section 2 are very similar to each other. In fact, any derivation of a formula in one of these three calculi can be transformed to a derivation of the same formula in any of the other two calculi [Gen35].

For example, a derivation of a formula F in the Frege–Hilbert calculus can be transformed to a derivation of the formula F, ie. of the judgement $\Longrightarrow F$ in the natural deduction calculus. The idea is to transform each formula A occurring as a member of the given derivation to the judgement $\Longrightarrow A$. Thus we obtain a finite sequence of judgements. This sequence is not yet a derivation in the natural deduction calculus because each step in this sequence may correspond to more than one step in natural deduction. But the sequence can be completed to a derivation of F in natural deduction by inserting the missing judgements. For example, the axiom A8 in the Frege–Hilbert calculus corresponds to the derivation

$$\cfrac{\cfrac{A \wedge B \Longrightarrow A \wedge B}{A \wedge B \Longrightarrow A}\ (\wedge\text{-E})}{\Longrightarrow A \wedge B \to A}\ (\to\text{-I})$$

in natural deduction, and the rule R2 of the Frege–Hilbert calculus corresponds to

$$\cfrac{\cfrac{\cfrac{A \Longrightarrow A \qquad \Longrightarrow A \to F\{x\backslash a\}}{A \Longrightarrow F\{x\backslash a\}}\ (\to\text{-E})}{A \Longrightarrow \forall x F}\ (\forall\text{-I})}{\Longrightarrow A \to \forall x F}\ (\to\text{-I})$$

which is a derivation of $\Longrightarrow A \to \forall x F$ from $\Longrightarrow A \to F\{x\backslash s\}$.

Similarly, a derivation of a formula in the natural deduction calculus can be transformed to a derivation of the same formula in the sequent calculus. Namely, a judgement is essentially a special form of sequent, and each step in natural deduction can again be transformed to one or more steps in the sequent calculus.

To close the cycle, a derivation of a formula in the sequent calculus can be transformed to a derivation of the same formula in the Frege–Hilbert calculus by replacing each sequent $A_1, \ldots, A_\mu \Longrightarrow B_1, \ldots, B_\nu$ with the formula $A_1 \wedge \cdots \wedge A_\mu \to B_1 \vee \cdots \vee B_\nu$, and by suitably replacing each deduction step of the sequent calculus by one or more deduction steps in the Frege–Hilbert calculus.

3.3 The sequent calculus and tableaux

If we look at the rules of the cut-free sequent calculus then we see that the structural rules do nothing except producing irrelevant formulas in a sequent, removing formulas occurring twice in a sequent, and interchanging formulas. The really interesting steps in a derivation in the cut-free sequent calculus are the introduction of axioms and the operational inferences. We shall see that these steps are exactly mirrored in a derivation in the tableau calculus —

$$
\begin{array}{llll}
Pa \Longrightarrow Pa & \text{by Axiom} & Pb \Longrightarrow Pb & \text{by Axiom} \\
\forall x Px \Longrightarrow Pa & \text{by } \forall\text{-IA} & \forall x Px \Longrightarrow Pb & \text{by } \forall\text{-IA}
\end{array}
$$

$$
\begin{array}{rl}
\forall x Px \Longrightarrow Pa \wedge Pb & \text{by } \wedge\text{-IS} \\
\Longrightarrow \neg\forall x Px, Pa \wedge Pb & \text{by } \neg\text{-IS} \\
\Longrightarrow \neg\forall x Px, \neg\forall x Px \vee (Pa \wedge Pb) & \text{by } \vee\text{-IS} \\
\Longrightarrow \neg\forall x Px \vee (Pa \wedge Pb), \neg\forall x Px & \text{by Interchange} \\
\Longrightarrow \neg\forall x Px \vee (Pa \wedge Pb), \neg\forall x Px \vee (Pa \wedge Pb) & \text{by } \vee\text{-IS} \\
\Longrightarrow \neg\forall x Px \vee (Pa \wedge Pb) & \text{by Contraction}
\end{array}
$$

Fig. 15. A derivation tree in the cut-free sequent calculus

$$
\begin{array}{ccc}
 & \Longrightarrow \neg\forall x Px \vee (Pa \wedge Pb) & \\
 & \Longrightarrow \ldots, \ldots & \\
 & \Longrightarrow \ldots, \neg\forall x Px & \\
 & \Longrightarrow \ldots, \ldots & \\
 & \Longrightarrow \ldots, Pa \wedge Pb & \\
 & \forall x Px \Longrightarrow \ldots & \\
\ldots \Longrightarrow Pa & & \ldots \Longrightarrow Pb \\
Pa \Longrightarrow \ldots & & Pb \Longrightarrow \ldots
\end{array}
$$

Fig. 16. The tree turned upside down and with redundancies removed

mirrored also in the literal sense that the rules are applied in the backward direction. As an example, we consider the formula $\neg\forall x Px \vee (Pa \wedge Pb)$ from Section 2.5. A derivation of this formula in the cut-free sequent calculus is given in a tree representation in Figure 15.

We shall now transform this derivation tree to a proof of the given formula in the tableau calculus. This transformation removes some of the redundancies contained in the derivation tree in the cut-free sequent calculus. We first turn the tree upside down. Then we go through it along each of its branches from the root to the leaves. At each node we remove all redundant formulas from the left or right hand side of the sequent. A formula is redundant if it has already appeared as a member of the respective side of a sequent earlier on the branch. Only formulas are kept that appear new on the respective side of a sequent. In our example, we thus obtain the tree shown in Figure **??** in which no more than one formula has been kept in each sequent. In the sequents that are premises of structural inference steps, all formulas have been removed as redundant. Formally, we obtain from this tree a tableau proof of the given formula $\neg\forall x Px \vee (Pa \wedge Pb)$ by removing all these redundant sequents and replacing each sequent $A \Longrightarrow$ with the formula A and each sequent $\Longrightarrow A$ with the formula $\neg A$. This tableau is shown in Figure 17.

There is an exact formal correspondence between the operational inference rules of the sequent calculus and the rules of the tableau calculus. The axioms of the sequent calculus are mirrored in the tableau calculus by the condition that all branches of the tableau have to be closed. The transformation shown here can be inverted to obtain a derivation of a formula F in the sequent calculus from each proof of F in the tableau calculus. The idea is to consider, for each node ν in the given tableau, the set S of all formulas on the path leading from the root of the tableau to that node ν. From this set of formulas we obtain a sequent by putting a formula G on the left hand side of the sequent when $G \in S$ and putting G on the right hand side of the sequent when $(\neg G) \in S$.

As we have already pointed out, the tableau calculus is an analytic calculus in the sense that it works by splitting a given formula into its subformulas (modulo some kind of substitution).

$$
\begin{array}{ccc}
 & \neg(\neg\forall x Px \vee (Pa \wedge Pb)) & \\
 & \neg\neg\forall x Px & \\
 & \neg(Pa \wedge Pb) & \\
 & \forall x Px & \\
\neg Pa & & \neg Pb \\
Pa & & Pb
\end{array}
$$

Fig. 17. A tableau proof

This property of the tableau calculus is closely related to the subformula property of the rules of the cut-free sequent calculus.

3.4 Tableaux and the connection method

There is a close relation between the method of tableaux and the connection method. Namely, every closed tableau can be regarded as a connection derivation and vice versa. Take, for example, the formula $\neg\forall x Px \vee (Pa \wedge Pb)$ from Section 2.5 for which Figure 2 depicts a proof in the tableau calculus. In the connection method, the formula is represented as the matrix

$$
\begin{array}{cc}
Pa & \\
 & \neg Px \\
Pb &
\end{array} .
$$

These two clauses correspond to the formulas marked "3." and "4." in the tableau of Figure 2. Remember that the tableau method is formulated as a method of refutation rather than proving validity. So positive literals become negative and vice versa when going from a connection proof to a tableau proof. Now, in the connection method a clause is chosen as a starting clause, say the clause $\{Pa, Pb\}$ in our example. This corresponds to applying the β-rule to the formula $\neg(Pa \wedge Pb)$ marked with "3." in Figure 2, thus obtaining the formulas $\neg Pa$ and $\neg Pb$ marked with "5." and "6.", respectively. In the connection method, now the connection $\{Pa, \neg Px\}$ is encountered proving all paths (there is only one) through Pa to be complementary. To this corresponds the closing of the left hand branch in Figure 2 by application of the γ-rule to $\forall x Px$. Next, in the connection method, the second literal Pb of the clause $\{Pa, Pb\}$ is treated the same way, where a new instance of the clause $\{\neg Px\}$ is needed. Correspondingly, in the tableau method, the second branch of the tableau is closed by a new application of the γ-rule.

This correspondence between proofs in the connection method and in the tableau method holds for proofs of arbitrary formulas of first-order predicate logic, not only for clausal form. To each path considered in the connection proof there corresponds exactly one branch in the tableau, and vice versa. To each connection considered in the connection proof there corresponds a complementary pair of literals in a closed branch of the tableau. The condition that every path has to contain a connection is mirrored in the condition that every branch has to be closed.

Note, however, that unification is built into the connection method whereas arbitrary substitutions are allowed in the method of tableaux. This fact, of course, amounts to a big advantage for the efficiency of the connection method. A slight change in the method of tableaux allows the incorporation of unification in it which renders it a useful calculus for automated theorem proving as well.

If one combines the transformations of the present section with those presented in the previous two sections then one obtains a transformation from connection proofs into natural deduction proofs and vice versa.[17] This relationship is important for interactive proof systems.

[17]Note, however, that a natural deduction proof has to be expanded to its tree representation before trans-

$$
\begin{array}{cccc}
 & & & \neg R \\
 & \neg P & Q & Q \\
P & \neg Q & R & S \\
\neg R & R & S & P \\
 & S & &
\end{array}
$$

Fig. 18. A matrix in propositional logic

In such a system the machine part of the proof search would be coded in terms of the more compact connection method while the proofs or partial proofs would be presented to the user in the style of natural deductions (as further discussed in Section 6.4) so that the user can easily interact and guide the search.

The relationship between the connection method and natural deduction has therefore been studied extensively, especially in the context of the connection (or matings) method deriving from natural deduction systems. Work focusing on this topic comprises [Bib80, Bib87a, Mil87, Pfe87].

3.5 Resolution and the connection method

Despite the quite different approaches to automated reasoning taken by resolution and by the connection method, a closer look at these two methods reveals close relationships.

3.5.1 Simulation of the extension calculus by resolution

In this subsection it is shown how any proof of a normal-form formula in the extension calculus (see Sections 2.7 and 4.2) can be simulated in resolution step by step in the sense that to every chosen connection in an extension step there corresponds exactly one resolution step.

The idea is to associate a set of literals, called the set of unsolved subgoals, with each stage of the proof process in the extension calculus. These sets of literals then form the clauses of a resolution refutation of the given set of clauses.

To demonstrate this, let us consider the matrix in Figure 18 as a simple example from propositional logic. A proof attempt for this matrix in the extension calculus proceeds as follows. In the first step, an arbitrary clause of the matrix, say the first clause, is chosen as a starting clause. Both literals, namely P and $\neg R$, in this clause are unsolved subgoals at this stage.

In the second step, an arbitrary literal of this clause, say the literal P, is chosen. The proof process now concentrates on showing that all paths through the matrix that pass through this literal are complementary while the treatment of the paths passing through its second literal $\neg R$ is deferred to the future. We say that $\{P\}$ is the *active path* and that $\neg R$ is an *unsolved subgoal*. Now, from the last literal P of the active path a connection is drawn to a complementary literal of a new clause of the matrix, say to the first literal $\neg P$ of the second clause. All other literals of this second clause are regarded as unsolved subgoals. So, at this

forming it to a connection proof if a connection calculus is used that does not admit the use of lemmata.

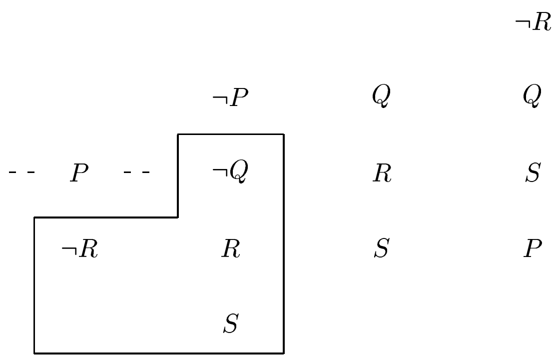


Fig. 19. Active path and unsolved subgoals after the second proof step

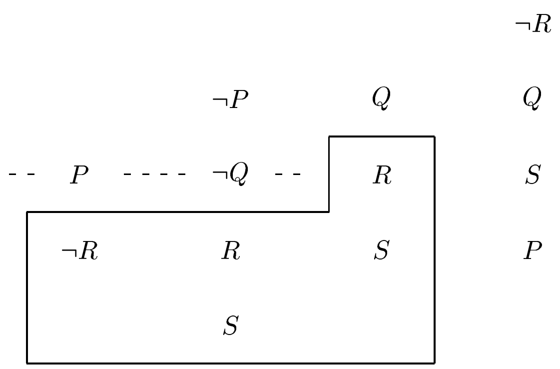


Fig. 20. Active path and unsolved subgoals after the third proof step

stage the unsolved subgoals are the literal $\neg R$ in the first clause and the literals $\neg Q$, R, and S in the second clause. This completes the second proof step. In Figure 19, the active path is indicated by a dashed line, and the set of unsolved subgoals is indicated by a box enclosing them. Note that the set of unsolved subgoals is a resolvent of the first two clauses $\{P, \neg R\}$ and $\{\neg P, \neg Q, R, S\}$ of the given matrix.

Similarly, in the third proof step a yet unsolved subgoal, say $\neg Q$, is chosen in the second clause, and from it a connection is drawn to a complementary literal in a new clause of the matrix, say to the literal Q in the third clause. After this step has been performed, the active path consists of the literal P in the first clause and the literal $\neg Q$ in the second, and the unsolved subgoals are $\neg R$ in the first clause, R, and S in the second clause, and R and S in the third clause, as shown in Figure 20. Note that the set of unsolved subgoals at this stage is the resolvent of the previous set of unsolved subgoals $\{\neg R, \neg Q, R, S\}$ and of the third clause $\{Q, R, S\}$ of the given matrix.

In the fourth proof step two connections can be drawn, namely from the literal R in the third clause to the literal $\neg R$ in the fourth clause, and from the literal $\neg Q$ in the second clause to the literal Q in the fourth clause. The resulting active path and set of unsolved subgoals are shown in Figure 21.

The set of unsolved subgoals at this stage of the proof process can be obtained from the sets

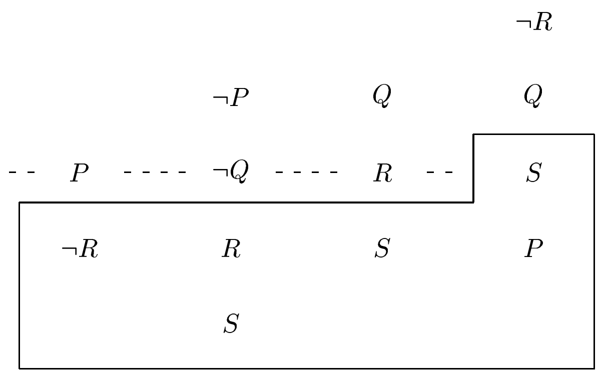


Fig. 21. Active path and unsolved subgoals after the fourth proof step

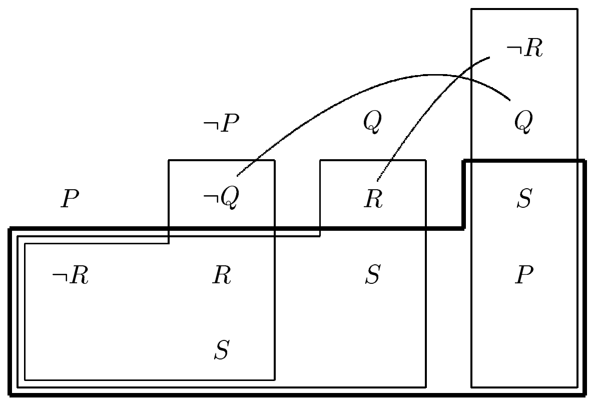


Fig. 22. The fourth proof step

if unsolved subgoals of the two previous stages and from the fourth clause of the matrix by two resolution steps corresponding to the two connections involved. Namely, first the fourth clause of the matrix has to be resolved with the set of unsolved subgoals of one of the two previous stages, and then the resulting resolvent with the set of unsolved subgoals of the other stage. The four involved sets of literals and the two involved connections are shown in Figure 22 where the box drawn with thick lines indicates the new set of unsolved subgoals.

This way, at each stage of the proof process, the set of unsolved subgoals can be obtained from the clauses of the given matrix and from the sets of unsolved subgoals of previous stages by resolution. Since a successful proof in the extension calculus always ends with no unsolved subgoals left, a resolution refutation of the set of clauses of the given formula is thus obtained. In the case of first-order logic a resolution refutation is obtained in quite a similar way. The essential difference is that the clause corresponding to a proof stage is not just the set of unsolved subgoals but obtained from this set by application of the most general unifier of all the connections considered up to that stage.

3.5.2 Non-simulatability of resolution in the extension calculus

In this section we give examples of formulas which show that we cannot in general p-simulate resolution refutations in the version of the connection calculus given here. This is to say that there is no such simulation at polynomial expense in time. In fact, the number of derivation steps increases exponentially in the worst case when going from the resolution method to the connection method.

As a first example we take the formula

$$Pa \wedge \forall x(Px \to Pfx) \to Pffffffffa$$

where P is a predicate symbol and f a function symbol both of arity 1, and where we have omitted parentheses wherever this does not lead to ambiguities. We shall adopt the notation $f^n t$ for $\underbrace{f\ldots f}_{n \text{ times}} t$ for any nonnegative integer n and any term t. Then the formula has the matrix representation

$$\begin{matrix} & Px & \\ \neg Pa & & Pf^8a \\ & \neg Pfx & \end{matrix}$$

There exists a short resolution proof of this formula: Resolving the second clause with itself we obtain the resolvent $\{Px, \neg Pf^2x\}$. Resolving this clause with itself we obtain $\{Px, \neg Pf^4x\}$, and resolving this clause with itself we obtain $\{Px, \neg Pf^8x\}$. By resolution with the first and the third clause we get the empty clause. As we shall see, however, the shortest proof in the connection calculus takes 9 extension steps. In general, for the matrix

$$\begin{matrix} & Px & \\ \neg Pa & & Pf^na \\ & \neg Pfx & \end{matrix}$$

there is a resolution refutation in $2k$ steps where k is the smallest integer such that $2^k \geq n$, whereas the shortest connection derivation has $n + 1$ extension steps.

In order to prove that there is no shorter connection derivation we define the *degree* of a valid matrix M to be the cardinality of the smallest (in terms of cardinality) set of variants of clauses of M which admits a unifiable spanning set of connections. Equivalently, the degree of M is the cardinality of the smallest set N of ground instances of clauses of M such that N is valid. Of course, N can be considered as a formula of propositional logic since it has no variables. The proof of the following proposition is obvious.

Proposition 3.5.1. *Let M be a matrix of degree d and let Δ be a connection derivation of M. Then the number of extension steps that Δ contains is greater than or equal to $d - 1$.*

It is easily seen that the degree of the matrix

$$\{\{\neg Pa\}, \{Px, \neg Pfx\}, \{Pf^na\}\}$$

is $n + 2$. Thus any connection derivation of it must have at least $n + 1$ extension steps.

Theorem 3.5.2. *The connection calculus cannot p-simulate resolution.*

One reason why resolution proofs cannot be p-simulated by connection derivations in the calculus given here is that resolution can make repeated use of lemmata generated in the course of the deduction. In our first example these are the clauses $\{Px, \neg Pf^jx\}$ for $j = 2,4,8,\ldots$. Another, more practical, example for the use of lemmata in resolution, namely the simplest Prolog program for computing the n-th Fibonacci number for some given nonnegative integer n, can be found in [Ede87]. It is important to note here that the use of lemmata is closely connected to the dag structure of resolution derivations. If we would expand the derivation dag into a tree then this source of efficiency of resolution would be lost. In the calculus of the connection method presented here there exist only very limited possibilities for use of lemmata. The concept of splitting introduced by W. Bibel [Bib87b] expands these possibilities greatly when used in a nested way. It is, however, not at all obvious how to define a concept of nested splitting that would be sound and that would incorporate the possibilities of use of lemmata offered by resolution. The concept of connection structures can be regarded as such a concept, and as a generalization and extension of the concept of splitting. A first step towards the use of lemmata in actual implementations of the connection method has been done in the theorem prover SETHEO [BEK$^+$87]. This implementation is restricted, however, to lemmata consisting of just one literal.

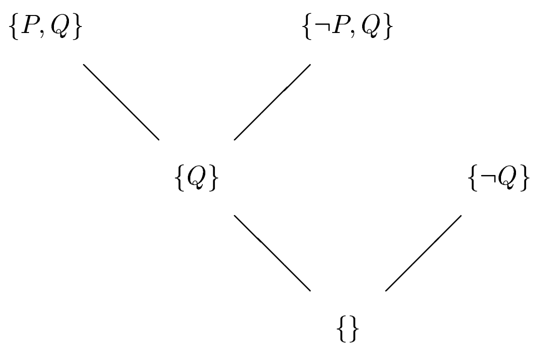
Another source of inefficiency of the connection calculus becomes apparent with the following example. Suppose we are given a matrix

$$\begin{array}{ccc} & \neg Py & \\ Px & & \ldots \\ & Q & \end{array}$$

where the predicate symbol P does not occur in the rest of the matrix. In resolution we can resolve the first two clauses to obtain the clause $\{Q\}$ which does not contain x and y any more. In fact, a resolution prover will then apply subsumption and pure clause elimination and just throw the first two clauses away. So resolution has the ability to just forget the variables x and y. On the other hand, a theorem prover using the connection calculus introduced here will generate a current substitution $\{y \leftarrow x\}$ when encountering the connection $\{Px, \neg Py\}$, and henceforth x and y will be shared variables, and the binding will be carried along all the time in the current substitution. Our previous two examples also show the necessity for forgetting of variables, here in connection with the use of lemmata. In these examples, the use of lemmata is only possible efficiently if the variables that do not occur in the formulation of a lemma, are forgotten. Otherwise, we have again an exponential explosion of the size of the derivation.

3.6 Resolution and consolution

There is a very simple way to simulate resolution by consolution step-by-step. To see this, consider again the matrix in Section 2.8. A resolution refutation of this matrix is given by the following tree.



In this tree we only have to replace each literal L with the one-element path $\{L\}$ in order to obtain the consolution proof of Figure 6 in Section 2.8. This amounts to replacing every clause c marking a node in the resolution refutation tree with the path set $\mathcal{P}_c$ as defined in Section 2.8. Every resolution refutation can thus be transformed to a consolution proof, but not vice versa.

3.7 Resolution and the connection structure calculus

In this section we show how resolution refutations can be simulated in the connection structure calculus. We take as an example the matrix

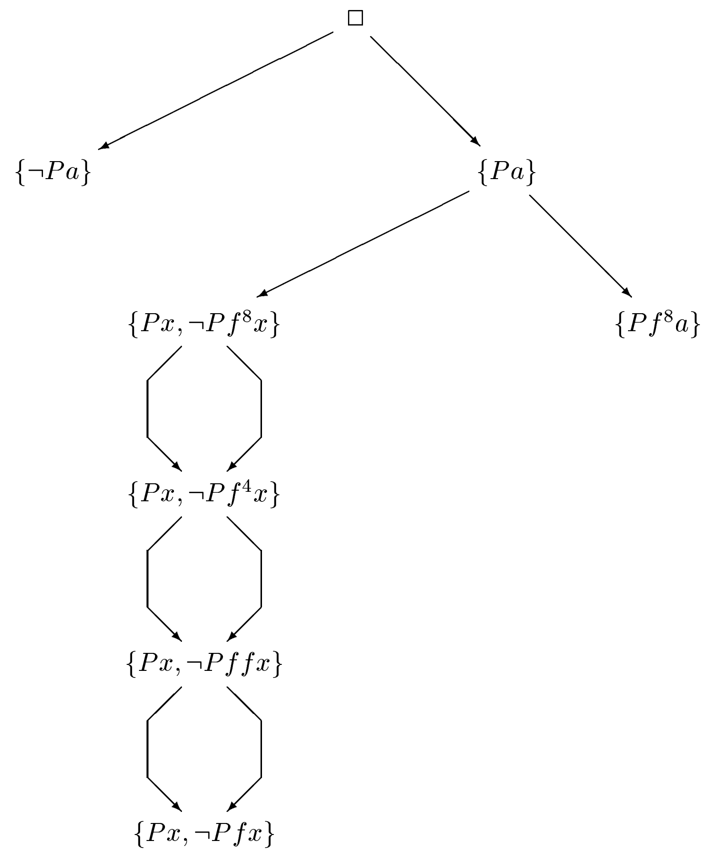


Fig. 23. The dag representation of a resolution refutation

$$\begin{array}{ccc} & Px & \\ \neg Pa & & Pf^8a \\ & \neg Pfx & \end{array} \; .$$

A resolution refutation of this matrix is shown in a dag representation in Figure 23 where we have written the input clauses at the bottom and the empty clause at the top. So the direction of the derivation in the dag is from the bottom to the top. Remember the connection structure we gave for this matrix in Figure 13 in Section 2.10. By expanding its root into two nodes we obtain the connection structure shown in Figure 24.

From the dag representation of a resolution refutation shown in Figure 23, the connection structure depicted in Figure 24 can be obtained as follows. Each edge of the dag is first marked appropriately with a selector ℓ or r. Each clause marking an inner node ν of the dag is a resolvent of the two clauses marking its successors. In these parent clauses we consider the two literals resolved upon. Each of these literals must be inherited from one literal in one of the input clauses. Let K and L be these two literals of the input clauses. The inheritence of each of these literals goes along a path from a leaf in the dag to the node ν. Let s and s' be the respective selector sequences corresponding to these two paths, going the opposite direction (from ν to the respective leaves). Then the node ν in the dag is marked with the pair $(s.K, s'.L)$. If, as is not the case in our example, there are several literals merged into one by a factorization in the resolution refutation, then each node in the dag will be marked not

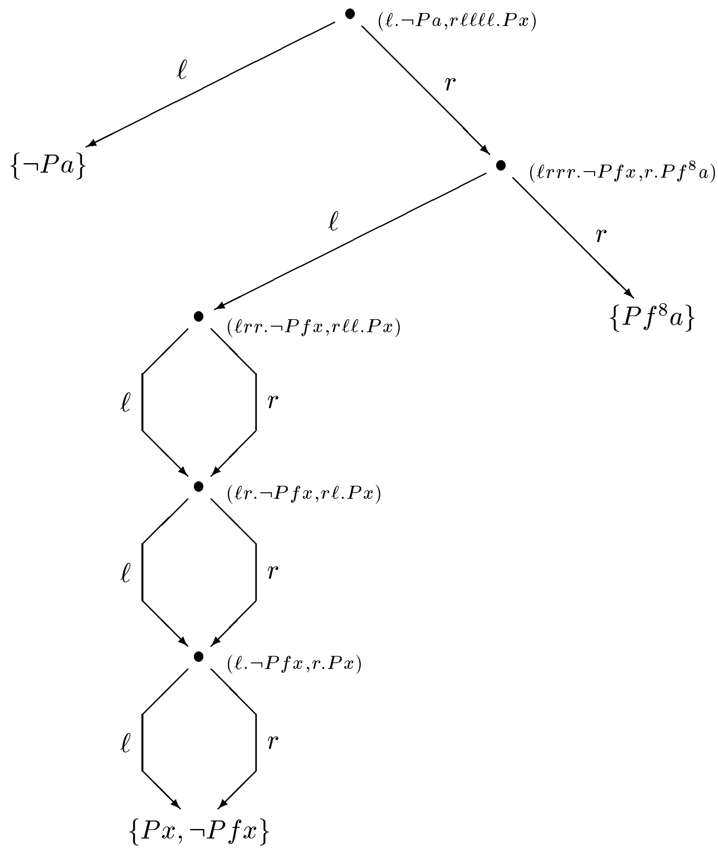


Fig. 24. A connection structure

only with a connection, but also with a number of factorization links.

To obtain a proof in the connection structure calculus, we still have to compute the most general unifiers and the path sets. The most general unifiers are obtained from the resolution steps. All paths are cut to length 1 in each step, and the path sets correspond exactly to the clauses occurring in the resolution refutation. For example, the node marked with the resolvent $\{Px, \neg Pffx\}$ in the resolution refutation will have the path set $\{\{Px\}, \{\neg Pffx\}\}$ associated to it in the connection structure proof.

So, by restricting the connection structure calculus to binary dags and to paths of length one we essentially obtain resolution. Similarly, by restricting it to trees and imposing an "input strategy", we obtain the connection calculus.

Note that a connection structure proof contains path sets as essential ingredients and that these path sets constitute a consolution proof. So, every connection structure proof can in a trivial way also be regarded as a consolution proof.

3.8 Conclusions and open questions

There has been an ongoing dispute on the merits of different calculi. From the results reported in the present section we can extract the following lessons. At this point in the presentation such lessons are restricted to properties of the calculi and therefore ignore any strategic considerations not treated until the subsequent section. Nevertheless such properties of calculi are extremely relevant for the comparison of different proof methods.

The first lesson is that all these calculi are very closely related. It is possible to transfer proofs of one calculus to another. It is also possible to import a certain feature absent in one calculus into it from another. It could even be that, by such import, in a sense any calculus may be extended to a calculus with the power of another one. From this perspective, the dispute just mentioned seems irrelevant.

On the other hand some calculi lend themselves more easily to the understanding by humans of the general features of reasoning and thus to improvements of the deduction technology. Apparently the general belief is that resolution serves well for this purpose, since most researchers in automated deduction are using resolution for studying these features.[18] The present authors do not share this belief. Resolution merges a number of important distinct features into the single concept of the resolution rule. For instance, resolution blurs the distinction between new instances of clauses that are crucially needed and those for which for example the connection method or the connection structure calculus would not need to generate a new instance. Also the use of lemmata is much more explicit in the latter two proof methods than in resolution.

All this justifies the use of the terminology of the connection method in the explanation of the strategic concepts introduced in most of the remainder of this chapter.

[18] Note that we are not talking here of natural representations of particular proofs as discussed in Section 6.4.

4 Strategies

In our survey of calculi and their comparison in Sections 2 and 3 we have largely ignored computational aspects of how to *find* proofs in any of these calculi. Since automation is a major focus in our presentation as we pointed out in Section 1, we have thus ignored an essential aspect of our topic to which we will turn our attention now.

For automation a proof system has to be empowered by some control, or strategy, which directs the application of the rules of the calculus. For each of the calculi discussed so far a different sort of strategy is required. With our insight from Section 3 we know however that strategies useful in one calculus may well be transported into others. This is particularly true for strategies in the connection structure calculus which has been shown to relate to any of the others. For the purpose of presentation, however, it is easier to describe strategies in terms of its close relative, the connection method, and from there point to their relevance for the other calculi. This is the purpose of the present section.

Strategic considerations to some extent depend on the view that one has of an ideal proof system. Our view in this regard is discussed in the next subsection. Further, experience with existing proof systems and theoretical insight have shown that strategic considerations are based on a few basic issues which will therefore be explained before the discussion of familiar strategies in the remainder of the section.

4.1 Adequate proof methods

Assume, we are given two theorems, T_1, T_2, and we know already their easiest proofs, $\mathcal{P}_1, \mathcal{P}_2$ (in accordance with some measure of simplicity). Further assume that $\mathcal{P}_1$ is an easier proof than $\mathcal{P}_2$ (possibly carried out in different calculi). In this sense, T_1 may be regarded as an easier theorem than T_2. Intuitively, one would expect a proof method to find a proof for T_1 faster than a proof for T_2. Let us speak of an *adequate* proof method in case it behaves in this way for any such pair of theorems. A proof method might be adequate for a restricted class of theorems only, in which case we would call it *adequate for* that class. Adequate proof systems seem a desirable goal to strive for.

Measuring the difficulty of theorems is a complicated issue. Recall that there are many calculi. A given theorem may have a short proof in one calculus while its easiest proof in another calculus may be quite long. An example for such a situation is the class of pigeonhole formulas. In [Hak84] Haken has shown that the shortest proofs by resolution of the theorems in this class are exponential. In other calculi, such as in some connection calculus with renaming, the same theorems have linear proofs [Bib90].

We see that first of all any measure of difficulty that is tied to a particular calculus might be of limited value. Also, simple measures, such as the length of proofs, might not be satisfactory since, while the proofs might be short, their discovery might still be hard. To understand why recall from Section 2.2 that some proof systems have even infinite branching factors.

Intuitively, the best measure for the difficulty of a theorem seems to be the size of the minimal search space remaining, after all available information is taken into account for its limitation. With such a measure, if made precise, the pigeonhole theorems would be classified as rather easy, since their structure strongly suggests the techniques used in their shortest proofs mentioned above with which the remaining search space becomes rather small. Unfortunately, a precise definition of such a measure would have to consider quite a few things such as different calculi, lengths of proofs, branching factors, and so forth.

4.2 Linear chains

There are a number of issues of strategy that have proven to be essential in automated deduction. The first of these is the idea of chaining the deductive steps in a *linear* way which we will first explain in terms of the connection method.

Recall from Section 2.7 that formulas may be transformed into sets of clauses. Since the idea of linear chaining is basically propositional in nature we may ignore any effects caused by the terms in the literals of the clauses. This allows us to restrict the discussion to propositional literals for simplicity, although all the following applies to the general case.

Consider the clauses $\{P\}, \{\neg P, Q\}, \{\neg Q, R\}, \{\neg R\}$ as an example. In Section 2.7 we have introduced a two-dimensional form of representation as a matrix as shown in the following picture.

$$\begin{bmatrix} \\ P \end{bmatrix} \begin{bmatrix} \neg P \\ Q \end{bmatrix} \begin{bmatrix} \neg Q \\ R \end{bmatrix} \begin{bmatrix} \neg R \\ \end{bmatrix}$$

Often we dispense with the brackets clarifying the clause structure, in particular in simple cases like the present one where no confusion may occur. So the same matrix without brackets looks as follows.

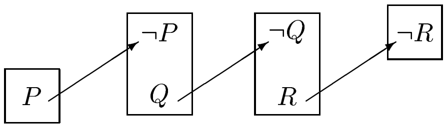
$$\begin{array}{cccc} & \neg P & \neg Q & \neg R \\ P & Q & R & \end{array}$$

Again from Section 2.7 we know that a spanning mating is required to establish the proof in this propositional case, ie. a set of connections such that each path contains one of its members. The next picture shows the example with all its connections.

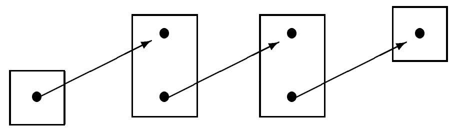
$$\begin{array}{cccc} & \neg P & \neg Q & \neg R \\ P & Q & R & \end{array}$$

The mating consisting of the three depicted connections obviously spans the matrix since each of its four different paths contains one of them. The question is how to check this spanning property in an algorithmic way. One solution is to check each path, one after the other, whether it contains a connection. Since there are exponentially many paths in a matrix, this is bound to be a bad solution. A better solution is to focus on the connections rather than the paths and (i) select some connection from the mating and delete it, (ii) determine the subset of all paths containing the selected connection, (iii) remove this subset from the set of paths, and, recursively, (iv) test whether the remaining connections span the remaining paths.

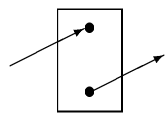
In order to illustrate this basic procedure let us distinguish the three connections of the present example as P-connection, Q-connection, and R-connection. Say, we first select the P-connection. It spans two paths. In order to test the remaining two paths, we have a choice of selecting the Q-connection or the R-connection. The idea of linear chaining amounts to selecting *first* the Q-connection and only thereafter the R-connection. Note that this also imposes a direction in which the matrix is traversed as illustrated in the next picture.



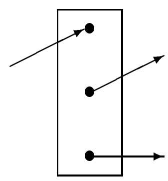
The picture obviously reminds of a chain whence the notion of a *linear chain*. Let us abstract from the particular literals. Then our picture becomes the following one.



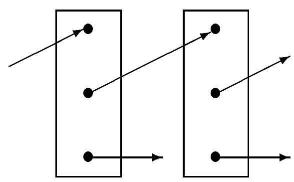
This shows that a chain-link has the form



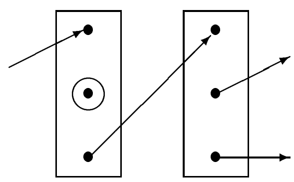
In the general case there may be more than two literals in a clause, so that in the case of three literals a chain-link has the form



Assume for this general case that a connection, assumed to belong to a spanning mating, has been selected which enters this clause by its head literal. Then one can easily see that each of the remaining two literals in the clause must be a member of some connection from this spanning mating as well [And76, Bib87a]. Since all connections of the spanning mating have to be considered anyway to establish the proof, one of two outgoing connections (like the Q-connection in the previous example) may be considered next, without loss of generality or efficiency, rather than some other connection (such as the R-connection above) from the remaining matrix. An illustration of the result of the operation is



Proceeding this way will result in a linear chain as above except that it has loose ends. These are stacked connections to be considered after successful completion of the chain initiated in the operation, so that eventually the operation



will be performed where the circled dot illustrates the previously *solved subgoal*. Looking at the general case from a global viewpoint shows that we are actually dealing with a *tree* structure. However, since at any one time only one branch (*path* in [And76]) within this tree is pursued, the operation itself is linear in nature which justifies our terminology.

As a further important remark, for any subgoal in the clause there may be more than one outgoing connection. The strategy has to select one of them, stack the remaining ones, and, in case of failure to find a proof, backtrack later to the point and restart with an alternative connection.

In talking of *head* literals (the negated literals in the top row of the example), of *subgoals* (the remaining ones in each column, also referred to as the *body* of the clause), of *queries*, and of *programs* (rather than examples or theorems), we follow the popular terminology as used in the context of PROLOG (see Volume 4).

In our illustrations we have shown the case of ground level formulas only. The main feature of linear chaining remains the same on the first-order level as already mentioned, with two modifications to be taken into account. First, a substitution has to be determined (by unification at each step) such that all connections of a spanning mating become complementary. Since this has already been illustrated in Section 2.7 and is done in the same way as in resolution, discussed at great length in the subsequent chapter, we will not go into further details here. Second, we have to allow a chain to involve a given clause in more than just one instance as illustrated already in Section 2.7. In such a case, the dynamically linear chains statically adopt a cyclic flavour to be discussed in Section 4.5.

Linear chaining is present in most deductive methods in use. We find it in linear resolution (including SLD– or LUSH–resolution used in PROLOG; see Chapter 8 in Volume 4), in calculi based on the connection method (see 2.7), including model elimination [Lov78] where it is called *extension*, in analytic tableaux (see Section 2.5), and so forth. A connection calculus using this rule as its essential proof mechanism is called an *extension* calculus. We may say that extension has become a standard, even to an extent that often this standard is used in compiled form (as for instance in many PROLOG systems; see Section 5.3). While being a standard, linear chaining is not a necessary, but a computationally convenient way of controlling the proof search, as we have explained above. In fact, Maslov's method treated in Section 2.9 uses a different strategy.

4.3 Hinged loops

Linear chaining is very attractive from a computational point of view but not general enough to cover all cases. In order to attain generality we are now going to discuss one further feature.

It is easy to see that this feature must introduce some cyclic structure into proofs as illustrated in the following example, obtained from the one in the previous subsection by some added elements. This example demonstrates that a cyclic proof structure appears in such a more general case.

$$\begin{bmatrix} & \neg P & \neg P & \neg Q & \neg R \\ P & N & \neg Q & R & \neg N \end{bmatrix}$$

Here we still have got the chaining structure from the previous proof. In addition, however, there are two further connections that close a loop, called a *hinged loop*. Such a loop contains a unique literal, here the goal P, that (statically) occurs in two different connections, which, for obvious reasons, we call the *hinge*. Starting with the hinge in linear chaining mode may eventually lead us back to it. Because of this latter property we see that there must at least be one clause involved where the chain does not switch signs. In our example this is the case with the rightmost clause where both literals are negative ones (thus making the clause non–Horn; see Section 5.3). This explains why in logic programming (which is restricted to Horn clauses) this feature is absent.

Hinged loops of this sort have been studied extensively in a number of papers, including [Lov69, And76, Sho76, Sic76, Bib87a], under different names (such as *merge loops* in [Sic76]) and with slight variations (such as in [And76]). It turns out that any proof (ie. spanning set of connections) consists of linear chains and possibly hinged loops, so that the general case has been attained with this additional feature.

There seem to be basically five ways to handle hinged loops. The first one extends the linear chaining technique by taking into account for each subgoal the possibility of closing a loop. In resolution this is called *ancestor resolution* [CL73], in the connection calculus presented in [Bib87a] it is integrated within the extension step since all "active path" literals may be used for extension there, in model elimination it is called *reduction*. In our example this would mean, we select one of the P–connections, say the one into the N–clause, proceed to chain along $N, \neg R, \neg Q$ to arrive at $\neg P$. At this point the *active path* consists of the set of all subgoals in the present line of reasoning, ie. of the set $\{P, N, \neg R, \neg Q\}$. Checking the active path for the complement of each subgoal, the process thus extended notices at this point that P is indeed a member of the active path. Whenever this situation occurs the loop is closed (as it duly happens in our example).

The first solution just described does not make a difference as to which way around the loop is traversed. The second solution is a refinement thereof committing itself to a certain direction, which saves some testing. Here the process initiated for the example as before with the first P–connection would be suspended rather than pursue the N–connection as before. This is because an (arbitrary) preference is given to the literal $\neg R$ in the rightmost clause to play the role of a head literal (as in PROLOG), and positive subgoals are chosen to extend with head literals only. Previous alternatives for extensions, such as the other P–connection, are taken into account instead. So the process here chains along P, Q, R, this way encountering the negative body literal, $\neg N$. When such a situation occurs connections with suspended positive subgoals are sought (or new alternative linear chains are triggered until they produce such subgoals). The resulting connections are called *body connections* for obvious reasons.

This technique, while somewhat more complicated than the first, has the advantage of retaining as much of the linear chaining technique as possible. It also avoids to keep a list of *all* active path literals and to test for *each* subgoal the possibility of a body connection. Suspended processes may accumulate instead though. However, since in practice connections are usually precomputed anyway, those positive literals contained in body connections (like N), and thus the points of possible suspension, are usually known to the process. Further, in application there are typically few of them so that the savings may be substantial.

The technique is of course complete. Note the following two points in this respect. First, the process might have selected the second P–connection when it started. So no suspended process would be available when $\neg N$ is encountered in which case this process would have to be suspended (until the alternative chain produces the necessary subgoal as already mentioned above). In each case of suspension the body connections involved would become activated and provided with the necessary information (such as bindings, etc.). Second, suspension points such as N in general might allow alternative continuations with linear chains, which have to be treated as proper alternatives to suspension.

This is a different (and, to these authors' belief, conceptually simpler) way of stating the gist in Plaisted's modified problem reduction format [Pla88]. This is not the place to introduce this format in all details. We only provide the interested reader with a few pointers. In this method each clause, without restriction of generality, is assumed to be of the form $L \leftarrow L_1 \wedge \ldots \wedge L_n$ (recall the arbitrary choice of $\neg R$ as head literal above). It is encoded as a *clause rule*

$$\frac{\Gamma_0 \to L_1 \implies \Gamma_1 \to L_1, \ \ldots, \ \Gamma_{n-1} \to L_n \implies \Gamma_n \to L_n}{\Gamma_0 \to L \implies \Gamma_n \to L}$$

This kind of rule is a variant of Gentzen's sequent rules (see Section 2.4). The Γ's are sequences of literals, the L_i's are literals, and L is an atom. The idea here is to represent a program by a set of such rules along with axioms and further rules to be given shortly and, for a given query, search for a proof in a backward chaining way (see Section 4.5). Closer inspection reveals that establishing one connection in the chaining mode described above amounts to applying such a sequent rule in a backward way.

Apart from such clause rules Plaisted also has the following *case analysis (splitting) rules*.

$$\frac{[\Gamma_0 \to L \implies \Gamma_1, \neg M \to L],\ [\Gamma_1, M \to L \implies \Gamma_1, M \to L],\ |\Gamma_0| \le |\Gamma_1|}{\Gamma_0 \to L \implies \Gamma_1 \to L}$$

Here M denotes an atom, $|\Gamma|$ denotes the number of literals in Γ, all other symbols have their previous meaning. These rules serve exactly the purpose as establishing body connections in the procedure as described above. Put the other way around, body connections serve the same purpose as case analysis in a natural form of reasoning. Finally, the system takes into account *assumption axioms*

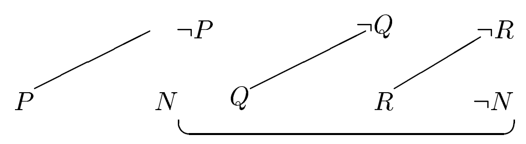
$$\begin{array}{c}\Gamma \to L \implies \Gamma \to L \ \text{ if } \ L \in \Gamma \\ \Gamma \to \neg L \implies \Gamma, \neg L \to \neg L\end{array}$$

which in our view serve the purpose of signaling loop closure.

As we said, the purpose of mentioning these rules is to indicate for interested readers the connections with the technique of handling hinged loops just outlined. It is not meant to introduce Plaisted's formalism in sufficient detail for understanding it thoroughly. But even with these few indications of its form and nature, the reader will be able to understand the lesson to be learned from this comparison. Basic ideas in automated deduction often appear in disguises making them hard to be understood in comparison with known techniques. For the benefit of progress of the field, it would be desirable that authors explained their ideas with reference to some canonical representation agreed upon by the community.

This completes our discussion of the second solution to handling hinged loops. The remaining three solutions all apply ways of dissolving the loops. The first has been suggested by Stickel [Sti88] (and used also in [LSBB92]). It amounts to introducing for each clause the set of all its contrapositives (ie. the same clause but a different literal selected as its head). This introduces enough redundancy to avoid the need for considering loops (as the reader may see with our example). The second among these three solutions has been suggested in [BCEF84] which is to apply Prawitz' matrix reduction (see Section 2.7) to clauses (and their connections) that are non-Horn. Recently, [Lov87] has presented a similar solution in different disguise.

The last of these solutions to the handling of loops so far has never been taken into consideration in actual implementations. By factoring the *mates* of the hinge, resulting in a non-normal form matrix, the loop (in the sense defined above with a single literal contained in two of its connections) in effect disappears as well. Applying this transformation to our example results in the following matrix.



With the method described in Section 3.1 such a matrix may be brought back into normal form (without losing the property just acquired). In either case the hinge has disappeared. What remains may be treated by linear chaining or something very close to it.

As we see, no standard has emerged so far for handling hinged loops as in the case of linear chains. It might evolve from one of the alternatives just described (we believe the second solution to be the best candidate for it). Needless to say, all our considerations have to be seen in the light of their generalizations to first-order logic, as pointed out before in the previous subsection.

The necessity for backtracking described in the previous section remains in the presence of hinged loops as far as first-order logic is concerned. But it does so for a different reason. Namely, on the propositional level backtracking is no more required if all active path literals may be used for extension as in the first solution above. On the first-order level, however, an extension also implies a binding of variables which may be the wrong binding from the viewpoint of the final proof. Backtracking is thus needed to undo the binding and choose the one implied by the alternative connection.

4.4 Simplifications

In the previous two subsections we have presented the two basic ideas that form the core of virtually any deductive system. In principle this is all needed for enabling a system to prove any first-order theorem. In practice, however, a more refined control strategy is needed in order to minimize the damaging effect caused by the exponential complexity from which all known theorem proving techniques suffer. An important part of any such strategy is the application of a number of simplification (or reduction) rules.

A number of these rules, viz. *purity*, *factoring* or *merging*, *tautology*, and *subsumption*, will be discussed in Chapter 4 in the context of resolution, although they are relevant for any proof system (also those based on the connection method [Bib81]) in the same way. Under certain circumstances, *unit resolution*, which requires one parent clause to be a unit clause, also belongs to the set of simplification rules. Their usefulness and importance lie in the fact that they *simplify* the stated problem so that the resulting problem requires strictly less deductive effort.

In terms of matrices one may uniformly characterize such *simplifications* as operations on the set of formulas that

- can be performed fast (typically in linear time);
- can be applied only a number of times that is polynomial (typically linear) in the size of the formula;
- reduce the number or length (or both) of paths in the matrix to be tested for complementarity (without increasing the matrix); and thereby
- retain the truth value, or at least validity, of the formula.

For any reasonable deductive method, like resolution or the connection method, the result of one or more simplifications of this kind reduces the effort required for the main proof system (even if the proof were not shortened through the simplification, which may well happen, it is still easier to handle a smaller object).

On the first-order level simplifications have to be handled with some care inasmuch as they require non-trivial substitutions in order to become applicable. One way to cope with this problem is to restrict the substitutions used in simplifications to renamings, and consider simplifications at various points during the deduction process.

The potential of simplifications is much richer than suggested by the ones mentioned above. The linear chains discussed in Section 4.2 offer a number of possibilities for the simplification of formulas. Let us start with the simplest observation of this kind. In the following example we assume that there are no further connections of the entire matrix containing either $P(f(x))$ or $\neg P(y)$.

$$\begin{array}{cccc} & \neg L & \neg Py & \\ \cdots & Pfx & D & \cdots \\ & C & & \end{array}$$

We may call the depicted connection an *isolated* one. In this case the two shown clauses may be *replaced* by the resolvent (while in general the resolvent may only be *added*), thus strictly simplifying the problem in the sense defined above (fewer and shorter paths). In the same vein consider the following situation.

$$\begin{array}{ccccc} & \neg Qx & \neg Pa_1 & \cdots & \neg Pa_n \\ \cdots & & & & \\ & Px & \cdots & & \end{array}$$

Px is assumed to be contained in no connections other than those depicted. In this case, the problem may be simplified to the following one.

$$\cdots \qquad \neg Qx^{\{a_1 \ldots a_n\}}$$

The notation $x^{\{a_1 \ldots a_n\}}$ indicates that the value of x is restricted to the range given by the associated set. Similarly,

$$\begin{array}{ccccccccc} \neg Rx & \neg Qa_1 & \cdots & \neg Qa_m & \cdots & \neg Qa_n & \neg Pa_1 & \cdots & \neg Pa_n \\ Qx & \cdots & & \cdots & & & & & \\ Px & & & & \cdots & & & & \end{array}$$

reduces to

$$\cdots \qquad \neg Rx^{\{a_1 \ldots a_m\}}$$

In [Bib88a, BLS87] this idea has been presented as *DB–reduction*.

It is straightforward to formalize the two simplifications thus illustrated in precise terms. Moreover, they are instances of a more general principle of the following kind. It suggests that one "shift" the potential values of variables along linear chains of connections (especially isolated ones, but others as well) closer to the "core" of the deductive problem. Once this information is available there, many alternatives that might otherwise seem viable could well turn out to be excluded by this information concerning the range of variables. Let us refer to this idea as the *compression principle*. We will meet it once again in the context of sorts and theory connections in the following subsection as well as in later ones.

4.5 Control and compression

Any formula (or matrix) has a polynomial (at most quadratic) number of connections that potentially contribute to its proof. Usually, a matrix has more connections than those required for the proof. So searching for a proof means searching for the right subset of connections.

Since there are exponentially many subsets in any set, we see why *controlling* the search is of crucial importance for the feasibility of theorem proving.

A different way of looking at this possibly exponential explosion is that any connection may have to be considered exponentially many times in the course of demonstrating the spanning property of a mating even in the propositional case. At present it is not known whether or not this kind of explosion can be avoided in principle. The question is equivalent with whether there exists a *super-proof system* [CR74a] or not. Most proof systems discussed in Section 2 (including resolution, the connection method, and the connection structure calculus) suffer from this problem. Only *extended* resolution (Section 2.11) and connection (structure) calculi with *renaming* techniques (Section 2.11) are provably superior to all others in this respect and thus remain candidates for avoiding this problem in principle. In both cases proof information is *compressed* and reused in the resulting compact form rather than elaborated again in detail.

A second source of an additional exponential explosion enters on the first-order level only. It is caused by the need (explained in Section 2.7) to consider an initially unknown number of instances of (parts of) the matrix in order to establish its proof. In this case the previously described explosion applies to each of the iteratively expanded (or indexed) matrices, ie. for the n–fold disjunction of variants of it (for some n). Unless this expansion is controlled with the utmost caution, any prover is quickly lost in an infeasible search space.

As we see, both control and compression are key issues in ATP. Especially control (in various disguises) has been the focus of many studies while the importance of compression is only recently beginning to be more widely recognized. In the present section we review and evaluate some of this work. While again we look at this work from the viewpoint of the connection method, everything (probably with no exception) is of the same relevance for any proof method, given the close relationships established in Section 3.

4.5.1 Basic control strategies

Among the simple control regimes, *top-down* (or *backward-chaining* or *goal-oriented*) search is the most popular one. In the given problem statement (ie. program and query in PROLOG terminology), this regime focuses on the final assertion of the problem as the initial subgoal and uses the deductive relationships to eventually trace it back to the given facts (or ground axioms). All PROLOG systems use this control regime. *Set-of support* (see Section 4.5.1) as well falls into this category.

Bottom-up (or *forward-chaining*) search goes the opposite direction, starting with the facts and deriving from them new relationships until the final assertion is among them. An obvious drawback of this regime consists in its unselective production of relationships that may have nothing to do with the given problem. For this reason top-down search is mostly preferred but, unfortunately, has its drawbacks as well [BLS87]. Therefore, advanced systems necessarily need employ more sophisticated regimes taking into account a number of cooperating regimes. For instance, while eventually the dominant proof process might follow a top-down regime, in a preparatory phase, say, a limited process might rather be guided by a bottom-up control, in this combination avoiding disadvantages of each regime applied by itself.

Simplifications as discussed in the previous subsection may be regarded as bottom-up regimes in this sense. Namely, by simplifications some redundant connections are discarded first so that the subsequent proof process is restricted to those connections remaining in the simplified formula. This observation includes the simplifications noted there in connection with the compression principle.

Additional sophistication for the strategic behavior of the proof search is achieved by what could be phrased dynamic simplification. In resolution subsumption is used in this way, for instance. Since the connection method does not explicitly generate new clauses, comparable measures here look different (although one could of course translate those from resolution into the connection method via the relationships discussed in Section 3). For instance, (dynamic) *factorization* (see Section IV.6 in [Bib87a]) is among the techniques that together achieve the same effect. Namely, if an extension step resulted in a new subgoal that already is among the set of previous subgoals yet to be solved then the previous one needs not be solved any more and thus may be discarded from the set of subgoals. Another important measure of control here is the avoidance of *tautological cycles* as defined in 4.5.3. In contrast to hinged loops (Section 4.3) such cycles close the cyclic structure *not* in a single literal but in a *different* one. Both features, factorization and avoidance of tautological cycles, may be combined [BH82, Joh89].

4.5.2 Theory connections

Let us now turn our attention to compression. Several kinds of bottom-up regimes falling into this category have recently attracted a lot of attention in the literature. As done in the next chapter, they may all be regarded as *theory resolution* (or *theory connections*; see below). Among them is the treatment of *sorts* (or even more general representational structures). This work is rarely recognized as falling under the issue of control. As we will argue below it preferably should be seen in this context.

Take sorts first and let us refer once again to the example from the previous subsection.

$$\begin{array}{cccccc} & \neg Qx & \neg Pa_1 & \cdots & \neg Pa_n \\ \cdots & Px & & & \end{array}$$

By considering the depicted connections, resulting in

$$\cdots \qquad \neg Qx^{\{a_1 \ldots a_n\}},$$

prior to any other proof steps, we obviously follow a bottom-up regime. It results in sort information for the variable x (now restricted to the domain $\{a_1, \ldots, a_n\}$). This information may further be propagated along the connections of the entire matrix resulting in sort information for all the affected variables in the matrix. This way, in the "actual" subsequent top-down proof process, many lines of reasoning may turn out not to be viable simply because of conflicting sort information, thereby pruning the search space. In certain areas this simple idea results in impressive performance enhancements [BLS87, Bib85]. It seems rather straightforward to show that such a strategic approach to sorted logic rather than the ones discussed in the subsequent chapter amount to the same computational result (although no one so far has bothered to write this down in detail).

Let us further consider a more general theory than just sorts in the following example.

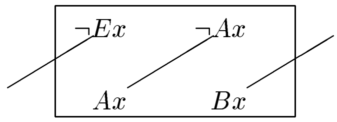
$$\begin{array}{cccc} & \neg Ex & \neg Ax & \neg Bc \\ Ec & Ax & Bx & \end{array}$$

In matrix representation it expresses (for instance) that the cardinal is a bird, birds are animals, animals eat, hence the cardinal eats. As we see, there is a linear chain of connections. As an isolated connection the one for A may be removed as described in the previous subsection. But we may even go a step further in simplifying the remaining deductive process.

Assuming that the eating habits of the cardinal are subject to many inquiries to the system, it may pay to "compile" the entire chain of connections into a single one of a new kind.

$$\begin{array}{ccc} & \boxed{EB} & \\ & & \neg Bc \\ Ec & & \end{array}$$

The little box associated with the connection, besides identifying its kind, is a reminder of the details



that have been abbreviated by it. This is one more instance of our principle of compression from the previous subset.

The more popular way of looking at this abbreviation consists of considering the boxed clauses as a theory under which the new kind of connections is applied. Hence we speak of *theory connections*. They were first introduced in [Bib87a][19]. Since any set of clauses makes up a theory, the idea has become popular [Sti85, Pet90].

Our view of theory connections suggests a technical distinction between theory connections of the kind just described and the sorts discussed previously. The latter are concerned with connections into leaves of the connection graph, while the former involve "inner" connections. Obviously both kinds may be combined, thus opening even more room for specialized investigations. This brings us to the crucial question of what the perspectives are which are opened up by such investigations for future general[20] ATP systems (of any kind, ie. resolution, connection method, etc. — recall what we concluded in Section 3.8). Three different such perspectives seem possible.

In the first perspective we let the user specify, along with the problem, any theory from a toolbox of theory connections that the proof system should apply. Users are already asked too much in their interaction with complex systems. Hence, while available information from the user's side is welcome, relying on it seems impractical.

In the second perspective we may think of a toolbox of tens or even hundreds of different theory connections available to the system. The figures are not exaggerated since any combination of theories makes a new theory. In other words, any subset of theories defines a new theory connection, thus causing another exponential explosion. Also, all these specialized treatments of theory connections have to be elaborated and coded in a uniform way, amounting to a huge effort. Now, given a particular problem, the system has to figure out in an initial phase whether any of the theories in the toolbox are part of the problem statement. This is by no means a trivial task. In fact, since the user may use different notation, it is equivalent to testing graph isomorphism, for which no polynomial algorithm is known.[21] If this search is successful, only then may, at any given point during the proof process, the system try to apply any of the identified theory connections. If the reader is willing to add up the total of all the costs involved, he may, like these authors, develop quite a bit of skepticism for this perspective.

The third perspective retains the attraction of logic that lies in its uniformity, and still takes advantage of the features that we are talking about. But in contrast with the previous approach, this one aims at a system based on uniform principles as outlined further above. This way the desired computational effect may be achieved flexibly customized for each given problem.

One might think in this context in terms of two different levels of representations. There is the level of problem specification where the user has no constraints other than those posed by logic. In a uniform compilation process this specification will be compiled into a form on a second, lower level, so that the subsequent proof process in effect will take advantage of sort and theory information but without applying specialized procedures. This behavior will rather be caused in a uniform way by the principles of simplifications, control, and compression

[19] Consider the first (or second) edition, Section V.1 for databases, Section V.3 for the theory of equality, and Section V.4 (2nd edition only) for other theories.

[20] not for truly specialized systems as those discussed in Section 6

[21] The situation might not be as bad as stated here in order to make our point clear. Algorithms for graph isomorphism work quite well in practice and the graphs involved might be rather limited in size.

discussed in this whole section. Even if this seems harder to achieve (which should not be taken for granted), the advantages of this perspective make it worth the effort. Some future system might, however, integrate limited aspects of the first two perspectives into the dominant and favored third one.

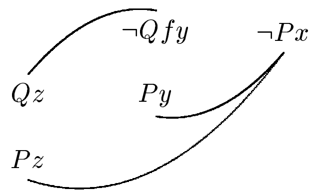
What we said here in the context of theory connections, applies in a similar way to *generalized unification* and *rewriting*. It is argued in this spirit in Section V.4 of [Bib87a] that our "two level system" model should be used here as well for the integration of the ideas underlying these methods. That is, both methods in effect realize techniques of control and compression. This is fine on the lower level, as long as it results from a uniform compilation of the higher-level problem statement represented in the uniform logical formalism.

4.5.3 Cycles

Many problems are stated using recursive rules. As a simple example, think of the definition of natural numbers. Here, for a constant 0 and a unary function $^+$, the successor function, it is specified that 0 is a natural number, $Nat(0)$, and that with n also its successor is a natural number, $Nat(n) \rightarrow Nat(n^+)$. Recursive rules like the one just given deductively have a cyclic connection structure which, once recognized, can be used for an enormous speed-up of proofs. In the present section we provide the basic concepts in this context, starting with that of a cycle.

Definition 4.5.1. In a set of clauses a *cycle* is a set $\{c_1, \ldots, c_n\}$ of connections $c_i = \{L_i, R_i\}$, $i = 1, \ldots, n$, such that R_i and L_j with $j - i = 1 \bmod n$ are different literals in one clause. A cycle is called *linear* if there is no literal that occurs in more than one of its connections.

Recall from 2.7 that unifiability of the terms in the literals is not inherent in the notion of a connection, that is, only the predicate symbols determine connections. Our notation $\{L_i, R_i\}$ is meant to indicate an implicit existential quantification that orders the otherwise unordered pairs of literals in the connections.[22] The following example illustrates a cyclic set of connections that would not qualify as a cycle in the sense of our definition, since Px is contained in two connections next to each other.



The cycle formed by the two connections in transitivity (or ancestor) rule $\neg Axz, Axy, Ayz$ is not linear since $\neg Axz$ occurs in both of them. There are two different types of such cycles, the tautological and the recursive ones. Let us first consider tautological cycles.

Definition 4.5.2. A cycle is called *tautological* if there exists a clause C involved in the cycle and a substitution which on C is the identity substitution, such that all connections in the cycle become complementary pairs of literals.

For instance, the cycle in each of the following clause sets is tautological,

[22] More precisely, for two such connections c_i and c_j there exist literals R_i and L_j with the property mentioned.

$$\begin{bmatrix} \neg Px \\ Qy \\ Px \end{bmatrix} \begin{vmatrix} \neg Px & \neg Qz \\ Qy & Pz \end{vmatrix} \begin{vmatrix} Pxy & \neg Qza \\ Qyz & Pza \end{vmatrix}$$

while this does not hold for the following clause sets.

$$\begin{bmatrix} \neg Px \\ Py \end{bmatrix} \begin{vmatrix} \neg Axz \\ Axy \\ Ayz \end{vmatrix} \begin{vmatrix} \neg Px & Qa \\ Qx & Pb \end{vmatrix} \begin{vmatrix} \neg Px & \neg Qza \\ Qxy & Pua \end{vmatrix}$$

In no case can tautological cycles ever give rise to a recursion since the identity substitution on C in the definition prevents any progress while traversing the cycle in a deductive process. Under certain conditions even some or all of the connections in a tautological cycle may be deleted without affecting provability. On the ground level such conditions have been stated in Lemma 3.6 of [Bib81] for a class of tautological cycles called simple circuits. The generalization to the first-order level follows standard techniques [Bib87a], which always require some care, however. The deletion of such connections may leave some of the previously connected literals pure in the sense of [Bib81] so that whole clauses may become obsolete. Altogether this is a powerful generalization of the well–known reduction of deleting tautologies mentioned already in Section 4.4.

Definition 4.5.3. A cycle is called *potentially recursive*, if it is not tautological and if there exists a weak unifier for all its connections.

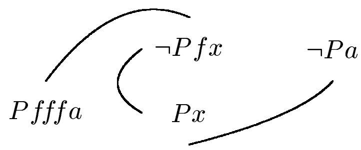
Weak unification was defined in [Ede85b] and captures whether in a proof a pair of literals is potentially unifiable by possibly renaming variables so that a variable occurring at both ends of a connection adopts a name at one end that is different from the one at the other. For instance, the connection in

$$\begin{matrix} \neg Pfx \\ Px \end{matrix}$$

is weakly unifiable in this sense since x will be renamed to, say, x' in one of the two literals. The obvious idea behind this concept is to take into account the possibility of considering more than a single copy of the clauses involved.

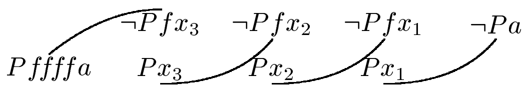
Definition 4.5.4. A cycle is called *recursive* if there is a positive integer n and a proof of the formula with a spanning set of connections that contains n instances of the whole cycle, ie. of all its connections.

For instance, there are two such instances (depicted here with one connection only) from the potentially recursive cycle of the previous example in the following proof.



Note that this example is actually the one from the outset of the present subsection defining natural numbers except that we used more conventional logical symbols here.

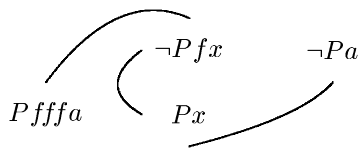
If the instances of the clauses involved in a recursive cycle are represented explicitly, then there is actually no cycle anymore. This may be seen in the following picture where, along with the substitutions involved, the three instances of the rule in the previous proof here are represented explicitly.



$$\left\{\begin{array}{l} x_1\backslash a \\ x_2\backslash fa \\ x_3\backslash ffa \end{array}\right\}$$

Nevertheless there is this cyclic property of coming back again and again to a copy of the same clause.

In [Bib88a] an efficient way of computation has been outlined for the case of recursive cycles. To give the idea, note that in

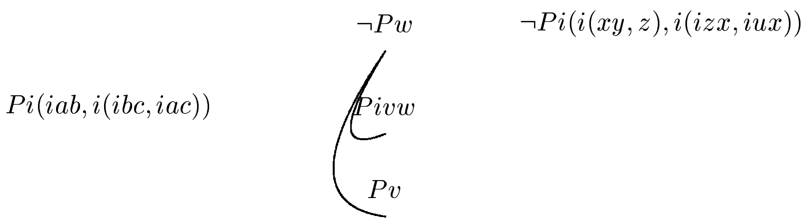


the only unknown is the number n of times the cycle is involved. This can be determined by solving equations extracted from the terms to be unified via the connections. In the present case we get the following equations, one for each connection.

$$x_1 = a\,,\ \ x_{i+1} = fx_i\,,\ \ x_n = fffa$$

Finding the proof reduces to determining n in these equations.

Often there is more than one cycle involved in a problem. Apart from the transitivity example given above we display another much discussed problem of this kind which originates from Łucasiewicz (see [Pfe88] for example).



The terms represent implicational formulas and P asserts derivability of its argument. Thus the second clause expresses modus ponens. It is this clause which causes exponential growth of the search space since each of its subgoals may instantiate it again. Not only are there two cycles represented by the two depicted connections, but also the proof may swap one cycle for the other an unknown number of times. Whenever this happened, the cycle connection will be instantiated an unknown number of times. So we get an unknown number of unknowns here. Still the equations can be given similarly as illustrated above and probably solved more easily than exploring the exponential search space following a conventional proof method [Bib91].

4.5.4 Instances

As we know from Section 2.7 psssibly more than one instances of (parts of) the matrix are needed for its proof. In the previous section we have seen how this works in the presence of recursive rules. The Łucasiewicz example given there has demonstrated how explosive the expansion of the matrix by the addition of further instances of clauses is for the search space involved. Of course, this explosion is even more disastrous if the cycle structure is not yet known as in the case of recognized recursions. In consequence, expansion must be handled with utmost caution from a strategic point of view. Note that the use of an index increase as described in Section 2.7 instead of an explicit expansion does not make any difference for the search space in this respect (while it may save a lot of storage space).

The importance of the issue of multiple instances is not restricted to the connection method as we already emphasized generally in the outset of this entire section for all issues treated in it. It is one of the mysteries of the field why in the context of resolution this issue has never been recognized in its full importance separate from other strategic issues. Part of an explanation is the fact that resolution carelessly generates new instances of clauses whenever they become involved in a resolution step.

In the context of the connection (or matings) method the importance of this issue has been recognized from the very beginning. The adjective "systematic" in the title of the very first paper [Bib74] in this line refers to the attempt of getting the procedure to do its job with the least number of instances of (parts of) the matrix possible. The issue has by far not been settled since. The following progress can be reported though.

One way to diminish the damaging effect of expansion consists in transforming the formula to be proved into an equivalent one that requires less instances for its proof than the original one. So the procedure would be able to detect the proof with less expansions. The most important transformation with such an effect is the narrowing of the scope of (affirmatively) universal quantifiers [Bib87a]. A simple idea in this context is the expansion by instances of formula parts rather than by a new instance of the entire formula. More intricate transformations of this kind seem possible [Bib77] but have never been explored.

Especially if non-normal form formulas are taken into consideration then splitting a formula into two conjunctive parts may also have the effect of reducing the number of instances required. Unfortunately, in other cases this may also be of disadvantage in terms of the resulting search space. For this reason a complete strategy called *splitting by need* has been developed [Bib87a] which offers the advantage and avoids the mentioned disadvantage.

This strategy is defined for arbitrary formulas in mind which are not necessarily brought into any normal form, ie. not even into Skolem normal form. In other words, the introduction of Skolem functions is avoided in this approach. This is made possible by way of a relation among quantified variables [Bib87a, Nos85] which might be called the *Skolem relation*. It allows the avoidance of circular unifications which alternatively would be excluded by Skolem functions. This alternative of using such a Skolem relation has advantages other than those mentioned here in the context of coping with instantiations.

In special cases it is possible to deduce from the structure of a given formula that the expansion by certain of its parts can never contribute to a proof; such cases are given in [BEF83]. If expansions are needed regardless of all the precautions mentioned so far, then one should duplicate the smallest possible part such as a clause rather the entire matrix (as it was naively done in the earliest systems).

Further, the additional clause should be associated with the active path (see Section2.7) in which the need for expansion arose, rather than simply added to the formula. This way the search space will not automatically be increased for paths other than those incident with the active one. In [Iss90] it is shown that this precaution provides a substantial improvement in the effectiveness of povers. In a depth-first search the technique of stepwise expansion, which was used in the earliest connection method systems such as [Bib74], today is known as the *iterative-deepening* strategy.

4.5.5 Clause linking

Most proof procedures for first-order logic are obtained by generalization of some procedure for propositional logic. Clause linking (CLIN) [LP92] is a remarkable exception in this respect. It is based on the following idea.

If a clause of the given matrix is involved with some instance in the proof, each of its literals must be contained in one of the connections of the spanning set establishing the proof. So why not first checking all the possible *hyper-links* for each clause, ie. the set of connections containing the clause's literals with one connection for each literal such that there is a substitution which makes all connections complementary, and code the resulting consequences in an instance of the clause? This idea of controlling the search for a spanning and complementary set of connections leads to the following three steps to be performed at each iteration of CLIN.

1. Generate new clause instances as described below and add them to the previous clause set.
2. In preparation of step 3 substitute in a copy of the new clause set any variable by some new constant.
3. Test the resulting propositional formula for tautology; in case of success the original matrix is valid; otherwise continue at step 1 with the new clause set.

For illustration of the instance generation mechanism mentioned in step 1 consider once again the matrix

$$\begin{array}{ccc} & Px & \\ \neg Pa & & Pf^8a \\ & \neg Pfx & \end{array}$$

from Section 2.10, used again in the subsection on cycles above. There is only a single clause here that can produce new instances, namely the second one. Each of its literals is involved in two different connections available in the matrix. Hence we have to consider four different potential hyper-links resulting from all possible combinations satisfying the properties of a hyper-link. Let us choose, for instance, for the first literal Px the connection with the leftmost clause and for the second literal $\neg Pfx$ the one with the rightmost clause. These two connections are not compatible though, because there is no substitution which makes both complementary, ie. these two connections do not qualify as a hyper-link in the above sense. The remaining three hyper-links do, however. One of them uses the same connection for the first literal as just before, but for the second literal $\neg Pfx$ takes the other possible connection with (a different variant of) the second clause. There is a substitution which makes both connections complementary, viz. $\{x\backslash a, x'\backslash fa\}$ where x' is the variable from the other instance of the second clause. The application of this substitution to the chosen clause $\{Px, \neg Pfx\}$ results in the new clause instance $\{Pa, \neg Pfa\}$. In the same way, the remaining two hyper-links yield the new clause instances $\{Pfx', \neg Pffx'\}$ and $\{Pf^7a, \neg Pf^8a\}$. The addition of these three new instances to the original matrix completes step 1 in this iteration. An alert reader might note that this step is in fact a special case of a consolution step discussed in Section 2.8.

In preparation of step 3 the two occurring variables x and x' are substituted in a copy of these clauses by some new constant c. Then the resulting propositional matrix is tested for tautology with some algorithm. In CLIN the Davis-Putnam procedure (see Section 2.7) is chosen for this purpose.

In our present example it takes two more iterations until such a tautology testing algorithm succeeds. This illustrates that CLIN shares with the connection structure method of Section 2.10 the quality of avoiding the unnecessary duplications inherent in the extension calculus. It also allows to take full advantage of *factoring* in the sense of handling a literal occurring in more than one clause in a single step by extracting it in the form of a common factor as usual in mathematics. CLIN realizes this by use of the Davis–Putnam procedure which has this feature incorporated. Finally, CLIN combines a top-down strategy with a bottom-up one and this way offers the advantages described in Section 4.4.

The prize to be paid by this relatively simple method for offering all these attractive features is the excessive generation of clause instances. In comparison, a connection structure calculus with factoring and other simplifications as described in Section 4.4 would offer the same advantages without this drawback. It is more complicated to describe and implement, though.

4.5.6 Lemmas

To complete this subsection on control and compression we mention several other disguises of the principle of compression in the literature. One is based on the following idea. If a subgoal has been solved (say with some connection calculus), it may well happen that the identical literal turns up as a subgoal within similar structural context again. Instead of redoing the proof for it, one might rather associate with the literal the information from the earlier proof attempt and this way possibly save a lot of computation. This idea has been applied in [Fro85, LSBB92, Pla88]. Plaisted has used the name "caching" for this technique.

The connection structure method advances the connection method by integrating a feature of the compression kind which resolution incorporates by its very nature (see (Section 2.10 for details). One may think of the usual concept of lemmas as in mathematics in this context. A limited form of storing and reusing lemmas is realized in SETHEO [LSBB92]. Also [Pfe87] touches upon this issue.

[Bib90] has demonstrated that a more general form of lemma application, which allows renaming, leads to linear proofs for the pigeonhole formulas (already mentioned in Section 4.1). These formulas are of interest since [Hak85] has shown that resolution requires exponential proofs for them. For the recognition of such a lemma no polynomial algorithm is known though, since one can show [Pip90] that this problem is equivalent from a complexity point of view with the well-known graph isomorphism problem[23]. But this, of course, does not preclude the integration of such generalized lemma applications in proof systems. An attempt in such a direction is currently pursued with SETHEO under the topic of *proof schemes* [LSBB92].

4.5.7 Omissions

The sheer size of the entire subsection indicates the extension of its topic of control and compression. We could not begin to treat it in an exhaustive way. For instance, we did not even touch upon the issues of control in *parallel* (rather than sequential) systems [BKA+87]. Further, the ATMS technique [RdK87] might have well been included here as a technique of controlling the search for proofs. Also, a treatment of calculi with cut or with extensions would be desirable under this topic. Literally nothing exists on this topic in the literature though (to the best of these authors knowledge). Even the obviously close relationship of the extra features in these calculi with the use of lemmas is only badly understood. Finally, the reader is referred to the material in the subsequent chapter once again.

[23] [Bib90] contains a faulty statement in this regard in its concluding section.

4.6 Semantic guidance

Introspection seems to suggest that mathematicians (or any human reasoner for that matter) do much of their reasoning with concrete structures (a view of the surroundings, a diagram, etc.) in mind rather than in a syntactic way as provided by logical formalisms. A number of researchers in ATP have therefore argued that such *semantic* guidance would as well be indispensible in automated proof search. The first such work is [Gel59, Gel63a], a major contribution. Taking into account concrete geometrical diagrams allowed him to delete subgoals. By pruning the search space in this fashion, interesting theorems could be proved at a very early point in the brief history of ATP. Chapter 10 of [Bun83] provides a nice account of this approach. Other work of such nature is reported in [Sla67, Wan85].

Gelernter's work is restricted to Horn clause logic. [NP90] has generalized it to full first-order logic. This general case will be briefly discussed in the remaining part of the present subsection.

A structure like a geometric diagram, logically may be regarded as an interpreted logical formula, say F. Hereby interpretations I are defined as usual (see eg. [Bib87a]). So taking into account such (semantic) structures within a logical formalism means testing $I \models F$, for certain I, F, at particular points in the proof search. [NP90] have realized this within the formalism briefly discussed in subsection 4.3 by adding to the premises of each rule the additional premise $I \models \exists . L$ where I is an interpretation of the given set of clauses, $\exists . L$ denotes the existential closure of the literal L mentioned in the conclusion of the rules. With these modifications, in searching for a proof in a backward chaining way, no subgoal has to be pursued that evaluates to false in the interpretation for some instantiation of its variables. Obviously this prunes the search space in a most desirable way.

This is not the place to give a full axiomatization of geometry. For illustrating our point with a simple example just the following two axioms will suffice.

$$seg(x,y) = seg(u,v) \rightarrow seg(x,y) = seg(v,u)$$

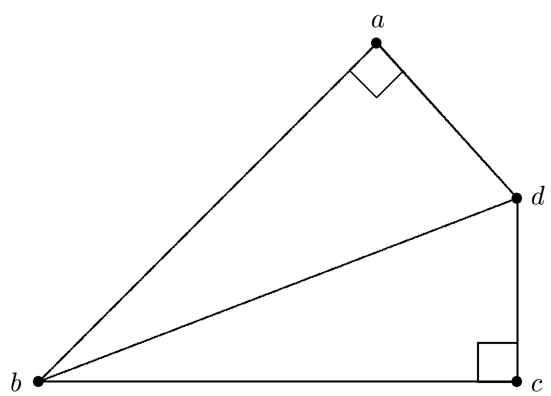
ie. segments are equal independent of the ordering of their endpoints.

$$tri(x,y,z) \equiv tri(u,v,w) \rightarrow seg(x,y) = seg(u,v)$$

ie. corresponding sides of congruent triangles are equal, whereby

$$\begin{aligned} angle(x,z,y) = angle(u,w,v) \quad & \wedge \quad angle(z,x,y) = angle(w,u,v) \\ & \wedge \quad seg(y,z) = seg(v,w) \\ & \quad\rightarrow \quad tri(x,y,z) \equiv tri(u,v,w) \end{aligned}$$

ie. triangles are congruent if they have two equal angles and a corresponding equal side.



Assume that, for a diagram a, b, c, d, as in the previous picture (taken from [Bun83]) and on the basis of the axioms just given, we want to prove the angle bisector to be equidistant from the rays, ie. $seg(a,d) = seg(c,d)$. Then a connection proof might be attempted as follows.

$$\begin{array}{ccc} & \neg seg(x,y) = seg(v,u) & \neg seg(x,y) = seg(u,v) \\ seg(a,d) = seg(c,d) & seg(x,y) = seg(u,v) & tri(x,y,z) \equiv tri(u,v,w) \end{array}$$

The resulting subgoal $tri(a,d,z) \equiv tri(d,c,w)$ (recall from Section 2.7 that the substitutions have to be determined by unification so as to make all connections complementary) would obviously fail to be true in the model provided by the pictured diagram. Any instantiations of the remaining variables z and w fails to result in congruent triangles as the reader (or the system) may check out. As Gelernter has shown, a subgoal, like the present one, failing to satisfy a given model can never lead to a proof. For our present example this means that the attempted line of proof can safely be abandoned in favour of an alternative one like the following.

$$\begin{array}{ccc} & \neg seg(x,y) = seg(v,u) & \neg seg(x,y) = seg(u,v) \\ seg(a,d) = seg(c,d) & seg(x,y) = seg(u,v) & tri(x,y,z) \equiv tri(u,v,w) \end{array}$$

The new subgoal $tri(a,d,z) \equiv tri(c,d,w)$ is actually true in the present model (ie. for the diagram above) if both variables are instantiated by b, so that we may proceed from here more successfully with the search for a proof.

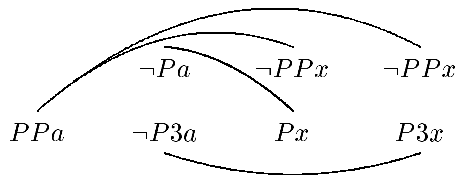
In precise terms, an extension step in the connection calculus (or in model elimination or in SLD resolution; recall from Section 4.2 that there are no essential differences among all these) for a Horn formula can safely be dispensed with, if the resolvent associated with the resulting structured matrix (recall this notion from Section 3.5) fails to be satisfiable in any given model. The same holds in the general (non-Horn) case unless there is a hinged loop containing one of the unsatisfiable subgoals in this resolvent and a literal from the active path as the hinge (recall the respective concepts from Section 4.3), in which case the extension cannot be dispensed with. Rather this loop has to be investigated for possibly providing part of the attempted proof. Obviously there might be more than one such loop to be considered if there were more unsatisfiable subgoals in the resolvent.

For illustration of this result, consider the following example, again taken from [Bun83]. In the following matrix, read Px as "x is a prime", PPx as "x is the product of primes", and $P3x$ as "x is a power of 3". The axioms state that a number is a product of primes if it is a prime (third clause) or a power of 3 (fourth clause). Under the condition that a is either a prime or a power of 3 (second clause), we want to prove that a is a product of primes (first clause), given these axioms. The matrix representing this setting looks as follows (ignore the connection for a moment).

$$\begin{array}{cccc} & \neg Pa & \neg PPx & \neg PPx \\ PPa & \neg P3a & Px & P3x \end{array}$$

Because of the second clause this is not a Horn matrix. If we interpret all predicates so as to adopt their intended meaning and let a be 9 then this results in a model for these clauses.

Assume we attempt to prove the formula using a top-down strategy (recall from Section 4.5) by starting with the main goal PPa. With linear chaining (as described in Section 4.2) along the depicted connection the subgoal Pa is considered after the first step. In this case, the resolvent associated with the structured matrix at this point simply consists of this subgoal. The subgoal fails to be true in the selected model (since 9 is not a prime). In the Horn case, this line of proof could therefore be abandoned according to the result stated above. Since the matrix is non–Horn, however, we have to check the "unless" clause. That is, we have to check whether there is a hinged loop containing this subgoal Px and PPa, as the only member of the active path, as the hinge. And indeed there is the hinged loop depicted in the following figure which has these two properties.



In fact, in this example no further subgoal is left over so that the loop establishes the entire proof. Note that starting out from the goal as before, but selecting the alternative connection (with the fourth clause) would as well have left us with a false subgoal when a is interpreted as 2 (a product of primes but no power of 3). But the same loop would have allowed us to proceed in an analogous way.

The result stated above for the general case is an improvement of Plaisted's result in so far as it does not require any contrapositives of the clauses to be considered. As already mentioned in Section 4.3, we also consider this way of representation more suitable than the sequents used by Plaisted.

Involving selected examples in searching for proofs simulates the human behavior dispensing with a line of proof whenever an example demonstrates that it cannot lead to anything useful. It is therefore quite promising to explore this particular strategic direction much further. For instance, the selection of useful examples, ie. interpretations, is still a difficult and unsettled problem [Hör81, BB82, Ble83, BH84, WOLB84, Wan85, CZ90].

Models have important applications in other areas of Intellectics (see Section refintroduction) such as in vision. An example is the logic of depiction developed in [?] where the interpretation of a picture is given in terms of such models.

4.7 Integration and implementation

One of the major challenges in the field of automated deduction is the integration of the numerous ideas and results available in regard to calculi and strategies many of which have been presented so far in this chapter. One would like to incoporate them all within a single system. But, to begin with, we still lack a deep enough understanding of the relationships of the various calculi so as to be able to select the undoubtedly best of them as a basis for such an ambitious implementation.

A first step towards ordering the space of different calculi has been attempted in [Bib91]. But it does not include any strategic considerations. Note in this regard that, while the strategies discussed in the present section cover all general ones, there are still many more results of detail as those presented in the subsequent chapter in the context of resolution. It is likely that among all these some subsume others. But little is known about this.

In this situation systems at present are developed with a limited selection of results incorporated into them. We mention a few of these general first-order deductive systems in the remainder of this section.

One of the best performing systems is OTTER (Organized Techniques for Theorem-proving and Effective Research) [WWM$^+$90]. It is the last in series of systems developed over the period of nearly three decades at the Argonne National Laboratory under the leadership of Larry Wos. It is written in C and available for many machines free of charge. The inference rules it uses are resolution, hyperresolution, unit-resulting resolution, and paramodulation. Various simplifications such as subsumption and demodulation are available. While being a truly remarkable system in comparison with others, OTTER has many limitations in the perspective taken in the present chapter. A parallel version of OTTER, called ROO, is also available.

PTTP (PROLOG Technology Theorem Prover) [Sti90] is a system written in PROLOG and developed by Mark Stickel at SRI International. It is based on the model-elimination inference rule, or on linear chaining to use the terminology of the present chapter (Section 4.2), and uses a depth-first iterative-deepening strategy. By expanding the set of clauses with all their contrapositives the system is made complete for first-order logic.

A system based on a similar approach as PTTP is SETHEO (SEquential THEOrem prover) [LSBB92] written in C and developed at the Technical University München. Its connection method viewpoint, a more sophisticated simplification module, the compilation into Warren abstract machine code, and the possible choice among a variety of different depth bounds are some of its special features. A parallel version, called PARTHEO, is also available. SETHEO is currently the most advanced in a series of provers developed on the basis of the connection method, starting with an unnamed prover developed in 1970/71 [Bib74].

The TPS Theorem Proving System [AINP90], developed at Carnegie Mellon University, is based on the matings or connection method. It is unique in that it realizes a theorem prover not only for first-order but also for higher-order logic (see Section 5.5).

Since the emphasis in this chapter is on methods and calculi, not on systems, our selection of systems is so limited, naming only one of the leaders in this technology and a few systems with a design philosophy close to the one favored in the present chapter. There are numerous other systems which would require a full section in its own right for a fair coverage which we dispense with for the reason mentioned and for space limitations. Some of them will be mentioned at other places within the present chapter as well as in other chapters (such as the subsequent one).

5 Restrictions and extensions of logic

In the previous three sections we have focused our attention on first-order logic, on the various attempts to capture the notion of validity in it, and on ways to guide the search for proofs. Although it is not by chance that when talking about logic the default meaning is first-order logic (cf. the previous Chapter), there are other logics of great importance. Taking first-order logic as an orientation point, we consider (some of) these other logics as restrictions or extensions in the present section.

The need for logics, other than first-order ones, has two opposing sources. One is the computational intractability of first-order logic with the means available today. For practical applications one has to think of restricted forms of logic to get results efficiently. On the other hand, a rich expressiveness provides comfort to the user so that even first-order logic appears too restricted, let alone the more restricted forms of it. One of the major research issues of this field is therefore the quest for a good trade-off between these opposite goals. Note that this trade-off necessarily is a function of the progress made in developing efficient proof procedures for fragments of first-order logic.

In the present section we will review a list of logics with increasing expressiveness. The first is in fact so weak that usually one would not actually recognize it as a logic. We are talking of formalisms for constraint satisfaction. They are of great (and neglected) relevance for automated deduction since their task is in fact part of the more general task of theorem proving. Further we discuss datalogic, that part of logic which is needed to provide convenient access to data-bases. Also, we mention Horn-logic as well as its extensions. Finally, we briefly review logics that go beyond the expressiveness of first-order logic, and also mention a few more in the summary.

5.1 Constraint satisfaction

Many tasks in Intellectics (see Section 1) are naturally seen as constraint satisfaction problems. Especially has the interpretation of images often been modelled as a constraint satisfaction problem. This type of problem has therefore been studied in vision quite extensively (see [Mac87] for an overview and for references). In such a setting, problems of this type are formalized in the following way.

We consider a set $V = \{v_1, \cdots, v_n\}$ of variables. With each variable v_i a set L_i of values (or labels) is associated. Further, binary predicates P_{ij} are considered such that $P_{ij}(k, \ell)$ expresses that the assignment of label k to the variable v_i is compatible with the assignment of label ℓ to the variable v_j. No function symbols are taken into account. With these notions the *Constraint Satisfaction Problem (CSP)* is defined as the problem of finding an assignment of labels to the variables that does not violate the constraints given by P_{ij}. See [Mac87, Bib88b] for examples and more details.

Any CSP can naturally be regarded as a logical problem of the following form [Bib88b].

$$F \rightarrow \exists x_1, \ldots, x_n\, G$$

where F is a conjunction of ground atoms (facts) of the form $P_{ij}(k, l)$ and G is a conjunction of atoms of the form $P_{ij}(x_i, x_j), 1 \leq i, j \leq n$. While in general problems might be stated with m–ary predicates, $m \geq 1$, the restriction to (unary and) binary ones is not really a restriction of generality [Mac77]. Whether we consider one or the other does not make any difference for what follows.

It is known that the class of constraint satisfaction problems is NP–complete [MF85]. Techniques have been developed to solve a CSP in a stepwise fashion so as to narrow the range of possible assignments with little effort yet as quickly as possible [Mac87, Bib88b]. These would deserve an entire section for their treatment (to some extent covered by Volume 4, Chapter 10).

We only mention that these techniques can be divided into two different groups. One is characterized by *constraint propagation*, *filtering*, or *network consistency* algorithms, and the other one by *implicit enumeration* or *tree search* algorithms. The former aim at narrowing the space of possibilities by applying what we called the compression principle (Section 4.4), with differences only in the details of organizing the steps towards compression. The latter focus on optimizing the enumeration of the space of possibilities so as to guide the process as closely as possible towards the solution within the represented data.

For automated deduction it is essential to note that virtually any deductive problem also involves (among other things) a CSP (since in virtually all cases we have a number of facts as well as existential quantifiers). It therefore seems natural to integrate these specialized techniques within deductive systems. Experiments of such a nature have demonstrated that this way the performance of such systems may be enhanced significantly [Hen87].

5.2 Datalogic

A *database* is a collection of explicitly given data organized in some way (see [Rei83] for a more detailed definition). Logically it may naturally be regarded as a conjunction F of facts (ie. ground atoms) exactly as in a CSP as defined in the previous subsection. Again no function symbols are considered. A query is very much like the existential part of a CSP except that typically a database query is of the much simpler form $\exists x_1, \ldots, x_n \; Px_1, \ldots, x_n$. We will therefore restrict the use of the notion of a *database query* to this special case where the existential formula involves a single atom only. In the more general case with more than one atom described in the previous section we will rather speak of a *constraint query*. The general notion of a *query* will then cover the case of the existential closure of a conjunction of literals. More complicated existential formulas are not usually considered in such a context.

While database queries pose the simplest possible problem of a deductive nature (which simply amounts to a matching problem), *datalogic* extends this kind of problems considerably by allowing additionally a set (or conjunction) R of (possibly recursive) rules, ie. Horn clauses. So a datalogic problem has the form

$$F \wedge R \rightarrow \exists x_1, \ldots, x_m \; G$$

whereby G might be restricted to a database query (although more general cases are sometimes taken into consideration as well).

For this case of datalogic, again, a number of special techniques have been developed that aim at speeding up the evaluation of queries. Section 4.5 in [Bib88a] provides some details of concepts and pointers to the extensive literature on this particular subject, while [Min88] is entirely devoted to this subject.

5.3 Horn logic

Horn logic is treated within the present volume in Chapter 6. Further, much of Volume 4 is concerned with this particular fragment of logic. In the present section it is included merely for completeness and for its relevance to the more expressive logics as far as the proof techniques are concerned.

From a theorem proving point of view the attraction of Horn logic lies in the following features. First there is a linear algorithm to handle the ground case of Horn logic (just keep applying unit resolution and subsumption). So the complexity of the propositional part poses no challenge at all in the case of a Horn problem (in contrast to the general case which is NP-complete). Second, it lends itself very naturally to a top-down regime whereby the prover starts with the given goal and proceeds from there until a proof is found. In the course of this procedure only a single clause (representing the intermediate subgoals) has to be kept at any one time (apart from the input clauses) which, thirdly, amounts to a relatively simple data structure necessary for storing the information accumulated during a proof. Finally, a given logic program may be compiled into directly executable machine code. This may speed up the execution considerably. All these features are a consequence of the non-existence of hinged loops in Horn logic so that linear chaining (as used in an extension calculus — see Section 4.2) is the only required proof technique. The features based on this technique have been realized in a number of quite successful PROLOG systems. As an aside we mention that Horn logic enjoys other interesting features such as the existence of minimal models which is discussed in detail in Chapter 6.

In the context of PROLOG or logic programming a different form of representing formulas is usually preferred. So far we have looked at matrices such as

$$\begin{array}{cccc} & \neg P & \neg Q & \neg R \\ P & Q & R & \end{array}$$

These have been explained in Section 2.7 as affirmative clause form representations. In logic programming a refutational approach is often preferred for which the (refutational) clause form for the same example reads

$$\begin{array}{cc} R & \\ Q & \neg R \\ P & \neg Q \\ & \neg P \end{array}$$

as explained in Section 2.7. Note that this representation is obtained from the previous one by way a 90^o counter-clockwise rotation and by switching negation signs. As always rows represent disjunctions so that the second row is to be understood logically as $Q \vee \neg R$ which is equivalent to $Q \leftarrow R$. So the last representation may also be represented as

$$\begin{array}{lcl} R\,. & & \\ Q & \leftarrow & R\,. \\ P & \leftarrow & Q\,. \\ & \leftarrow & P\,? \end{array}$$

which is the way preferred in logic programming. As we see the correspondence between the (affirmative) matrix representation preferred in the present chapter is very simple so that the reader should have no difficulty in easily switching from one to the other.

Horn logic in general and PROLOG in particular have their drawbacks as well. First of all, they lack the expressiveness of richer logics such as first-order logic needed in many applications. The extensive literature on the treatment of negation in PROLOG (see Chapter 5 in Volume 4) implicitly expresses the discomfort resulting from this restriction. Further, PROLOG uses an incomplete search strategy (depth-first) and an unsound unification algorithm, both resulting in unexpected behavior. Finally, there are a number of drawbacks discussed in Section 4.5 of a purely top-down control regime as used in PROLOG. One, therefore, would prefer "to remedy these deficiencies while retaining as fully as possible the high performances of well-engineered PROLOG systems" [Sti88]. There are quite a few attempts in such a direction, some of which will be briefly discussed in the next subsection.

5.4 Extensions of Horn logic

Given the limitations of Horn logic, it is not surprising that many attempts have been made to characterize logics more expressive than Horn, yet computationally less complex than first-order logic. We may distinguish essentially three different directions of such extensions. One — historically the first — consists of treating *negation* (in the body of rules) *as failure* (in proving the unnegated goal). That is, the proof technique of linear chaining is retained but applied in a modified way in the presence of negation. This has the undesirable consequence of changing the semantics of the formula. For details the reader is referred to Chapter 5 in Volume 4.

The second direction of extension involves a modification of the formula but one that leaves the semantics unchanged. A first instance of this consists in renaming negated predicates $\neg P$ as P' which in certain cases results in a Horn formula. A second instance of this direction is the way PROLOG treats disjunction in the body of its rules. It uses for this purpose the technique described in Section 3.1 of linear transformations to normal form, which consists in introducing names for the disjuncts and defining them appropriately. For instance, if the clause reads $L \leftarrow L_1 \vee L_2$ then it is (coded as if) replaced by the three clauses $L \leftarrow L'$, $L' \leftarrow L_1$, and $L' \leftarrow L_2$. Three further instances of this direction have already been discussed in Section 4.3, namely the replacement of each clause by all its contrapositives, the application of Prawitz' matrix reduction, and the application of factoring.

A third direction of extension retains the formula as it is but applies linear chaining beyond Horn clauses. This characterizes a class of formulas, called *case-free* programs in [WP90], provable this way. Recall from Section 4.3 that the body connections introduced there are needed for obtaining a proof whenever reasoning by cases would occur in a natural form of deductive argument. From this we may understand that a case-free program together with a query is a theorem of first-order logic that admits a connection proof the mating of which does not involve a hinged loop (as defined in Section 4.3). The following is an example of such a case-free program.

$$\begin{array}{rcl}
\mathit{Math.maj}(\mathit{mike}) & \leftarrow & \\
\neg\mathit{Math.maj}(\mathit{jon}) & \leftarrow & \\
\mathit{Take}(x, \mathit{math200}) & \leftarrow & \mathit{Math.maj}(x) \\
\mathit{Take}(x, \mathit{math100}) & \leftarrow & \neg\mathit{Math.maj}(x)
\end{array}$$

Whatever the query is, the body connection $(\mathit{Math.maj}(x), \neg\mathit{Math.maj}(x))$ will not be useful for establishing any proof, that is no hinged loop may become involved in any proof. So, while the formula is non-Horn and cannot be made Horn by renaming predicates, it is still accessible for proofs with linear chains.

The class of case-free programs has recently been characterized also semantically in [WP90]. Note that it is very close to the sets of clauses that have unit resolution proofs.

The disjunctive logic programs discussed in Chapter 3 of Volume 3 might seem to belong to the category discussed in the present subsection. But actually they have the full expressiveness of first-order logic. This is because any formula in first-order logic can be transformed into clause form as explained in Section 3.1. Any clause is of the form $L_1 \vee \ldots \vee L_m \vee \neg K_1 \vee \ldots \vee \neg K_n$ with atoms L_i, K_j, $i, j \geq 0$. But such a clause is equivalent to $L_1 \vee \ldots \vee L_m \leftarrow K_1 \wedge \ldots \wedge K_n$ which is exactly the kind of rules of which *disjunctive logic program* consist. Therefore everything discussed in this chapter in view of first-order formulas applies to disjunctive programs in the same way. However, in the literature of disjunctive logic programs aspects are discussed that are not treated here (such as fixpoint semantics).

5.5 Extended logics

One of the basic tasks of logic is the formalization of mathematical reasoning. The task may be regarded as a test-bed for the more general task of formalizing human reasoning beyond mathematics. It has always been felt among logicians that one would have gone a long way towards the latter more general goal if the more specific goal were achieved to full satisfaction.

First-order logic is expressive enough to formalize virtually all of mathematics. This is so because axiomatic set theory (see Section 6.3) can be formalized as a first-order theory and most of the rest of mathematics can be established on the basis of set theory. *Higher-order logic* (otherwise known as *type theory*) offers an alternative formalization of mathematics. Those people familiar with both alternatives recognize that in some ways the *hol* approach provides a more natural vehicle than the set theoretic one for formalizing what mathematicians actually do. Put more strongly, encoding set theory in first-order logic is awkward and unnatural for theorem provers (and humans). [And86] discusses both ways in some detail and gives a full account of the higher-order logic approach.

In the vein of what was previously said, one might be tempted to conclude that also for modeling more general human reasoning *hol* might be the appropriate candidate. The rare experiences of this type seem to support this conclusion. In any case, *hol* presents itself as a serious candidate for study of its deductive properties. Again, the presentation within this chapter must be restricted to providing a very few glimpses at this logic and pointers to the literature. As a comprehensive source for a detailed study of this logic, Andrews' book [And86] is warmly recommended for further reading as is [vBD83]. Chapter 10 in the present volume is devoted to the topic from a slightly different perspective.

As a first addition to the language of first-order logic we introduce *λ-abstraction*. It provides a convenient way of naming functions. Suppose, we wish to name the function that adds to any natural number the value 2. With λ-notation the name would be $\lambda x.x+2$. This might be read as "the set of pairs $(x, x+2)$". More than one bound argument can occur as in $\lambda xy.(x+y)\cdot 2$, naming the function that takes the sum of its arguments and multiplies the result by two. So λ-abstraction pairs a tuple of variables with a term wich in the present context is first-order. The result may be considered as a name for a function just as any usual first-order function symbol and may be used in exactly the same way within first-order logic.

As with any function, such a *λ-expression* (or *λ-term*) may be applied to particular argument values and evaluated. For instance, taking 3 and 4 as arguments for the latter function yields $[\lambda xy.(x+y)\cdot 2](3,4)$. The operation is called *application*. Once particular arguments are given, as in this case, the expression may be reduced by replacing the bound variables by their values, an operation called *λ-conversion*. In the present example we obtain the expression $(3+4)\cdot 2$. Altogether we thus are dealing with the operations λ-abstraction, application, and λ-conversion.

As we have seen, λ-notation can be integrated even within first-order logic. The λ-expressions are allowed in the place of function symbols within the recursive definition of terms when specifying the respective first-order language. Now we go a step further.

Recall that the quantifiers in first-order logic may bind individual variables, but no function or predicate symbols. The sentence "there exists a 2-place function f such that $f(3,2)=5$" could therefore not easily be translated into first-order logic since the obvious translation into $\exists f\ f(3,2)=5$ results in an illegal expression. This expression is a well-formed formula of *second-order* logic, however. In this logic (universal or existential) quantification is allowed on any n-ary function or predicate variable with $n\geq 0$, in addition to all features of first-order logic with λ-expressions.

Further the need naturally arises to relate, say, functions (ie. symbols of first-order) and individual values (ie. symbols of zero order). For instance, one might wish to express that all functions of a certain class on a specific argument take the same value, $\forall f$ *SAMEVALUE(f,arg)*. These new predicate and function variables of the next higher, in this case second, order could again be quantified leading to third-order logic; and so forth.[24]

[24] This way of counting orders follows Section V.6 in [Bib87a] rather than [And86], the latter taking two steps where the former takes only one. In essence it is compatible with the definition of "order" given in [vBD83] except for a difference of 1 in the order of a type, since we prefer to assign a term of type 0 also the order 0 rather than 1.

In practice a few orders suffice which, from a deductive point of view, can be handled very much like first-order logic as discussed shortly. In fact, (much of) higher-order logic may be viewed as a sorted version of first-order logic as done in Chapter 4. Also Frege has taken a single-order view of this construction [Fre79]. In a more general view, first taken by Russel [Rus08], arbitrary *types* of relation and function symbols are introduced for the systematic distinction of these symbols leading to an arbitrary-order logic, technically called *ω-order logic* or *(finite) type theory*. This logic avoids inconsistencies which might otherwise arise when carrying out this generalization with less care. But in its full generality the terminological apparatus is somewhat frightening for the non-specialist (but considered beautiful by the specialist). The inconsistencies can be avoided without the introduction of types and sorts as shown in [Per85] which is more in the tradition of Frege (cf. also Chapter 11 in the present volume).

The automation of proof search in type theory may be based on a generalization of the fundamental theorem 2.7.1 for the connection method in first-order logic which can be found in [And81, And86, Bib87a]. A less formal account of many of the issues involved is given in [And89]. Such an approach has first been realized by Andrews and his collaborators with a system called TPS [MCA82, AMCP84, AINP90] (see also Section 3.4).

In brief, all paths through the formula have to be tested for complementarity in this approach the way explained in Section 2.7. As on the first-order level, a unification algorithm is involved in this test. Since unification here has to account for λ-conversion (in the form of *term expansions*), matters become much more complicated. In fact, in contrast to the first-order case, unification becomes undecidable already on the second-order level [Gol81]. This does not at all mean that we could not carry out proofs in practice as TPS demonstrates which uses Huet's unification algorithm [Hue75] (see also [JP76, SG89]).

Another important family of extensions of first-order logic is *modal logic* [HC68]. Again there is an entire chapter (viz. 8) in the present volume devoted to this topic so that we may just mention it here for completeness, thereby emphasizing connections with the material discussed so far.

In brief, the language of modal logic is extended from a first-order language by the two unary logical operators $\Box$ and $\Diamond$. In the presence of negation, one may be expressed by the other (as in the case of quantifiers). There are a number of interpretations of the meaning of these operators. One is that $\Box$ means the same as $\vdash$ and thus denotes derivability. In this interpretation, modal logic may be regarded as the meta-language of first-order logic (cf. Chapter 11 in the present volume). In the same vein, $\Box$ may also be regarded as a higher-order predicate *ALWAYS.TRUE*. In other words, there is a close, but badly understood relationship of modal logic with higher-order logic. From what was said above, we can thus conclude that also modal logic may be treated as a first-order logic [Per85].

This reduction of many kinds of modal logics to first-order logic has technically been realized by Wallen in an extremely elegant way [Wal90]. As with higher-order logic, proofs again are spanning sets of connections as on the first-order level (discussed in Section 2.7). The only change applies to the unification part on which an extra burden is imposed in order to account for the modal operators. In consequence, this extra burden enters already on the propositional level. The way it is handled depends on the kind of modal logic under discussion. This is extremely satisfactory from a deductive point of view since now most techniques developed for first-order apply naturally to its extensions.

Based on the work of Wallen, these first-order proof methods have now also been imported into the framework of resolution [Ohl88]. This is not the first time that an important insight has been gained through a study within the framework of the connection method which was not discovered within the resolution framework despite a number of vested attempts.. These experiences support our thesis that the perspective provided by the connection method is more revealing than that of resolution.

While higher-order and modal logics are extensions of first-order logic, containing the latter as a subsystem, the relation of *intuitionistic logic* and classical logic is more complicated. On the one hand, intuitionistic logic is a restriction of classical logic in the sense that every intuitionistically valid formula is also valid in classical logic. On the other hand, a formula in intuitionistic logic expresses more than it does in classical logic. Also we know that the intuitionistic connectives can be faithfully translated into modal logic.

Intuitionistic logic has been regarded as *the* logic of computation. Higher-order versions of the logic are used as bases for meta-logics, ie. formalisms in which other logics may be defined and encoded for representation on, and manipulation by, computer. We mention the constructive logic developed by Martin–Löf [ML82, ML84] as a particularly important representative of this family. It may be regarded as an intuitionistic version of the type theory discussed before. A variant of it is realized in the NuPRL system [CAB$^+$86]. A similar approach is pursued by Huet with his calculus of constructions [Hue87, CH88]. A proof checker, called AUTOMATH [dB80], is also based on some variant of these logics. The calculi used in all these systems are variants of the natural deduction calculus discussed in Section 2.3. The systems typically do not aim at a full automation of reasoning, but are rather meant to support the human activity of reasoning in an interactive way.

Given the importance of these logics especially in the context of programming, it is very satisfying that Wallen could apply his technique to intuitionistic logic as well by exploiting the similarity between the modal logic S4 and intuitionistic logic [Wal90]. This means that the deductive techniques discussed in the present chapter apply to most, if not all the extensions mentioned in this section.

Recently, Girard has proposed what he calls *linear logic* [Gir87]. This is a constructive logic like intuitionistic logic and is as strong as intuitionistic logic which thus can be embedded into linear logic. But in contrast to intuitionistic logic it shares a number of nice features with classical logic among other attractive properties. Linear logic thus challenges intuitionistic logic for its role of *the* logic of computation. There are a number of striking similarities of linear logic and the linear proofs used for plan generation [Bib86] which are worthwhile for further investigation.

Yet another class of logics is concerned with non-monotonic reasoning. Since Volume 2 is devoted to this topic we may skip it here. So we appear to have at least mentioned most important categories of logics in this section.

6 Special theories and heuristics

The thesis on which logic rests is that human reasoning can be modeled by a logical formalism independent of any particular application. A particular theory is specified in this approach by a number of axioms and new properties are derived under these axioms as premises. It is conceivable that for such a particular theory proof search along the lines discussed in the previous sections may be optimized considerably by taking advantage of the special structure of the axioms. Such optimization clearly does fit well into the model just described. Yet relatively little has been done so far in such a direction.

This is not to say that there is no work on deduction in special mathematical theories. Rather there is even too much to be adequately covered in the present section. But either this kind of work specializes on the theory to such an extent as to ignore the general logical mechanisms of deduction altogether; or it bends the logic ending up in a specialized logic (eg. in the way already discussed in Section 4.5.2) rather than optimizing the deductive treatment of the one general logic. This way we have reached a situation where there are hundreds of different logics or special procedures known in the literature, a figure that is expanding rapidly every day.

Since there are infinitely many theories, where is this development meant to lead us? Each of these special developments could be translated back into the general logical framework with the same performance characteristics. There it would mean that special structural features of the axioms translate into strategical changes of the deductive mechanism (a single while rather complex one, mind you). Different theories share structural features. It would therefore be much more economical in the first place to focus on such features within the general mechanism rather than studying the theories independently of each other as mostly done these days. Also problems often arise in practice with no label attached as to which theory they belong. In other words, there is no indication in the literature of how, for a given problem, the appropriate specialized logic or procedure is to be selected from the hundreds (or thousands, if the trend continues) available.

Not only is the work reported in the literature too extensive to be fairly covered in the present section, but also is there a discrepancy between the motivation underlying a great deal of it and the motivation behind the work covered in the previous sections. We will therefore focus on important theories and mention some selected work rather than pretend to provide a balanced coverage of the known work.

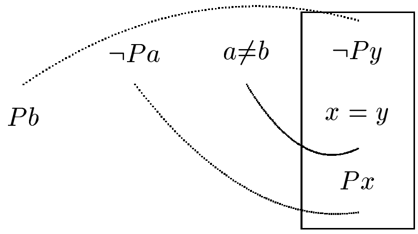
It is generally felt that in pure logic it does not make much sense to apply heuristics. This is because logic is intrinsically symmetric by nature, in the sense that for any heuristic speeding up the search for a proof for some class of theorems there is an infinite class of theorems for which it slows down the search if not even prevents it. Matters change radically in a particular theory. Here the symmetry is destroyed by the axioms of the theory. It thus makes a lot of sense to consider specialized strategies or heuristics exploiting the asymmetry in such a case. This will be the second issue briefly discussed in the present section.

Both issues just discussed are of great importance which is not reflected by the size of this section. Faced with the choice of ignoring the topics altogether or restricting the section to a very limited presentation we opted for the latter in order to emphasize its relevance.

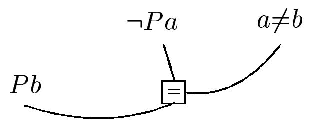
6.1 Equality and term rewriting

While it seems impractical to develop hundreds of different special deductive mechanisms for as many different theories, some theories are so fundamental and ubiquitous that in contrast to what we said above it is indeed quite worthwhile to develop specialized mechanisms for them. One of the most fundamental among these is the theory of equality. Equals may be substituted for equals which leads us to the rewriting of terms studied extensively for many particular areas. Since Chapter 5 of the present volume is devoted entirely to the treatment of equality and to term rewriting, the reader is referred to it for any details. We only point out the following connections with the material discussed so far.

Without any specialization, equality could be treated as any other predicate symbol with any of the methods presented in Section 2 by including the usual axioms of equality. These are reflexivity, commutativity, transitivity, substitutivity. If we continue to use the connection method as our canonical representation, then this means that the matrix would include a number of additional clauses representing instances of (some of) these axioms. In the following example the last clause represents an instance of the substitutivity axiom. The depicted connections provide a connection proof of the kind explained in Section 2.7.



In principle there is no problem with this kind of proof. However, the fact that in the presence of equality this kind of proof structure will appear in exactly the same way again and again suggests to shortcut the operations involved in the spirit of the principle of compression presented in Section 4.5. Exactly as in Section 4.5.2 we therefore abstract from the details of the boxed clause and introduce an *equality connection* depicted as follows.



The meaning of this connection is defined by the previous proof, as the latter picture is to be regarded simply as an abbreviation of the earlier one. Such equality connections were first introduced in [Bib87a] where a more detailed treatment of the topic under the present viewpoint may be found.

The compressed version of this kind of proof may now also induce a considerably enhanced computational treatment in terms of both finding the connection structure and computing the appropriate substitution. A solution to the latter problem has been offered in [GNRS92]. This kind of treatment is analogous to what paramodulation does in the context of resolution (see the subsequent chapter). In [Bib87a] it is demonstrated that term rewriting also is a form of compressing logical proofs in a more compact form.

6.2 Induction

Induction is treated in detail in Chapter 9 of the present volume. Therefore we restrict ourselves again to a few remarks.

The problem with induction was already mentioned in Section 2.4 which is that cut-elimination can longer be performed in its presence. Connection proofs may still be carried out for inductive problems in many restricted, but practically important cases (see [Bib87a].

Apart from such restricted cases, inductive problems may be attacked in a heuristic way only. The most important approach of this nature is the Boyer–Moore prover [BM90] which has been used to prove the correctness of large systems. [Bre86] reports a limited induction prover based on the connection method.

6.3 Mathematical theories

There are several ways to build the edifice of mathematics. Here we rely on the fact that virtually all of mathematics can be based on set theory. In other words a formalization of mathematics, needed for its deductive treatment, reduces to the formalization of *set theory*. It immediately induces a formalization of the natural and real numbers, ie. of *analysis*.

6.3.1 Set theory and analysis

There are basically three different ways of formalizing set theory within logic [vOQ63]. The first and most popular way consists in providing a set of axioms within first-order logic. There are basically two such sets, one due to Zermelo–Fraenkel (containing infinitely many axioms), the other to Gödel–Bernays. The first can be found in many logic texts (eg. in [Sho67]), for the second see [BLM+86].

The second way is known under the term *logicism* and has been originated by Frege and Russel. Its thesis is that basic mathematical concepts such as "number" are logical concepts so that the theorems concerning natural and real numbers are tautologies of logic. Recent work along this line of formalizing mathematical theories is presented in [Fef84, Gil86].

The third way of formalizing set theory is related to the second and consists in building set and number theory within the framework of higher-order logic as discussed in Section 5.5 already.

For any of these approaches we are already well equipped with sufficient deductive machinery discussed in the preceding sections in order to prove theorems automatically. For instance, the axioms and a first-order theorem prover is all needed for the first approach. Unfortunately it turns out that the strategies presently realized in deductive systems fail to prove deeper theorems. None of these provers, however, exploits the special structures inherent in the axioms, as indicated in the outset of the section, which may at least be one of the reasons for this relative failure.

This is not to say that interesting theorems could not be proved automatically so far in the area of set theory and analysis. [Sla72] represents early work in this area. More recently, especially Bledsoe and his group have achieved remarkable results in that they have been able to prove a number of interesting theorems in these areas automatically [Ble84]. Bledsoe takes the third of the three approaches and uses a number of heuristics in order to cope with the complexity of the search space.

Some of these heuristics seem particularly useful for these theories. For instance, one of the main problems in the search for proofs in higher-order logic is the instantiation of the existentially quantified variables. Since λ-conversion is involved here, the task becomes much more complicated than in first-order logic where unification does the work easily. In order to cope with this problem, Bledsoe has developed what he calles "ad hoc methods", or heuristic approaches, which provide reasonable instantiations of set variables in restricted cases. This most significant work on this problem has evolved in [Ble79, Ble83, Hin90].

There has been a longterm project, AUTOMATH [dB80], which laid its emphasis on *checking* rather than *finding* proofs. An entire book on analysis has been translated into AUTOMATH's formalism (see Section 5.5) and proved correct [vBJ77].

6.3.2 Arithmetic

The first theorem prover ever programmed was specialized for Presburger arithmetic [Dav60]. A much better method, known as the sup-inf method, has later been developed [Ble75, Sho77, Sho79] for dealing with arithmetic problems.

While the work just mentioned has been successful by restricting generality and drawing attention to special cases, one could alternatively focus on weaker formal systems of arithmetic. Primary candidates for this purpose are those reported in [Bus86, Tak90]. So far no work is known into such a direction.

6.3.3 Logic, algebra, and lattice theory

The combinatorial nature of logic and algebra make them primary candidates for automating proofs of their theorems. Because of that it is not surprising that the first milestone in automated deduction was achieved in the area of logic. In the years 1958/59 Hao Wang was able to implement a proof procedure which automatically proved all the 350 theorems in the book Principia Mathematica by Russel and Whitehead within a few minutes [Wan60].

A second milestone in the field is the automatic discovery of a proof for SAM's lemma which solved an open algebraic problem [GOBS69], the first instance of a machine solving a major open problem. Many of the theorems proved by OTTER (see Section 4.7) fall into the areas of logic and algebra [WWM$^+$90]. Some of them were also unproven conjectures until the machine discovered the proof.

6.3.4 Geometry

Another early system in the field is the geometry machine by Gelernter [Gel59, Gel63b, Gel63a], already mentioned in Section 4.6. As discussed there, it has drawn its strength from the use of semantic information involved in geometrical diagrams.

Geometry reasoning is particularly useful in the context of robotics. It is therefore of great interest for applications as demonstrated by the special Volume *37* of the Artificial Intelligence Journal on geometric reasoning as well as by the special Issue *2(4)* on the same subject. Due to a specialized technique, the Ritt–Wu Principle [Wu86], hundreds of non-trivial geometry theorems, some rather deep, could be proved automatically [CG90]. It is related to term-rewriting techniques applied to polynomials in several variables. The success is "handmade" in the sense that no general deductive method is integrated. Especially in this successful case, it would be extremely illuminating if one could understand the success in terms of a general logical mechanism in order to be able to carry over this success to other areas as well. This remains to be done though.

6.4 Proof representation

Proofs by mathematicians, even if they are rigorous, lack the precision to be understood by today's computer systems. Put the other way around, computer proofs are unintelligible for humans because they are overloaded with formalistic details which hide the relevant ideas behind the formal deductive chains. While this gulf is wide, it could be bridged, but there is a long way to go until this will happen in a completely satisfactory way. Here we only can only report very first steps in such a direction.

Gentzen's calculi of natural deduction are considered closest to natural proofs as presented by mathematicians. An important first step in bridging the gulf just discussed is therefore to provide transformations that relate any implemented proof method with the calculi of natural deduction. This has been the topic of Section 3 within the present chapter. On the basis of the relations reported there, any resolution or connection proof, say, can be transformed into a natural deduction proof, and vice versa. [And91] provides a rather comprehensive list of work on this topic including [Bib87a, Mil84, Pfe87, Ede92].

An important feature of human proofs is their structuring in the form of lemmas. [Lin90] has shown how to extract suitable lemmas from automatically generated proofs. Appropriate conceptualization is another important aspect of human proof representation which at present is not accessible by automatic means. Finally, one would like to translate even natural-deduction proofs into natural language proofs (if possible even including figures and drawings), and vice versa, a topic studied (with regard to the first direction) in [Che76].

6.5 Abstraction and analogy

A significant feature of human proof understanding and discovery is the use of abstraction and analogy. Once we have discovered a particular proof, humans would use their structure as a guidance in the search for proofs of analogical problems. While these features have been given some attention, little practical progress can be reported at this point.

[Pla80] studies basic concepts of abstractions. The proof plans discussed in [BvHHS91] are a form of abstracted proofs. One may think of abstracted proofs as of the connection structures remaining after abstraction from the details of the literals involved in the proof.

Analogy relates different things by abstraction. [Kli71] was the first attempt to incorporate analogy into a theorem prover. Little progress has been made on formal grounds in this respect (but see [BC87]). There is however a considerable literature on analogy in the context of learning from which we mention [Win80] as one example.

6.6 Deduction, inference and reasoning

A *deduction* is a relation of symbolic expressions, which is based on some calculus. The relation is established by applications of the rules of the calculus. The word “deduction” is also used to denote the area in which deductions and techniques to find them are studied, but in the following we will stick to the narrower sense.

Since deduction is a relation, no direction is inherent in its meaning. A formula may be deduced from a set of axioms (as in natural deduction systems), axioms may be deduced from a given formula (as in resolution where the empty clause is the axiom), or a structured formula may be deduced from some other structured formula (as in a connection calculus), and so forth.

Induction is a form of deduction. If we mean by induction mathematical induction, then this is obviously deduction, only in a special kind of calculus. If we understand induction in the sense of generalization and theory formation, then this form is of course also based on some rules of generalization which are applied in an inductive argument. These rules relate symbolic expressions, so we are dealing again with deduction.

Abduction is another form of deduction which relates with a given formula other formulas so that the usual deductive relation can be established among them. As we see, deduction is a very general concept. Any attempt to define it in a reasonable way such that induction and abduction is excluded is bound to fail.

Often precise calculi are not known to capture a form of reasoning in which case we prefer to speak of *inference* rather than deduction. In other words, deduction is “calculized” inference. The notion of *reasoning* is reserved to denote the corresponding *human* activity. However, the distinction is not followed by all authors in the literature in exactly the same way.

7 Conclusions

While there has been considerable progress and success in the short history of the field of automated deduction, spectacular breakthroughs have not taken place so far. The fundamental reason for the moderate success is the enormous complexity involved in the enterprise. The present volume as well as this chapter clearly demonstrate the amount of knowledge relevant for deductive systems. One simply cannot expect any longer a graduate student solving "the" deductive problem.

Fashion is changing over time. First, special systems were praised as the answer, then resolution along with a simple strategy raised high expectations, only to be condemned soon after as infeasible. More recently heuristics and knowledge became fashionable. We believe that the following conclusions can be drawn from all the experience accumulated over the first three decades of research in the area.

The most promising basis is still a uniform mechanism, especially one that integrates all the progress made in eliminating redundancy from proofs. On the basis of the results reported in Section 3 resolution is no longer the only candidate for such a mechanism. The relative and moderate success of resolution in a way has hindered faster progress.

What was wrong with the belief in resolution in the early sixties was not the uniform mechanism as such but the naive strategy applied to it. The strategy is the crucial issue as discussed in Section 4. It has to cope flexibly with different special cases and exploit their peculiarities. Unfortunately this is an extremely hard problem from which many people shy away. Alternatively, special systems are built for particular applications. This leaves us with the additional problem of how to integrate these systems and/or their ideas into a more uniform system. The two-level model discussed in Section 4 seems a promising way towards such ends.

Ultimately, the answer will be the trinity of, first, a uniform and compact formalism or calculus, leaving no room for much redundancy; second, a truly smart strategy, avoiding stupid search in irrelevant parts of the search space; and, finally, the incorporation of knowledge and heuristic guidance based on the accumulated experience of humans in their work in special areas of mathematics and science. All three of these are equally important which makes the entire enterprise so complicated since few people manage to cover them all.

In other words, while we share the emphasis of Bledsoe on heuristics in many of his writings, it is not a "switch" [Ble77] that will do the trick by itself. While a deeper understanding of how to integrate heuristic knowledge into the uniform mechanism may indeed lag behind, the basis of the entire trinity needs to be raised to a higher level, in order to come up with a more impressive success record.

Since the field has grown so huge, perhaps the most important obstacle for faster progress is the lack of coherence in the research worldwide. Coherence can only be achieved by insight into the problem as a whole. We hope that this chapter will have contributed to provide such insight into the problem of deduction and, thus, lead to a better coherence of future research in this field.

Acknowledgements. The two readers of the present chapter, Peter Andrews and David Plaisted, have provided numerous suggestions for improvements in content, style and presentation. They share a great deal of the quality in this work while the authors take full responsibility for any remaining weaknesses. We greatly appreciate their invaluable cooperation.

Thanks are also due to Bernd Jünger and Michael Thielscher who have coded many of the figures throughout the text partially using a matrix style developed by Gerd Neugebauer.

References

[AINP90] Peter B. Andrews, Sunil Issar, Dan Nesmith, and Frank Pfenning. The TPS theorem proving system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 641–642, Berlin, 1990. Springer.

[AMCP84] Peter B. Andrews, Dale A. Miller, Eve Longini Cohen, and Frank Pfenning. Automating higher-order logic. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years*, pages 169–192. Contemporary Mathematics *29*, American Mathematical Society, Providence RI, 1984.

[And76] Peter B. Andrews. Refutations by matings. *IEEE Transactions on Computers*, C-25:193–214, 1976.

[And81] Peter B. Andrews. Theorem proving via general matings. *Journal of the ACM*, 28:193–214, 1981.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Academic Press, Orlando, 1986.

[And89] Peter B. Andrews. On connections and higher-order logic. *Journal of Automated Reasoning*, 5:257–291, 1989.

[And91] Peter B. Andrews. More on the problem of finding a mapping between clause representation and natural-deduction representation. *Journal of Automated Reasoning*, 7:285–286, 1991.

[BB82] A. M. Ballantyne and W. W. Bledsoe. On generating and using examples in proof discovery. In *Machine Intelligence, vol. 10*, pages 3–39. Horwood, 1982.

[BC87] Thierry Boy de la Tour and Ricardo Caferra. Proof analogy in interactive theorem proving: A method to express and use it via second order pattern matching. In *Proc. AAAI-87*, pages 95–99, San Mateo CA, 1987. Morgan Kaufmann.

[BCEF84] W. Bibel, R. Caferra, E. Eder, and B. Fronhöfer. Extension of PROLOG through matrix reduction. In T. O'Shea, editor, *ECAI–84, Proceedings of the European Conference on Artificial Intelligence*, pages 101–104, Amsterdam, 1984. North–Holland.

[BdlT90] Thierry Boy de la Tour. Minimizing the number of clauses by renaming. In Jörg Siekmann, editor, *Proceedings of the 10th International Conference on Automated Deduction, Kaiserslautern, FRG*, pages 558–572, Berlin, July 1990. Springer-Verlag.

[BEF83] W. Bibel, E. Eder, and B. Fronhöfer. Towards an advanced implementation of the connection method. In *IJCAI-83, Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 920–922, Los Altos, 1983. Morgan Kaufmann.

[BEK+87] Stephan Bayerl, Elmar Eder, Franz Kurfeß, Reinhold Letz, and Johannes Schumann. An implementation of a prolog-like theorem prover based on the connection method. In Ph. Jorrand and V. Sgurev, editors, *Artificial Intelligence II—Methodology, Systems, Applications (AIMSA'86), Varna, Bulgaria*, pages 29–36, Amsterdam, 1987. Association pour la Promotion de l'Informatique Avancée (APIA), North-Holland.

[Bet55] Evert W. Beth. Semantic entailment and formal derivability. *Mededlingen der Koninklijke Nederlandse Akademie van Wetenschappen*, 18(13):309–342, 1955.

[Bet59] Evert W. Beth. *The Foundations of Mathematics*. North-Holland, Amsterdam, 1959.

[BH82] W. Bibel and K. M. Hörnig. Improvements of a tautology-testing algorithm. In D. Loveland, editor, *CADE–6, 6th Conference on Automated Deduction*, pages 326–341, Berlin, 1982. Springer, Lecture Notes in Computer Science *138*.

[BH84] W. Bibel and K. M. Hörnig. LOPS — A system based on a strategical approach to program synthesis. In A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 69–89. MacMillan, New York, 1984.

[Bib74] W. Bibel. An approach to a systematic theorem proving procedure in first-order logic. *Computing*, 12:43–55, 1974. First presented to the GI Annual Conference in 1971.

[Bib77] W. Bibel. A syntactic connection between proof procedures and refutation procedures. In H. Tschach, editor, *GI - 3. Fachtagung für Theoretische Informatik*, pages 215–225, Berlin, 1977. Springer, Lecture Notes in Computer Science *48*.

[Bib79] W. Bibel. Tautology testing with a generalized matrix reduction method. *Theor. Comput. Sci.*, 8:31–44, 1979.

[Bib80] W. Bibel. A theoretical basis for the systematic proof procedure. In P. Dembinski, editor, *Mathematical Foundations of Computer Science, Proceedings of the 9th Symposium MFCS'80*, pages 154–167, Berlin, 1980. Springer, Lecture Notes in Computer Science *88*.

[Bib81] W. Bibel. On matrices with connections. *Journal of ACM*, 28:633–645, 1981.

[Bib83] W. Bibel. Matings in matrices. *Comm. ACM*, 26:844–852, 1983.

[Bib85] W. Bibel. Automated inferencing. *Journal of Symbolic Computation*, 1:245–260, 1985.

[Bib86] W. Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986.

[Bib87a] W. Bibel. *Automated Theorem Proving*. Vieweg Verlag, Braunschweig, 2. edition, 1987.

[Bib87b] Wolfgang Bibel. *Automated Theorem Proving*. Artificial Intelligence. Vieweg, Braunschweig/Wiesbaden, second edition, 1987.

[Bib88a] W. Bibel. Advanced topics in automated deduction. In R. Nossum, editor, *Advanced Topics in Artificial Intelligence*, pages 41–59, Berlin, 1988. Springer, LNCS *345*.

[Bib88b] W. Bibel. Constraint satisfaction from a deductive viewpoint. *Artificial Intelligence Journal*, 35:401–413, 1988.

[Bib90] W. Bibel. Short proofs of the pigeonhole formulas based on the connection method. *Journal of Automated Reasoning*, 1990.

[Bib91] W. Bibel. Perspectives on automated deduction. In R. S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 77–104. Kluwer Academic, Utrecht, 1991.

[Bib92] W. Bibel. Intellectics. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. John Wiley, New York, 1992.

[BKA+87] W. Bibel, F. Kurfess, K. Aspetsberger, P. Hintenaus, and J. Schumann. Parallel inference machines. In P. Treleaven and M. Vanneschi, editors, *Future Parallel Computers*, pages 185–226, Berlin, 1987. Springer.

[Ble75] Woodrow W. Bledsoe. A new method for proving certain Presburger formulas. In *Proceedings of IJCAI-75*, pages 15–21. Morgan Kaufmann, 1975.

[Ble77] W. W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence Journal*, 9:1–35, 1977.

[Ble79] Woodrow W. Bledsoe. A maximal method for set variables in automatic theorem proving. In *Machine Intelligence 9*, pages 53–100. Ellis Horwood, Chichester, 1979.

[Ble83] Woodrow W. Bledsoe. Using examples to generate instantiations of set variables. In *Proceedings of IJCAI-83*, pages 892–901, Los Altos CA, 1983. Kaufmann.

[Ble84] Woodrow W. Bledsoe. Some automated proofs in analysis. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years*, pages 89–118. American Mathematical Society, Providence, Rhode Island, 1984.

[BLM+86] Robert Boyer, Ewing Lusk, William McCune, Ross Overbeek, Mark Stickel, and Lawrence Henschen. Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning*, 2:287–327, 1986.

[BLS87] W. Bibel, R. Letz, and J. Schumann. Bottom–up enhancements of deductive systems. In I. Plander, editor, *Proceedings of 4th International Conference on Artificial Intelligence and Information–Control Systems of Robots*, pages 1–10, Smolenice, CSSR, October 1987. North–Holland.

[BM90] Robert S. Boyer and J Strother Moore. A theorem prover for a computational logic. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 1–15, Berlin, 1990. Springer, Lecture Notes in Artificial Intelligence, vol. 449.

[Bre86] Michael Breu. Einbeziehung einfacher Induktionsbeweise in den Konnektionenkalkül. Diplomarbeit, Technische Universität München, 1986.

[BS75] W. Bibel and J. Schreiber. Proof search in a Gentzen–like system of first–order logic. In E. Gelenbe and D. Poitier, editors, *Proceedings of the International Computing Symposium*, pages 205–212, Amsterdam, 1975. North–Holland.

[Bun83] Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, London, 1983.

[Bur87] R. M. Burstall. Research in interactive theorem proving at Edinburgh University. In *Proceedings of the 20th IBM Computer Science Symposium, Shizuoka, Japan*, 1987.

[Bus86] Samuel R. Buss. *Bounded Arithmetic*. Bibliopolis, Napoli, 1986.

[BvHHS91] A. Bundy, F. van Hermelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. Mark Bromley, W. Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, Jim T. Sasaki, and Stuart F. Smith. *Implementing Mathematics with the NuPRL proof development system*. Prentice–Hall, Englewood Cliffs, NJ, 1986.

[CG90] Shang-Ching Chou and Xiao-Shan Gao. Ritt–Wu's decomposition algorithm and geometry theorem proving. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 205–220, Berlin, 1990. Springer.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Che76] Daniel Chester. The translation of formal proofs into English. *Artificial Intelligence Journal*, 7:261–278, 1976.

[CL73] C. L. Chang and R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

[CR74a] S. A. Cook and R. A. Reckhow. On the lengths of proofs in the propositional calculus. In *Proceedings 6th STOC*, pages 135–148. ACM, 1974.

[CR74b] Stephen A. Cook and Robert A. Reckhow. On the lengths of proofs in the propositional calculus. In *Proceedings of Sixth Annual ACM Symposium on Theory of Computing, Seattle, Washington*, pages 135–148, 1974. Corrections for this paper are in Sigact News, July 1974, Vol. 6, No. 3, pages 15–22.

[CR79] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(1):36–50, March 1979.

[CZ90] Ricardo Caferra and Nicolas Zabel. Extending resolution for model construction. In *European Workshop on Logics in Artificial Intelligence, JELIA'90*, Berlin, 1990. Springer.

[Dav60] Martin Davis. A computer program for Presburger's procedure. In *Summaries of talks presented at the Summer Institute for Symbolic Logic 1957*, Princeton NJ, 1960. Institute for Defense Analysis.

[Dav73] G. V. Davydov. The synthesis of the resolution method with the inverse method. *Journ. Soviet Mathematics*, 1:12–18, 1973. First published in: Zap. Naucn. Sem. Leningrad. Otdel. Mat. Inst. Steklov, 20, 24–35 (1971).

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of ACM*, 7:201–215, 1960.

[Ede85a] Elmar Eder. An implementation of a theorem prover based on the connection method. In W. Bibel and B. Petkoff, editors, *Artificial Intelligence, Methodology, Systems, Applications (AIMSA'84)*, pages 121–128, Amsterdam, New York, Oxford, 1985. European Coordinating Committee for Artificial Intelligence, North-Holland.

[Ede85b] Elmar Eder. Properties of substitutions and unifications. *Journal for Symbolic Computation*, 1:31–46, 1985.

[Ede87] Elmar Eder. Theorem proving and the connection method—the state of the art. *Rendiconti del Seminario Matematico, Università e Politecnico di Torino, Logic and Computer Science (1986)*, pages 93–114, 1987. Fascicolo Speciale.

[Ede91] Elmar Eder. Consolution and its relation with resolution. In *Proceedings IJCAI-91*, Los Altos, 1991. Morgan Kaufmann.

[Ede92] Elmar Eder. *Relative Complexities of First Order Calculi*. Vieweg, Braunschweig, 1992.

[Fef84] Solomon Feferman. Towards useful type-free theories, I. *Journal for Symbolic Logic*, 49:75–111, March 1984.

[Fit90] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[Fre79] Gottlob Frege. *Begriffsschrift*. Louis Nebert, Halle, 1879.

[Fro85] Bertram Fronhöfer. On refinements of the connection method. In J. Demetrovics, G. Katona, and A. Salomaa, editors, *Algebra, Combinatorics and Logic in Computer Science*, pages 391–401, Amsterdam, 1985. North-Holland.

[Gal86] J. H. Gallier. *Logic for Computer Science: Foundations of Automated Theorem Proving*. Harper and Row, New York, 1986.

[Gel59] H. Gelernter. Realization of a geometry theorem-proving machine. In *Proc. IFIP*, pages 273–282. Paris UNESCO House, 1959.

[Gel63a] H. Gelernter. Empirical explorations of the geometry theorem-proving machine. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 153–163. McGraw Hill, New York, 1963.

[Gel63b] H. Gelernter. Realization of a geometry theorem-proving machine. In *Computers and Thought*, pages 134–152. McGraw Hill, New York, 1963.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210 and 405–431, 1935. Engl. transl. in [Sza69].

[Gil86] Paul C. Gilmore. Natural deduction based set theories: A new resolution of old paradoxes. *Journal of Symbolic Logic*, 51:393–411, 1986.

[Gil89] Paul C. Gilmore. How many real numbers are there? Technical Report TR 89-7, The University of British Columbia, Dept. Computer Sc., 1989.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[GNOP82] S. Greenbaum, A. Nagasaka, P. O'Rorke, and D. A. Plaisted. Comparison of natural deduction and locking resolution implementations. In D. Loveland, editor, *Proceedings of the 6th Conference of Automated Deduction*, pages 159–171. Springer Lecture Notes in Computer Science 138, 1982.

[GNRS92] Jean Gallier, Paliath Narendran, Stan Raatz, and Wayne Snyder. Theorem proving using equational matings and rigid E-unification. *J.ACM*, 39(2), 1992. To appear.

[GOBS69] J. Guard, F. Oglesby, J. Bennet, and L. Settle. Semi-automated mathematics. *J. ACM*, 18:49–62, 1969.

[Göd30] Kurt Gödel. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37:349–360, 1930. Engl. transl. in [Hei67], pp. 582–591.

[Gol71] W. D. Goldfarb, editor. *J. J. Herbrand — Logical writings*. Reidel, Dordrecht, 1971.

[Gol81] W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[Hak84] Armin Haken. *The intractability of resolution.* PhD thesis, University of Illinois, Urbana, 1984. PhD Thesis.

[Hak85] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.

[HB39] David Hilbert and P. Bernays. *Grundlagen der Mathematik*, volume 1 und 2. Springer, Berlin, 1934-39.

[HC68] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* Methuen, London, 1968.

[HD83] Jieh Hsiang and Nachum Dershowitz. Rewrite methods for clausal and non-clausal theorem proving. In Joseph Díaz, editor, *Automata, Languages and Programming, 10th Colloquium, ICALP 83, Barcelona, Spain, July 1983*, pages 331–346, Berlin, Heidelberg, New York, Tokio, 1983. Springer-Verlag.

[Hei67] Jean van Heijenoort. *From Frege to Gödel. A Source Book in Mathematical Logic, 1879–1931.* Source Books in the History of the Sciences. Harvard University Press, Cambridge, Massachusetts, 1967.

[Hen87] Pascal Van Hentenryck. *Consistency Techniques in Logic Programming.* PhD thesis, University Namur, Belgium, 1987.

[Her30] J. J. Herbrand. Recherches sur la théorie de la démonstration. *Travaux Soc. Sciences et Lettres Varsovie, Cl. 3 (Mathem., Phys.)*, page 128 pp., 1930. Engl. transl. in [Gol71].

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the Symposium on Logic in Computer Science, Ithaca, New York, June 1987*, pages 194–204. IEEE Computer Society Press, 1987.

[Hin90] Larry Hines. Str$\dot{+}$ve$\subseteq$: The Str$\dot{+}$ve-based subset prover. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 193–206, Berlin, 1990. Springer.

[HLO$^+$80] Larry J. Henschen, E. Lusk, R. Overbeek, B. T. Smith, R. Veroff, S. Winker, and L. Wos. Challenge problem 1. *SIGART Newsletter*, 72:30–31, 1980.

[Hör81] K. M. Hörnig. Generating small models of first order axioms. In J. Siekmann, editor, *Proc. German Workshop on Artificial Intelligence*, pages 248–255, Berlin, 1981. Springer, Fachberichte Informatik 47.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hue87] Gérard Huet. Deduction and computation. In W. Bibel and Ph. Jorrand, editors, *Fundamentals of Artificial Intelligence — An Advanced Course*, pages 39–74. Springer, LNCS *232*, Berlin, 1987.

[Iss90] Sunil Issar. Path-focused duplication: A search procedure for general matings. In *AAAI-90*, pages 221–226, Palo Alto CA, 1990. American Association for Artificial Intelligence.

[Joh89] C. A. Johnson. Factorisation and circuit in the connection method. *J.ACM*, 1989. Submitted.

[JP76] D. C. Jensen and T. Pietrzykowski. Mechanizing ω-order type theory through unification. *Theoretical Computer Science*, 3:123–171, 1976.

[KK84] W. Kneale and M. Kneale. *The Development of Logic*. Clarendon Press, Oxford, 1984.

[Kli71] R. E. Kling. A paradigm for reasoning by analogy. *Artificial Intelligence Journal*, 2:147–178, 1971.

[Koh70] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, 1970.

[Lin90] Christoph Lingenfelder. *Transformation and Structuring of Computer Generated Proofs*. PhD thesis, Univ. Kaiserslautern, 1990.

[Lov69] D. W. Loveland. A simplified format for the model elimination theorem-proving procedure. *J.ACM*, 16(3):349–363, 1969.

[Lov78] D. W. Loveland. *Automated Theorem Proving: A logical basis*. North-Holland, Amsterdam, 1978.

[Lov87] D. W. Loveland. Near Horn Prolog. In J. Lassez, editor, *4th International Conference in Logic Programming*, pages 456–469, Cambridge, 1987. MIT Press.

[LP92] Shie-Jue Lee and David A. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9(1):25–42, 1992.

[LSBB92] Reinhold Letz, Johannes Schumann, Stephan Bayerl, and Wolfgang Bibel. SETHEO — A high-performance theorem prover for first-order logic. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[Mac77] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence Journal*, 8:99–118, 1977.

[Mac87] A. K. Mackworth. Constraint satisfaction. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205–211. J. Wiley and Sons, New York NY, 1987.

[Mas68] S. J. Maslov. The inverse method for establishing deducibility for logical calculi. *Trudy Mathemat. Inst. AN USSR*, 98:26–87, 1968.

[MB88] J. C. Madre and J. P. Billon. Proving circuit correctness by formally comparing their expected and extracted behavior. In *Proceedings 25th Design Automation Conference*, pages 205–210. IEEE Comp. Society, 1988.

[MCA82] Dale A. Miller, Eve Longini Cohen, and Peter B. Andrews. A look at TPS. In D. W. Loveland, editor, *6th Conference on Automated Deduction*, pages 50–69, Berlin, 1982. LNCS *138*, Springer.

[MF85] A. K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence Journal*, 25:65–74, 1985.

[Mil84] D. A. Miller. Expansion trees and their conversion to natural deduction proofs. In R. E. Shostak, editor, *7th International Conference on Automated Deduction*, pages 375–393, Berlin, 1984. Springer.

[Mil87] Dale A. Miller. A compact representation of proofs. *Studia Logica*, 46:345–368, 1987.

[Min88] Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Los Altos, 1988.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, vol. IV*, pages 153–175. North–Holland, Amsterdam, 1982.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory Lecture Notes. Bibliopolis, Napoli, 1984.

[Mor82] B. M. E. Moret. Decision trees and diagrams. *ACM Computing Surveys*, 14(4):593–623, 1982.

[MR91] Neil V. Murray and Erik Rosenthal. Dissolution: Making paths vanish. *Journal of Automated Reasoning*, 1991.

[Nos85] Ralf Nossum. Automated theorem proving methods. *BIT*, 25:51–64, 1985.

[NP90] Xumin Nie and David A. Plaisted. A complete semantic back chaining proof system. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 16–27, Berlin, 1990. Springer.

[Ohl88] H. J. Ohlbach. A resolution calculus for modal logics. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 500–516, Berlin, 1988. LNCS *310*, Springer.

[Per85] Donald Perlis. Languages with self-reference. *Artificial Intelligence Journal*, 25:301–322, 1985.

[Pet90] Uwe Petermann. Towards a connection procedure with built in theories. In *European Workshop on Logics in Artificial Intelligence, JELIA-90*, pages 444–453, Berlin, 1990. Springer LNCS vol. 479.

[Pfe87] Frank Pfenning. *Proof transformations in higher-order logic.* PhD thesis, Carnegie Mellon University, Pitsburgh, 1987.

[Pfe88] Frank Pfenning. Single axioms in the implicational propositional calculus. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, pages 710–713, Berlin, 1988. LNCS 310, Springer.

[PG86] David Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[Pip90] Nick Pippenger, 1990. private communication.

[Pla80] David A. Plaisted. Abstractions mappings in mechanical theorem proving. In W. Bibel and R. Kowalski, editors, *5th International Conference on Automated Deduction*, pages 264–280, Berlin, 1980. LNCS 87, Springer.

[Pla88] D. A. Plaisted. Non-horn clause logic programming without contrapositives. *Journal of Automated Reasoning*, 4(3):287–325, 1988.

[Poo84] D. L. Poole. Making clausal theorem provers non-clausal. In *Proc. CSCSI/SCEIO Conf.*, pages 124–125, London, Ontario, 1984.

[Pra60] Dag Prawitz. An improved proof procedure. *Theoria*, 26:102–139, 1960.

[Pra65] Dag Prawitz. *Natural Deduction.* Almquist and Wiksell, Stockholm, 1965.

[Pra70] Dag Prawitz. A proof procedure with matrix reduction. In M. Laudet et al., editor, *Symposium on Automatic Demonstration, Lecture Notes in Mathem. 125*, pages 207–214. Springer, Berlin, 1970.

[RdK87] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems. In *AAAI'87*, pages 183–188, Los Altos, 1987. AAAI.

[Rec76] Robert Allen Reckhow. *On the Lengths of Proofs in the Propositional Calculus*. PhD thesis, University of Toronto, 1976.

[Rei83] Raymond Reiter. Towards a logical reconstruction of relational database theory. In M. L. Brodie et al., editor, *On Conceptual Modeling*, pages 191–238. Springer, Berlin, 1983.

[Rus08] Bertrand Russel. Mathematical logic as based on the theory of types. *American J. of Mathematics*, 30:222–262, 1908. Reprinted in [vH67].

[SC77] H. Saya and Ricardo Caferra. A structure sharing technique for matrices and substitutions in Prawitz' theorem proving method. Rapport 101, University of Grenoble, France, 1977.

[Sch77] Kurt Schütte. *Proof Theory*. Grundlehren der mathematischen Wissenschaften 225. Springer-Verlag, Berlin, 1977. English and revised version of: Beweistheorie (1960).

[SG89] Wayne Snyder and Jean Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1,2):101–140, 1989.

[Sha38] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of AIEE*, 57:713–723, 1938.

[Sho67] Joseph R. Shoenfield. *Mathematical Logic*. Addison–Wesley, Reading MA, 1967.

[Sho76] Robert E. Shostak. Refutation graphs. *Artificial Intelligence Journal*, 7:51–64, 1976.

[Sho77] Robert E. Shostak. On the sup-inf method for proving Presburger formulae. *J. ACM*, 24(4):529–543, 1977.

[Sho79] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *J. ACM*, 26(2):351–360, 1979.

[Sic76] Sharon Sickel. A search technique for clause interconnectivity graphs. *IEEE Transactions on Computers*, 35:823–835, 1976.

[Sko20] Thoralf Skolem. Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theoreme über dichte Mengen. *Videnskapsselskapets skrifter, I. Matematisk-naturvidenskabelig klasse*, (4), 1920.

[Sla67] J. R. Slagle. Automatic theorem proving with renamable and semantics resolution. *Journal of ACM*, 14(4):687–697, 1967.

[Sla72] James R. Slagle. Automatic theorem proving with built-in theories including equality, partial ordering, and sets. *J. ACM*, 19(1):120–135, 1972.

[Smu71] Raymond M. Smullyan. *First-Order Logic*. Ergebnisse der Mathematik und ihrer Grenzgebiete. Springer-Verlag, Berlin, Heidelberg, New York, 1971.

[Sti85] Mark E. Stickel. Automated deduction by theory resolution. *Journal of Automated Reasoning*, 1:333–355, 1985.

[Sti88] M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

[Sti90] Mark E. Stickel. A Prolog technology theorem prover. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 673–674, Berlin, 1990. Lecture Notes in Artificial Intelligence *449*, Springer.

[Sza69] M. E. Szabo, editor. *The collected papers of Gerhard Gentzen*. North-Holland, Amsterdam, 1969.

[Tak90] Gaisi Takeuti. s_3^i and $\overset{o\,i}{V}_2$ (bd). *Archiv f. Mathematische Logik*, 1990.

[Tse70] G. S. Tseitin. On the complexity of derivations in propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125. Consultants Bureau, New York, 1970. part II.

[vBD83] Johan van Benthem and Kees Doets. Higher-order logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, chapter 4, pages 275–329. Reidel, Dordrecht, 1983.

[vBJ77] L. S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system*. PhD thesis, Eindhoven University of Technology, Eindhoven NL, 1977.

[vH67] J. v. Heijenoort, editor. *From Frege to Gödel*. Harvard University Press, Cambridge Mass., 1967.

[vOQ63] Willard van Orman Quine. *Set Theory and its Logic*. Belknap Press, Cambridge MA, 1963.

[Wal90] Lincoln A. Wallen. *Automated Deduction in Non-Classical Logics*. MIT Press, Cambridge, Mass., 1990.

[Wan60] Hao Wang. Proving theorems by pattern recognition, Part I. *Comm. ACM*, 3:220–234, 1960.

[Wan85] T. C. Wang. Designing examples for semantically guided hierarchical deduction. In *IJCAI'85*, pages 1201–1207, Morgan Kaufmann, Los Altos CA, 1985.

[Wil52] Raymond L. Wilder. *The foundations of mathematics*. Wiley, New York, 1952.

[Win80] Patrick H. Winston. Learning and reasoning by analogy. *Comm. ACM*, 23(12):689–703, 1980.

[WOLB84] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. Prentice-Hall, Englewood Cliffs, 1984.

[WP90] Toshiro Wakayama and T. H. Payne. Case-free programs: An abstraction of definite horn programs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 87–101, Berlin, 1990. Springer.

[WR13] A. N. Whitehead and Bertrand Russel. *Principia Mathematica*, volume 1 – 3. The University Press, Cambridge, 1st edition, 1910-13.

[Wu86] Wen-tsün Wu. Basic principles of mechanical theorem proving in geometries. *Journal of Automated Reasoning*, 2(4):221–251, 1986.

[WWM+90] L. Wos, S. Winker, W. McCune, R. Overbeek, E. Lusk, and R. Stevens. Automated reasoning contributes to mathematics and logic. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 485–499, Berlin, 1990. Springer.

[Zam90] Nail Kalimovitch Zamov. On the connection between the resolution method and the inverse method. *Computers and Artificial Intelligence*, 9:151–157, 1990.

Index